Herschel Data Analysis Guide

Formerly known as HowTo Documents

Version 3.0, Document Number: HERSCHEL-HSC-DOC-1199
06 May 2010

Herschel Data Analysis Guide: Formerly known as HowTo Documents

Table of Contents

Preface

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

- Data input/output tools
- Data display tools
- Plotting tools
- Image analysis tools
- Spectral analysis tools
- External tools

Each chapter, except the last one, contains three main sections:

- **Summary:** shows you what you will find in the chapter.
- **How to:** quick instructions to carry out the most common tasks. Look at this section if you want to become familiar with the software, or if you need to get something done quickly.
- **In depth:** thorough explanation of all the features. Read this if you are looking for advanced options not covered in the previous section, or if you want to become a Herschel data analysis guru.

Chapter 1 contains an additional section, *Basic concepts*.

Tip

For last-minute documentation fixes and updates, please see this page:

http://herschel.be/twiki/bin/view/Public/DocumentationErrata

Tip

If you are interested in more advanced features, including scripting, batch processing and data analysis, please have a look at the *Scripting and Data Mining* guide.

Note

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

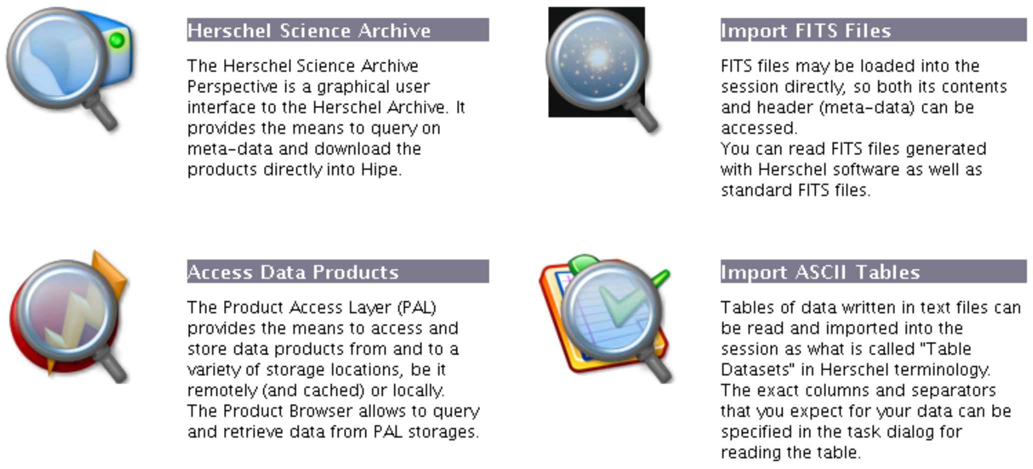
Chapter 1. Data input/output

1.1. Summary

This chapter tells you everything you need to know about getting data *into* HIPE from a variety of sources and exporting data *from* HIPE to a variety of destinations.

1.1.1. The four pillars of data exchange

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



Herschel Science Archive

The Herschel Science Archive Perspective is a graphical user interface to the Herschel Archive. It provides the means to query on meta-data and download the products directly into Hipe.

Import FITS Files

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

Access Data Products

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

Import ASCII Tables

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

These are two pillars you will use most often:

- **Herschel Science Archive:** this is the place you get your data from.
- **Access Data Products:** a mechanism made of *storages* and *pools* to help you store, query and retrieve Herschel data on your computer.

The other two pillars will come in handy especially if you want to exchange data with external applications:

- **Import FITS files:** save your data to FITS and import external FITS files. HIPE will do its best to determine what's inside the file and act accordingly.
- **Import ASCII tables:** save and read data as text-only files in a variety of formats.

1.1.2. Typical procedures

The following lists show some procedures you will likely follow during your work on Herschel data, and give links to the sections explaining each step.

- **Quick inspection of data from the Herschel Science Archive.** Recommended for inspecting the contents of observations and retrieving small data sets. See Section 1.3.1, Section 1.3.2 and Section 1.3.3. Data are browsed online; for the recommended ways of saving one or many observations on your computer, see the next two points.
- **Retrieval of a single observation from the Herschel Science Archive.** See Section 1.3.1, Section 1.3.2, Section 1.3.4, Section 1.3.6.
- **Retrieval of multiple observations from the Herschel Science Archive.** The recommended method for downloading observations from the Herschel Science Archive. See Section 1.3.1, Section 1.3.2, Section 1.3.5, Section 1.3.6.

- **Saving data products into local storage.** Storing your data into *pools* on your computer. See Section 1.3.9 and Section 1.3.10 (via a graphical interface) or Section 1.4.1 (via the command line).
- **Querying and retrieving data products from local storage.** Searching for data products on your computer and loading them into HIPE. See Section 1.3.7 (via a graphical interface) or Section 1.4.1 (via the command line).

1.2. Basic concepts

1.2.1. Data structures

Data in Herschel is characterised by an *onion-like* structure, made of several layers. The basic data structure is the *product*. A product contains *metadata* and one or more *datasets*.

Datasets contain numeric data organised in tabular form, in one or more dimensions (up to five). There are many types of dataset: the most common is perhaps the *table dataset*, which holds *columns* organised in tabular form. More specialised datasets exist as well (for instance, `Spectrum1d` and `Spectrum2d`). Finally, a *composite dataset* is a special dataset that contains other datasets.

Products can themselves be grouped into a *context*, a special product that can hold references to other products (including other contexts). The most obvious example is the *observation context*, which contains an entire Herschel observation.

For advanced information on products and datasets, see the *Scripting and Data Mining* guide: Chapter 2.

1.2.2. Pools and storages

Data products are stored into *pools*, which are grouped into *storages*. A *pool* is a mini-database which you can use to save, retrieve and query data products. Every pool must be *registered* to a storage.

There are many types of pools, for handling both local and remote data. The *local store* is probably the one you will use most often. As the name suggests, this pool is held locally on your system, usually in a `.hcss/lstore` directory under your home directory. Although products are stored as FITS files, you should use the graphical tools provided by HIPE and described in this chapter (see for instance Section 1.3.7) rather than manipulating the files directly.

For advanced information on pools and storages, see the *Scripting and Data Mining* guide: Appendix A.

1.2.2.1. Update of index format for local stores

Warning

Please read the contents of this section carefully. **Failure to do so may result in permanent corruption of your data.**

The internal format of local store indexes has changed in HIPE 3.0. When you first execute HIPE 3.0, a dialogue window appears, asking you to update your local stores to the new format. The window shows a list of the local stores found by HIPE. Click on the stores you want to update at this stage. Hold **Ctrl** to select multiple stores. To select a contiguous range, click on the first store of the range, then click on the last one while holding **Shift**.

Once you have selected the stores to update, click *Yes* to start the update, or *No* to close the window without taking any action. Note that the time needed for the update varies roughly linearly with the size and number of products in the local stores, and it can range from a few seconds to **several hours.** During the update, another window shows the progress of the operation. You can click *Cancel* at any moment to interrupt the update.

Turning on automatic updates. You can set a property so that HIPE automatically updates a local store when it is accessed. To activate automatic updates, issue this command in the *Console* view of HIPE:

```
Configuration.setProperty("hcss.ia.pal.pool.lstore.index.autoupdate", -"true")
```

This setting will not be kept if you quit HIPE. You can make the setting permanent by adding the following line to your `user.props` file, located in the `.hcss` directory within your home directory (create the file if it does not exist already):

```
hcss.ia.pal.pool.lstore.index.autoupdate = true
```

Set the property to `false` to turn off automatic updates.

Note that, if you leave this property set as `true` in a configuration file, any future update of the index format of local stores will be done automatically. We recommend that you set the property back to `false` once you have updated all your local stores.

Accessing legacy local stores with HIPE 3.0. You may have chosen not to update your local stores at the first execution of HIPE 3.0, or there may be local stores in non-standard locations that HIPE did not find. If automatic updating is turned off, when you access a local store in legacy format HIPE will inform you that you need to update it manually (see *Updating a local store manually* below for how to do it).

Accessing updated local stores with HIPE 2.x. Local stores updated by HIPE 3.0 *will no longer be accessible with HIPE 2.x*. If you plan to access your data with a legacy HIPE version in the future, please make a backup before updating with HIPE 3.0 (see *Troubleshooting* below for how to make a backup).

Updating a local store manually. With automatic updating off, you can update a local store manually in HIPE with the following commands:

```
pool = LocalStoreFactory.getStore ("pool_name")
pool.rebuildIndex()
```

Troubleshooting. A backup of the index files is made automatically. You can of course still make your own backup copy of the entire pool if you like. Note that the data files themselves are not modified.

If you cancel the update before it has finished, or in the unlikely event that an error occurs, you can switch back to the legacy index format by following these steps:

- Enter the directory of the local store
- Rename the `.index` directory to `.index_new`
- Rename the directory `.index_bak` to `.index`

You can switch back to the new format by following the same steps in reverse order.

If you want to be able to switch versions, do not write to the local store using the HIPE 3.0. If you do, switching will not give correct results, because the old index will no longer reflect the contents of the local store. For maximum security, you may want to backup the entire directory of the local store, containing the index *and* the data.

1.2.3. Observation contents

Herschel observations (more formally, *observation contexts*) contain many data products, grouped in several contexts. The following is the structure of a typical observation, common to all instruments and all observing modes:

History:	Contains the automatically generated script of actions performed on your data, a history of the tasks applied to the data, and the parameters belonging to those tasks.
Auxiliary Context:	All Herschel non-science spacecraft data required directly or indirectly in the processing and analysis of the scientific data.
Calibration Context:	The parameters that characterise the behaviour of the satellite and the instruments. Used for reprocessing data.
Level-0 Context:	Raw data, minimally manipulated.
Level-0.5 Context:	Data processed to an intermediate point adequate for inspection
Level-1 Context:	Detector readouts calibrated and converted to physical units, in principle instrument and observatory independent.
Level-2 Context:	Scientific analysis can be performed. These data products are at a publishable quality level and should be suitable for Virtual Observatory access.
Level-3 Context (optional):	Publishable science products with level 2 data products as input. Possibly combined with theoretical models, other observations, laboratory data, catalogues, etc. Formats should be Virtual Observatory compatible.
LogObsContext:	A log of actions performed on the Products in the ObservationContext
Quality Context:	Issues flagged by the pipelines that indicate possible issues with the quality of the data or pipelining. An empty quality report indicates no problems in processing.
Trend Analysis Context	Products useful for tracking systematic changes in instrument response over time.
Telemetry Context:	Optional - only included when the HSC deems it necessary because of a serious problem in the processing to level-0 data.

1.3. How to

1.3.1. Accessing the Herschel Science Archive from HIPE

You can access the Herschel Science Archive User Interface (HUI), via the *Herschel Science Archive* view, shown in Figure 1.1. This view is part of the *Herschel Science Archive* perspective.

You can access this view in three ways:

- Open the *Welcome* perspective (Help → Welcome!), click on the *Access Data* icon and then on the *Herschel Science Archive* icon.
- Choose Window → Show Perspectives → Herschel Science Archive
- Choose Window → Show View → Herschel Science Archive

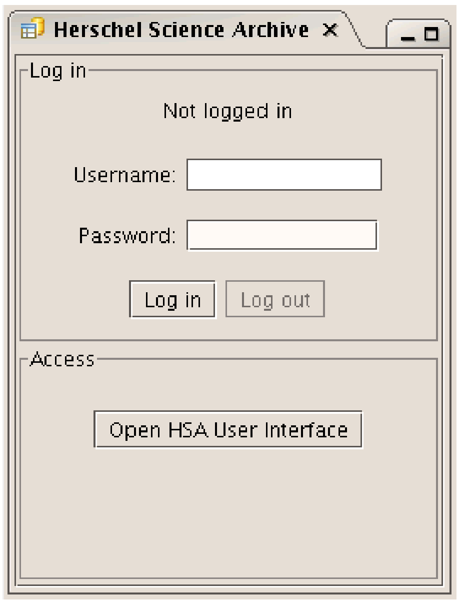


Figure 1.1. The Herschel Science Archive view

Login first, and then click on *Open HSA User Interface* to access data in the HSA. Your credentials will be automatically transferred to the HUI. Note that if an HUI was opened before starting HIPE, opening a new one is not needed as the Plastic connection will be established automatically between them. However, in this case you will need to login separately in both applications, HUI and HIPE.

Warning

If you get a message about *Java WebStart* (`javaws`) not being present, it probably means you are using a 64-bit version of Java prior to 1.6 update 12 (1.6u12). To find out which version of Java you have installed, issue this command from a terminal window:

```
java --version
```

Java WebStart is a piece of software needed to fetch from the Internet the HSA User Interface. You can obtain it either by switching to a 32-bit version of Java or (recommended) by updating to Java 1.6u12 or newer.

1.3.2. Querying the HSA

The Herschel Science Archive User Interface (HUI) opens on the *Query Specification* window: see Figure 1.2.

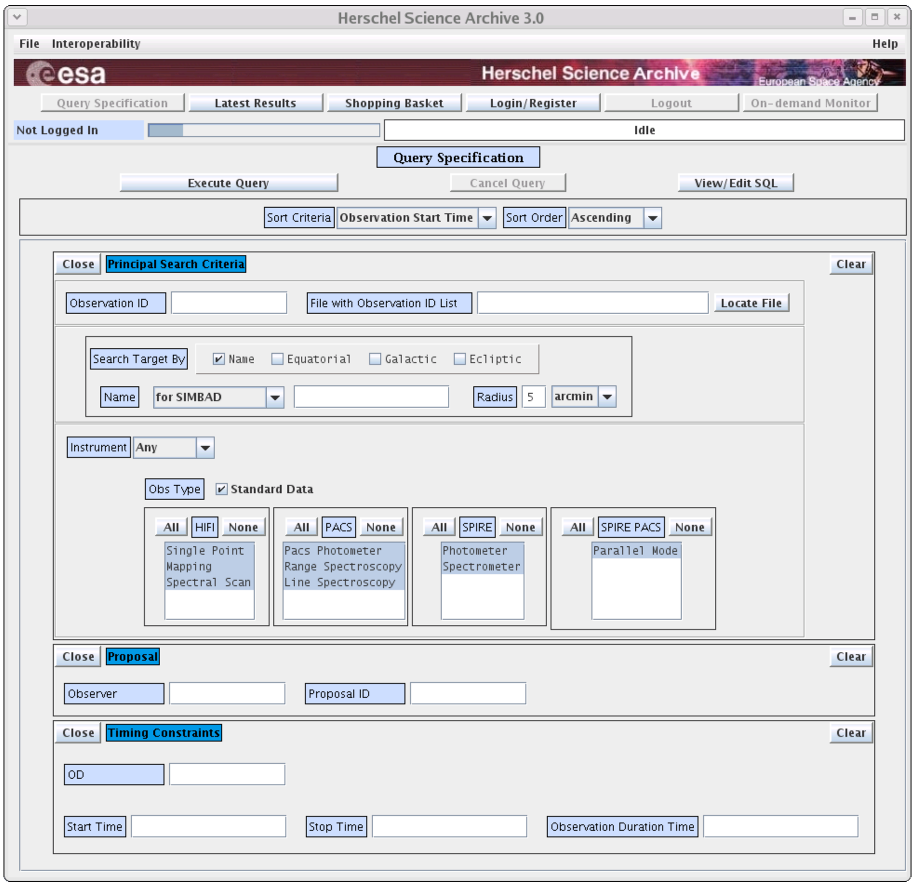
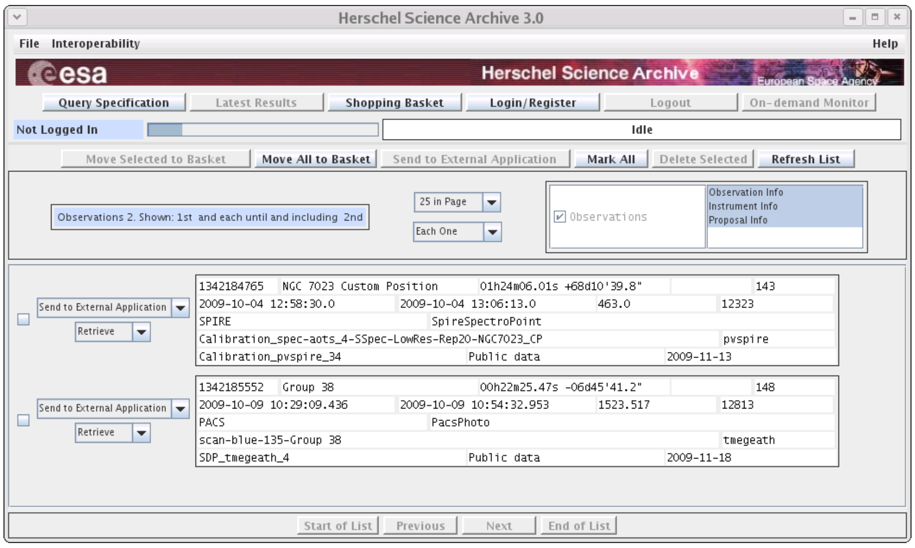


Figure 1.2. The HUI Query Specification window.

Any label in the HUI that changes colour under the mouse pointer has an associated help text. Click on the label to open the help window (note that some labels still do not have help text available). There is also a Help menu at the top right corner.

To define your query, set all the relevant fields in the *Query Specification* window. You can also click on *View/Edit SQL* to view and modify the corresponding SQL statement. Click on *Execute Query* to run your query. The *Latest Results* window opens (see Figure 1.3) which contains the list of observations matching the query.



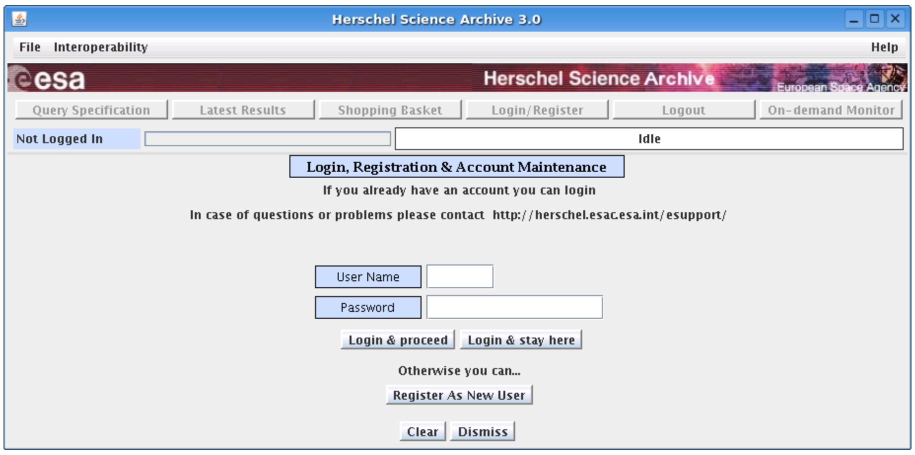
Observations 2. Shown: 1st and each until and including 2nd

```
1342184765  NGC 7023 Custom Position      01h24m06.01s +68d10'39.8"          143
2009-10-04 12:58:30.0      2009-10-04 13:06:13.0       463.0        12323
SPIRE                      SpireSpectroPoint
Calibration_spec-aots_4-SSpec-LowRes-Rep20-NGC7023_CP                   pvspire
Calibration_pvspire_34               Public data               2009-11-13

1342185552  Group 38                      00h22m25.47s -06d45'41.2"          148
2009-10-09 10:29:09.436    2009-10-09 10:54:32.953     1523.517     12813
PACS                       PacsPhoto
scan-blue-135-Group 38                                                  tmegeath
SDP_tmegeath_4                       Public data               2009-11-18
```

Figure 1.3. Result of query of the HSA

Note that you do *not* need to be a registered Herschel user to query the archive and browse its contents. You do need to register if you want to retrieve data. To register with the Herschel system please go to Herschel Archive Registration and follow the appropriate instructions. This registration page is also accessible through the *Login/Register* page of the HUI (*Register as New User*) Figure 1.4).


Herschel Science Archive 3.0

File Interoperability Help

Herschel Science Archive

Query Specification | Latest Results | Shopping Basket | Login/Register | Logout | On-demand Monitor

Not Logged In | Idle

Login, Registration & Account Maintenance

If you already have an account you can login

In case of questions or problems please contact http://herschel.esac.esa.int/esupport/

User Name

Password

Login & proceed | Login & stay here

Otherwise you can...

Register As New User

Clear | Dismiss

Figure 1.4. Login/registration in the HSA.

Only authorised users can access data covered by proprietary rights. The same rule applies to the viewable quick-look products of observations, as well as to proposal-related files. They can only be viewed by the observation owner, provided he or she has logged in with his/her registration identifier.

1.3.3. Browsing HSA data from HIPE

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

Open the HSA User Interface as described in Section 1.3.1. Query the HSA for the data to be retrieved. In the query result panel, close to every observation item, you will see a drop-down list called *Send to External Application*. With this you can choose whether to browse the whole observation or just a portion of it.

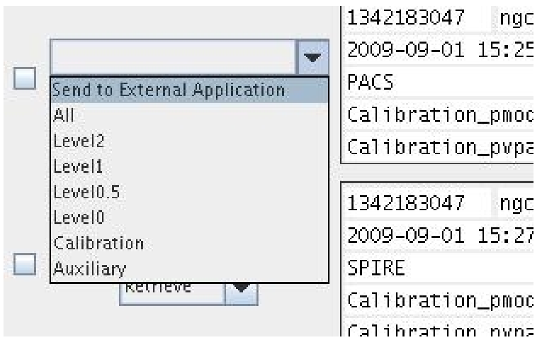
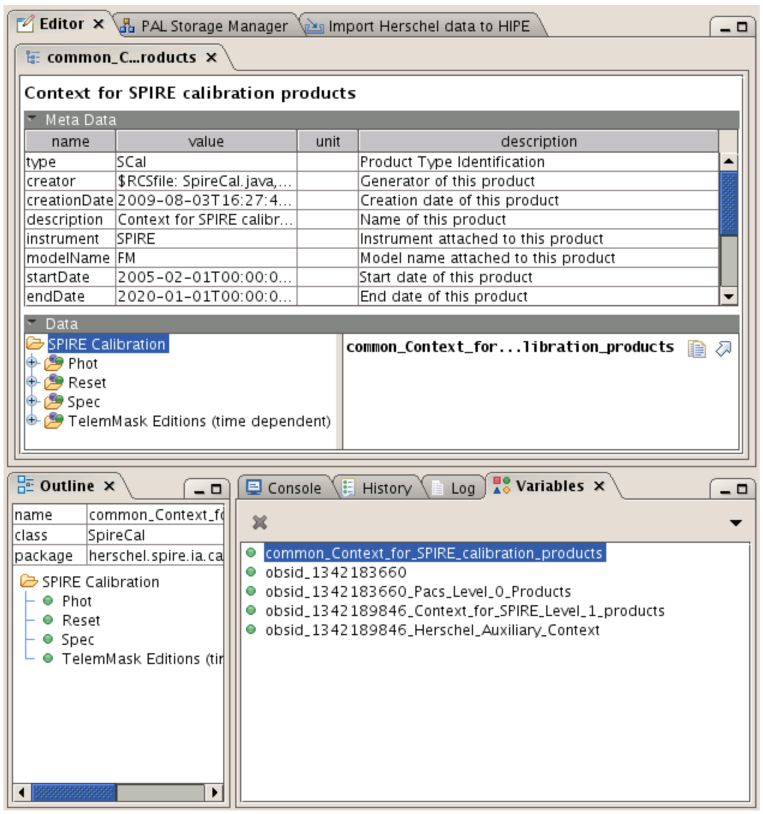


Figure 1.5. Selecting which part of an observation to browse.

Select an option and, if the connection between the two applications (HUI and HIPE) is well established, a pop-up window appears with the message *Request sent successfully to external application*. Data starts to load into HIPE automatically. During the operation an indicator shows in the HIPE display that loading is taking place and the system is busy.

If the option *All* was selected, a variable called `obsid_xxxxxxxxxx` is created in HIPE, with an actual observation number. Other options are also stored in different variables as illustrated in Figure 1.6.



Context for SPIRE calibration products

Meta Data

name	value	unit	description
type	SCal		Product Type Identification
creator	$RCSfile: SpireCal.java,…		Generator of this product
creationDate	2009-08-03T16:27:4…		Creation date of this product
description	Context for SPIRE calibr…		Name of this product
instrument	SPIRE		Instrument attached to this product
modelName	FM		Model name attached to this product
startDate	2005-02-01T00:00:0…		Start date of this product
endDate	2020-01-01T00:00:0…		End date of this product

Data

SPIRE Calibration
- Phot
- Reset
- Spec
- TelemMask Editions (time dependent)

common_Context_for...libration_products

name	common_Context_f
class	SpireCal
package	herschel.spire.ia.ca

SPIRE Calibration
- Phot
- Reset
- Spec
- TelemMask Editions (tir

```
common_Context_for_SPIRE_calibration_products
obsid_1342183660
obsid_1342183660_Pacs_Level_0_Products
obsid_1342189846_Context_for_SPIRE_Level_1_products
obsid_1342189846_Herschel_Auxiliary_Context
```

Figure 1.6. Product loaded into HIPE from the HSA.

Note again that in this way the data is not stored on your machine, but it is referenced for fetching as needed within your working session. So this simply makes the data available in the HIPE session. Products can be inspected, analysed, plotted, and so on. Note also that for this, *the internet connection must be kept open*, since the products are being read from the HSA.

The products can be saved/stored into pools later on (see Section 1.3.10).

1.3.4. Downloading a single observation from the HSA

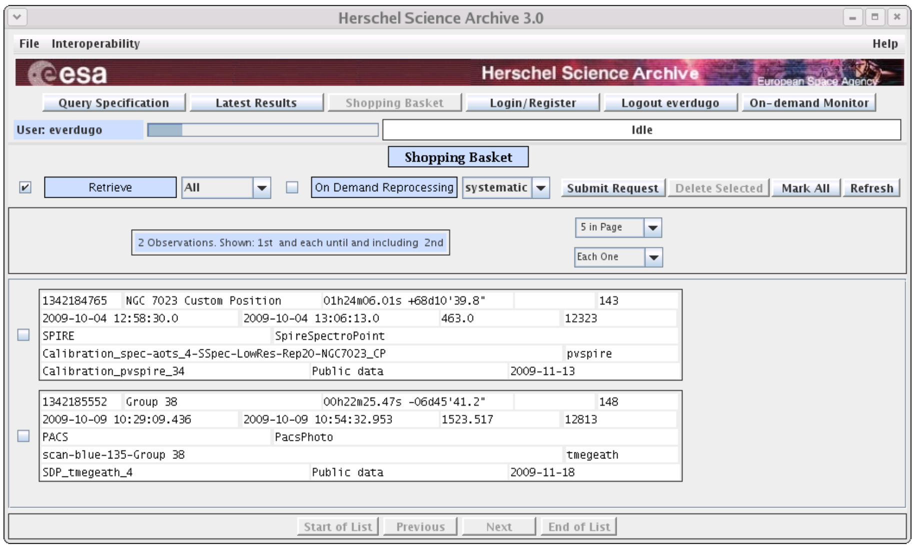
In the query results page of the HUI, next to each observation, there is a *Retrieve* drop-down list, with the same options as the *Send to External Application* drop-down list visible in Figure 1.5. Selecting an item activates an FTP session, downloading a tar file with the data products corresponding to that observation.

For information on how to load the observation into HIPE, see Section 1.3.6.

Using this method is only recommended for individual observations. To download many observations at once, see the next section.

1.3.5. Downloading more observations with the shopping basket

With this method the records of several observations can be transferred into a *Shopping Basket*. This method is envisaged for multi-observation requests. Once the shopping basket contains all the datasets to be retrieved, these can be transferred to a secure FTP area.



User: everdugo

Idle

[✓] Retrieve All [ ] On Demand Reprocessing systematic

2 Observations. Shown: 1st and each until and including 2nd

5 in Page

Each One

```
1342184765   NGC 7023 Custom Position   01h24m06.01s +68d10'39.8"          143
2009-10-04 12:58:30.0   2009-10-04 13:06:13.0   463.0   12323
SPIRE                   SpireSpectroPoint
Calibration_spec-aots_4-SSpec-LowRes-Rep20-NGC7023_CP        pvspire
Calibration_pvspire_34           Public data          2009-11-13

1342185552   Group 38                   00h22m25.47s -06d45'41.2"          148
2009-10-09 10:29:09.436   2009-10-09 10:54:32.953   1523.517   12813
PACS                    PacsPhoto
scan-blue-135-Group 38                                        tmegeath
SDP_tmegeath_4                   Public data          2009-11-18
```

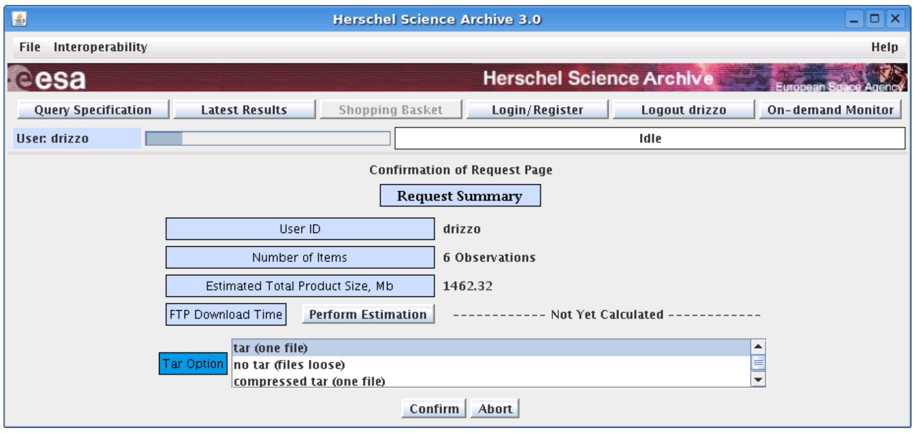
Figure 1.7. The shopping basket of data to retrieve from the HSA

To select data for retrieval, click on the check box to the left of each record of the observations list, and then click on *Move Selected to Basket* at the top of the panel. To see what is in your shopping basket, click the *Shopping basket* button. The observations moved have disappeared from the *Latest Results* page and appeared in the *Shopping Basket* page.

To move *all* the observations in the query results, click on the *Move All to Basket* button. This will move all the observations into the shopping basket and it will open the shopping basket page.

You can delete observations from the shopping basket by clicking on the check box to the left of them and then clicking the *Delete Selected* button to the top right.

Once you are happy with the contents of the shopping basket, click on *Submit Request* to proceed. The *Request Summary* page appears (see Figure 1.8).



User: drizzo

Idle

Confirmation of Request Page

Request Summary

User ID	drizzo
Number of Items	6 Observations
Estimated Total Product Size, Mb	1462.32
FTP Download Time	------------ Not Yet Calculated ------------

Tar Option

tar (one file)
no tar (files loose)
compressed tar (one file)

Figure 1.8. Data retrieval request in the HSA

You can go back to the *Query Specification* or to the *Latest Results* page to change the query or the shopping basket selection, respectively. An estimation of the total size of the request is given so that you can choose between the tar file option or a compressed tar file (files of type `.tar.gz`). Click on *Confirm* to confirm the request. A request ID is shown, meaning that the generation of the dataset has started. An e-mail will be sent to you as soon as the data are available on the FTP area for retrieval.

The tar file with the data retrieved from the HSA contains FITS files ordered in a well-specified (hierarchical) directory structure. Once the tar file is decompressed in a user directory, it can be registered in HIPE as a pool (see Section 1.3.6).

Warning

If you use WinZip (and possibly other compression programs) to decompress your tar files, your FITS files may be corrupted. For more information on how to solve this problem, please see Section 1.4.6.2.

1.3.6. Importing/exporting Herschel data to/from HIPE

The tar file provided by the Herschel Science Archive (HSA) can be registered in HIPE as a pool (see Section 1.4.1 for information on local stores and pools) using the view *Import Herschel data to HIPE*.

Note that currently the tar file provided by the HSA should contain the whole observation context (option *ALL* in HUI for retrieving data) in order to use the *Import Herschel data to HIPE* view. The directory structure, once the tar file has been decompressed, should look like the following:

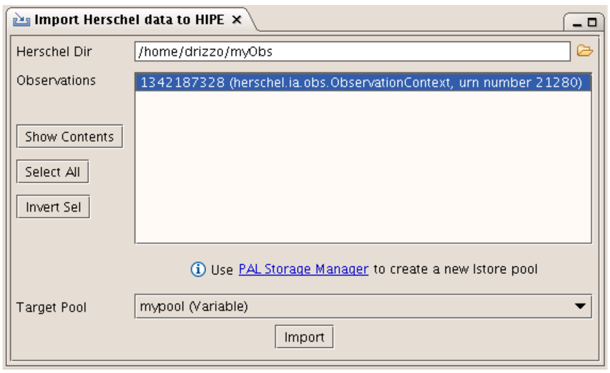
```
1342185538/
auxiliary/
calibration/
```

Open the *Import Herschel data to HIPE* view (see Figure 1.9). Select a directory in which the files coming from HSA are placed. Pressing the button *Show Contents* all the observations included in that directory will be shown. Select the ones you want to save into a pool, select the pool and press *Import*. The observations saved into the pool are referenced automatically in HIPE.

Note

- You need to specify the directory *above* which you have the directory of HSA data (so if you unpacked your data into `/Users/me/.hcss/stuff/134211111` then you need to select `/Users/me/.hcss/stuff`). Also, if `.hcss` (or any other hidden directory) does not come up in the directory listing, try typing in the directory listing panel `/Users/me/.hcss` and press **Enter**. Then run the browse again, and the directory should now appear. If it does not, you will have to write the full path in the *Herschel Dir* text box.

- You have to put your data into a *pool*, which must already have been defined to appear in the *Target Pool* drop-down list. For more information about creating pools, see Section 1.4.1.



Herschel Dir /home/drizzo/myObs

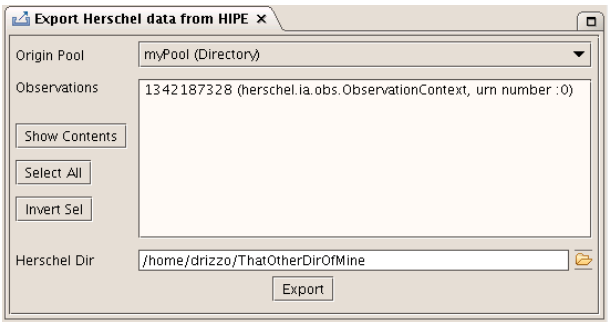
Observations 1342187328 (herschel.ia.obs.ObservationContext, urn number 21280)

Use PAL Storage Manager to create a new lstore pool

Target Pool mypool (Variable)

Figure 1.9. Product loading into HIPE from the HSA tar file.

In the same way, you can export observations in pools to the standard (hierarchical) directory structure by using the *Export Herschel Data From HIPE* view. Select one observation from a pool and an output directory in the view, and press *Export* (see Figure 1.10).



myPool (Directory)

1342187328 (herschel.ia.obs.ObservationContext, urn number :0)

/home/drizzo/ThatOtherDirOfMine

Figure 1.10. Product export from HIPE into standard Herschel directory structure.

Note

These views make use of the following two tasks, documented in the *User's Reference Manual*:

- `importUfDirToPal`: Section 2.209.
- `exportPalToUfDir`: Section 2.117.

The XML file needed by these two tasks is included in the HSA tar file under a directory called `.exported/`.

1.3.7. Data access via the HIPE GUI

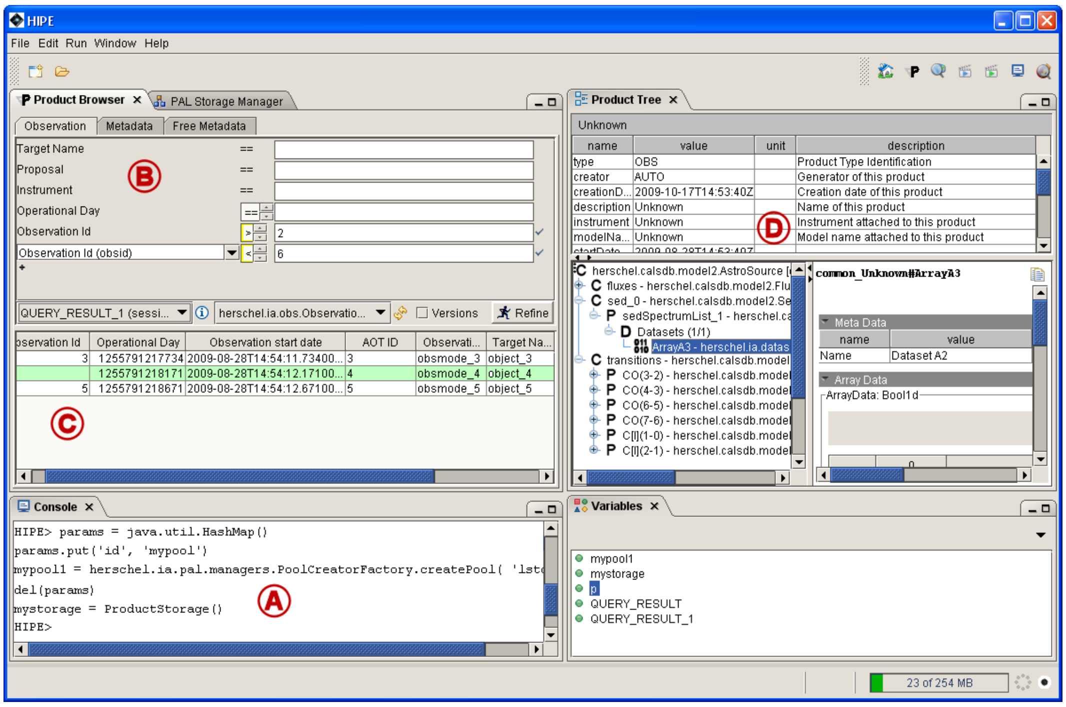
With the *Product Browser Perspective* you can query and explore all the data stored by HIPE within *storages* and *pools*. You can open the Product Browser perspective by clicking on the icon on the HIPE toolbar, or by choosing Window → Show Perspectives → Product Browser.

Note

The *Product Browser Perspective* will be replacing the *standalone Product Browser* which was used in JIDE as well as the *Data Access view* which is implemented as prototype only.

The main components of the perspective are the following:

1. The *Jython console* (A) can be used to load and initialize a product storage. Alternatively you can use the *PAL Storage Manager* tab in the main area (B).
2. The *query area* (B), where you enter query parameters.
3. The *result area* (C), where you view the query result.
4. The *result inspection area* (D), where you inspect a selected product.



Target Name	==	
Proposal	==	
Instrument	==	
Operational Day	==	
Observation Id	>	2
Observation Id (obsid)	<	6

QUERY_RESULT_1 (sessi... herschel.ia.obs.Observatio...

bservation Id	Operational Day	Observation start date	AOT ID	Observati...	Target Na...
3	1255791217734	2009-08-28T14:54:11.73400...	3	obsmode_3	object_3
	1255791218171	2009-08-28T14:54:12.17100...	4	obsmode_4	object_4
5	1255791218671	2009-08-28T14:54:12.67100...	5	obsmode_5	object_5

Unknown

name	value	unit	description
type	OBS		Product Type Identification
creator	AUTO		Generator of this product
creationD...	2009-10-17T14:53:40Z		Creation date of this product
description	Unknown		Name of this product
instrument	Unknown		Instrument attached to this product
modelNa...	Unknown		Model name attached to this product

```
C herschel.calsdb.model2.AstroSource [
  C fluxes - herschel.calsdb.model2.Flu
  C sed_0 - herschel.calsdb.model2.Se
    P sedSpectrumList_1 - herschel.ca
      D Datasets (1/1)
        ArrayA3 - herschel.ia.datas
  C transitions - herschel.calsdb.model
    P CO(3-2) - herschel.calsdb.model
    P CO(4-3) - herschel.calsdb.model
    P CO(6-5) - herschel.calsdb.model
    P CO(7-6) - herschel.calsdb.model
    P C[I](1-0) - herschel.calsdb.model
    P C[I](2-1) - herschel.calsdb.model
```

common_Unknown#ArrayA3

Meta Data

name	value
Name	Dataset A2

Array Data

ArrayData: Bool1d

```
HIPE> params = java.util.HashMap()
params.put('id', 'mypool')
mypool1 = herschel.ia.pal.managers.PoolCreatorFactory.createPool( 'lst
del(params)
mystorage = ProductStorage()
HIPE>
```

```
mypool1
mystorage
p
QUERY_RESULT
QUERY_RESULT_1
```

23 of 254 MB

Figure 1.11. The Product Browser Perspective

The following is a typical sequence of steps you will follow to load, query and inspect data products:

1. Create a ProductStorage in the *PAL Storage Manager* tab in the query area (B, seeSection 1.3.10) or in the Jython Console (A, see Section 1.4.2)
2. Specify attributes of a product in the query area (B).

   Each line in the query area is called query term. If you enter data for one query term a green tick at the end of the line will indicate if the term will be considered by the query. The query will return only those Products that match *all* specified query terms.

   Use the combo boxes and the check box at the bottom of the query area to further constrain your query:

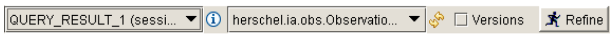
   

   Figure 1.12. Product Browser Perspective Search bar

   - Use the *first drop down box* to select the product storage or any previous query result. Previous results are stored in a variable called QUERY_RESULT_x, where x is a number.

     Note

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

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

   - Use the *second drop down box* to constrain the type of the product you are looking for. If you select "herschel.ia.dataset.Product" all product types will be returned. Note that for observation queries this field defaults to herschel.ia.obs.ObservationContext.
   - The *refresh icon* can be used to reload the Products. This may be required in some rare cases where you add a new Pool or some new Product type to a ProductStorage.

   - If the *version checkbox* is selected the query will include any version of a Product in its result. Versions of Products are created whenever a saved Product is modified and stored again.
   - The *Run or Refine button* is used to execute a query. If you select a ProductStorage (e.g. myStorage) in the first drop down box you can "Run" a query. If you select a previous result (e.g. QUERY_RESULT_1) you can "Refine" this result.

3. Click on the *Run* button to execute the query.

   - Your result will be stored in a variable called QUERY_RESULT_x, where x is a number.

4. Review the results in the result area (C).

   - Select a row to further inspect it in the left side panel.
   - Double click a row to create a new named variable in the variables view.
   - Right click a row to export the product to FITS. You can even do that for a selection of Products.

   You can change the layout of the result table to match you needs.

   - Click on the column header to sort the column ascending or descending. You can sort up to three columns. Double click a column to reset sorting.
   - Drag and drop a column header to move the column
   - Right click the column header to change the layout. The menu entry "Predefined..." offers two predefined table layouts.
   - Right click the column header to hide and display columns. Only checked columns will be shown.
   - Right click the column header to add columns. The menu entry "Add column..." offers an option to add a predefined column.

   Note

   Although you can configure the result table for your needs it currently lacks the possibility to save your changes. This will be provided in the next version of the perspective.

5. Inspect selected results in the *Product Tree View* (D).

   - Use the tree view to browse your product. Please be aware that if you open a Product the system may have to load it first. This may be a time and memory consuming operation.
   - The meta data panel at the top will show the meta data of the currently selected Product.
   - The left side panel will show viewers for selected Products or data sets.
   - We strongly advice to maximize the *Product Tree View* to use the left side panel.

1.3.8. Using the Data Access View

Warning

The Data Access view is deprecated and will be removed in a future HIPE version. Equivalent functionality is available in the Product Browser perspective (see previous section).

When selecting the Data Access view the user will have certain "pools" of data available. These allow access to data stored in registered data storage areas (basically areas accessible to the user on his/her own computer or via the internet to another computer). Storing data in user-named pools is described in Section 1.4.2. All pools currently need to be explicitly "registered" to tell the system where to look.

1.3.8.1. Using the Data Access View to query for products

There are several ways of searching through your stores of data to get the products you want. You can search for complete observations -- such as those you are PI on which exist in the Herschel Science Archive -- attriibutes or metadata values, or you can go into data mining which involves searches based on the data itself.

For all cases, setup of the data query can be done based on observation data, the attributes of data, meta data or all data (data mining). Once the query of the data store has been set up the search can be done by clicking the Search button to the bottom right of the Data Access view. If the user wishes to access all available data in a data storage then this can be obtained by placing nothing in any of the input boxes of the query.

When the search button is clicked the equivalent command-line version of the request appears in the Console view (see the *HIPE Owner's Guide* for more information on the Console view). This can be saved and edited and used in batch mode processing. This helps to avoid syntax errors by the user in setting up queries on data stores.

Doing a search

In order to do a search the user needs to do the following.

- Open the "Data Access" view.
- Select an available pool from the pull-down menu at the top of the view next to the word "Query". If none are available (greyed-out) then you need to first register a pool for access (see earlier sections of this chapter).
- After inserting an appropriate query, click on the "Accept" button to bottom right of the view. Note that if nothing is placed in the query then the total contents of the pool will be obtained. This is a good way to see the total contents of a pool.

Search by observation

In this case we are dealing with high-level information. The data is part of certain proposal or uses a particular instrument on a particular day. Clicking on the "Observation" tab in the Data Access view allows searches at this level based on instrument, proposal ID, proposal name, observation ID (unique observation numbers or operational day (See Figure 1.13).

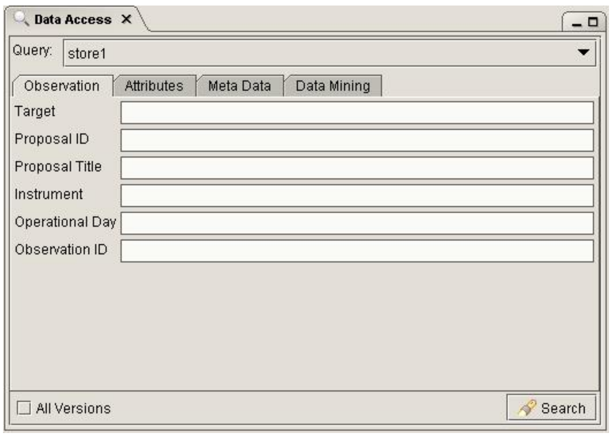
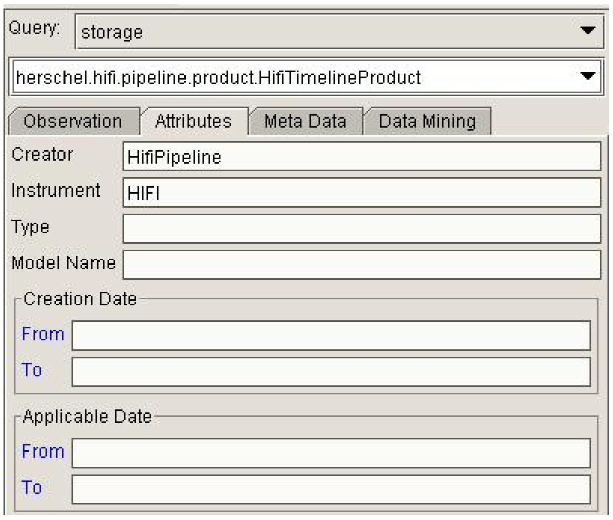


Figure 1.13. HIPE store selection and panel for searching by information on stored observation information in a product.

Search by attributes

The attributes of a set of data are standard to all (See Figure 1.14) and it is possible to do a search on values in this given set of attributes -- which are listed in the query interface.



Query: storage

herschel.hifi.pipeline.product.HifiTimelineProduct

Creator	HifiPipeline
Instrument	HIFI
Type	
Model Name	

Figure 1.14. Attributes available for search.

Search by metadata and data mining

These two options are not implemented.

1.3.8.2. Output from a query and searching a query result

The output from the first query produces a result "QUERY_RESULT". This will be a group of products (e.g., observations) which can then be looked at by the user. The "QUERY_RESULT" name is highlighted in the Variables view (where the name can also be edited to something more appropriate if desired). This result is also automatically fed back to the Data Access pulldown menu, allowing for a search to be made on the result of the initial search.

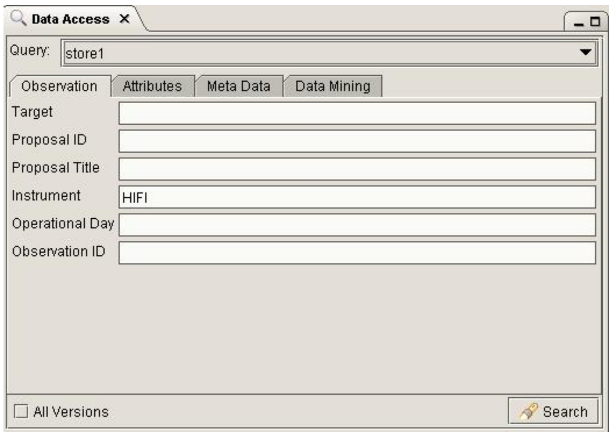
The query output can be viewed by double-clicking on the result variable, e.g. "QUERY_RESULT" in the Variables view. This brings up the query results viewer in the Editor view part of HIPE. This lists the selected items. It also makes the outline available in the "Outline" view.

Clicking on one of the results shown in the query viewer extracts the chosen result (for example, the first product in the list is then available as "prod_0" in the session). Clicking on the name of this extracted product when it appears in the "Variables" view allows further assessment of its contents and viewing of any datasets it contains.

1.3.8.3. An example of search to display of data

In this case, we have partially processed some HIFI data to level 1, which has the format of a HifiTimelineProduct, and stored several versions of this processing in a store given the handle under the HCSS of "store1". This appears under the Data Access view pulldown menu as a selectable store item. The following now leads to displaying some data that has been extracted from our data store.

1. We now intend to search for all HifiTimelineProducts (see second pulldown menu on the screen) with instrument=HIFI within this store by searching on these attributes. The setup should look like the screen shown in Figure 1.15.


Data Access

Query: store1

Observation | Attributes | Meta Data | Data Mining

Target:

Proposal ID:

Proposal Title:

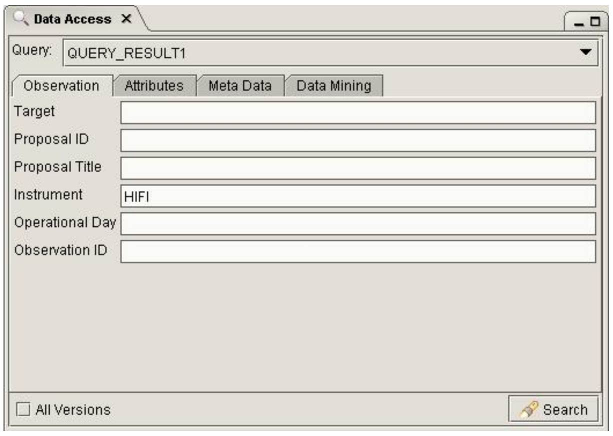
Instrument: HIFI

Operational Day:

Observation ID:

Figure 1.15. Set up of a query for data out of our store.

2. Once this has been setup we click the "Search" button and the appropriate results are extracted and placed in a query result (see Figure 1.16). A highlighted "QUERY_RESULT1" (the number automatically placed at the end will increase depending on the number of queries you make) appears and the data access store available for querying -- at the top of the Data Access view -- immediately changes to QUERY_RESULT1 ready for further searching on the initial query results.


Data Access

Query: QUERY_RESULT1

Observation | Attributes | Meta Data | Data Mining

Target:

Proposal ID:

Proposal Title:

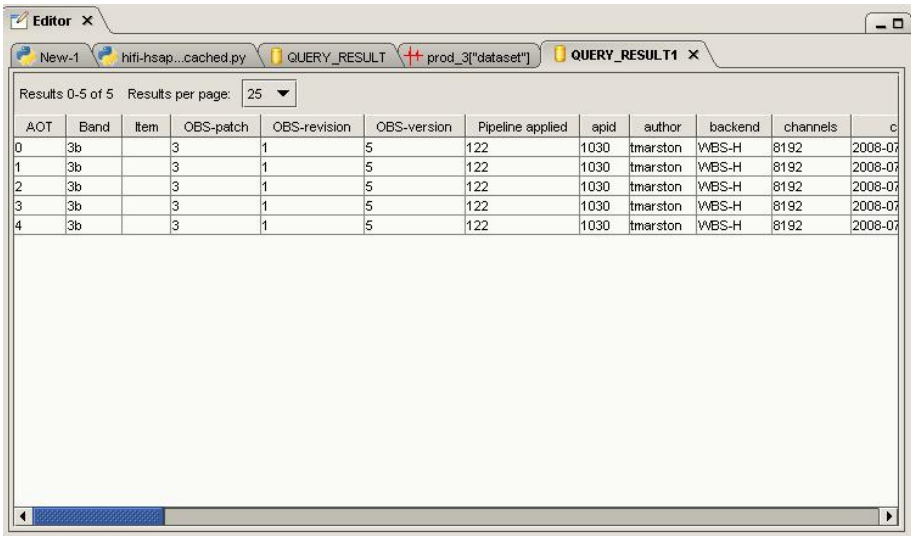
Instrument: HIFI

Operational Day:

Observation ID:

Figure 1.16. Query result obtained.

3. Select the query result in the Variable view (QUERY_RESULT1) via a double mouseclick. This provides a Query result viewer showing a listing in the Editor view of the query results items (see Figure 1.17).

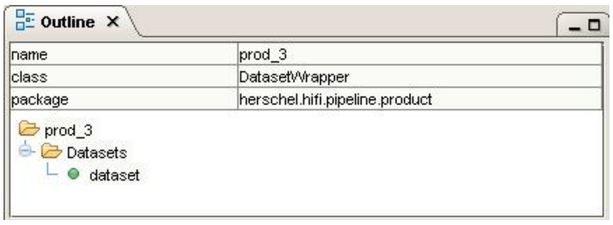


Results 0-5 of 5 Results per page: 25

AOT	Band	Item	OBS-patch	OBS-revision	OBS-version	Pipeline applied	apid	author	backend	channels	c
0	3b		3	1	5	122	1030	tmarston	WBS-H	8192	2008-07
1	3b		3	1	5	122	1030	tmarston	WBS-H	8192	2008-07
2	3b		3	1	5	122	1030	tmarston	WBS-H	8192	2008-07
3	3b		3	1	5	122	1030	tmarston	WBS-H	8192	2008-07
4	3b		3	1	5	122	1030	tmarston	WBS-H	8192	2008-07

Figure 1.17. List of query results appear in editor window.

4. Double-clicking on one of the results shown in the editor view creates the item (product) in the session. It allows us to pull out one of the selected products (e.g., "prod_3" for item number 3 in the query viewer) which can be manipulated in standard ways. For example, if we click on this product in the Variables view we get an outline of its contents in the Outline view (as in Figure 1.18).

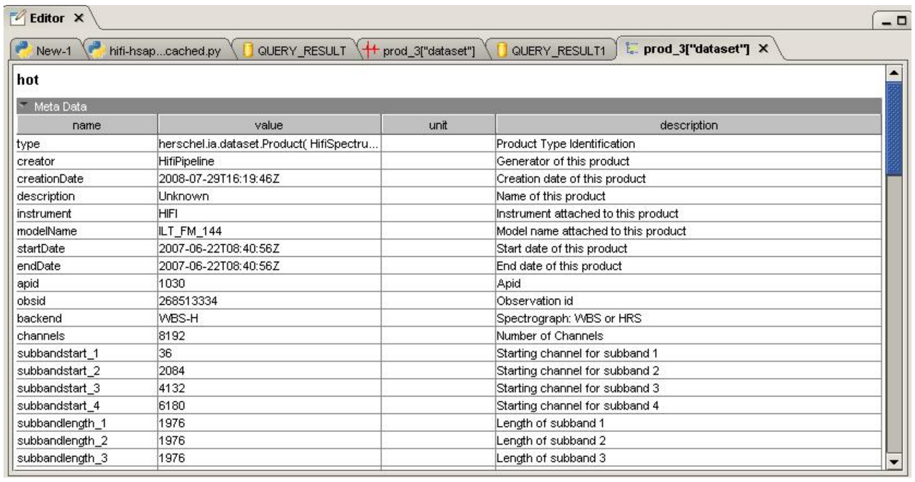


name	prod_3
class	DatasetWrapper
package	herschel.hifi.pipeline.product

Figure 1.18. One of the items is selected with outline of contents shown bottom left.

5. We see that it shows a single folder in the Outline view. Clicking on the first folder, it opens up to show its contents which include a single dataset (as in Figure 1.18).

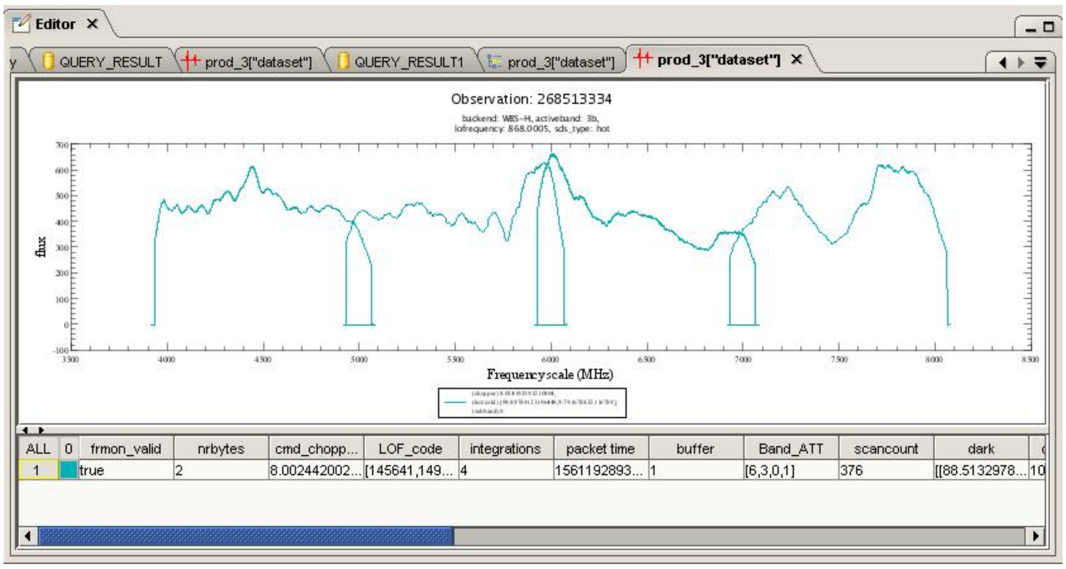
6. A right-click on the word "dataset" in the Outline view provides a set of viewer options. The Dataset viewer will show the associated metadata (header) information plus a table of various values associated with the spectrum, include flux/count values per channel (as in Figure 1.19).



name	value	unit	description
type	herschel.ia.dataset.Product(HifiSpectru...		Product Type Identification
creator	HifiPipeline		Generator of this product
creationDate	2008-07-29T16:19:46Z		Creation date of this product
description	Unknown		Name of this product
instrument	HIFI		Instrument attached to this product
modelName	ILT_FM_144		Model name attached to this product
startDate	2007-06-22T08:40:56Z		Start date of this product
endDate	2007-06-22T08:40:56Z		End date of this product
apid	1030		Apid
obsid	268513334		Observation id
backend	WBS-H		Spectrograph: WBS or HRS
channels	8192		Number of Channels
subbandstart_1	36		Starting channel for subband 1
subbandstart_2	2084		Starting channel for subband 2
subbandstart_3	4132		Starting channel for subband 3
subbandstart_4	6180		Starting channel for subband 4
subbandlength_1	1976		Length of subband 1
subbandlength_2	1976		Length of subband 2
subbandlength_3	1976		Length of subband 3

Figure 1.19. Metadata (header) display for the extracted spectrum.

7. Alternately, we can simply view the extracted spectrum dataset by selecting the "Spectrum Explorer" viewer instead (see Figure 1.20 and HowTo on displaying spectra).



ALL	0	frmon_valid	nrbytes	cmd_chopp...	LOF_code	integrations	packet time	buffer	Band_ATT	scancount	dark	(
1	🟦	true	2	8.002442002...	[145641,149...	4	1561192893...	1	[6,3,0,1]	376	[[88.5132978...	10

Figure 1.20. Displaying the extracted spectrum. Note that the view has been expanded using the capabilities of the "Spectrum Explorer" viewer.

1.3.9. Managing storages and pools

Warning

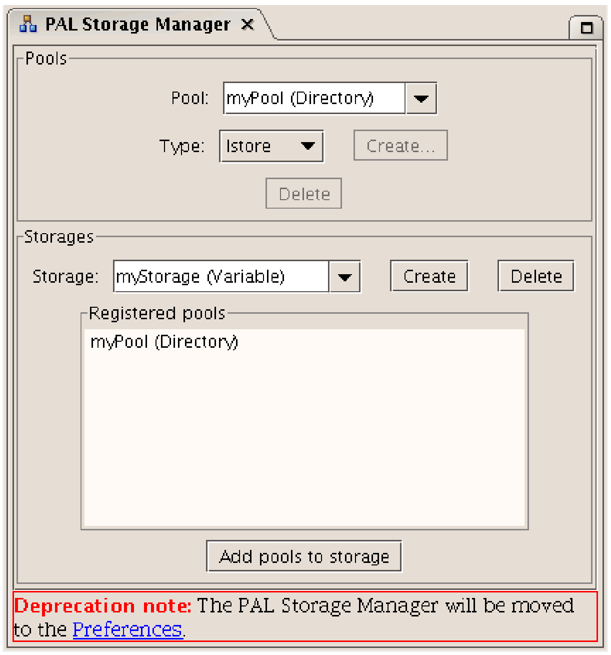
Please see Section 1.2.2 for important information about pools and compatibility between HIPE 2.x and 3.x.

Storages and *pools* are the two tools with which you can store and retrieve data on your computer. With the *PAL Storage Manager* view you can create, delete and associate storages and pools.

Note

This functionality will be moved to the *Preferences* dialogue window of HIPE.

To open this view, select Window → Show View → PAL Storage Manager.



Deprecation note: The PAL Storage Manager will be moved to the Preferences.

Figure 1.21. The PAL Storage Manager view

With this view you can accomplish three tasks:

- **Creating and deleting pools.** In the *Pools* pane, write the name of a new pool in the *Pool* field and choose a type from the *Type* drop-down list. If you are unsure about the type, choose `lstore` (*local store*, storing data on your computer). When you click on *Create*, a window appears with additional options, depending on the pool type. The only option always present is *Use local cache*. If you are working with remote data, this option will create a local cache, so that you can continue working offline. Clearly this is not needed if the pool is local, like a local store.

  To delete a pool, select it from the drop-down list and click *Delete*.

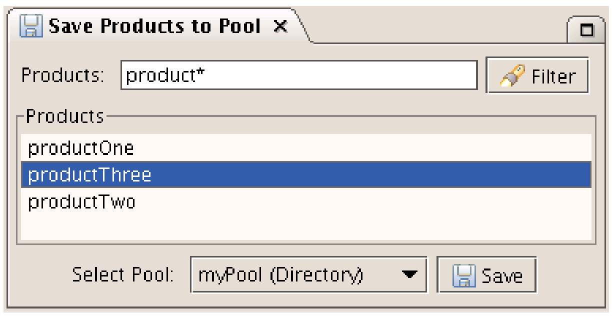
- **Creating and deleting storages.** To create a storage, write a name in the *Storage* field in the *Storages* pane, and click *Create*. A window appears, allowing you to *register* one or more pools with the new storage. You can choose not to register any pools at this stage

  To delete a storage, select it from the drop-down list and click *Delete*.

- **Registering pools to storages.** In the *Storages* pane, select a storage from the drop-down list and click *Add pools to storage*. A window appears with the list of existing pools. Select one or more pools to register and click *OK*.

1.3.10. Saving data to a pool

With the view *Save Products to Pool* you can store one or more data products into one of the available pools. To open the view, select Window → Show View → Save Products to Pool.



Products: product*

Products
productOne
productThree
productTwo

Select Pool: myPool (Directory)

Figure 1.22. Save Products to Pool view

1. Press *Filter* to display all the available products in your session. Alternatively, write a search expression in the *Products* text field and press *Filter* to only display some products. For example, in the previous image the search string `product*` is used, which means that only the products whose name begins by *product* are shown.

2. Select one or more products in the *Product* pane.

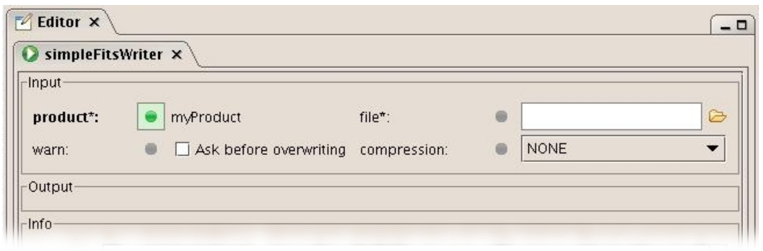
3. Select a pool from the *Select Pool* drop-down list.

4. Press *Save* to store the selected products into the pool.

1.3.11. Saving data to FITS files

You can save any kind of Herschel data to FITS files, as long as it is of type `Product`. All the raw and reduced data coming from the Herschel Science Archive are of type `Product`, so this should not be an issue. In case you have datasets that are not products, see Section 1.4.1 to learn how to wrap them into products.

To save a product as FITS file, select the product in the *Variables* view and open the *Applicable* folder in the *Tasks* view. Double click on the `simpleFitsWriter` task to launch it. The task dialogue window opens in the *Editor* view, as shown in the next figure.



product*: myProduct
file*:
warn: Ask before overwriting
compression: NONE

Figure 1.23. FITS save task dialogue window.

Write the name of the new FITS file, and optionally browse for a different directory. You can also specify a compression method (*ZIP* and *GZIP* are available) and whether you want to be warned if you try to overwrite an existing file. Press *Accept* to save the product to file.

1.3.12. Reading data from FITS files

You can use two tasks to read FITS files: `fitsReader` and `simpleFitsReader`. The `fitsReader` task (see Figure 1.24) will try to guess what the file contents are (by looking at the `XTENSION` keyword) and will put the contents in a variable of the appropriate type. If `fitsReader` does not recognise the file contents, it defaults to the `simpleFitsReader` task. This task is optimised to read data from FITS files as packaged by HCSS. If the file is not an HCSS FITS product, the contents are put in unformatted arrays. You can choose how to read the file or let the software choose.

To run `fitsReader` or `simpleFitsReader` from HIPE, go to the *Tasks* view, select the *All* tasks folder and scroll down to `fitsReader` or `simpleFitsReader`. A double-click on the name opens its dialogue window. Insert the input file name and click the Accept button to run the task and read in the FITS file.

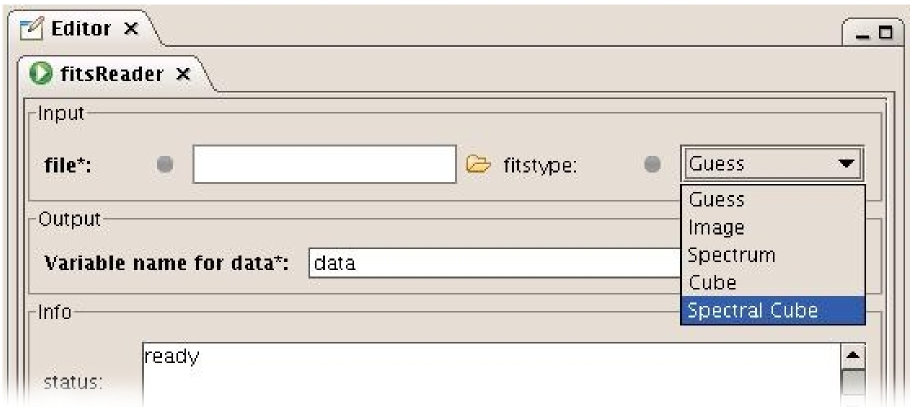

Editor
fitsReader
Input
file*:
fitstype: Guess
Guess
Image
Spectrum
Cube
Spectral Cube
Output
Variable name for data*: data
Info
status: ready

Figure 1.24. FITS read task dialogue window.

1.3.13. Creating and reading ASCII table files

You can save tabular data of type `TableDataset` to a text file. If you click on a variable of type `TableDataset` in the *Variables* view of HIPE, you will see `asciiTableWriter` in the *Applicable* folder of the *Tasks* view. Double-clicking on this task opens a dialogue window for creating an ASCII table. The simplest way of formulating an ASCII table is to take the defaults and simply fill in a name for the output table. But more sophisticated options are available (see Figure 1.25).

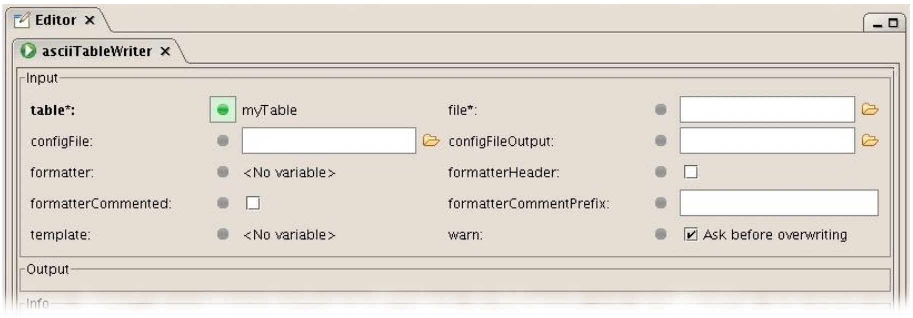

Editor
asciiTableWriter
Input
table*: myTable
file*:
configFile:
configFileOutput:
formatter: <No variable>
formatterHeader: ☐
formatterCommented: ☐
formatterCommentPrefix:
template: <No variable>
warn: ☑ Ask before overwriting
Output
Info

Figure 1.25. ASCII save task dialogue window.

The other possible inputs for the task are the following (this information is also available by hovering the mouse over the parameters shown in the dialogue window).

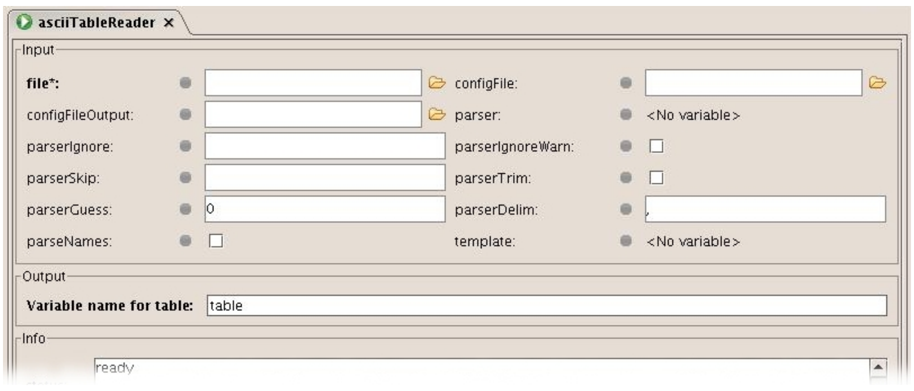
```
* file = output file name.
* table = TableDataset to write.
* configFile  = configuration file where the formatter
(AsciiFormatter), parser (AsciiParser) and table template
(TableTemplate) must be specified. When configFile parameter is specified,
any parameter related to parser or to table template are not allowed.
* configFileOutput = if a config file is specified, an output configuration
file will be created.
* formatter (default AsciiTableTool formatter) = AsciiFormatter object.
* formatterHeader (default AsciiFormatter header allowed) = Specifies
if header information to be provided (true/false).
* formatterCommented (default AsciiFormatter comments allowed) = Specifies
if there are comments when writing a file (true/false).
```

```
    * formatterCommentPrefix (default AsciiFormatter comments prefix value) =
    Specifies what the prefix is for identifying all comments.
    * template (INPUT, default value: extracted from the first file rows) =
    TableTemplate object for specifying the data structure (see the In depth
section for more details).
```

You can read an ASCII table into HIPE with the `asciiTableReader` task, available in the *All* folder of the *Tasks* view. For standard CSV tables you only need to provide the file name of the ASCII table to be read in. More options are given below:

```
    * file = input file containing ASCII table.
    * table = TableDataset object name for loaded table.
    * configFile = configuration file where the formatter (AsciiFormatter),
    parser (AsciiParser) and table template (TableTemplate) must be specified.
    When configFile parameter is specified, any parameter related to parser or
    to table template are not allowed.
    * configFileOutput = if a file is specified, an output configuration
    file will be created.
    * parser (default AsciiTableTool parser) = AsciiParser object.
    * parserIgnore (default AsciiParser ignore value)
    = String expression to ignore when parsing a file.
    * parserSkip (default AsciiParser skipping rows value) = Number of rows to
    skip when reading a file.
    * parserTrim (default AsciiParser trim rows value) = Specifies if the parser
    must trim each row when reading a file (true/false).
    * parserGuess (default value AsciiParser.GUESS_NONE) = specifies if
    the parser should guess column types. Files should not contain HCSS header
    (use skip=AsciiReader.HCSS_HEADER for skipping HCSS header or comment these
    lines)

      Valid options:
          o AsciiParser.GUESS_NONE: (default) file must contain template
   or template must be provided (no guess)
          o AsciiParser.GUESS_TRY: guess types based on the first 100 records
          o AsciiParser.GUESS_ALL: guess types based on all records
          o AsciiParser.ALL_STRING: each record is a string (no guess required)
          o AsciiParser.ALL_BOOLEAN: each record is a boolean (no guess required)
          o AsciiParser.ALL_BYTE: each record is a byte (no guess required)
          o AsciiParser.ALL_INTEGER: each record is an integer (no guess required)
          o AsciiParser.ALL_LONG: each record is a long (no guess required)
          o AsciiParser.ALL_FLOAT: each record is a float (no guess required)
          o AsciiParser.ALL_DOUBLE: each record is a double (no guess required)
          o AsciiParser.ALL_COMPLEX: each record is a complex (no guess required)
    * parserDelim (INPUT, default value: comma) = Specifies the field delimiter.
    If it is one character, a csvParser is selected. If it is an expression,
    a RegExpParser (regular expression) is selected.
    * template (INPUT, default value: extracted from the first file rows) =
    TableTemplate object for specifying the data structure. See TableTemplate.
```



asciiTableReader

Input

file*:		configFile:	
configFileOutput:		parser:	<No variable>
parserIgnore:		parserIgnoreWarn:	☐
parserSkip:		parserTrim:	☐
parserGuess:	0	parserDelim:	,
parseNames:	☐	template:	<No variable>

Output

Variable name for table: table

Info

ready

Figure 1.26. ASCII read task dialogue window.

Information on saving and reading tables from the command line is available in Section 1.4.7.

1.4. In depth

1.4.1. Creating and saving products in a pool

Warning

Please see Section 1.2.2 for important information about pools and compatibility between HIPE 2.x and 3.x.

Any product (such as a complete observation in the form of an ObservationContext) can be placed in a pool, or storage area, on your hard disk. You can find advanced information on this topic in the *Scripting and Data Mining* guide: Appendix A. This section simply illustrates how to set up a set of stores (which act a bit like mini databases) in which you can place any output data that is in the form of a product, such as an observation.

A pool can be set up and populated in the following fashion via the command line.

```
poolholder = ProductStorage()  # -"poolholder" can be any word.
myPool = LocalStoreFactory.getStore("myTestPool")
# -"myTestPool" is the directory name on disc where your data
# are to be put. -"myPool" can be any word.
# Now link the directory on disc to the -"poolholder" in HIPE
poolholder.register(myPool)
# At this point the pool is ready
poolholder.save(prod1)     # Now we add our products called
poolholder.save(prod2)     # -"prod1", -"prod2" and -"prod3" to the store.
poolholder.save(prod3)     #
```

Names for pools and storages can contain letters, numbers and the dot and underscore characters. Spaces are not allowed.

Note that if you start a new HIPE session you will need to register your pool again via something similar to the first three lines.

The data physically reside in the `.hcss/lstore`, under your home directory. You will see that the information is actually held as a hierarchical set of FITS files.

To rename a pool *created with HIPE 3.x or higher*, rename its directory. Pools created with HIPE versions prior to 3.0 *cannot* be renamed.

Note that you can only save products, which means that if you want to save a dataset of any kind (like a `TableDataset`, `Spectrum1d` or `Spectrum2d` and so on) you need to wrap them in a product as is shown in the following example:

```
# Create a TableDataset with two columns index and xvalue
table = TableDataset(description = -"A table")
table["index"] = Column(data=Int1d.range(100))
table["xvalue"] = Column(data=Double1d(100).apply(RandomUniform()))
# Wrap a product around the dataset
tProduct = Product(description="A table")
tProduct["myTable"] = table
store.save(tProduct)
```

Placing things into products allows for the proper header information to be included. Products can be wrapped within products (e.g., several images in a single product such as an observation) and each level has its own metadata/header information.

Restoring data using command line queries

You can search the local store for products with a given attributes. For example, querying the local store pool `myPool` for products with description matching `An image`:

```

query=MetaQuery(Product,"p","p.description=='An image'")
```

```
results2=store.select(query)
print results2
# [urn:MyPool1:herschel.ia.dataset.image.SimpleImage:0]

image = results2[0].product
```

The same as above, if there are more than one result then we can refer to it with the index.

1.4.2. Registering and accessing other data stores

It is possible to register other stores that can then be searched from the data access view, but they first have to be registered in the system (you need to tell the system where they are, in effect). For data stores elsewhere on your machine other than the default area this can be done by using the following lines of code which can be entered at the command line.

```

# Get a local store (or create a new one if not already existing) with
# an id of -"test". The Configuration command changes the directory
# where the store is
Configuration.setProperty('hcss.ia.pal.pool.lstore.dir', -'C:\\.hcss\\myData')
datastore = LocalStoreFactory.getStore("test")
myStore = ProductStorage()  # Create a product storage
myStore.register(datastore) # Register it
# -"myStore" is now one of the selectable data stores on the Data Access menu
myStore.save("myProduct")
# will save a Product in the DP session called -"myProduct" in the storage area
```

1.4.3. Saving to and loading from FITS files

The tool to write and read Products to and from FITS files is `FitsArchive`. In your HIPE session, you may have multiple instances of this tool, each with a different configuration.

In general, you can set up a FITS file for archiving, export products to it and retrieve back a product from a FITS file.

A generic FITS reader is also available. This generic reader can parse FITS files that were created by applications other than the HCSS software.

```
# Note that this example will fail
# unless you have a FITS file called input.fits!

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 1.1. Using FitsArchive

The `product` variable can be manipulated in a similar way as other arrays. In the above example, a 2D FITS image is displayed after having been imported.

A product containing data and meta data can be saved into a FITS file using the following command:

```
fits.save("output.fits", product)
```

In particular, you can save a `SimpleImage` as a multi-extension FITS file:

```

fits = FitsArchive()
myImage = SimpleImage(description="An image",image = Double2d(50,100), \
          error=Double2d(50,100),exposure=Double2d(50,100))
fits.save("myImage.fits", myImage)

```

The file will be saved in the directory from which you started HIPE. Provide the full path, instead of just the file name, if you want to save the file elsewhere.

Warning

The above code will generate a FITS file with the value 50 assigned to the `NAXIS2` keyword and 100 assigned to `NAXIS1`. In other words, the image size will be 50 pixels along the *y* axis and 100 pixels along the *x* axis. The coordinate values will be displayed in this order (*y*, *x*) in the Image Viewer. For an explanation of why the *y* size is specified *before* the *x* size, see the *Scripting and Data Mining* guide: Section 2.6.1.

1.4.4. Saving TableDatasets as FITS files

Once we have the TableDataset wrapped in a Product we can save it like all other products, like in the following example:

```
fits=FitsArchive()
myTable = TableDataset() # Create an empty table
myTable["X values"] = Column(Double1d([2,3.4,4])) # Create dummy column
myTable["Y values"] = Column(Double1d([2,4.5,4.8])) # Create second column
tProduct = Product(description="This is a table") # Create the product
tProduct["firstTable"] = myTable  # Add the table and give it a label
fits.save("test.fits", tProduct)
```

The resulting structure of the saved FITS file is:

```
   No. Type     EXTNAME      BITPIX   Dimensions(columns)

   0   PRIMARY                  32     0
   1   BINTABLE table            8     2(3)

 Column Name                          Format     Dims       Units     TLMIN  TLMAX
      1 X values                      1D
      2 Y values                      1D
```

The column names, named as "X values" and "Y values", are in the file.

1.4.5. Parameter name conversion and FITS header

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

Lookup dictionaries are used to convert well known FITS parameter names into a convenient and human readable name. The following dictionaries are in use:

- Common keywords widely used within the astronomical community, which are taken from HEASARC
- Standard FITS keywords
- HCSS keywords containing keywords that are not defined in the above dictionaries

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

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

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

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

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

# And finally, a product that has the composite dataset,
# TableDatset and array dataset.
p=Product(description="FITS demonstration",creator="demo.py")
p.creator="You?"
p.modelName="demonstration"
p.meta["sampleKeyword"]=\
StringParameter("Example keyword not in FITS dictionaries")
p.meta["observationInstrumentMode"]=StringParameter("UnitTest")
p["myArray"]=s
p["myTable"]=t
p["myNest"]=c

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

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

Example 1.2. FITS input/output example

1.4.6. Caveats

For more information see the FITS IO general documentation.

1.4.6.1. FITS header character limit

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.

1.4.6.2. Corrupted FITS file after unzipping

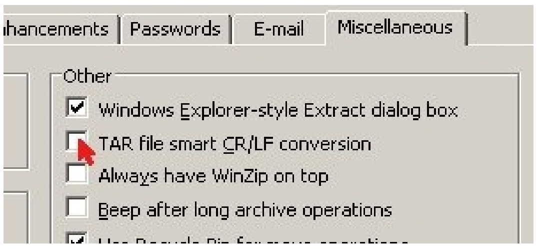
The Herschel Science Archive provides an option to download observations as a TAR (zipped) file. Windows users often extract such a file with the WinZip program and find that their FITS files are corrupted.

The default settings of WinZip tries to be smart and converts text files to DOS format, which means that the line feed (LF, or `\n`) character is replaced by line feed and carriage return (CR, or `\r`) character. Obviously this should not be done to binary files.

WinZip seems to determine whether a files is an ascii file by reading the first few characters of the file, if this is looks like plain text, it will do the conversion. Unfortunately all (binary) FITS files start with the word "SIMPLE". Hence the FITS file is interpreted as text file and conversion and therefore corruption takes place.

The above is the result of running WinZip with default settings. Fortunately WinZip provides a way to disable the conversion. The steps below describe the procedure for WinZip 12.0.

- Select Options → Configuration...
- Go to the *Miscellaneous* tab
- De-select the *TAR file smart CR/LF conversion* option (see the figure below).



☑ Windows Explorer-style Extract dialog box
☐ TAR file smart CR/LF conversion
☐ Always have WinZip on top
☐ Beep after long archive operations

Figure 1.27. The Configuration dialogue window in WinZip.

Note

It seems that 7-Zip does *not* cause this problem. If using another compression software, please consult its documentation. You may want to inform the Herschel Editorial Board of your findings so that they can be included in this section.

1.4.7. ASCII table import/export

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()
```

The `ascii` variable now represents 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 the HIPE console view:

```
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]))
```

You can now export it to an ASCII file with the following command:

```
ascii.save("table.output", table)
```

The file `table.output` looks 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 you did not set any.

To load the data back into HIPE use the following command:

```
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:

parser = yourParser	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).

1.4.7.1. 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 leading and trailing spaces. Default is 0 (false).

The following lines make the parser skip the first twenty lines and remove leading and trailing blanks:

```
ascii.parser.skip = 20
ascii.parser.trim = 1
```

The following input parsers are available:

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 two 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='*')
```

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:

sizes = array	An array `n` elements, where `n` 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])
```

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*")
```

1.4.7.2. Export formatters

A formatter controls how to format a row of cells into a line of ASCII. All formatters share the following attributes:

commented = 0|1	States whether comments will be allowed in the output or not. Default is 0 (false).
commentPrefix = string	Prefix used for all comments, default="# ".
header = 0|1	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="$$$ -"
```

The following export formatters are available:

- **Comma-separated-variable formatter.**

  Please read its counterpart `CsvParser` (see Section 1.4.7.1) 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='<')
  ```

- **Fixed-width formatter**

  Please read its counterpart `FixedWidthParser` (see Section 1.4.7.1) 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])
  ```

1.4.7.3. 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:

names = array	Specifies names that will be attached to the columns.
types = array	Specifies the types of all columns. If not specified, the template assumes that all columns are of type `String`. Allowed types are: `Boolean`, `Integer`, `Float`, `Double` and `String`.
units = array	Specifies the units of all columns. Uses SI units, and units that are accepted for use with SI.

descriptions = array	Specifies comments for all columns.

The following table template indicates a table with four 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"])
```

1.4.7.4. Examples of how to import/export ASCII tables

Section 1.4.7 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 HIPE. A number of ASCII tables are created and reimported. These can be viewed by opening them in HIPE (or within any other text editor). In order to run the program the user will also require an input file, which is given below. 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 Witdh parser
ascii.parser=FixedWidthParser(sizes=[8,16,8,30])
table=ascii.load("table.out2")
# ---- export using Fixed Witdh 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 1.3. 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
#####
```

1.4.8. Saving and restoring variables

Some or all of your variables can be saved to disk and restored later in the same session, or even a different session. 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

These can be saved from and brought back into a HIPE session using the `save` and `restore` commands. This is illustrated in the following example:

```
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 1.4. Using save and restore

Chapter 2. Data display

2.1. Summary

This chapter teaches you how to display and inspect data of several different types:

- Tabular data, especially as `TableDataset` objects. Table datasets are the main building blocks of the data products containing your observations.
- Images, which internally are just a particular type of table dataset.
- Spectra and data cubes.

For information on more sophisticated analysis tools for images, spectra and data cubes, see Chapter 4 and Chapter 5. In particular, for information on the WCS (World Coordinate System) see Section 4.3.1.

Note

Throughout the section, a JPEG image of NGC 6992 is used as example. This image can be fetched from the `data/ia/demo/data` folder of your HIPE installation (click here for a local link if you are viewing this document with the HIPE Help System: ngc6992.jpg).

2.2. How to

2.2.1. Viewing an image

To display an image in HIPE, double-click the image name (for instance in the *Variables* view). The standard image viewer display will appear in the *Editor* view (see Figure 2.1).

Tip

If you have a large image, you may want to undock the image viewer from the *Editor* view (by clicking and dragging the viewer tab) and then enlarge it.

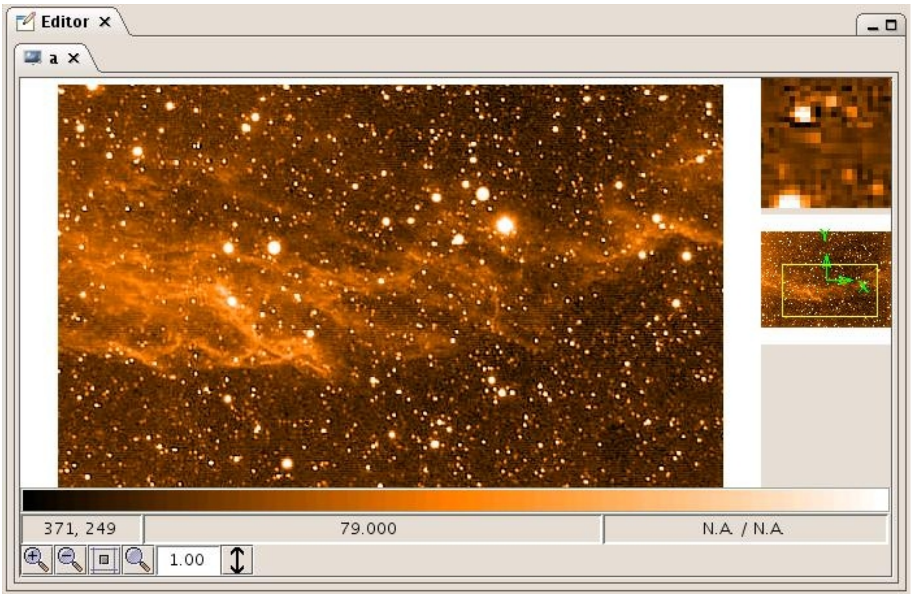


Figure 2.1. Viewing an image in HIPE.

The two smaller boxes on the right show the following:

- A zoomed image around the mouse position.

- An overview of the full image with the area shown in the main pane outlined by a rectangle. This box also illustrates the directions N and E on the display based on the WCS coordinates of the image (or X and Y if no WCS is present). You can change the position of the zoom/pan region by dragging it.

Click and drag the mouse pointer on the gradient bar below the image to change intensity levels.

With the four icons at the bottom left corner you can (left to right):

- Zoom in
- Zoom out
- Zoom to fit window
- Zoom to original size

The number next to the zoom icons is the current zoom level. You can modify it and press **Enter** to set a new zoom level. By clicking the double-arrow icon you can flip the direction of the Y axis.

The three boxes below the image show the following information (left to right):

- Pixel coordinates at mouse pointer, listed as (*y*, *x*)
- Pixel intensity value at mouse pointer
- WCS coordinates (if defined) at mouse pointer

Right-click on the image to display a context menu with additional options (see Figure 2.2). In particular, you can print the image or save it to file (with *Create screenshot*). You can save either the whole image or the current view, in one of four formats (JPG, PNG, BMP and PS).

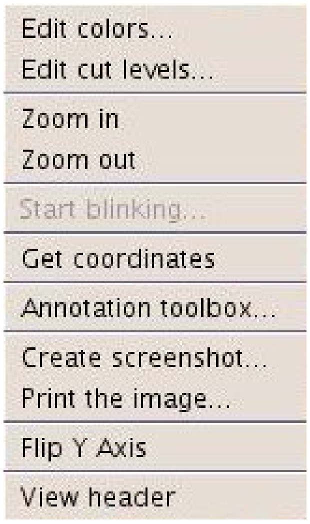


Figure 2.2. Image editing functions.

2.2.2. Simple image editing

This section describes simple tools to change the colours and the cut levels of an image, and to add drawings and annotations. For more sophisticated processing and analysis routines, see Chapter 4.

Editing the image colours

Choose *Edit colors* from the context menu (see Figure 2.2) to display the dialogue window in Figure 2.3. This window allows you to change the colour map, the intensity profile and the scale algorithm. All changes are immediately reflected on the image. Click Reset to return to the default scheme (*Real* colour map, *Ramp* intensity and *Linear Scale* algorithm).

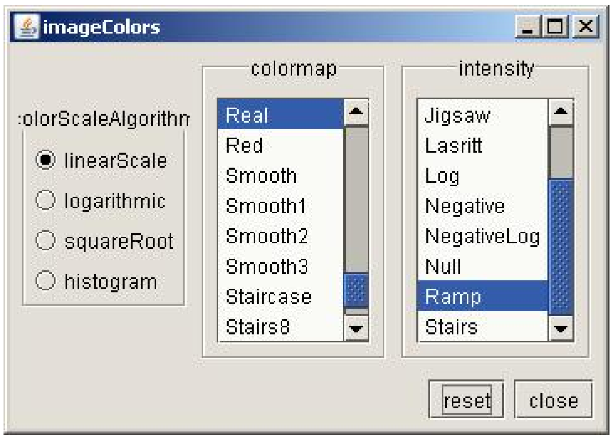


Figure 2.3. Colour map window.

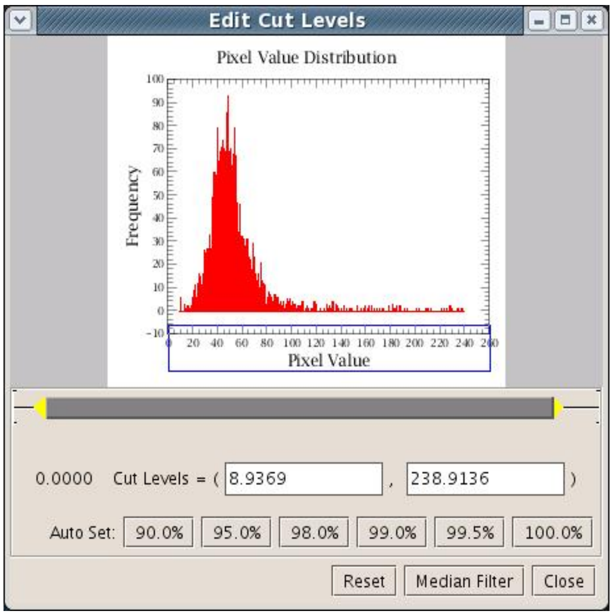
Editing the cut levels

Choose *Edit cut levels* from the context menu (see Figure 2.2) to display the dialogue window in Figure 2.4.

You can edit the cut levels in three ways:

- Click and drag the yellow arrows shown at either end of the histogram view to change the upper and lower level cutoffs.
- Enter the level values in the two text boxes.
- Click one of the *Auto Set* buttons

All changes are immediately reflected on the image and on the histogram plot. Click Reset to return to the default cut level of 99.5% of pixel values.



0.0000 Cut Levels = (8.9369 , 238.9136)

Figure 2.4. Cut level selection window.

2.2.2.1. Annotating an image

The annotation toolbox is shown in Figure 2.5.

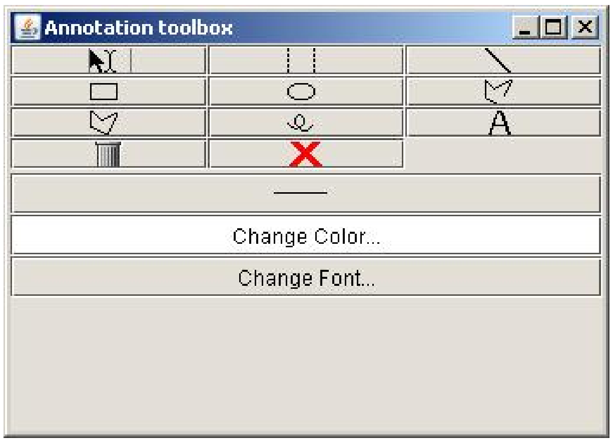


Figure 2.5. The annotation toolbox.

The buttons in the annotation toolbox appearing in Figure 2.5 have the following usage (from left to right and from top to bottom):

- Select annotation
- Select all annotations in a region
- Draw a line, a rectangle, an ellipse, a polyline or a polygon
- Draw 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. From top to bottom:

- Change the thickness of the line
- Change the colour of the annotation. The present colour of annotations is shown in the background.
- Change the font of the text annotation

Note

The *Select all annotations in a region* button only works when there are already annotations on the image. Pressing the button will select all the annotations which are in the selected region. This button can be used to change the colour or the line width of several annotations at once.

2.2.3. Viewing a data cube

When you double click on a variable representing a data cube, it will be opened like this:

- If the variable is of type `SimpleCube`, it will be opened in the *Cube Spectrum Analysis Toolbox*, described in more detail in Section 5.2.7.

- If the variable is of type `SpectralSimpleCube`, it will be opened in the *Spectrum Explorer*, described in more detail in Section 5.2.11 (see the heading *SpectralSimpleCube panel*).

You can also open your cube with another tool, like the *Standard Cube Viewer*, almost identical to the image viewer described in Section 2.2.1. To do so, right-click on the variable name and choose Open With → Standard Cube Viewer, or another of the available tools.

The cube viewer has some additional controls shown in Figure 2.6. With the slider and the two arrow buttons you can move through the layers of the cube. You can also input a layer number in the text box and press **Enter** to reach a specific layer. The rightmost box shows the wavelength of the current layer (the *2.0 LAYER* in the image below refers to a dummy data cube).

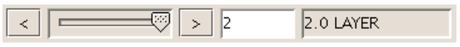


Figure 2.6. Additional controls for data cubes.

For information on how to display data cubes via the command line, see Section 2.3.4.

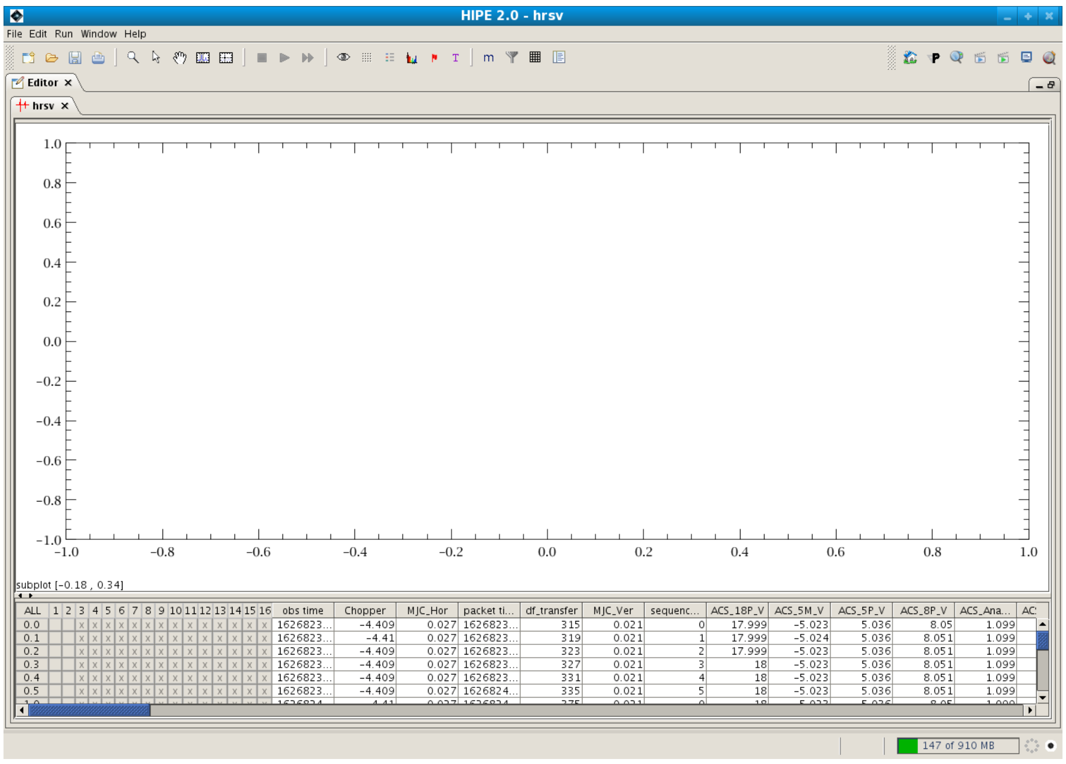
2.2.4. Viewing a spectrum

2.2.4.1. Starting the SpectrumExplorer

The SpectrumExplorer package allows you to visualise spectrum datasets. To activate it, click on a SpectrumDataset or Product in the Variables window or Observation Viewer with the right mouse button and select 'Open With' and 'Spectrum Explorer'. If this is the default, it suffices to double-click on the variable.

Initially an empty plot is displayed in the top part of the window that is opened and a selection panel is displayed in the bottom part.

The look of the selection panel depends on the SpectrumDataset type. A typical example is displayed in the following picture. When the added SpectrumDataset is a SpectralCube, a cube visualizer is displayed instead with which spectra can be selected.



subplot [−0.18 , 0.34]

ALL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	obs time	Chopper	MJC_Hor	packet ti...	df_transfer	MJC_Ver	sequenc...	ACS_18P_V	ACS_5M_V	ACS_5P_V	ACS_8P_V	ACS_Ana...	AC:
0.0			x	x	x	x	x	x	x	x	x	x	x	x	x	x	1626823...	−4.409	0.027	1626823...	315	0.021	0	17.999	−5.023	5.036	8.05	1.099	
0.1			x	x	x	x	x	x	x	x	x	x	x	x	x	x	1626823...	−4.41	0.027	1626823...	319	0.021	1	17.999	−5.024	5.036	8.051	1.099	
0.2			x	x	x	x	x	x	x	x	x	x	x	x	x	x	1626823...	−4.409	0.027	1626823...	323	0.021	2	17.999	−5.023	5.036	8.051	1.099	
0.3			x	x	x	x	x	x	x	x	x	x	x	x	x	x	1626823...	−4.409	0.027	1626823...	327	0.021	3	18	−5.023	5.036	8.051	1.099	
0.4			x	x	x	x	x	x	x	x	x	x	x	x	x	x	1626823...	−4.409	0.027	1626823...	331	0.021	4	18	−5.023	5.036	8.051	1.099	
0.5			x	x	x	x	x	x	x	x	x	x	x	x	x	x	1626823...	−4.409	0.027	1626824...	335	0.021	5	18	−5.023	5.036	8.051	1.099	

Figure 2.7. The Spectrum Explorer.

When a Product is selected for display, the bottom part will show a 'loading datasets...' message as long as the Product is being processed. Each SpectrumDataset found in the Product is added to the selection panel.

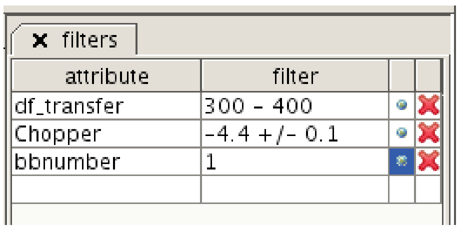
The location of the divisor between both panels can be changed through drag drop interaction. Clicking on one of the little black arrows displayed on the left edge of this divisor extents a single panel to its full size.

2.2.4.2. Selecting Spectra

The attribute columns in the selection panel can be used to find spectra that one wishes to plot. A single click on a header of such column sorts the rows according to that column's entries. Clicking it again inverts the sort order. A double click removes the sort and therefore brings the ordering back to its initial state.

With drag and drop, the columns themselves can be reordered. A right click on one the headers shows a dialogue box with a selection list of all column headers. With this list the columns can also be reordered or even hidden from view. Hold the shift button to hide/display a whole range of columns at once.

Furthermore, specific spectra can be selected by applying a filter on the attribute columns. Open the filter panel by selecting Dialogs -> Filter from the right-mouse click menu or by clicking on the filter icon in the button toolbar at the top of the HIPE screen. Specify the attribute name (from one of the column headers) and enter the filter values, that can be ranges, circular ranges or exact values. The filters are combined by applying the 'AND' operator. Clicking on the green circle next to a filter temporarily disables that filter. Clicking on the red cross removes it from the panel.



attribute	filter		
df_transfer	300 - 400	●	✗
Chopper	-4.4 +/- 0.1	●	✗
bbnumber	1	●	✗

Figure 2.8. Filters on attributes.

2.2.4.3. Displaying Spectra

In the general selection panel at the bottom, each row depicts an individual spectrum. The numbers in the first column show the index of the spectrum within the SpectrumDataset. If SpectrumExplorer was opened on a Product, the index is preceded by the index of the SpectrumDataset within the Product. For example, 2.3 denotes the fourth spectrum within the third SpectrumDataset within the Product (given that both indices start with 0).

Clicking the button in the first column displays all segments in that spectrum. A double-click removes them from the plot. The same accounts for the top row of buttons: clicking displays a single segment for all spectra, while double-clicking removes them from the plot. The 'ALL' button in the top left corner of the selection panel displays all segments of all spectra. Finally, individual segments can be displayed by the clicking the approprate box. The colour of the button is changed to the colour of the spectrum displayed in the plot. In case a Product is displayed with SpectrumDatasets containing different numbers of segments, the invalid segments are disabled and displayed with a grey 'x'. An example is shown in the figure above.

2.2.4.4. Button Bar

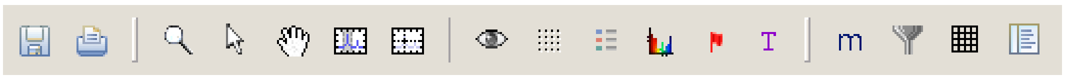


Figure 2.9. The button bar.

At the top of the HIPE screen, the SpectrumExplorer buttons following the 'New...' and 'Open File...' buttons have the following meaning:

- button 1: save the plot as a PNG, PDF, EPS or JPEG file
- button 2: send the plot to the printer
- button 3: zoom mode. This is the default mode when SpectrumExplorer is started. Change the horizontal and vertical plot ranges by drawing a rectangular box using the left mouse button. Control-left mouse button will un-zoom the plot (or use the Autorange option under the right mouse button).
- button 4: select spectra. A clicked spectrum will be displayed with a bold line. Any operation, such as the Tasks under the right mouse button, will then only apply to this particular spectrum. Also the selected spectrum can be dragged to a new panel (note that dragging to the left and top of the original panel is not possible). The spectrum can also be dragged to the Variables window where it will be stored as a new variable.
- button 5: pan mode. Pan through the spectrum by clicking the left mouse button and moving the mouse. If one only wants to pan along the x or y axes, click on the axis with the left mouse button and then move the mouse (or use the mouse wheel).
- button 6: select ranges. Click and drag to select ranges in a plot (the middle mouse button can be used anytime for this as well). This will create a vertical grey bar. Then in the spectrum selection mode (button 4), only this will be saved as a new variable.
- button 7: select points. Click and/or drag with the left mouse button to select one or more spectral points. These points can later be flagged or removed.
- button 8: (de-) activate preview mode. In preview mode a quick preview is displayed of all rows selected in the selection panel.
- button 9: display/hide grid in the active sub plot
- button 10: display/hide the plot legend
- button 11: switch between line and histogram mode
- button 12: display flagged channels
- button 13: show/hide the plot title
- button 14: open filter panel
- button 15: show metadata of the displayed SpectrumDataset
- button 16: open a raster panel showing all plots in the selection panel
- button 17: open the properties panel in the top-right part of the SpectrumExplorer to view and modify any plot parameter. The panel can also be opened using the 'Properties...' option under the right-click popup menu. If a paricular element in the context contains no changeable properties, the plot properties are displayed.

2.2.4.5. Plot Interactions

The Spectrum Explorer provides context-dependent plot interactions. The behaviour of mouse interaction depends on the location of the mouse cursor. The actual context is displayed in the left bottom corner of the plot panel. Next to the context you'll find the location of the mouse cursor in plot coordinates. The following table provides the some contexts and the mouse interaction behaviour.

Context	**Click**	**Ctrl-click**	**Drag**	**Scroll**
Subplot	Set as 'active'		Zoom/Select/Pan	Zoom

Context	Click	Ctrl-click	Drag	Scroll
Axis			Pan	Zoom
Spectrum	Select spectrum	Extend selection	Move spectrum to another subplot	
	Select point		Extract spectrum to a new variable	
			Use spectrum as task input parameter	
Selection			Same as above	
Marker edge			Resize marker	

A right click on a plot shows a popup menu with global and context specific options. Right clicking below or besides a plot gives the option to add another subplot in that place. The new subplot becomes 'active'. New selected spectra are displayed in the active subplot. To activate another subplot, right click on that subplot and check the radio button named 'active'.

2.2.4.6. Raster Panel

When SpectrumExplorer is used in raster mode (selected using the Raster button at the top button bar), a single spectrum is plotted plot for each row in the selection panel. This selection can be altered by making use of the filter panel. When all spectra contain pointing information, the plots are laid out on a latitude/longitude plane. Otherwise the plots are displayed in a rectangular grid.

The wave and flux ranges above the plot can be altered by textual input or by scrolling on top of the text field. After doing this, the slide bars below the ranges can be used to slide the sub range through the plots.

Use the scroll wheel on top of the plot to zoom. A single click on a plot opens the spectrum in the plot view of the SpectrumExplorer.

2.2.4.7. Preferences

Default SpectrumExplorer settings can be modified using the Edit-->Preferences button at the very top of the HIPE screen. The following options are available:

- Initial tool: specifies whether the Spectrum Explorer should start in zoom or select mode.
- ChartType: display plot in line style or histogram
- Display grid: on or off
- Display legend: on or off
- Start in preview mode: on or off

For a specific SpectrumDataset type, title/subtitle and legend element can be specified. Metadata fields and attribute fields can be filled in automatically by specifying the fields name between angular brackets. Optionally with a printf-style format suffix. For example, **`longitude%.2f"`** in the legend element field displays the value of the longitude attribute for each spectrum in the legend

2.2.5. Creating and viewing a TableDataset

A `TableDataset` is made up of columns. Each column contains an `ArrayDataset` (data), a description and a quantity value associated with the `ArrayDataset`. Each `ArrayDataset` can have up to five dimensions and can be of varying types.

These are the steps to follow to create, view, and plot graphs of a table dataset within HIPE.

1. Open the Workbench Perspective in HIPE.

2. If you already have a `TableDataset` loaded into HIPE skip to the next item. Otherwise type the following commands into the Console window. These instructions create a `TableDataset` with three columns, each containing a one-dimensional dataset. The first column contains the numbers from 1 to 100, the second column holds the sine value of the values in the first column, and the third column contains the values in the first column multiplied by 100. The column names are `x`, `sin` and `y` respectively.

   ```
   from herschel.share.unit import *
   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 the `x` column and its associated units, in this case a time duration measured in seconds.
   ❸ Here we have applied the `SIN` function from the Numeric library and added a description for the second column.

3. Next we wish to view the table we have created. Move your cursor over the item `t` in the *Variables* window and right-click on it. Choose Open With → Dataset Viewer from the menu. The table appears in the *Editor* window.

4. Now we wish to view the table in the TablePlotter task. Again right-click on the item `t` in the variable list and select Open With → TablePlotter. This opens the TablePlotter in the *Editor* window. You can find a complete guide to the TablePlotter in Section 2.3.7.

   When right-clicking a Table Dataset in the *Variables* window, the Open With menu offers two more options:

   - The *OverPlotter* can be thought of as a stack of TablePlotters, so that several graphs can be overlaid on top of each other. This tool is described in more detail in Section 2.3.8.

   - The *Power Spectrum Generator* computes a power spectrum for each column of a Table Dataset. This tool is described in more detail in Section 2.3.9.

2.3. In depth

2.3.1. Images and cubes

This section describes how you can store, display and analyse images and data cubes. You can find additional information in the developer (API) documentation of the herschel.ia.dataset.image, herschel.ia.gui.image and herschel.ia.toolbox.image packages.

All the image packages (herschel.ia.dataset.image, herschel.ia.dataset.image.wcs, herschel.ia.gui.image and herschel.ia.toolbox.image) are automatically loaded when starting HIPE. However, these might have to be imported by hand if executing scripts within a different environment.

An image is represented as a product of type `SimpleImage`, with the following components:

- The *image* itself, described by a `Numeric2d` (i.e. a 2D numeric array : this can be a `Double2d`, a `Float2d`, a `Long2d`, an `Int2d`, a `Short2d` or a `Byte2d`)

- The *errors* on the image, also described by a `Numeric2d`, but optional
- The *exposure* of the image (idem)
- A *flag*, described by a `Flag` (also optional)

Other information stored in the `Image` can be for example a `Wcs` (World Coordinate System) to do coordinate conversions, and the wavelength at which the image was taken.

When constructing an image, you usually start by making the `Wcs` and the `Numeric2d` that will be used as image data.

The following example shows how you can construct a `SimpleImage` with a valid `Wcs`, without errors and exposure, and with one pixel (55, 35) flagged out. It has 60 rows and 40 columns.

Note

The reference pixel is at position (*crpix1*, *crpix2*), with the pixels starting to count at (1,1). This corresponds to `row` = `column` = 0.

Note

The *crval* keywords for the pixel scaling are given in decimal degrees in RA en Dec.

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

# Construction of the image data (1)
myImageData = Float2d(60,40) #

for row in Int1d.range(60):
 for column in Int1d.range(40):
  myImageData.set(row, column, row + column)

# Construction of the flag (2)
myFlag = Flag(60,40) #
flaggedOut = Bool2d(60,40)
flaggedOut.set(55,35, True)
myFlag.setFlag("UNVALID", flaggedOut)

# Construction of the unit (3)
myUnit = FluxDensity.MILLIJANSKYS #

# Construction of the Wcs (4)
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = --22.5, \
 cdelt1 = 0.00028, cdelt2 = 0.00028, ctype1 = -"RA---TAN", ctype2 = -"DEC--TAN")
# Construction of the SimpleImage (5)
myImage1 = SimpleImage(description = -"test image", image = myImageData, \
 flag = myFlag, unit = myUnit, wcs = myWcs)
# Or using the ImportImageTask (6)
myImage2 = SimpleImage(wcs = myWcs)
importImage(image = myImage2, filename = -"[path]/ngc6992.jpg")
# where we now import our JPG image into the SimpleImage
```

1. The construction of a `Float2d` : at position (row, column) the pixel value is set to row + column
2. Pixel (55,35) is flagged out, using the *UNVALID* flag. Other flag types are possible (look in the subsection on flags).
3. Setting the unit for the pixel values. The flux associated with one count in the image (equivalent to *BUNIT* in a FITS image).
4. The construction of a `Wcs` object. The center pixel is set tot (29,29) and corresponds to the sky coordinate with right ascension 2h00m00s and declination -22d30'00". For more information, look into the subsection on `Wcs`.
5. Construction of a `SimpleImage` with the given image data and `Wcs`, but without errors and exposure.
6. Construction of another `SimpleImage` with the same `Wcs` applied to it. The `ImportImageTask` is used to load a JPEG image. There is no flag, no error, nor exposure in this case.

Example 2.1. Constructing a `SimpleImage`

Note

Using the `ImportImageTask`, data from *.jpeg, *.jpg, *.tiff, *.png, *.fits, *.fts or *.fit files can be loaded into an `Image`. When a FITS file is imported, the information in the header of the file is also included.

A `Cube` works in a very similar way to an `Image`. The only difference is that 3D datatypes should be given as parameters, instead of 2D. This holds for the cube data, as well as for the errors, exposures and flag. In the `Wcs` the parameters for the 3rd axis should also be specified.

2.3.1.1. Flagging out Pixels : the `Flag` Class

A `Flag` can be used to flag out pixels and specifying the reason for doing so. In the example below it is explained how you can do this.

```
myFlag = Flag(60,40) # (1)
myFlag.addFlagType("SATURATED", -"Saturated pixels") # (2)

flaggedOut1 = Bool2d(60,40)
flaggedOut1.set(55,35, True)
myFlag.setFlag("UNVALID", flaggedOut1) # (3)

flaggedOut2 = Bool2d(60,40)
flaggedOut2.set(50,35, True)
myFlag.setFlag("SATURATED", flaggedOut2) # (4)

print myFlag.getFlagTypes() # (5)
print myFlag.getFlag() # (6)
print myFlag.getFlag("UNVALID") # (7)
```

1. The `Flag` you create must be of the same dimensions as the `Image` to which you're going to attach it. In this case, it is a 60*40 `Flag`.
2. You can create up to 15 different flag types. Here, you create a new flag type with the name *SATURATED*. One flag type is standard available : *UNVALID*.
3. In these three lines is described how you can flag out the pixel with coordinate (55,35) with the *UNLVALID* flag type. Note that you have to construct a `Bool2d` for this and that this must be set to `True` at the appropriate position.
4. The saturated pixels are flagged out in a similar way. Note that you had to add the *SATURATED* flag type yourself.
5. Here you print the existing flag types for this `Flag`. In this case, these are *SATURATED* and *UNVALID*.
6. Here you print a `Bool2d` with the same dimensions as the `Flag`. All flagged pixels are marked as `True`. In this case, pixels (55,35) and (50,35) are marked as `True`, all the others as `False`.
7. Here you print a `Bool2d` with the same dimensions as the `Flag`. All pixels flagged as *UNVALID* are marked as `true`, all others as `false`.

Example 2.2. Constructing a `SimpleImage`

Note

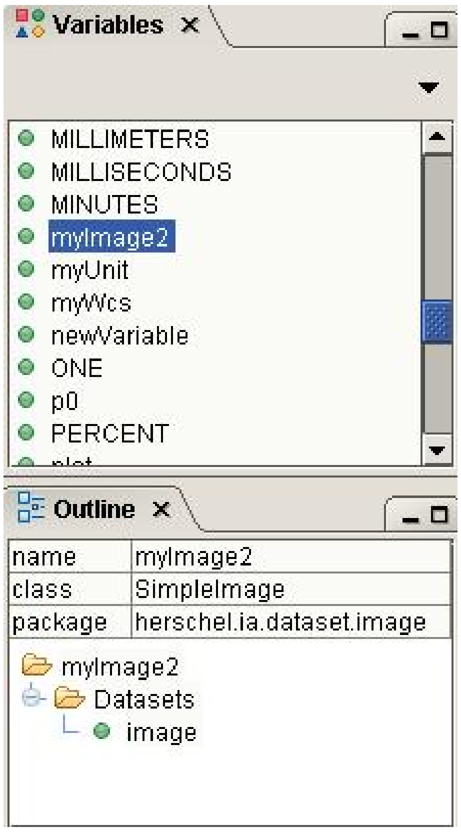
We are well aware of the fact that "unvalid" in not a true English word. In the future this flag type should be changed to *INVALID*.

2.3.2. Creating a test image

If you have no image available to test the utilities described here, you can use the following script to create one from any JPEG file and associate a WCS to it. Copy and paste the script into the *Editor* view after opening a Jython script window, or copy it into the *Console* view and run from there.

```
# Create some fake WCS information
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = --22.5, \
 cdelt1 = 0.00028, cdelt2 = 0.00028, ctype1="RA---TAN", ctype2 = -"DEC--TAN")
# Create a SimpleImage with WCS in it
myImage2 = SimpleImage(wcs = myWcs)
# Put the image into the SimpleImage
# *.jpeg, *.jpg, *.tiff, *.tif, *.png, *.fits, *.fts or *.fit
# files are accepted.
importImage(image = myImage2, filename="[path]/ngc6992.jpg")
```

You can also import the image via the `importImage` task available in the *Tasks* view list. Click on `myImage2` in the *Variables* view then double-click on `importImage`. Type or select a file name and click on Accept.



MILLIMETERS
MILLISECONDS
MINUTES
myImage2
myUnit
myWcs
newVariable
ONE
p0
PERCENT

name	myImage2
class	SimpleImage
package	herschel.ia.dataset.image

myImage2
Datasets
image

Figure 2.10. Variables view with `SimpleImage` variable highlighted.

Double-clicking on the variable "myImage2" in the "Variables" view will automatically display the image in a new Editor window. A single right click in the same place will indicate that this can be "Open(ed) with..." a Product display as well. This shows header information and the fact that there is a single image dataset in the `SimpleImage` product we have created.

2.3.3. Viewing an image

From the command line you can use an object of the `Display` class to view images and data cubes. To display an image use the following command:

```
myDisplay1 = Display(myImage1)
```

This will show the same graphical interface you would obtain by double clicking on an image in the *Variables* view.

You can then create a second `Display` object to view a second image:

```
myDisplay2 = Display(myImage2)
```

The variables `myDisplay1` and `Display2` allow you to refer to the two objects and their contents separately.

You can add an extra parameter when initialising the `Display`, which decides whether the window should be shown or not. This can be very useful in scripts, where you don't want all images to be shown on the screen, but where you want to look at some images after the execution of the script. This can be done like this:

```
myDisplay = Display(myImage, False)
```

You can make the window visible, typing

```
myDisplay.setVisible(True)
```

The following table shows some, but not all, the methods applicable on `Display` objects. For an exhaustive list of all methods, have a look in the `Display` javadoc. To execute a certain method, type **myDisplay2.<methodname>**.

Table 2.1. Useful methods on `Display`

getIntensity(int row, int column) -> double	Returns the intensity at the given pixel coordinates (row, column)
getIntensityFromWorldCoordinates(double ra, double dec) -> double	Returns the intensity at the given sky coordinates (ra, dec)
getUnit() -> Unit	Returns the unit of the shown image
setUnit(Unit<?> unit)	Sets the unit of the shown image
getZoomFactor() -> float	Returns the zoom factor of the shown image
setZoomFactor(float zoomFactor)	Sets the zoom factor
zoom(double row, double column, float zoomFactor)	Zooms on the given pixel coordinates (row, column) with the given zoom factor
zoomWorldCoordinates(double ra, double dec, float zoomFactor)	Zooms on the given sky coordinates (ra, dec) with the given zoom factor
zoomIn()	Zooms in
zoomOut()	Zooms out
getCutLevels() -> double[]	Returns the cut levels of the shown image
setCutLevels(double percent)	Sets the cut levels according to the given percentage
setCutLevels(double[] minmax)	Sets the cut levels
setCutLevels(double min, double max)	Sets the cut levels
flipYAxis()	Flips the y-axis
isFlipped() -> boolean	Returns whether the y-axis is flipped
getDepthAxis() -> int	Returns the depth axis
setDepthAxis(int depthAxis)	Sets the depth axis

2.3.3.1. Using different layers

It is possible to display several layers in one `Display`. This can be done by adding a layers to the existing `Image`, or by displaying a `Cube` or a `Numeric3d` datatype (`Double3d`, `Float3d`,...). Adding a layer can be done like this :

```
myDisplay.addLayer(myImage1)
```

This way we add `myImage1` to `myDisplay`. You also see that a slider appears in the status bar, which you can use to switch between the different layers. This is the same slider that appear when you display data cubes.

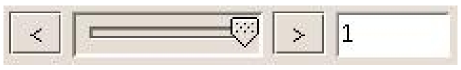


Figure 2.11. Slider to switch between layers.

Note

When you change the zoom factor of the displayed `Image`, it is important to know what will happen to other `Layers`. If the current `Image` was added to the `Display` separately (as an `Image`), then no other `Layers` will be affected. However, if the displayed `Image` is part of a `Cube`, all other layers in this `Cube` will be affected.

2.3.3.2. Placing annotations on an image

It is possible to draw figures and put text, so called annotations, on an `Image`, using `Display`.

You can place these kinds of annotations on an `Image` in `Display`, via the command line :

- *Regular text annotations*, using the **addAnnotation(...)**, **setAnnotationFont(...)** and **setAnnotationFontColor(...)** methods
- *Greek text annotations*, using the **addGreekAnnotation(...)**, **setAnnotationFont(...)** and **setAnnotationFontColor(...)** methods

Note

The **addGreekAnnotation(...)** method converts normal characters to Greek characters ('a' becomes 'alpha', 'b' becomes 'beta',...)

- *Figures as annotations*, using the **addEllipse(...)**, **addLine(...)**, **addPolygon(...)**, **addPolyline(...)** and **addRectangle(...)** methods

Note

The **addPolygon(...)** and **addPolyline(...)** methods need an array of doubles as parameter. In such an array, the coordinates should be added as polygon(([x1, y1, x2, y2,...]),...).

The following example shows how you can do this on the command line. Also the resulting `Display` is shown.

```
# Imports
from java.awt import Font
from java.awt import Color

myDisplay2 = Display(myImage2)

# Placing a text annotation at position (321, 224)
myDisplay2.addAnnotation("Veil nebula", 321, 224)
# Changing the font type and size of the annotations
myDisplay2.setAnnotationFont(321, 224, Font("Dialog", 0, 32))
# Changing the annotation colour
myDisplay2.setAnnotationFontColor(321, 224, Color(0,0,255))
# Adding an ellipse with center at (268.5,500.0), width = 38 and height = 37,
# linewidth = 3.0 and black colour
myDisplay2.addEllipse(268.5, 500.0, 38.0, 37.0, 3.0, Color.green)
# Adding a Greek text annotation at position (100,500)
myDisplay2.addGreekAnnotation("a = 12.34, d = +30.30", 100, 500)
# Changing the font and colour of the annotation
myDisplay2.setAnnotationFont(100, 500, Font("Dialog", 0, 20))
myDisplay2.setAnnotationFontColor(100, 500, Color(0,0,0))
# But white is more visible
myDisplay2.setAnnotationFontColor(100, 500, Color.white)
```

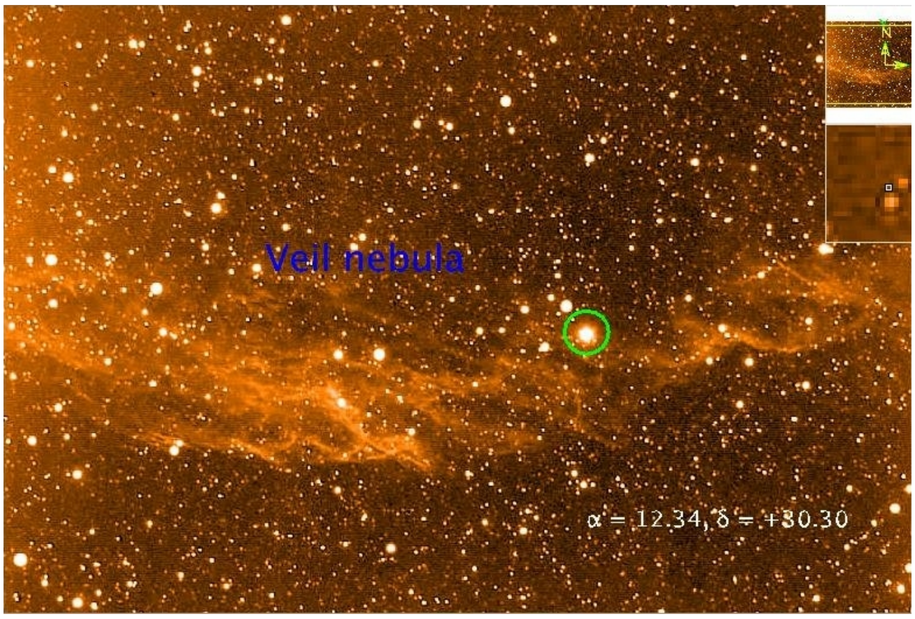
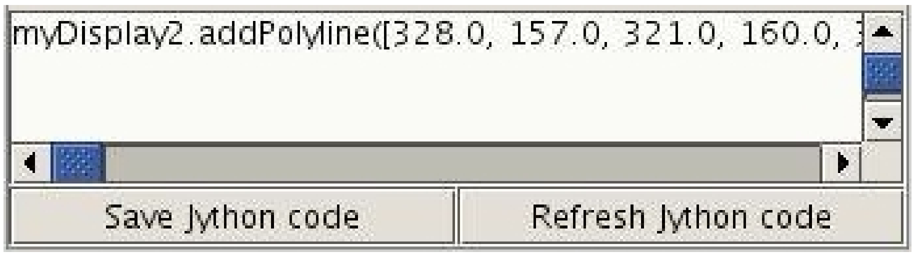


Figure 2.12. Adding annotations to a Display.

The annotation toolbox was covered in Section 2.2.2.1. You can open the annotation toolbox via the command line as follows:

```
myDisplay2.annotationToolbox()
```

When you open the annotation toolbox via the command line you get an additional window with the Jython code corresponding to your commands, and two buttons to save or refresh the code.



```
myDisplay2.addPolyline([328.0, 157.0, 321.0, 160.0,
```

Figure 2.13. Jython code appearing in the annotation toolbox.

Note

If you change the size of a text annotation, this will not be reflected in the Jython code.

2.3.3.3. Opening other dialogue windows via the command line

These commands display the dialogue windows for editing colours and cut levels:

```
HIPE> myDisplay.editColors()
HIPE> myDisplay.editCutLevels()
```

2.3.4. Viewing a data cube

A `SimpleCube` contains one or more 3d images and works in a very similar way to `SimpleImage`. A 3-dimensional datatype should be given as input.

The following example shows how to create and display a dummy data cube:

```
s = SimpleCube()
d = Double3d(3,4,5,20.5) # produces a cube of 3x4x5 all with values 20.5
s.setImage(d)  # include cube of information into our SimpleCube
# the depth of this cube is given by the first integer, 3.
# The cube can be displayed using
show = Display(s)
# The depth axis can be changed by the setDepthAxis method, e.g.
show.setDepthAxis(2)
# where the depth would now be the third dimension
# of the image available -, or 5.
# In each case the cube is shown as
# image layers. The current layer viewed is determined by a slider
# to the bottom right of the display screen. Moving the slider left or right
# shows the image stored in each of the layers along the depth axis.
```

For information on how to display data cubes via the HIPE graphical interface, see Section 2.2.3.

2.3.5. Viewing metadata and array data associated to an image

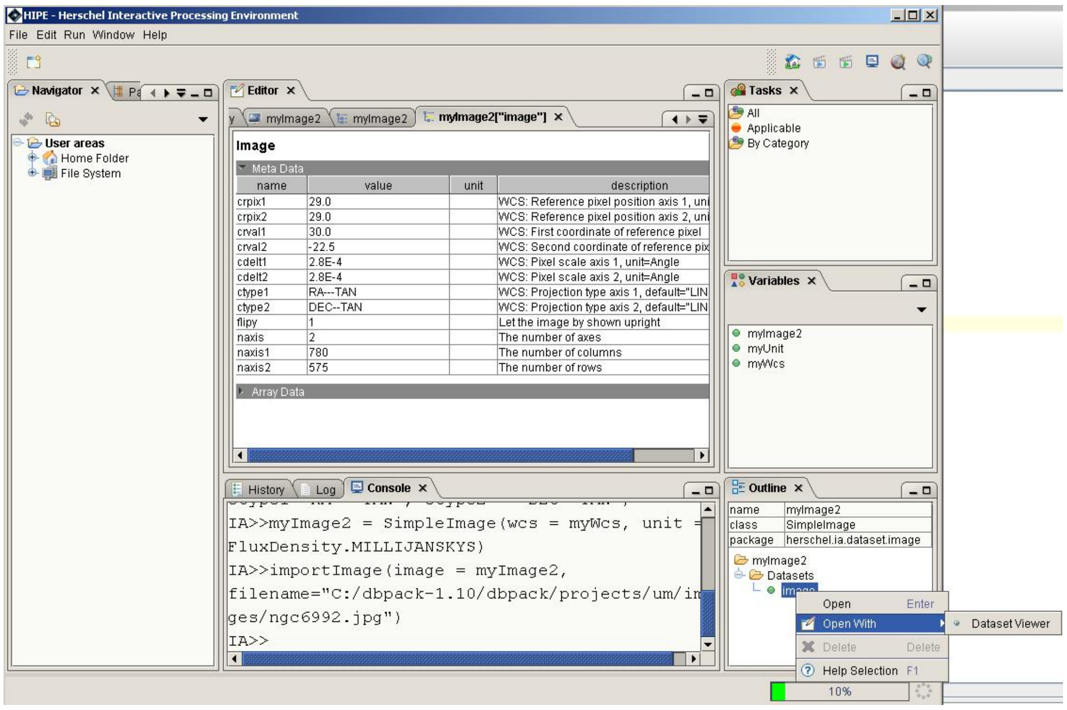
An image can have several datasets. For example, we can include a flag image dataset for flagging bad pixels (see Section 4.3.5 for more information). Each of these datasets have associated metadata, which has the same role as header information in a FITS file. It indicates associated flux and coordinate information plus processing history (if appropriate) etc.

To view the metadata (and array data) associated with an image dataset requires opening a Dataset viewer. This can be done in two ways.

- First a right-click on your image variable name in the "Variables" view (e.g., on "myImage2"). A short menu including "Open With...." appears. Choose the product viewer. The product view is shown which includes some overview information/metadata plus a list of datasets (at the bottom of the datasets -- and could include a number of image layers). Do a right-click on one of the datasets to see the "Open With..." in the short menu. Select Dataset viewer.

- A single click selection of the image in the *Variables* menu list shows its outline in the *Outline* view. Opening the folder in the *Outline* view to see the datasets in it and right-click on a dataset to see the context menu menu with "Open With...." and the dataset viewer selectable. Note that the *Outline* view is only available in the *Full Work Bench* perspective. See the *HIPE Owner's Guide* for more information on views and perspectives.

Any of the above will provide a view of the metadata plus the data values of the array making up the dataset within a window in the "Editor" view. View of either the metadata or array data can be toggled using the arrows to the left of the metadata/array data names in the "Editor" window (see Figure 2.14).



name	value	unit	description
crpix1	29.0		WCS: Reference pixel position axis 1, uni
crpix2	29.0		WCS: Reference pixel position axis 2, uni
crval1	30.0		WCS: First coordinate of reference pixel
crval2	-22.5		WCS: Second coordinate of reference pix
cdelt1	2.8E-4		WCS: Pixel scale axis 1, unit=Angle
cdelt2	2.8E-4		WCS: Pixel scale axis 2, unit=Angle
ctype1	RA---TAN		WCS: Projection type axis 1, default="LIN
ctype2	DEC--TAN		WCS: Projection type axis 2, default="LIN
flipy	1		Let the image by shown upright
naxis	2		The number of axes
naxis1	780		The number of columns
naxis2	575		The number of rows

```
IA>>myImage2 = SimpleImage(wcs = myWcs, unit =
FluxDensity.MILLIJANSKYS)
IA>>importImage(image = myImage2,
filename="C:/dbpack-1.10/dbpack/projects/um/im
ges/ngc6992.jpg")
IA>>
```

name	myImage2
class	SimpleImage
package	herschel.ia.dataset.image

Figure 2.14. Metadata and Array data view using the Dataset viewer with an image.

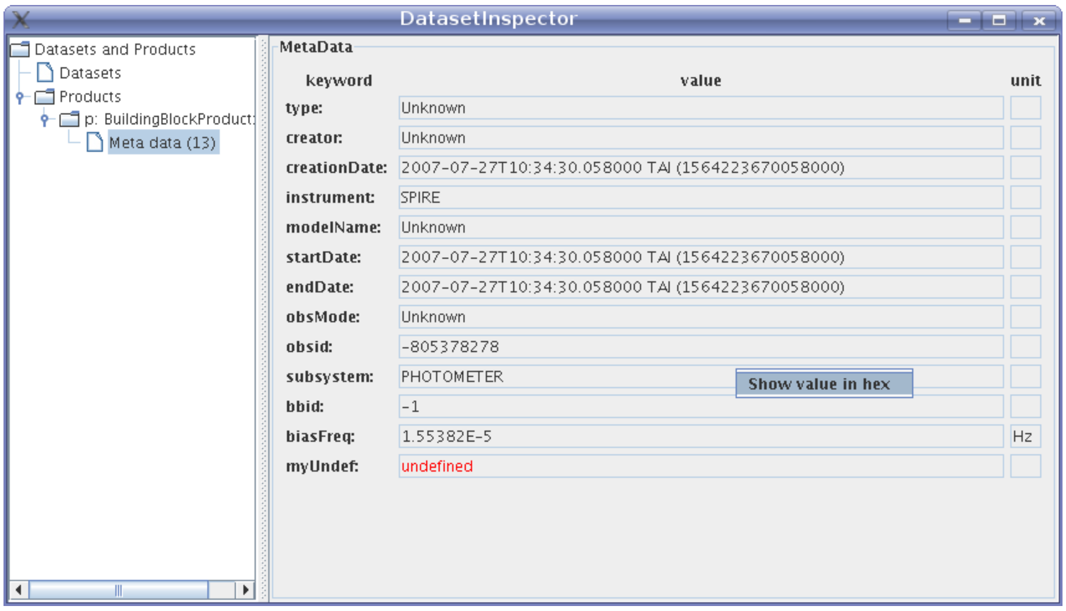
2.3.6. The Dataset Inspector

This section describes in detail a graphical tool to inspect table datasets, the *Dataset Inspector*. You can start it with the following commands:

```
HIPE > from herschel.ia.dataset.gui import DatasetInspector
HIPE > DatasetInspector()
```

The main window is 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 2.15 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.



MetaData

keyword	value	unit
type:	Unknown	
creator:	Unknown	
creationDate:	2007-07-27T10:34:30.058000 TAI (1564223670058000)	
instrument:	SPIRE	
modelName:	Unknown	
startDate:	2007-07-27T10:34:30.058000 TAI (1564223670058000)	
endDate:	2007-07-27T10:34:30.058000 TAI (1564223670058000)	
obsMode:	Unknown	
obsid:	-805378278	
subsystem:	PHOTOMETER	
bbid:	-1	
biasFreq:	1.55382E-5	Hz
myUndef:	undefined	

Figure 2.15. 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 you 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 how time is represented see the *Scripting and Data Mining* guide: Chapter 6). By right-clicking on the parameter values you 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. Three of them are described in the following sections.

2.3.7. The TablePlotter

The *TablePlotter* utility is a GUI tool to view and analyze table datasets which are organized in columns with an equal number of rows, for instance time ordered detector signals. In addition the tool provides advanced means of interactively selecting subsets of this data and create new table datasets from these selections. The TablePlotter appears as a tab in the *Editor* view.

TablePlotter does *not* support other types of datasets.

2.3.7.1. Invoking TablePlotter

- **Invoking TablePlotter as a Viewer in HIPE**

  The TablePlotter works with Table Datasets and products that contain Table Datasets. For example, double clicking on a FITS binary table file in the Navigator view of HIPE will load the file into a product containing a table dataset and automatically bring up the product viewer. Right clicking on the table dataset within the product and selecting *Open With* leads to a choice of viewers and tools that can be applied (see Figure 2.16).

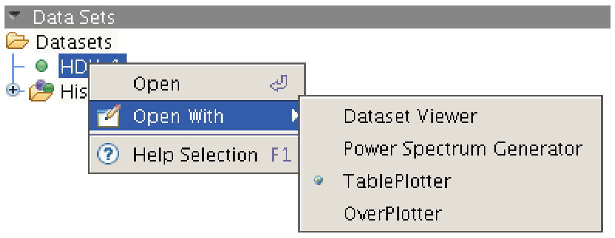


Figure 2.16. Viewers available for a table dataset in the product viewer, among them TablePlotter and OverPlotter.

Selecting "TablePlotter" opens the table dataset and brings up a view with the main TablePlotter screen (seeFigure 2.17).

- *Invoking TablePlotter from the command line or from a script*

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

```
from herschel.ia.gui.explorer.table import TablePlotter
from herschel.share.component import WindowManager
```

  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(tbs).component, 1)
```

  If you have a product created by reading in a FITS file containing a binary table, the first table dataset can be easily extracted with the default method. For instance, if a FITS file was read by double clicking on it in the navigator view, a product will appear as a variable. Assuming the variable name is "Myfile", the following command lines send it to TablePlotter.

```
wm = WindowManager.getDefault()
wm.addWindow("test", TablePlotter(Myfile.default).component, 1)
wm.addWindow('test', TablePlotter(TablePlotterExerciseFile["HDU_1"]).component,
1)
```

  If the product contains more than one dataset, the desired table dataset can be retrieved by its name. If you don't know the name of the dataset, a list of datasets can be obtained with the keySet method. In the following example the list of dataset names is obtained and printed, then the first dataset is chosen and displayed in TablePlotter.

```
wm = WindowManager.getDefault()
datasets = Myfile.keySet()              #Get the names of the datasets
print datasets                          #Here you see the names of the datasets
within the product
datasetName = datasets[0]               #Choose your dataset, in this case the
first with index 0
wm.addWindow("test", TablePlotter(Myfile[datasetName]).component, 1)
```

  If invoked from the command line, the TablePlotter will appear in its own window, instead of a HIPE view.

  If the name of the dataset is unknown, but its sequence number is known, the following shortcut can be used, in this case for the first dataset with index 0:

```
wm = WindowManager.getDefault()
wm.addWindow("test", TablePlotter(Myfile[Myfile.keySet()[0]]).component, 1)
```

2.3.7.2. Layout of the TablePlotter

When TablePlotter is invoked, it displays an X/Y-plot of the first two columns of the selected Table Dataset (See Figure 2.17). The TablePlotter GUI contains three major components: the plot display area, the plot control panel on the right, and axis selection boxes on the bottom. Sometimes it is necessary to adjust the window size and the sizes of the sections to see all components.

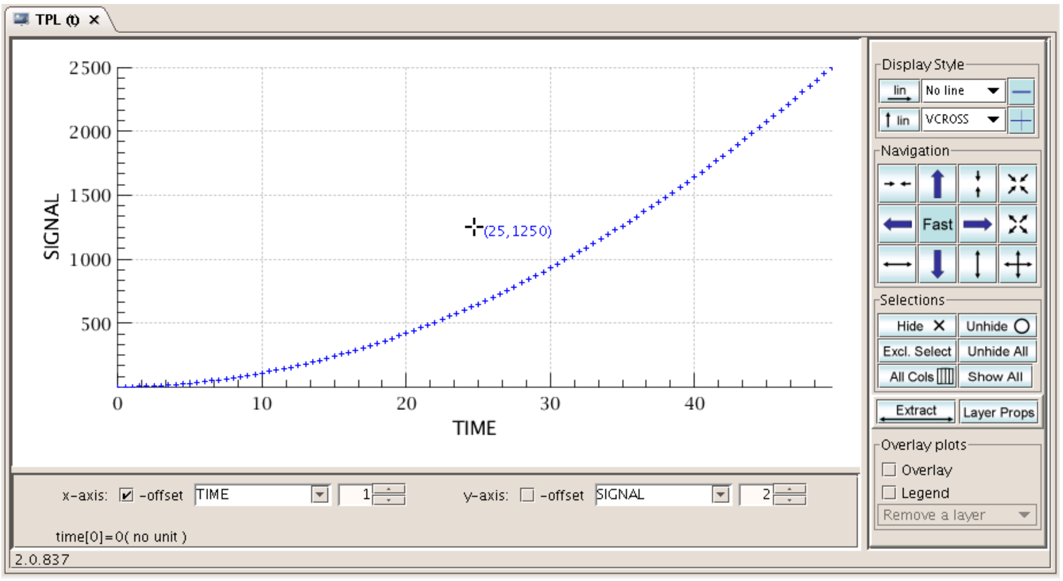


Figure 2.17. Layout of the TablePlotter GUI.

2.3.7.3. Controls and functions

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

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

  Under the graphics display area, two selector arrangements allow to assign columns in the table to the X and Y-axis of the plot. The elements of each selector are a combo box and a spinner.

  By default the first column of the TableDataset is associated with the X-axis. The second column is initially associated with the Y-axis.

  Clicking the arrow on the right of the combo box invokes a drop down menu with the displayable columns of the table dataset. Holding down the left mouse button and moving the mouse up or down scrolls through the columns if more than 8 columns are present. A column is selected by clicking on it. This list can be quite large. To help with the selection, a substring can be entered after clicking into the white name field of the combo box. Only columns whith names containing this substring will be shown in the drop down menu. No distinction is made for upper or lower-case characters in this selection.

  Columns can also be selected by index using the spinner, either by entering the index number directly after clicking into the index field, or by clicking on the up or down arrow buttons of the spinner. Fast forward/backward selection of columns in the spinner can be achieved by holding the left mouse button down and moving the mouse up or down.

The axis selector provide an additional "virtual" index column that allows to plot columns against the order in which they appear in the table dataset. This column only exists for convenience and is for instance not part of the extracted dataset, as shown further below.

In addition, two checkboxes named *"- offset"* allow you to subtract offsets from the data along both axes. This is useful, for example, if an axis corresponds to absolute times like TAI that start at an Epoch some time ago and bear a large offset compared to the time period covered by the data. When a checkbox is activated, the value of the subtracted offset appears below it.

- **Display style:**

  The control buttons in this section change the type of scaling of the X- and Y-axes, as well as the syles of lines and symbols used in the plot.

  The linear scale is selected for the X-axis. Clicking on the button will switch to logarithmic scale.

  The linear scale is selected for the X-axis. Clicking on the button will switch to logarithmic scale.

  The linear scale is selected for the Y-axis. Clicking on the button will switch to logarithmic scale.

  The linear scale is selected for the Y-axis. Clicking on the button will switch to logarithmic scale.

  The two pull-down menus select line- and symbol-styles. The selection of symbol styles is only available when the line styles are either *MARKED*, *MARK_DASHED* or *NONE*.

  / Increase/decrease symbol sizes.

- **Navigation:**

  The navigation field contains several buttons to zoom and pan within a plot. In addition the view can be controlled with the mouse pointer. Left clicking into the field, and pulling across an area with the left mouse button held down selects this area. This is called furtheron a hold-and-drag operation. When the mouse button is released, this area will be scaled so that it now fits the plot window (zoom-in).

  / Zoom in/out simultaneously in X- and Y-axis.

  / Zoom out along the X/Y axis only.

  / Pan the view towards the left/right.

/ Pan the view up/down.

The size of each zooming or panning step is controlled by a toggle button at the center of the Navigation field as follows:

Fast This button signifies that the fast mode is selected. Clicking on it toggles to slow mode.

Slow This button signifies that the slow mode is selected. Clicking on it toggles to the fast mode.

Layer Props This button opens the Preferences menu. The first entry in this drop-down menu opens a Properties window, where the factors can be changed that control fast and slow zooming and panning (for details see the *Preferences* section below).

This button switches into free-scale mode. It is one of the most frequently used buttons. The displayed ranges on X- and Y-axis are selected automatically to show all visible datapoints of the currently selected columns with optimal zoom parameters.

/ Switch the X/Y axis into free-scale mode.

- **Selections:**

  Table Plotter is not only a display tool for table datasets, but also a data selection tool. The selection feature can be used to hide or select a particular portion of the data points, to make use of the fast automatic scaling when scanning through many columns of data.

  The data selection feature, is also very useful for unplanned, ad-hoc, interactive data analysis tasks. Subsets of data in a table can be selected and extracted into new table datasets, that can then be sujected to other tools or tasks like the power spectrum tool. Typical applications would be for instance to manually remove glitches from a signal time stream, or to extract a specific period of a signal time stream out of a sequence of instrument configurations.

  The following buttons are relevant in this respect:

  Show All This button signifies that all data points are being displayed. De-selected data points are replaced by a small red cross. The automatic scaling takes also de-selected data into account. Clicking on this button switches to "Selected Only" display mode.

  Sel Only This button signifies that only selected data points are being displayed. De-selected data points are not shown. The automatic scaling takes only selected data into account. Clicking on this button switches to "Show All" display mode.

Hide ✕ Clicking this button first, and then performing a drag-and-hold operation within the plot hides all selected data points within the selected rectangle. In "All Columns" mode only the X-axis range is taken into account (see below).

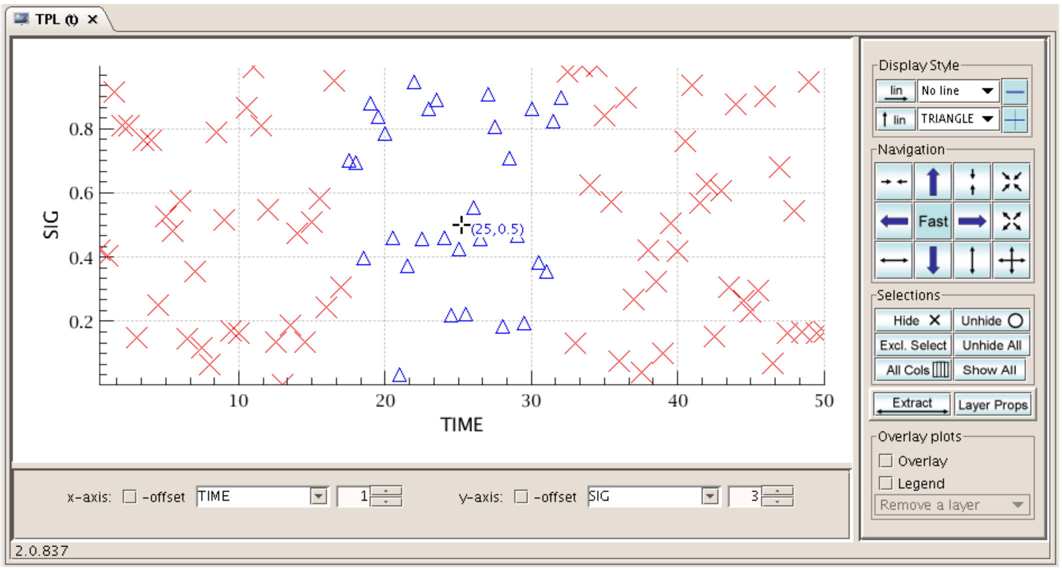
Unhide ○ Clicking this button first, and then performing a drag-and-hold operation within the plot selects all hidden data points within the selected rectangle. In "All Columns" mode only the X-axis range is taken into account (see below).

Excl. Select Clicking this button first, and then performing a drag-and-hold operation within the plot selects all data points within the selected rectangle and de-selects everything outside. In "All Columns" mode only the X-axis range is taken into account (see below).

Unhide All This button will re-select all hidden data points.

Current Col This button signifies that selections and de-selections only affect the two columns used for the plot. Clicking on this button will switch Table Plotter into "All Colum" mode.

All Cols This button signifies that selections and de-selections affect all columns of the table. The selection is based on the range on the X-axis, while the selected Y-axis range is ignored. Clicking on this button will switch Table Plotter into "Current Colum" mode.



lin No line
lin TRIANGLE

x-axis: ☐ -offset TIME 1
y-axis: ☐ -offset SIG 3

2.0.837

Figure 2.18. The plot with selected (blue) and hidden (red crosses) data points.

- **Printing and saving the plot:**

  Right-click on the plot to display a context menu with the entries *Save as* and *Print*. You can save your plot in PDF, PNG, JPEG or EPS format.

- **DataseteExtraction:**

  Besides visualization, the Table Plotter can be handy for creating new datasets out of existing ones. Typically this is done in data analysis where a specific portion of interest is selected and saved into another dataset for subsequent analysis. The result becomes another table dataset. The extracted columns are the two being displayed while in "Current Columns" mode, or an arbitrary user selection of columns in "All Columns" mode. As a general rule, any row, where at least two columns represent a valid datapoint (X,Y), will appear in the result. Data that were "hidden" in such a row are replaced by NaNs. All other rows will be purged from the resulting table dataset.

The selection of datapoints is internally done with flags that exist for each datum. Making selections while choosing different columns for the X-axis can have sometimes results that first appear confusing, but make perfect sense in a logical way. Especially the **Exclusive Select** button and the **Unhide** button should be used with due consideration of the side effects.

Extract This button extracts a subset of the data that remains selected after all prior selection operations. The selected data will be extracted into a new table dataset that will be fed back into the session. A name can be assigned to the new variable, which will appear in the Variables view.

If *Current Col* is selected, only the selected data points in the currently displayed column will be extracted.

If *All Cols* is selected, the selected data points in all the columns become available for extraction. After clicking *Extract*, a column selection window (see Figure 2.19) pops up, allowing to **Add** individual columns or **Add All** columns to a list. Individual columns can be also **Remove** again from the selection. The **Remove All** button allows to start over. **Up** and **Down** buttons are available to change the order of columns in the new dataset (see Figure 2.19).

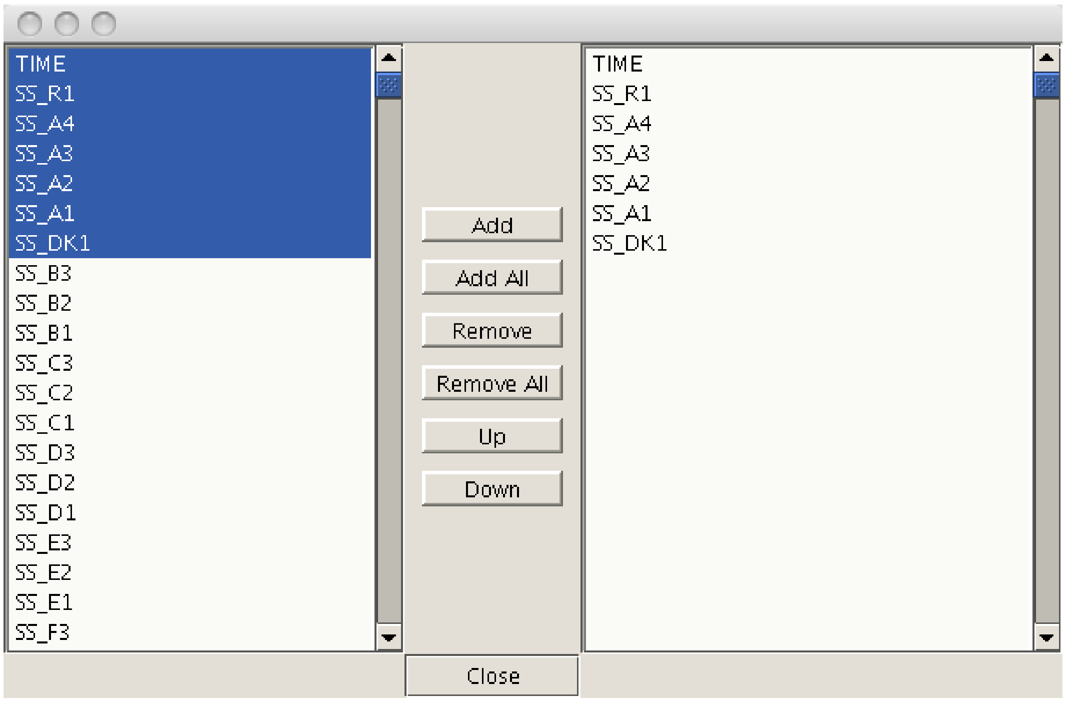


Figure 2.19. Extract Selected Data from Multi Columns to a New DataSet.

Clicking Close completes the extraction. You can then enter a name for the new dataset or accept the default (see Figure 2.20).

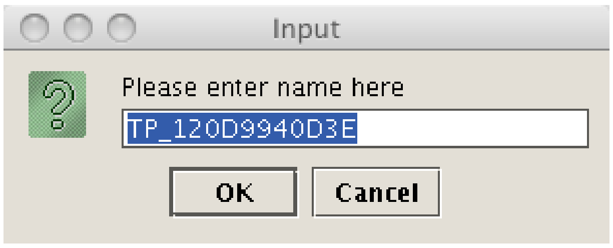


Figure 2.20. Giving a name to the new dataset.

The new table dataset appears as a new variable in the *Variables* view of HIPE.

- *Overlay Plots:*

  Even though the TablePlotter was primarily designed for single X-Y scattergram display, there is limited overlay capability available. For any more complex overlay plotting, the **Over Plotter** was created that is described in detail further down.

  Simple overlay plots are created by marking Overlay in the Overlay plots panel on the lower right, and selecting another column for the Y-axis. The old plot stays on display and the new X/Y-plot is overlaid with a different color. If different symbols, symbol sizes or line styles are required, they must be selected now. They can not be selected at a later stage. While Overlay is on, the Y-axis will have the same scale for all overlays and it is not possible to select another column for it. The only way to change a plot that was done earlier, is to remove the overlay in question with the **Remove a layer** drop-down menu, and selecting the column for the Y-axis again. Activating the **Legend** button shows the relation between color and name of the overlay in a legend (see Figure 2.21).

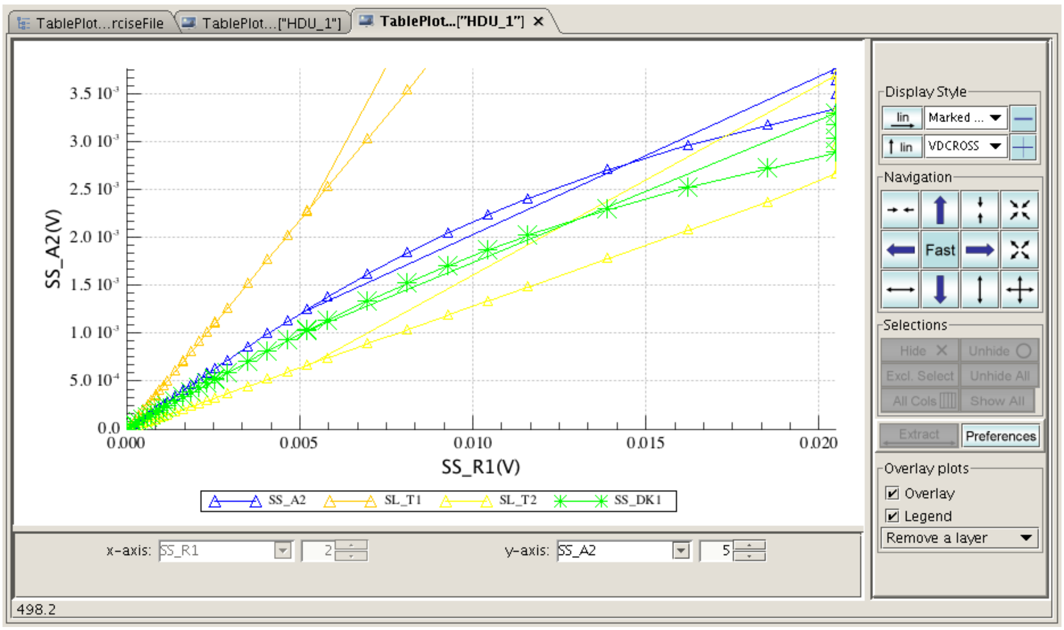


Figure 2.21. Simple overlay plots of different columns plotted against the same X-axis are created by marking the Overlay field.

- *Layer Props:*

  This button provides a drop-down menu, giving access to the display rules for complex data (the Appearances entry is still under development). See ???.

The Table Plotter is able to show complex data in four different representations: the modulus, the real part, the imaginary part and the phase.

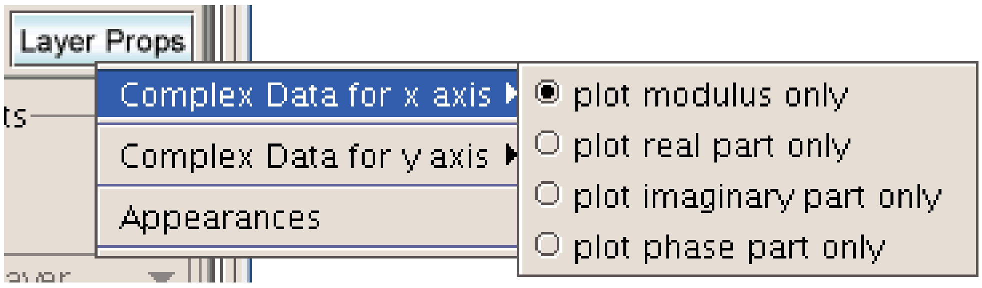


Figure 2.22. Preferences: Complex data can be displayed in four different ways as shown in this properties menu.

The selected preferences are stored in a properties file and will be remembered the next time you open Table Plotter.

- *Advanced command line control of TablePlotter*

  After invoking Table Plotter from the command line or a script, its display can be further controlled, allowing for integration of this tool into other applications that require interactive X/Y display and/or data selection. As stated before, the following imports must be performed first.

```
from herschel.ia.gui.explorer.table import TablePlotter
from herschel.share.component import WindowManager
```

  A Table Dataset tbs would be plotted as follows in a Jython script or from the command line. Note that in this case we retain the object tpl inbetween. This link enables us to access the Table Plotter and its components from the command line.

```
wm = WindowManager.getDefault()
tpl = TablePlotter(tbs)
wm.addWindow('test', tpl.component, 1)
```

  Now we should see a Table Plotter window as before, coming up detached of the HIPE window. We can now go about our business in HIPE. In case we make selections, we can get the result back into the session with the following commands.

```
extbl = tpl.activeLayerStruct.extractedTableDataset
flags = tpl.activeLayerStruct.flags
```

  The variable extbl now contains the resulting TableDataset after selection. It contains only rows with at least two valid entries. Deselected entries are replaced by NaNs. Sometimes however it is more convenient to just return the flags that were actually set for the original table dataset. This is done by the second line, where the flag array is saved in the variable flags. The dimensions of this flag array match those of the original table dataset tbs, but the type is a 2 dimensional Boolean array.

  The Table Plotter can also be pre-loaded with a flag array, which can be convenient in programmed applications.

2.3.8. The Over Plotter

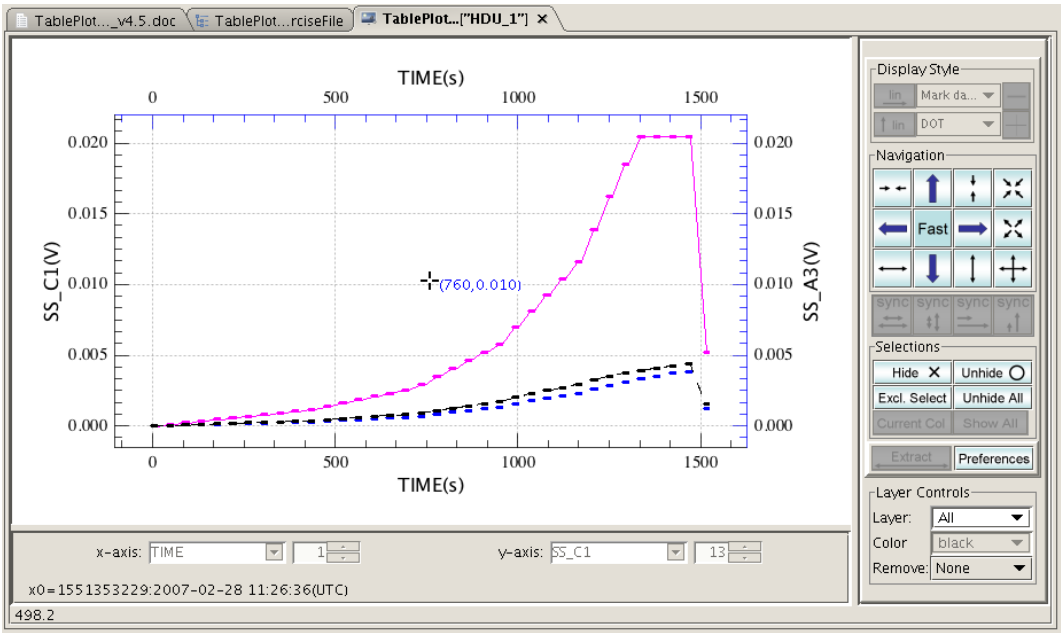
The Over Plotter is a consequential evolution out of the Table Plotter. It can be thought of as a stack of individual Table Plotters with the same individual functionalities so that several graphs can be overlaid on top of each other with their individual scaling, panning, and data point selections. In addition, the OverPlotter provides capabilities to navigate the stack of layers in a coordinated fashion, i.e. like a stack of glued together transparencies. It further allows for synchronization of axis scales of different layers and synchronous selection of data across layers. As the basic Table Plotter functionalities apply to the single layers of Over Plotter as well, they will not be repeated here. Please refer to the applicable Table Plotter sections instead. This section will focus on all the functionalities that are specific to Over Plotter.

2.3.8.1. Invoke Over Plotter

A table dataset can be opened also in Over Plotter. Right clicking of the table dataset within a product in the product viewer and selecting "Open With" leads to a choice of viewers and tools that can be applied (see Figure 2.16). To bring a table dataset into Over Plotter, just choose the respective option. Note that at any time there can exist only one instance of Over Plotter in a session, while Table Plotter can exist in many instances. In other words, selecting the option Table Plotter will always create a new view in HIPE, while selecting Over Plotter will create a new view for Over Plotter only once and after that send any further dataset to the same Over Plotter view as new layer.

2.3.8.2. Layout of Over Plotter

The Over Plotter main view looks very similar to the Table Plotter, but also shows a few important differences.(seeFigure 2.23). The main differences are the "Layer Controls" panel, which replaces the "Overlay Plots" panel, and the addition of four synchronization buttons. The plot area now contains obviously more graphs and a second pair of axes to the top and right sides.



x-axis: TIME 1 y-axis: SS_C1 13

x0=1551353229:2007-02-28 11:26:36(UTC)

498.2

Layer: All
Color: black
Remove: None

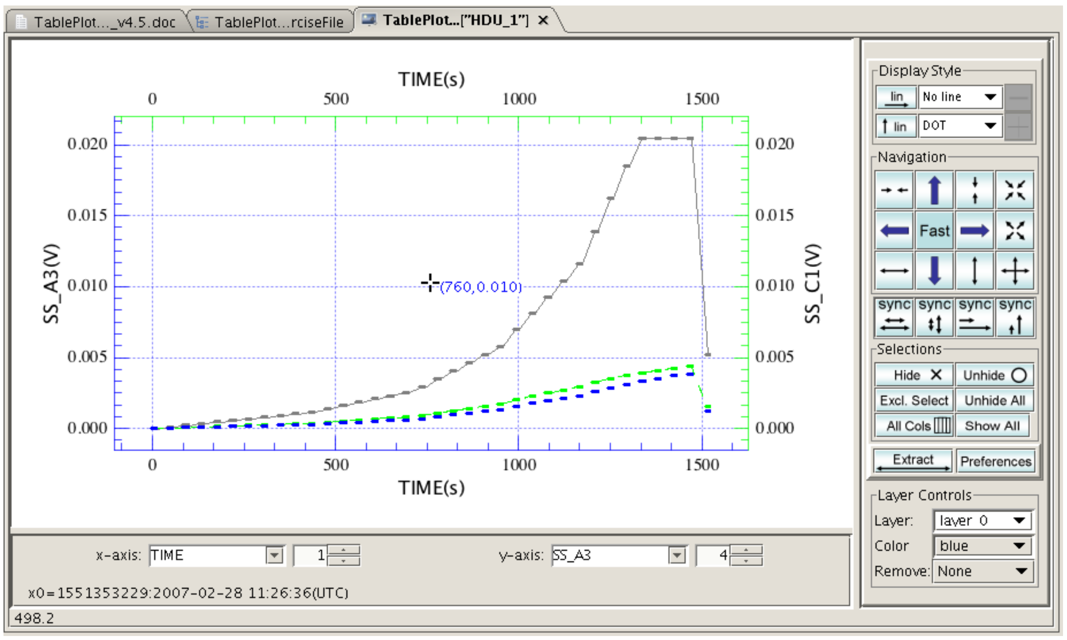
Figure 2.23. The main panel of Over Plotter is very similar to that of the Table PLotter. New features include the Layer Controls panel and the synchronization buttons. This Over Plotter is in "All Layers" mode.

The Over Plotter works in two main modes that can be chosen through the selection of layers: 1) a "Single Layer" mode and 2) an "All Layer" mode. The "Layer" drop down menu shows all the available layers, i.e. all the table datasets that have been sent to the Over Plotter so far. In addition, it contains an "All" entry. If selected the Over Plotter is switched to "All Layers" mode.

Please note that the same dataset can be sent to Over Plotter more than once. This makes sense as one may want to overlay diagrams of different pairs of columns of the same table dataset. A limitation of the Over Plotter is that a pair of columns of two different datasets can not be combined into one diagram,as the equal number of rows of both datasets is not guaranteed. However, columns of two different datasets can easily be combined on the command line into two one table and then plotted into one diagram, provided the tables have the same length. For instance, if tbl1 and tbl2 were two related table datasets of equal length and we wanted to plot the column RA from one dataset against the column DEC from the other dataset, then we would execute 3 simple command lines like the following and then display the newly created table dataset in Table-Plotter.

```
# tbl1 and tbl2 are table datasets
tbl = TableDataset()            #create new empty table dataset
tbl['RA'] = tbl1['RA']          #add column RA
tbl['DEC'] = tbl2['DEC']        #add column DEC
#now open tbl in Table- or Over-Plotter.
```

In Figure 2.23 the Over Plotter is in "All Layers" mode and the graphs are shown in their selected colors. Only for two graphs the axes can be shown. These are called the primary and the secondary layers. The axes of the primary layer are the ones on the bottom (X-axis) and to the left (Y-axis), while the axes of the secondary layer are the ones on the top (X-axis) and to the right (Y-axis). The axes are shown in the color of the respective layers.



x-axis: TIME 1 y-axis: SS_A3 4

x0=1551353229:2007-02-28 11:26:36(UTC)

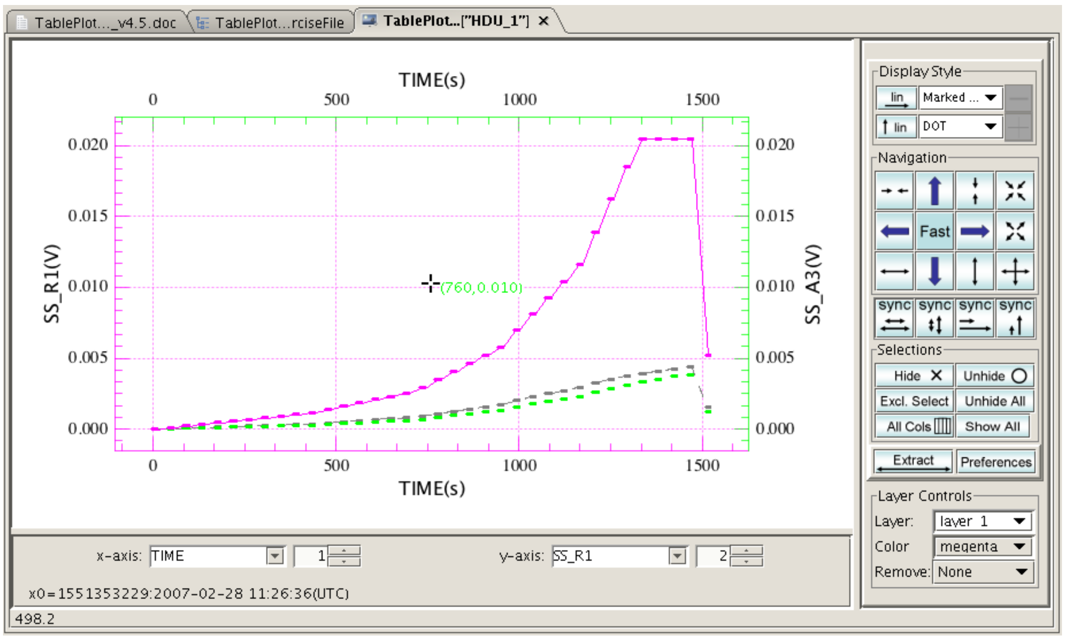
498.2

Layer: layer 0
Color blue
Remove: None

Figure 2.24. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected color and the secondary layer is displayed in green. All other layers are displayed in grey color.

In Figure 2.24 the Over Plotter is in "Single Layer" mode. In this case only the primary layer is shown in its selected color. The secondary layer is always green and all other layers are all displayed in gray.

The assignment of primary and secondary layer is dynamic and changes when another layer is selected. Then the layer that was prime before becomes the secondary layer and will be displayed in green. The previously secondary layer changes to grey color, unless it has been selected to be prime again, and the new prime layer is shown in its selected color. An example is shown in Figure 2.24, where the third layer that was gray in the previous example is now chosen to be prime, and the colors change accordingly.



x-axis: TIME 1 y-axis: SS_R1 2

x0=1551353229:2007-02-28 11:26:36(UTC)

498.2

Layer: layer 1

Color megenta

Remove: None

Figure 2.25. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected color and the secondary layer is displayed in green. All other layers are displayed in grey color. These are the same layers as in the previous figure, but after selecting Layer 1 to become prime.

2.3.8.3. Controls and Functions

Layer: layer 0 This drop down menu button shows the currently selected layer. If a single layer is selected, all actions apply to the selected layer only. Individual zooming, panning etc. is performed in this mode. ALL indicates that all layers are selected and actions are performed on all layers simultaneously. A number of buttons are not applicable in this mode and are grayed out.

Remove: None This drop down menu button allows to remove specific layers. This menu is available in any mode.

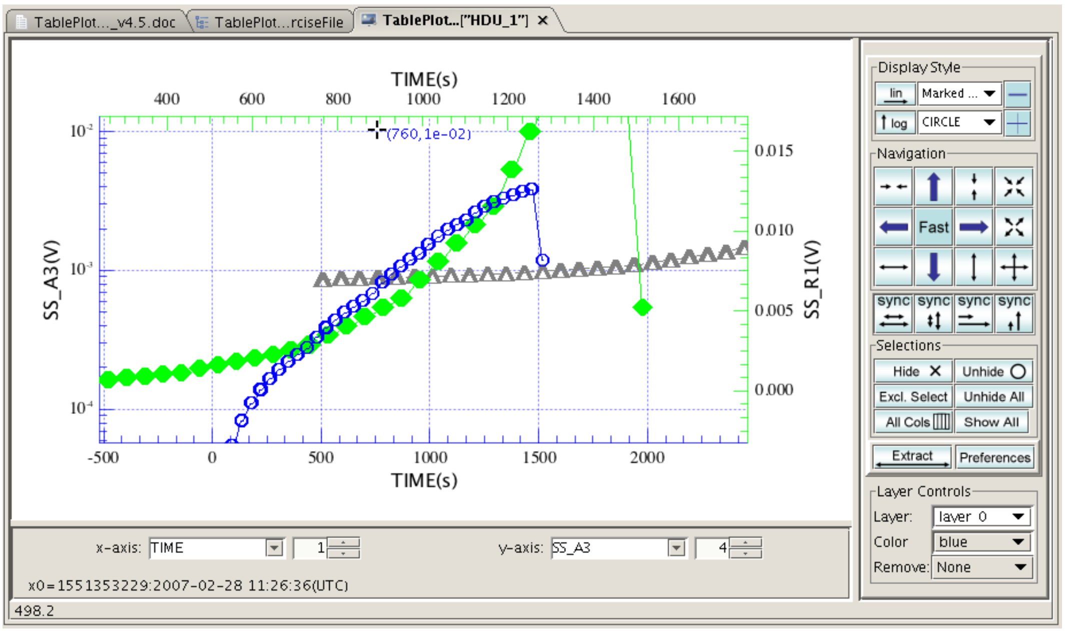
sync This button synchronizes the scale of the X-axis of the primary layer to the scale of that of the secondary layer, i.e. the distances between equal intervals on the X-axis display on the same scale.

sync This button synchronizes the scale of the Y-axis of the primary layer to the scale of that of the secondary layer, i.e. the distances between equal intervals on the Y-axis display on the same scale.

sync This button synchronizes the offset of the X-axis of the primary layer to the offset of the secondary layer, i.e. the primary layer is shifted in X-direction such that the values where the left Y-axis cuts the primary and secondary X-axes become the same.

This button synchronizes the offset of the Y-axis of the primary layer to the offset of the secondary layer, i.e. the primary layer is shifted in Y-direction such that the values where the bottom X-axis cuts the primary and secondary Y-axes become the same.

With all the possibilities of Table Plotter, except for the overlay function, available for each layer, many combinations are possible. In Figure 2.26 an overlay of 3 layers with different scaling and panning is shown. These are the same layers as in the previous plots, just with several display parameters changed to illustrate the possibilities. In addition the first layer (Layer 0) has a Y-log axis, and the blue circles are connected by solid lines. The second layer (Layer 1) has selected enlarged magenta filled diamonds, which are shown in green, because this is the secondary layer at this time and we are in single layer mode. The third layer (Layer 2) has selected blue enlarged triangles connected with a dashed line, which in this case is shown in gray color, because this layer is neither primary nor secondary layer right now.



Display Style: lin Marked ...; log CIRCLE

x-axis: TIME 1 y-axis: SS_A3 4

x0=1551353229:2007-02-28 11:26:36(UTC)

498.2

Layer: layer 0
Color: blue
Remove: None

Figure 2.26. A complex example for illustration. The Over Potter is in "Single Layer" mode. The primary layer is displayed in blue with large symbols and connected by a line. The Y-axis is set to logarithmic mode. The secondary layer is displayed in green with large filled diamonds. The third layer is displayed in grey color.

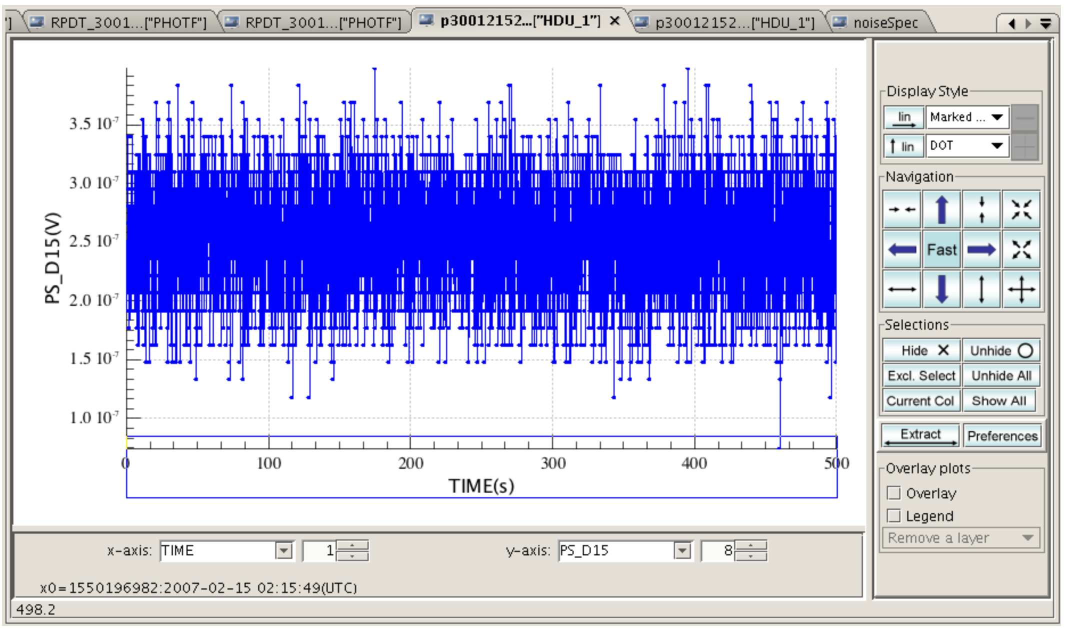
Due to the many logical combinations that are possible, mastering the Over Plotter can be a challenge at times, especially when it comes to synchronizations of plots. Some serious training with the tool is recommended. It should also be mentioned that at the time of writing (HIPE V1.1) there are still known issues with overplots involving log scales, or log/lin overplots, that will have to be fixed in the future.

2.3.9. The Power Spectrum Generator

The Power Spectrum Generator computes a power spectrum for each column of a Table Dataset. You can access it by right-clicking on a Table dataset in the *Variables* window in HIPE and choosing Open With → Power Spectrum Generator.

This interface is a wrapper around a command-line tool described in the *Scripting and Data Mining* guide: Section 3.10.3. Please see that section for more details on the available options.

A time column must be selected in the main menu. The result is another table dataset, that can be displayed with the TablePlotter. An example of a signal timeline is shown in (Figure 2.27, below).



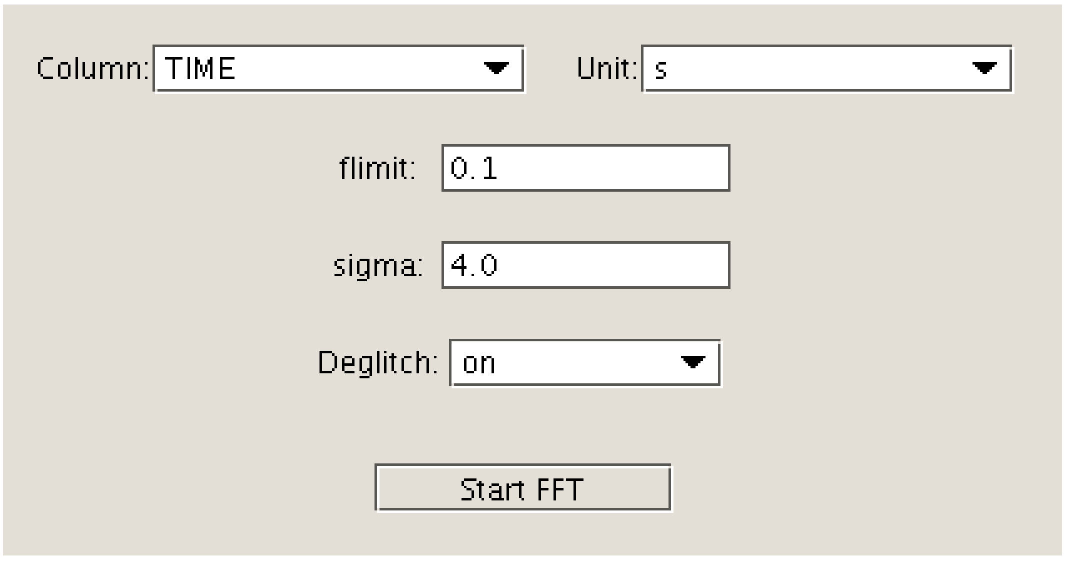
x-axis: TIME 1 y-axis: PS_D15 8

x0=1550196982:2007-02-15 02:15:49(UTC)

498.2

Figure 2.27. A signal timeline displayed in Table Plotter that the Power Spectrum generator can be applied to.

When the Power Spectrum Generator is invoked a menu appears. It consists of selectors for the time column in the dataset and its unit, in case that is not available or incorrect. There are two text boxes labelled flimit and sigma, controlling the deglitcher, which can be de-activated in another selector below. The button Start FFT initiates the processing, which results in a new table datatset (see Figure 2.28, below).


Column: TIME

Unit: s

flimit: 0.1

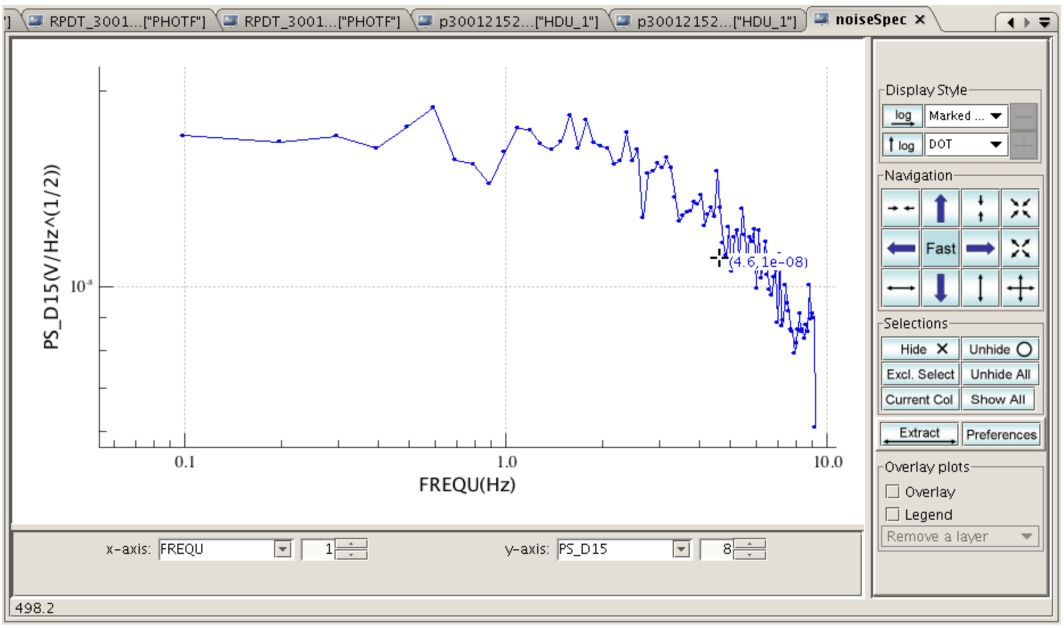
sigma: 4.0

Deglitch: on

Figure 2.28. Main view of the Power Spectrum Generator.

Two text boxes are pre-filled with default values for the cut off frequency (flimit) and the deglitcher threshold (sigma). Both flimit and sigma can be changed in the menu.

After clicking the Start FFT button, and a short processing time, a widget appears that allows naming of the newly created table dataset. After pressing the OK button, the dataset is fed back into the session and appears in the Variables view of HIPE. The TablePlotter can be used to display the dataset as shown in Figure 2.29.



x-axis: FREQU 1

y-axis: PS_D15 8

Figure 2.29. Displaying the newly created power spectra in the Table Plotter.

Chapter 3. Plotting

3.1. Summary

This chapter provides several examples that show you how to create and customize plots from the command line and from the HIPE graphical interface.

You can plot table datasets with the *TablePlotter* tool. For more information about TablePlotter see Section 2.3.7 in Chapter 2).

3.2. How to

The following examples show how to create and modify plots from the command line. Note that, with the exception of the first two steps, you can do everything via the plot properties window. To open the plot properties window, right-click on the plot and choose *Properties...* from the menu.

In order to illustrate the steps to produce simple plots we need input x and y variables. The plotting package 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:

```
x = Double1d.range(11) # Creates array with values from 0.0 to 10.0
y = x*x
```

1. Simple plot:

   ```
   plot = PlotXY()
   plot.autoBoxAxes = 1
   layer = LayerXY(x,y)
   plot.addLayer(layer)
   ```

2. Overplot a second x and y dataset:

   ```
   x1 = 10.0*Double1d.range(11)/10.0 -- 5.0
   y1 = x1**3.0
   ```

   Note that we do not need to repeat all plotting commands from the above example, we simply add a new layer:

   ```
   layer2 = LayerXY(x1,y1)
   plot.addLayer(layer2)
   ```

   And we note that the axis ranges are expanded correspondingly and that the new layer is with a different colour.

3. Change the plot title and subtitle

   ```
   plot.setTitleText("Example plot")
   plot.setSubtitleText("two layers")
   ```

   or if you don't want to have plot title and subtitle you can switch them off

   ```
   plot.title.setVisible(0)
   plot.subtitle.setVisible(0)
   ```

4. Change the axis labels:

   ```
   plot.xaxis.title.text = -"X-values"
   plot.yaxis.title.text = -"Y-values"
   ```

   the above commands are also equivalent to the following:

```
plot.xaxis.setTitleText("X-values")
plot.yaxis.setTitleText("Y-values")
```

or even the following:

```
plot.getXaxis().setTitleText("X-values")
plot.getYaxis().setTitleText("Y-values")
```

Which one of the variants to use is a matter of preference.

5. Change the axis ranges:

```
plot.xaxis.setRange([-2.0,2.0])
plot.yaxis.setRange([-10.0,10.0])
```

or go back to the auto range

```
plot.xaxis.setAutoRange(1)
plot.yaxis.setAutoRange(1)
```

6. Change the tick marks spacing and then the number of minor tick marks:

```
plot.xaxis.getTick().setInterval(3.0)
plot.yaxis.getTick().setInterval(30.0)
```

and to have 5 minor tick intervals between the major tick marks (which means 4 minor ticks):

```
plot.xaxis.getTick().setMinorNumber(4)
plot.yaxis.getTick().setMinorNumber(4)
```

7. Draw grid lines

```
plot.xaxis.getTick().setGridLines(1)
plot.yaxis.getTick().setGridLines(1)
```

Note that the grid lines are drawn at the major tick marks.

8. Change the axis from linear to log

```
plot.xaxis.setType(Axis.LOG)
plot.xaxis.setType(Axis.LINEAR)
```

Warning

The axis ranges need to be positive otherwise values are ignored in the LOG plot. When returning the plot back to LINEAR, all points are made plotted again even if some had been dropped in the LOG plot.

9. Change the line style for a given layer

```
layer.setLine(Style.NONE)
```

The line styles for setLine() can be

- Style.NONE - symbols only
- Style.MARKED - symbols connected with lines
- Style.SOLID - solid line, no symbols
- Style.DASHED - dashed lines
- Style.MARK_DASHED - symbols connected with dashed lines

Note that in the MARKED styles the default plotting symbol is used

10. Change the plotting symbol and its size. In order to have an effect you need to change the line style first to be one of NONE or MARKED styles

```
layer.setLine(Style.NONE)
layer.setSymbol(Style.FSQUARE)
layer.setSymbolSize(10)
```

The symbols can be:

Table 3.1. Symbols codes

DOT = 1	a dot	VCROSS = 2	a "+" sign
DCROSS = 3	an "x" sign	VDCROSS = 4	a "+" + "x" sign
CIRCLE = 5	an empty circle	TRIANGLE = 6	an empty triangle
UTRIANGLE = 7	an empty upside-down triangle	SQUARE = 8	an empty square
SQUARE_CROSS=9	an empty square + "x"	DIAMOND = 10	an empty diamond
DIAMOND_CROSS=11	a diamond + "+"	OCTAGON=12	an empty octagon
STAR = 13	an empty star	FCIRCLE=14	a filled circle
FTRIANGLE=15	a filled triangle	FSQUARE = 16	a filled square
FDIAMOND=17	a filled diamond	FOCTAGON=18	a filled octagon
UARROW = 19	an up arrow	DARROW = 20	a down arrow
RARROW=21	a right arrow	LARROW = 22	a left arrow
DARROW_LARGE=23	a large down arrow	UARROW_TRIANGLE = 24	a large up triangular arrow
DARROW_TRIANGLE = 25	a large down triangular arrow		

Note

You can use either the code or the numeric value for the symbol, that is, setSymbol(Style.FSQUARE) is equivalent to setSymbol(16).

11. Change the colour of the symbols and lines for a given layer

```
layer.setColor(java.awt.Color.RED)
```

12. Show or remove the legend for the layers

```
plot.getLegend().setVisible(1)
```

and we can also remove it

```
plot.getLegend().setVisible(0)
```

13. We can also change the legend name for a given layer

```
layer.setName("Test 1")
```

and we can also remove the legend for a particular layer if we don't want it to appear on the plot

```
layer.setInLegend(0)
```

14. Histogram mode. You need to be in MARKED or SOLID line style for this mode to work:

```
layer.setLine(Style.MARKED)
layer.style.setChartType(Style.HISTOGRAM)
```

The chart type can be HISTOGRAM - the data point is in the middle of the histogram horizontal bar, HISTOGRAM_EDGE - the data point is on the edge of the histogram horizontal, LINECHART - the data points are connected with lines.

15. Add error bars to x and/or y values. First we need to create arrays with errors

```
xerr = SQRT(x)
yerr = SQRT(y)

layer.setErrorX(xerr,xerr)
layer.setErrorY(yerr,yerr)
```

Note that the upper (the first argument to setError() method) and the lower (the second argument to setError() method) error limits can be different.

16. Add an annotation

```
plot.addAnnotation(Annotation(6.5,-10,"Test",color=java.awt.Color.GREEN))
```

here an annotation (the text "Test") will be put in the plot at position (6.5, -10).

17. You can use math and special symbols for text labels in your plot. It is possible to use TeX-like formatting of strings. In particular, entering math mode using a `$` symbol it is possible to insert Greek characters, e.g. using `\\alpha` or `\\beta`. Superscripts are preceded by the `^` symbol and subscripts by the _ symbol. For example the following can be used to set the title of the *x* axis:

```
plot.xaxis.title.text="$A_{1.3}^{b-3/2}$"
plot.xaxis.title.text="$\\alpha_{1.3}^{\\beta-3/2}$"
```

Note that it is necessary to use "\\" to escape the "\" symbol *from the command line*. A single backslash should be used in the *Property Panel* window instead.

Warning

Not all special symbols are available. If the symbol is not available it will be treated as normal text by the interpreter. For example, `$\\Alpha$` will be rendered as *\Alpha*.

The available special symbols are the following:

- All the lower-case Greek letters.
- The following upper-case Greek letters: `\Gamma, \Delta, \Theta, \Lambda, \Xi, \Pi, \Sigma, \Upsilon, \Phi, \Psi, \Omega`.
- The `\angstrom` and `\micro` symbols.

To insert other symbols you can use the Unicode escape sequence `\uxxxx`, where `xxxx` is the hexadecimal code of the symbol. For example, `\u2299` corresponds to the *circle dot operator*, which can also be used as symbol for the Sun.

For a list of Unicode sequences see for example http://www.utf8-chartable.de/.

18. Change the plot window size. You can resize the window with the mouse or you can specify the desired window size once you have added layers to the plot

```
plot.setWidth(400)
plot.setHeight(300)
```

19. Save the plot to file:

```
plot.saveAsJPG("myfile.jpg") # JPEG format
plot.saveAsEPS("myfile.eps") # Encapsulated PS
plot.saveAsPNG("myfile.png") # PNG format
```

20. Printing of a plot. In the menu which pops up when you click with the right-hand side mouse button you have "Print..." menu which allows you to send the plot directly to a printer (if you have configured one for your system).

21. Saving the plot. In the menu which pops up when you click with the right-hand side mouse button you have "Save as..." menu which allows you to save the plot in different image formats: Encapsulated PostScript file (EPS), JPG or PNG files.

22. Multiple plots per window.

    When we add layers to the plot we can specify their position on a grid as in the example below which places 4 layers onto a 2x2 grid (running indeces from 0,0 to 1,1).

```
plot = PlotXY()
layer = LayerXY(x,y)
layer1 = LayerXY(x1,y1)
layer1x = LayerXY(x1,y1/5.0)
layer1y = LayerXY(x1/5.0,y1)
plot.addLayer(layer,0,0) # top left
plot.addLayer(layer,0,1) # top right
plot.addLayer(layer,1,0) # bottom left
plot.addLayer(layer,1,1) # bottom right
```

    Now, if we open the plot properties GUI we have all four layers and we can change each one of them if necessary. We can interact with each layer and change its properties following the command line methods too.

23. Create a plot in batch mode.

    This is useful when you have many layers to add to the plot and you want to avoid to have the plot window redrawn and reajusted each time a new layer is added. From the above example:

```
plot = PlotXY()
plot.setBatch(1)
layer = LayerXY(x,y)
layer1 = LayerXY(x1,y1)
layer1x = LayerXY(x1,y1/5.0)
layer1y = LayerXY(x1/5.0,y1)
plot.addLayer(layer,0,0)
plot.addLayer(layer,0,1)
plot.addLayer(layer,1,0)
plot.addLayer(layer,1,1)
plot.setBatch(0)
```

24. Make the aux axis (top X or right Y) in different units.

    This is illustrated with one example where the main x-axis is Wavelength in µm and the top x-axis is changed to show the wavenumber (=1/wavelength) in cm^{-1}. The logic is a bit complicated so the exmaple is split into steps.

    a. Create a standard plot.

```
x = 100.0 + 6*Double1d.range(100)
y = x*x
plt = PlotXY()
l1 = LayerXY(x,y)
plt.addLayer(l1)
plt.xaxis.setTitleText("Wavelength ($\\mu$m)$")
plt.yaxis.setTitleText("F$_\\lambda$ (Jy)")
```

b. Get the aux axis (top X), make it free (i.e. ticks not identical to main axis), remove the autoadjustment of the ticks, make its title visible and set the new title.

```
xaux = plt.xaxis.getAuxAxis(0)
xaux.setTickIdentical(0)
xaux.getTick().setAutoAdjustNumber(0)
xaux.getTitle().setVisible(1)
xaux.setTitleText("Wavenumber (cm$^{-1}$)")
```

c. Get the top axis labels, make them visible, set the new values and set the new labels. In addition we add some minor ticks too.

```
xauxlab = xaux.getTick().getLabel()
xauxlab.setVisible(1)
vals = Float1d([10.0,20.0,30.0,40.0,50]) # these are the wavenumbers we want
to show
valsMinor = Float1d([15.0,25.0,35.0,45.0]) # these are the minor ticks
vals = 1.0e4/vals # convert the wavenumbers to wavelength in microns
valsMinor = 1.04/valsMinor
xaux.getTick().setFixedValues(vals,valsMinor)
# string values to each label
svals = ["10.0","20.0","30.0","40.0","50.0"]
xauxlab.setFixedStrings(svals)
```

3.3. In depth

This section 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 Javadoc documentation for the herschel.ia.gui.plot package.

Four main classes are described in this section: 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).

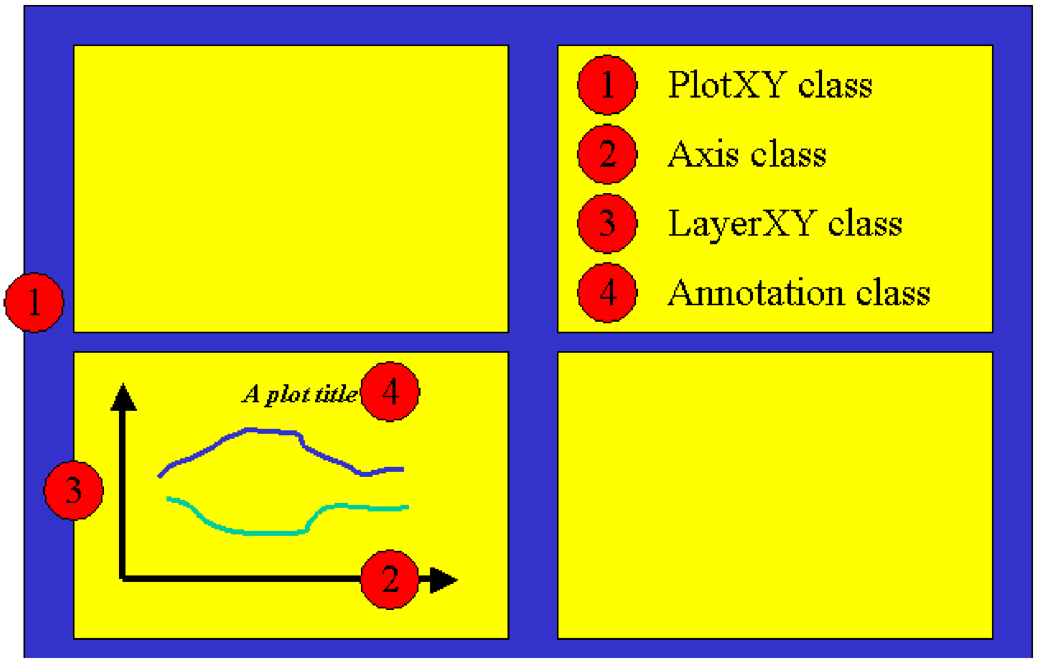


Figure 3.1. Classes involved in plot operations.

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.

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.

Note

PlotXY does *not* store the data values. This makes it more memory efficient but can lead to unexpected behaviour. For example, if you change the arrays "n" or "e" in the previous example, the plot will automatically update to the new values of "n" or "e".

```
n += 2   # adds 2 to every value in the array -"n"
```

If the above line is executed at the end of the sequence in the example then values along the plotted x-axis will be shifted by 2 and automatically updated in the plots displayed.

3.3.1. Properties

Plot properties allow the definition of items such as *colours*, *linetypes* etc. with your personal preferences.

You can open the properties window in two ways:

- Right-click on the plot window and choose *Properties...* from the menu.
- From the command line, if the plot correspond to variable p, use this command:

```
p.props()
```

The plot properties window (see Figure 3.2) 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:

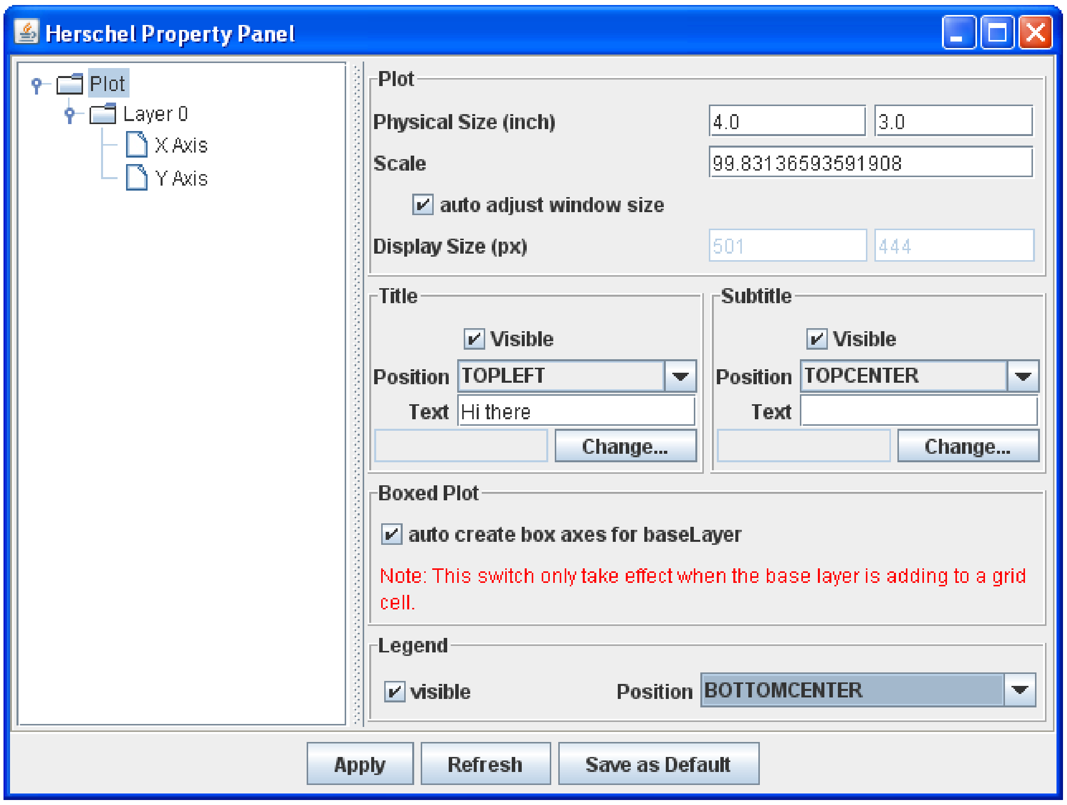
Apply	Applies any changes to the plot, without closing the properties window.
Refresh	Reads in the properties of the visible register card (plot, layer or axis). This button is useful if you have the plot property GUI visible and change properties from the command line. Refresh updates the GUI afterwards.
Save as default	Saves the properties as default. 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.

3.3.1.1. Plot properties

The plot properties available for a "PlotXY" object are shown in Figure 3.2. 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.


Herschel Property Panel

Plot
Layer 0
X Axis
Y Axis

Plot
Physical Size (inch) 4.0 3.0
Scale 99.83136593591908
☑ auto adjust window size
Display Size (px) 501 444

Title
☑ Visible
Position TOPLEFT
Text Hi there

Subtitle
☑ Visible
Position TOPCENTER
Text

Boxed Plot
☑ auto create box axes for baseLayer
Note: This switch only take effect when the base layer is adding to a grid cell.

Legend
☑ visible Position BOTTOMCENTER

Figure 3.2. The `Plot` section of the `PlotXY` properties dialog.

3.3.1.2. 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 3.3).

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.

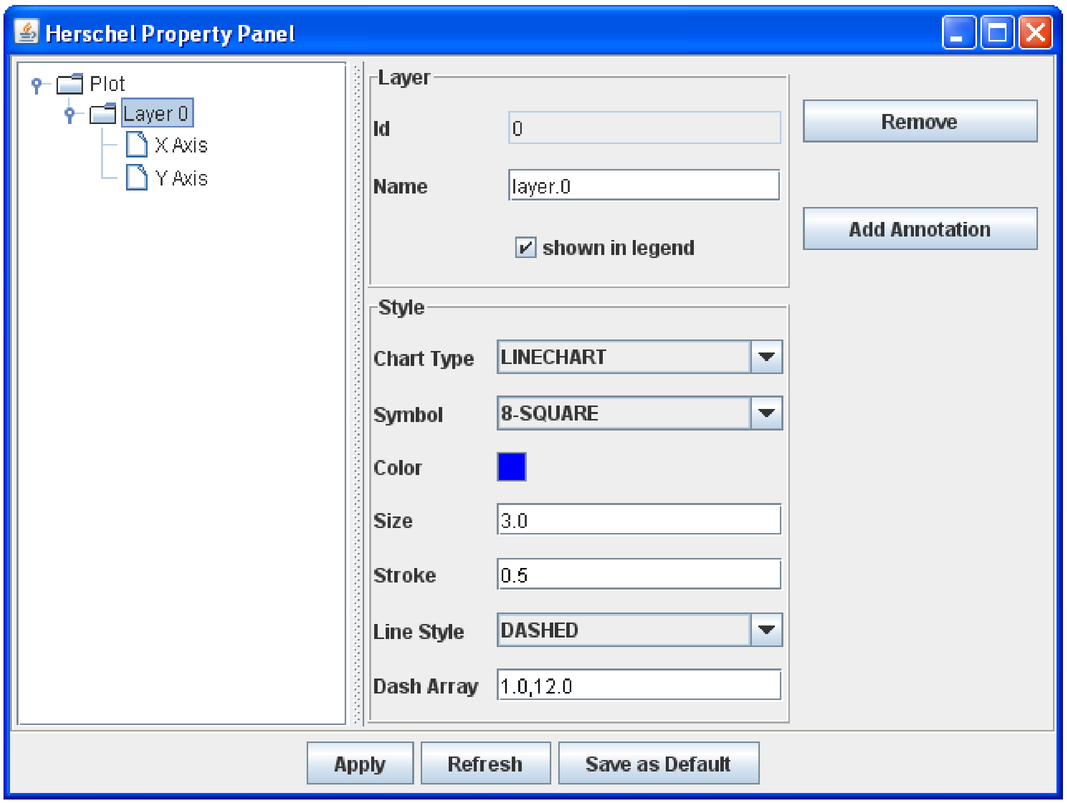
Chart Type	The pulldown allows for either a LINECHART or a HISTOGRAM plot.
Symbol	The symbol type to be used for points on a plot can be chosen from 25 possibilities in the pulldown menu. The symbol type number is also given (SQUARE = "8").
Color	The colour can be changed by clicking on the coloured 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 3.4).

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).



Layer

Id: 0

Name: layer.0

[✓] shown in legend

Style

Chart Type: LINECHART

Symbol: 8-SQUARE

Color: 🟦

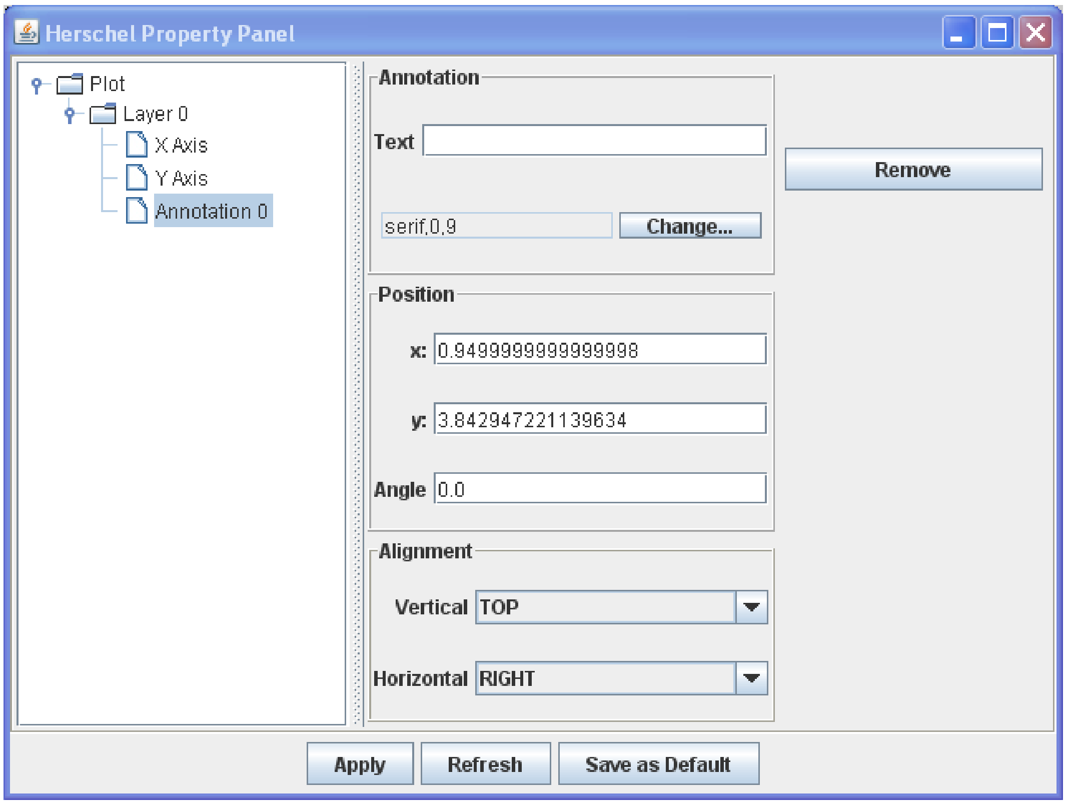
Size: 3.0

Stroke: 0.5

Line Style: DASHED

Dash Array: 1.0,12.0

Figure 3.3. The `Layer` section of the `PlotXY` properties dialog.


Herschel Property Panel

Plot / Layer 0 / X Axis / Y Axis / Annotation 0

Annotation
Text:
serif,0,9 [Change...]

Position
x: 0.9499999999999998
y: 3.842947221139634
Angle: 0.0

Alignment
Vertical: TOP
Horizontal: RIGHT

Figure 3.4. Dialog for adding an annotation to a `Layer`.

3.3.1.3. Axis properties

The `Axis` properties dialog (see Figure 3.5) 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 3.5). 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.

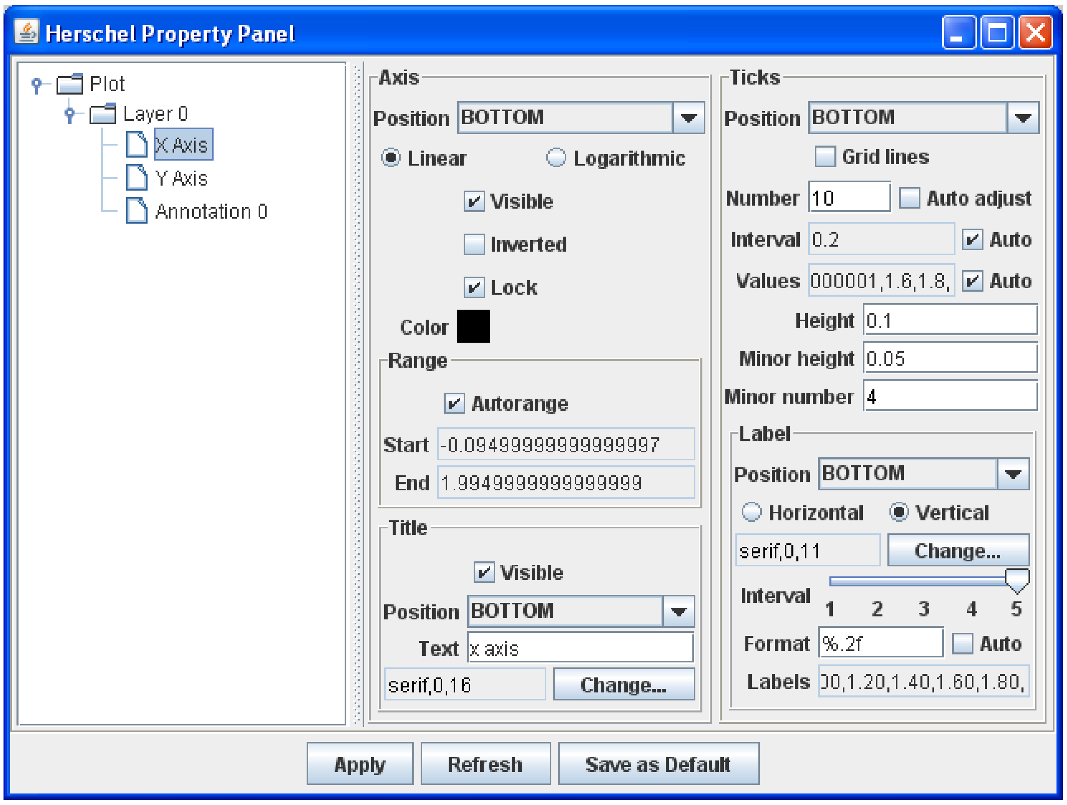
  The range can be set or left to be generated automatically.

  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.



Axis
Position BOTTOM
(● Linear) ○ Logarithmic
✓ Visible
☐ Inverted
✓ Lock
Color ■
Range
✓ Autorange
Start -0.09499999999999997
End 1.9949999999999999
Title
✓ Visible
Position BOTTOM
Text x axis
serif,0,16

Ticks
Position BOTTOM
☐ Grid lines
Number 10 ☐ Auto adjust
Interval 0.2 ✓ Auto
Values 000001,1.6,1.8, ✓ Auto
Height 0.1
Minor height 0.05
Minor number 4
Label
Position BOTTOM
○ Horizontal (● Vertical)
serif,0,11
Interval 1 2 3 4 5
Format %.2f ☐ Auto
Labels 00,1.20,1.40,1.60,1.80,

Figure 3.5. The `Axis` section of the `PlotXY` properties dialog.

3.3.1.4. How to use properties

The result of a property setup procedure (with a defined set of properties) is given in Example 3.1. This can be used to set up properties from the command line window of HIPE or for generating plots from within scripts.

```
n = Double1d.range(20) -/ 10.
e = EXP(n)
p = PlotXY(n, e)
p.props()  # (1)
p[0] = None  # (2)
p[0] = LayerXY(n, n*n, name="anotherLayer")  # (3)
p[0].style.stroke = 5  # (4)
p[1] = LayerXY(n, 2*n*n, name="yetAnotherLayer")  # (5)
p[1].style.stroke = 7  # (6)
```

1. this command allows graphical interface property setup, it fires the Plot Property GUI.
2. removes the first (and only) layer of the plot. Press the Refresh button in the Properties window to see the change
3. 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.
4. sets the line stroke for overlay plot `anotherLayer`
5. adds yet another layer to the plot "`p`"...
6. ...and changes the line stroke on this plot too!

Example 3.1. Command line control of properties

The result of running above example is shown below.

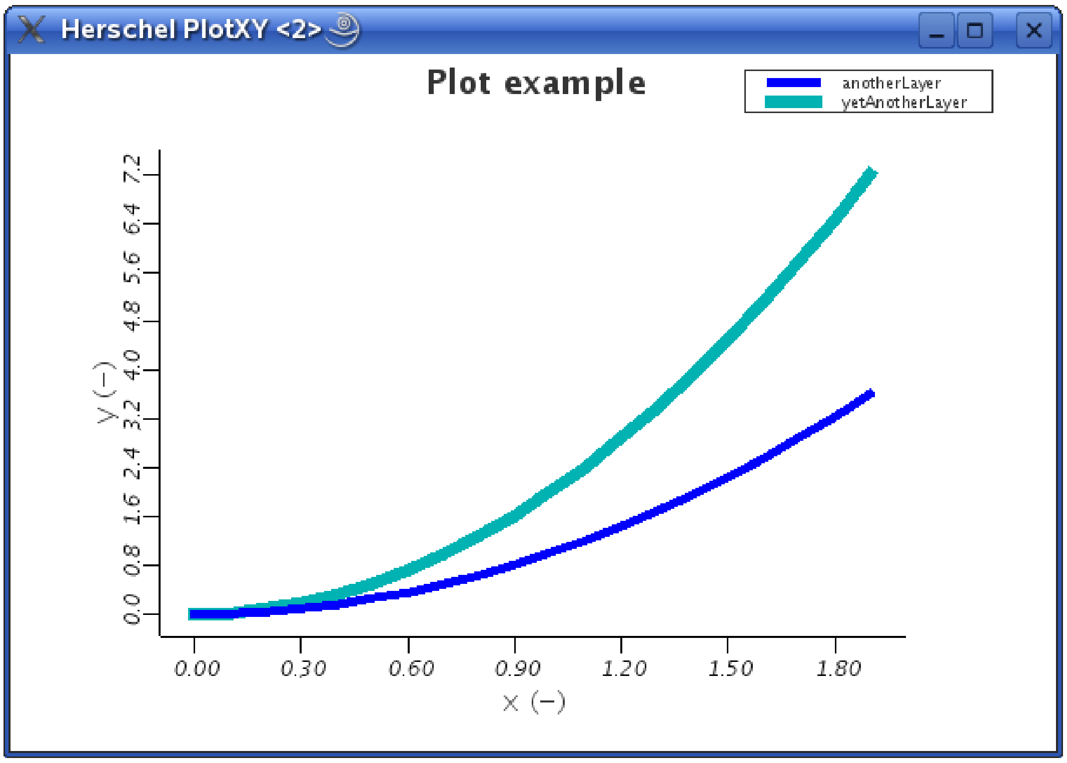


Figure 3.6. This plot is the result of Example 3.1.

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.

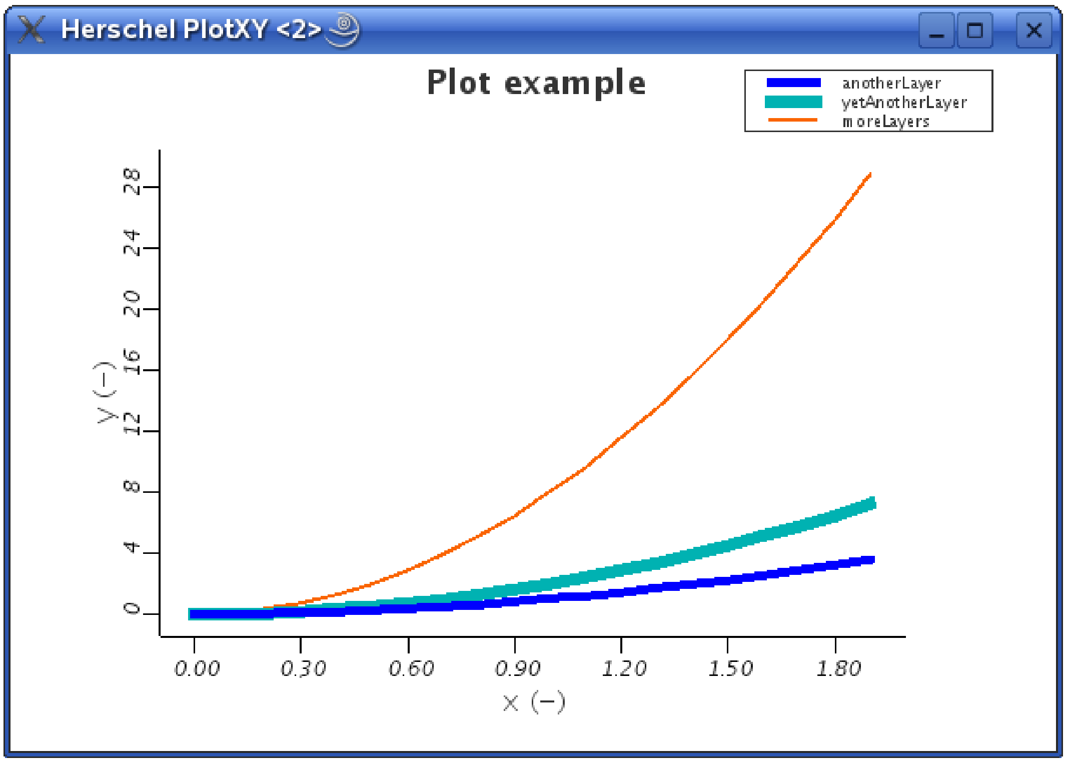


Figure 3.7. Adding in another layer gives the orange curve (see text).

3.3.1.5. Resizing a plot

The `width` and `height` properties are available to set the size of a plot in pixels. However, using these properties on their own could cause unwanted side effects, like in the following example:

```
x = Int1d([0, 1, 2, 3]) # Setting up sample data
y = x
plot = PlotXY(width = 600, height = 400)
layer = LayerXY(x, y)
plot.addLayer(layer)
```

Adding the layer causes the plot window to grow to a very large size. This can be avoided by setting the `autoAdjustWindowSize` property to `0`:

```
x = Int1d([0, 1, 2, 3]) # Setting up sample data
y = x
plot = PlotXY(width=600, height=400, autoAdjustWindowSize=0)
layer = LayerXY(x, y)
plot.addLayer(layer)
```

Adding the layer in this case does not cause problems.

Another solution is to set the plot size *after* all the layers have been added, using the `setSize` method:

```
x = Int1d([0, 1, 2, 3]) # Setting up sample data
y = x
plot = PlotXY()
layer = LayerXY(x, y)
plot.addLayer(layer)
plot.setSize(600, 400)
```

3.3.2. Plot 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 3.2 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()  # (1)
rn = Double1d(20).apply(r) -- 0.5 # (2)

n = Double1d.range(20)/10
e = EXP(n)  # (3)
en = e+rn  # (4)

p = PlotXY(layers=[LayerXY(n, e, name="e", color=java.awt.Color.red)], \
 titleText="Exponential plot")  # (5)
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")  # (6)

layer_en = p.getLayerXY(1)  # (7)
layer_en.setLine(Style.NONE)
layer_en.setSymbol(Style.FCIRCLE)
layer_en.setColor(java.awt.Color.red)

layer_en.setLine(1)  # (8)
```

1. Produces random numbers between 0 and 1.
2. generates a set of 20 random double (real) numbers between -0.5 and 0.5.
3. 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.
4. adds the random numbers to the array `e` i.e. add noise to the data.
5. 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.
6. Add the noise data to the plot as a layer with name `en`
7. 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".
8. reset the layer back to a line plot. Note how setting the line to "solid" (code 1) the symbols automatically

Example 3.2. 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 3.2. Methods for handling annotations. Note that these methods must be applied to a *plot* object, not to a *layer* object: for instance, `myPlot.clearAnnotations()`, not `myLayer.clearAnnotations()`.

`addAnnotation(Annotation annotation)`	Adds an `Annotation` object to the layer.
`addAnnotations(Annotation[] annotations)`	Adds several `Annotation` objects to the layer. The input Annotations are passed as an array.
`setAnnotation(int id, Annotation annotation)`	Sets an annotation to a given id, replacing what was there before.
`setAnnotations(Annotation[] annotations)`	Replaces all the annotations with the ones provided in the array.
`getAnnotation(int i)`	Retrieves one annotation from the layer.
`getAnnotations()`	Retrieves all the annotations from the layer. The annotations are returned as an array.
`removeAnnotation(int id)`	Removes the annotation with the specified id.
`clearAnnotations()`	Removes all the annotations.

Table 3.3. Methods for handling error bars in layers.

`appendErrorX(double low, double high)`	Appends a low and high error value of x.
`appendErrorX(Ordered1dData low, Ordered1dData high)`	Appends a set of low and high error values of x.
`setErrorX(Ordered1dData[] error)`	Sets low and high error values of x.
`setErrorX(Ordered1dData low, Ordered1dData high)`	Sets the low and high error values of x.
`getErrorX()`	Returns an array of `Ordered1dData` with length equal to 2.

Table 3.4. Axis-related methods of the `Layer` class. All can equally be applied to the y-axis by replacing "X" with "Y".

`setXaxis(Axis axis)`	Sets the x axis to the specified `Axis` instance.

	Note: the x axis will be reinstantiated with its default settings plus whatever is indicated in the `Axis` instance. So any prior manipulations of the axis are lost.
`setXrange(double[] `*`range`*`)`	Sets the range of the x axis. Get method available.
`setXtitle(String `*`title`*`)`	Sets the title of the x axis. Get method available.
`setXtype(Axis.Type `*`type`*`)`	Sets the type of the x axis based on the axis types available. LINEAR is type 0, LOG is type 1. Get method available.
`setXy(Ordered1dData[] `*`xy`*`)`	Sets the x and y values, passed as elements of an "array of arrays" of size two. Get method available. Note that there is no `setYx` method!
`setXy(Ordered1dData `*`x`*`, Ordered1dData `*`y`*`)`	Sets the x and y values, passed as two separate arrays. Note there is no `setYx` method!
`setY(Ordered1dData `*`y`*`)`	Sets the ordinate values. Get method available. Note there is a `getX` method but not a `setX` method.
`shareXaxis(Axis `*`axis`*`)`	Removes the x axis and uses the given axis as a shared one.

Table 3.5. Miscellaneous setters of the `Layer` class.

`setName(`*`text`*`)`	Changes the name (and thus the legend) of the layer. Get method available.
`setLine(`*`line code`*`)`	Changes the plot to a line plot for the specified layer. Get method available.
`setSymbol(`*`symbol code`*`)`	Changes the plot to a scatter plot for the specified layer. Get method available.
`setSymbolSize(int `*`size`*`)`	Sets the size of a the symbol. Get method available (note that it returns a `double` rather than an `int`.
`setSymbolShape(SymbolShape `*`shape`*`)`	Sets the shape of the symbol. The input parameter is an instance of the class `SymbolShape`. Get method available.
`setColor(`*`colour`*`)`	Sets the colour of the symbols and lines for the specified layer. Get method available.
`setStroke(`*`stroke`*`)`	Sets the stroke of the line for the specified layer (only for line plots). Get method available.
`setStyle(Style `*`style`*`)`	Sets the style of the layer. The input parameter is an instance of the `Style` class. Get method available.

Table 3.6. Other methods of the `Layer` class.

`addPoint(double `*`x`*`, double `*`y`*`)`	Adds a point to the layer.
`addPoint(Ordered1dData `*`x`*`, Ordered1dData `*`y`*`)`	Adds a set of points to the layer.
`getCoords()`	Waits for mouse click and returns the coordinates of the pointer. Returns a `double[]`.

`getCoords(int n)`	Like the previous method, but this one does the job for `n` successive clicks. Returns a `double[][]`
`getDataCoords()`	The difference with respect to the previous two methods is that this time the coordinates of the layer point closer to the mouse pointer are returned. Returns a `double[]`.
`getDataCoords(int n)`	Like the previous method, but this one does the job for `n` successive clicks. Returns a `double[][]`.
`getId()`	Returns an `int` representing the index of the current layer inside the `PlotXY`.
`setInLegend(boolean)`	`True` if the layer is shown in the legend.
`isInLegend()`	Returns `True` if the layer is shown in the legend.
`setNotifyWarningAsExceptional (boolean)`	`True` if exceptional values like NaN and infinity are notified as errors, `False` if they are only logged.
`isNotifyWarningAsExceptional()`	Returns `True` if exceptional values like NaN and infinity are notified as errors, `False` if they are only logged.

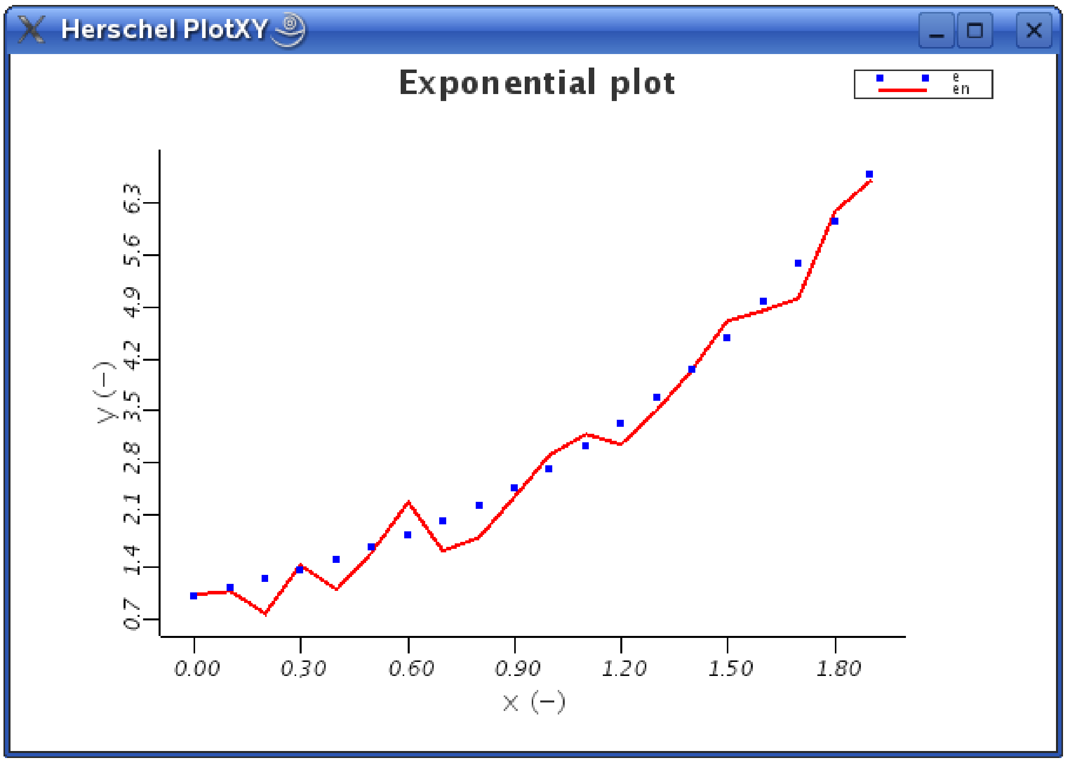


Figure 3.8. Plot showing the result of manipulation of layers from the command line.

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 Javadoc documentation for `LayerXY` lying next to you.

3.3.3. Plot axes

As with Layers most manipulations of both X and Y axes can be done through the `Axis` class.

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")
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 3.3. Axes, labels and grid lines

It is also possible to use TEX-like labelling for subscripts and superscripts. For example:

```
p.xaxis.title.text="$x_1^{2a}$"
```

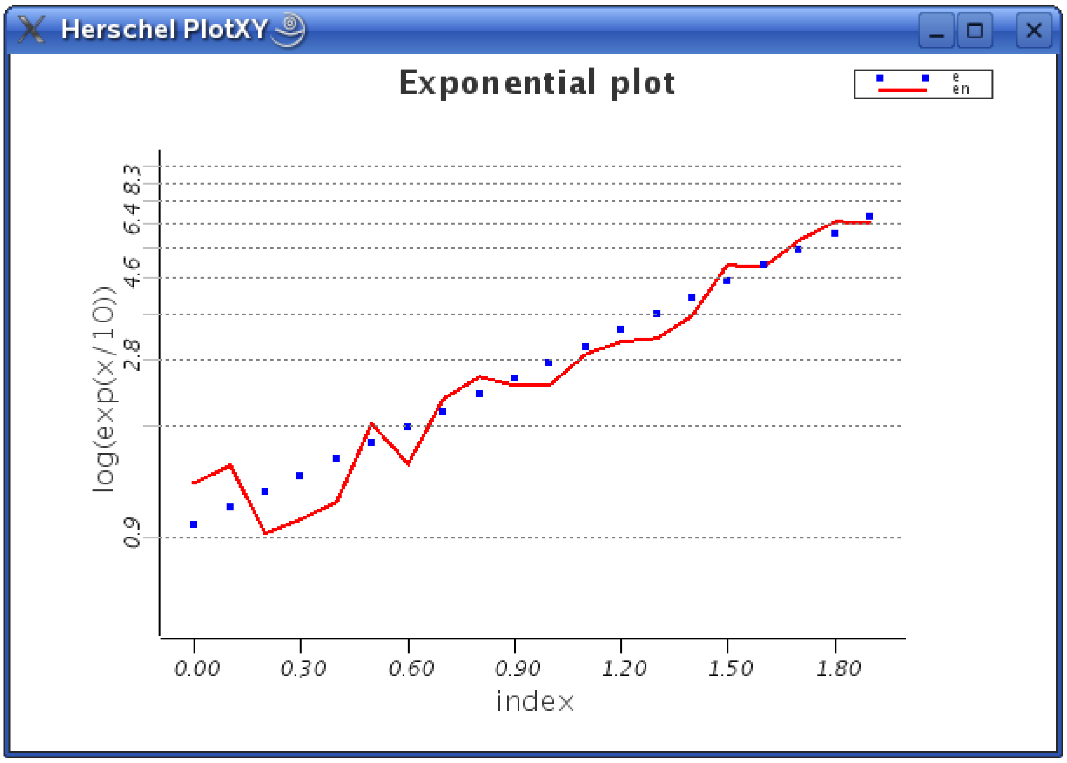


Figure 3.9. Changing Axes, labels and added grid lines.

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 = 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")
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.getLayerXY(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 3.4. 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 HIPE 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 3.10.

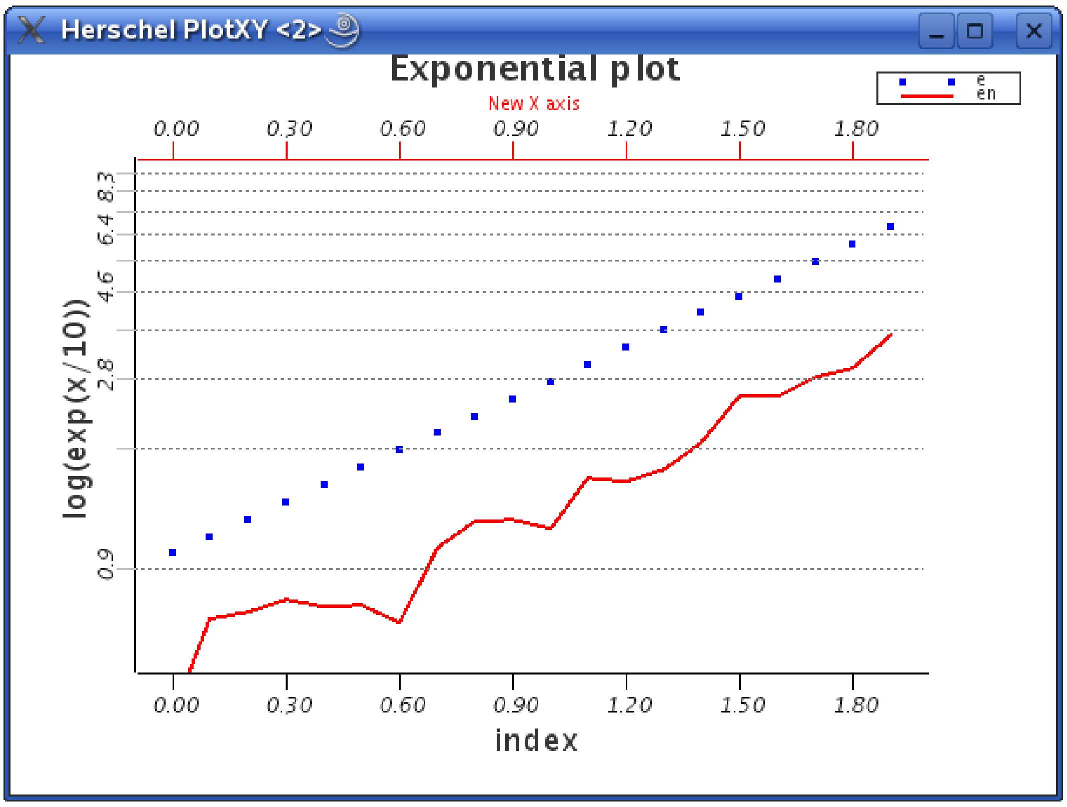


Figure 3.10. Example of a second X-axis label relevant to the red line plot.

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 Javadoc documentation for the `Axis` class.

Table 3.7. Useful ways of manipulating axes from the command line

`axis = layer.getXaxis()` or `getYaxis()`	Gets the X or Y Axis object to do direct manipulations on the corresponding axis
`setAutoRange(`*`flag`*`)`	If `flag` is true, adjusts the range of the specified axis so that all datapoints will be shown
`setRange([`*`lower, upper`*`])`	Set the range of the specified axis to values between lower and upper. Note that we no longer have two arguments for the lower and upper limits, but one array argument containing both values.
`setGridlines(`*`flag`*`)`	Show grid lines for the specified axis if flag is true, hide the grid lines if flag is false.

Table 3.8. Methods for handling labels on axes.

	Sets the colour of labels. Get method available.

`getTick().getLabel().setColor (java.awt.Color` *colour*`)`	
`getTick().getLabel().setFont (java.awt.Font` *font*`)`	Sets the font of labels. Get method available.
`getTick().getLabel().setFontSize (double` *size*`)`	Sets the physical size of labels. Get method available.
`getTick().getLabel().setInterval (int` *n*`)`	Sets the interval (in ticks) between successive labels. Get method available.
`getTick().getLabel(). setOrientation(int` *n*`)`	Sets the orientation of the labels (0 for horizontal, 1 for vertical). Get method available.
`getTick().getLabel().setStrings (String[]` *labels*`)`	Replaces the current labels with the values in an array of `String` objects. Get method available.
`getTick().getLabel().setPosition (AxisConstants.Position` *position*	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 3.9. Methods for handling ticks on axes.

`getTick().setColor(java.awt.Color` *colour*`)`	Sets the colour of ticks. Get method available.
`getTick().setHeight(double` *size*`)`	Sets the physical height of the major ticks. Get method available.
`getTick().setInterval(double` *interval*`)`	Sets the interval (in axis units) between ticks. Get method available.
`getTick().setPosition (AxisConstants.Position` *position*	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.
`getTick().setNumber(int` *ticks*`)`	Sets the number of major ticks displayed on the axis. Get method available.
`getTick().setMinorNumber(int` *minors*`)`	Sets the number of minor ticks displayed between two major ticks. Get method available.
`getTick().setValues(Double1d` *values*`)`	Sets the values where ticks are to be placed. Get method available.
`getTick().setAutoAdjustNumber (boolean)`	`True` if the number of ticks on the axis is set automatically.
`getTick().isAutoAdjustNumber()`	Returns `true` if the number of ticks on the axis is set automatically.
`getTick().setAutoValues(boolean)`	`True` if the positions of the ticks on the axis are chosen automatically.
`getTick().isAutoValues()`	Returns `true` if the positions of the ticks on the axis are chosen automatically.

Table 3.10. Miscellaneous setters/getters of the `Axis` class.

`setType(Axis.Type` *type*`)`	Sets whether the axis is linear (0) or logarithmic (1). You can also use `Axis.LINEAR` and `Axis.LOG` as input parameters. Get method available.
`setLinear()`	

	Sets the axis to a linear scale. Equivalent to `setType(Axis.LINEAR)`.
`setLog()`	Sets the axis to a logarithmic scale. Equivalent to `setType(Axis.LOG)`.
`setColor(java.awt.Color` *`colour`*`)`	Sets the colour of the axis. Get method available.
`setAutoRange(boolean` *`isAutoRange`*`)`	Sets whether the range is automatically determined. Get method `isAutoRange` available.
`getTick().setGridLines(boolean)`	Sets whether grid lines are displayed. Get method `isGridLines` available.
`setInverted(boolean)`	Sets whether values on the axis are displayed in inverted order (e.g. right to left for abscissa). Get method `isInverted` available.
`setPosition (AxisConstants.Position` *`position`*	Sets the position of the axis with respect to the plot. Possible values are `TOP` or `BOTTOM` for abscissa axis and `LEFT` or `RIGHT` for ordinate axis. Get method available.
`setRange(double[]` *`range`*`)`	Sets the range of the axis. The lower and upper limit are passed inside an array. Get method available.
`setRange(double` *`low`*`, double` *`high`*`)`	Sets the range of the axis. The lower and upper limit are passed as separate `double` parameters.
`getTitle().setPosition (AxisConstants.Position` *`position`*	Sets the position of the axis title with respect to the axis. Possible values are `TOP` or `BOTTOM` for abscissa axis and `LEFT` or `RIGHT` for ordinate axis. Get method available.
`setVisible(boolean` *`isVisible`*`)`	Sets whether the axis is visible. Get method `isVisible` available.

It is also possible to set the Axis in one go using GUI plot' `Axis` class. An example of this is:

```
x = Double1d.range(10)
y = x*x
plt = PlotXY()
plt[1] = LayerXY(x,y)
plt[1].xaxis = Axis(titleText="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(titleText="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"
```

3.3.4. Error bars

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(1) # show the legend
p.setTitleText("Error bar example plot") # give the plot a title
```

Example 3.5. Adding error bars to plots

The above example produces the plot shown in Figure 3.11.

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 3.12.

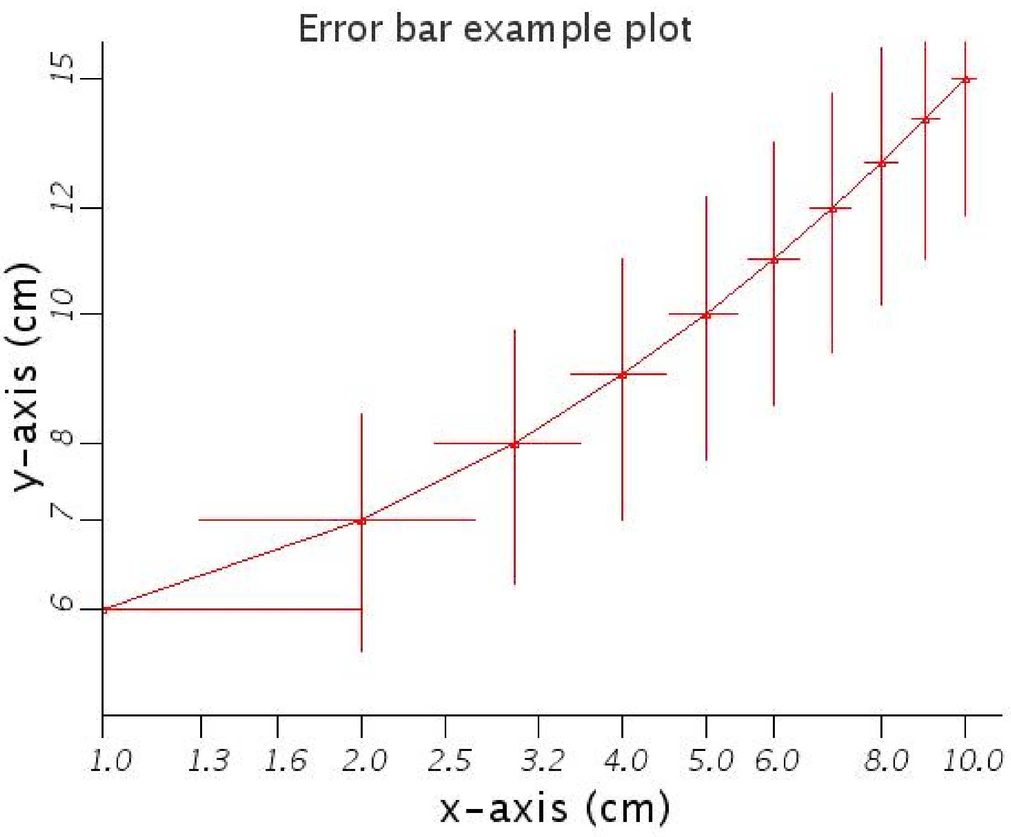


Figure 3.11. Setting errors in a plot

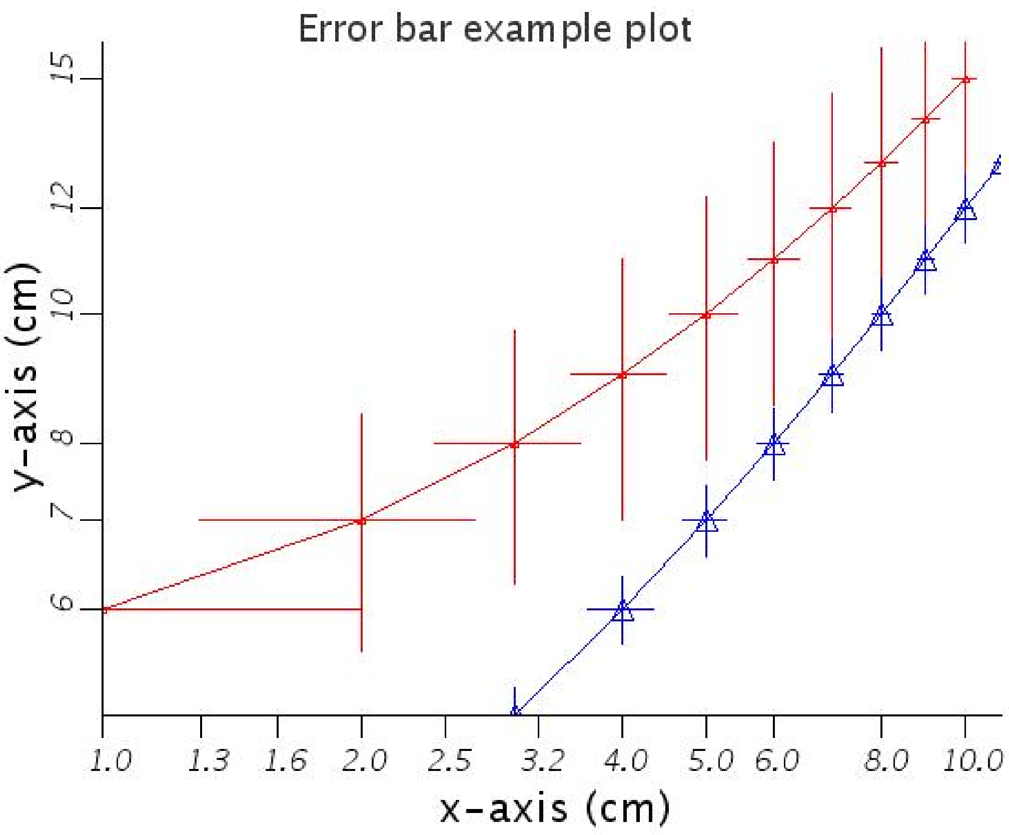


Figure 3.12. Applying errors to a specific layer of a plot

3.3.5. Decorating and saving plots

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 3.13). 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 = Double1d(20).apply(r) -- 0.5
n = Double1d.range(20)/10
e = EXP(n)  #
en = e+rn
p = PlotXY(n, e, name="e2", color=java.awt.Color.red, \
titleText = -"Exponential plot",subtitleText = -"a layered plot", \
xaxis=Axis(titleText="Index"), \
yaxis=Axis(titleText="log(exp(x/10))",type=Axis.LOG))
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.getLayerXY(1)
# Change the name (and the legend) for this layer to say what we want
layer.setName("exp+noise")
# Get the next layer we want to change
layer = p.getLayerXY(0)
# Change the name (and the legend) for this layer to say what we want
layer.setName("exp")
# Make sure the legend is visible
p.getLegend.setVisible(1)
# Set a new style
layer.setStyle(Style(line = Style.MARKED, symbol = Style.FTRIANGLE, \
 color = java.awt.Color.green, symbolSize=7))
# 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 3.6. Decorating and saving 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` layer we have changed the appearance of the datapoints to a line with triangles on top of it. Please refer to Section 3.3.2 for information on basic manipulation methods for layers.

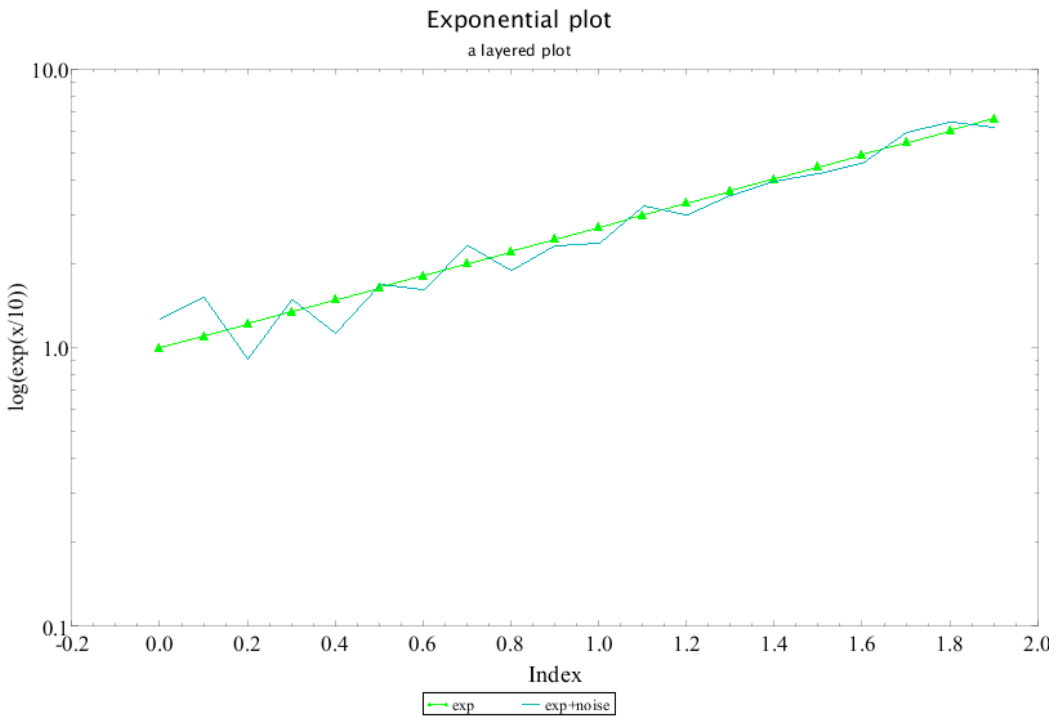


Figure 3.13. Final plot from "Decorating and saving plot" example.

3.3.6. Colours in plots

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. 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))
```

3.3.7. File output and printing 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", 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
  ```

3.3.7.1. 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.

3.3.8. 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 3.7.

```
# Create the data
data = Double1d.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 3.7. Multiple plotting

The above code produces the multiple plot window shown in Figure 3.14. 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)
```

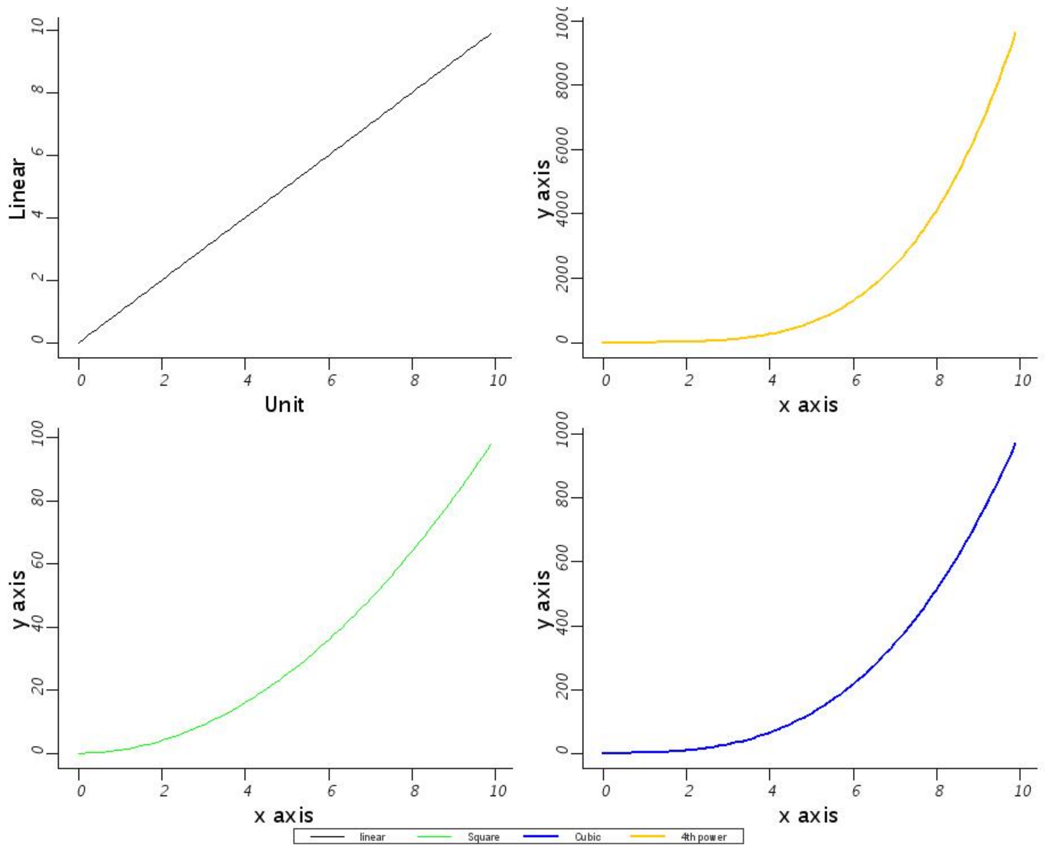


Figure 3.14. Example of multiple plots in a window from Example 3.7.

The properties of any one of the layers in the PlotXY window can be adjusted, e.g.,

```
p.props()
```

3.3.9. 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.

Chapter 4. Image analysis

4.1. Summary

This chapter describes the following tools for manipulating images:

- Basic tools for clipping/clamping, cropping, rotating, scaling, translating, transposing.
- Image arithmetics tools: adding, subtracting, multiplying, dividing, computing logarithms, exponentials and square roots, and so on.
- Aperture photometry with a circular target aperture and an annular or rectangular sky aperture.
- Histograms of the whole image or of a region bounded by a circle, an ellipse, a rectangle or a polygon.
- One-dimensional profile plotting.
- Contour plotting and overlays.
- Source extraction.
- Flagging saturated pixels.

All these tasks work on a `SimpleImage` that can be derived from a FITS file import, or even from an image file such as a JPEG.

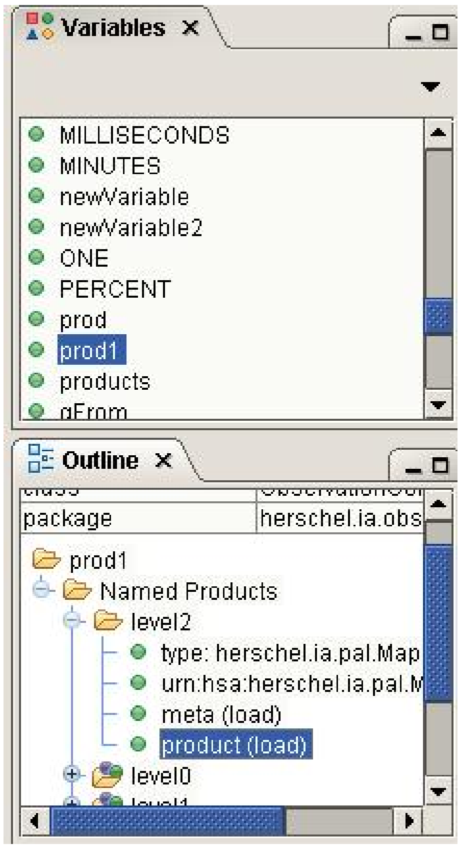
It should be noted that the overview and zoomed images displayed to the right of the displayed image during basic image analysis are the reverse for those when just displaying the image.

If you need a test image to test the utilities described in this chapter, see Section 2.3.2.

4.2. How to

4.2.1. Getting images from the Herschel Science Archive (HSA)

When downloading a product out of the science archive we access images from an `ObservationContext`. An `ObservationContext` contains all the information associated with a single observation and its processing (including all associated calibration files). A download from the HSA contains products made available from several levels of processing.

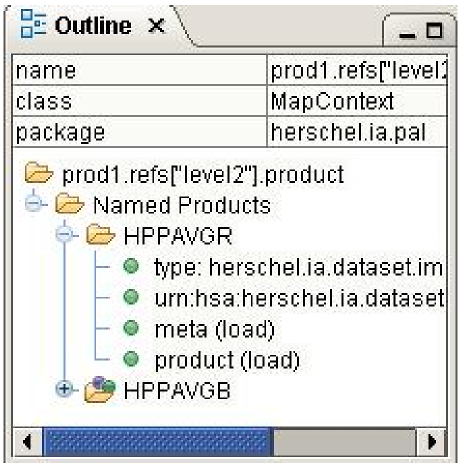


MILLISECONDS
MINUTES
newVariable
newVariable2
ONE
PERCENT
prod
prod1
products
qFrom

package	herschel.ia.obs

prod1
Named Products
level2
type: herschel.ia.pal.Map
urn:hsa:herschel.ia.pal.M
meta (load)
product (load)
level0
level1

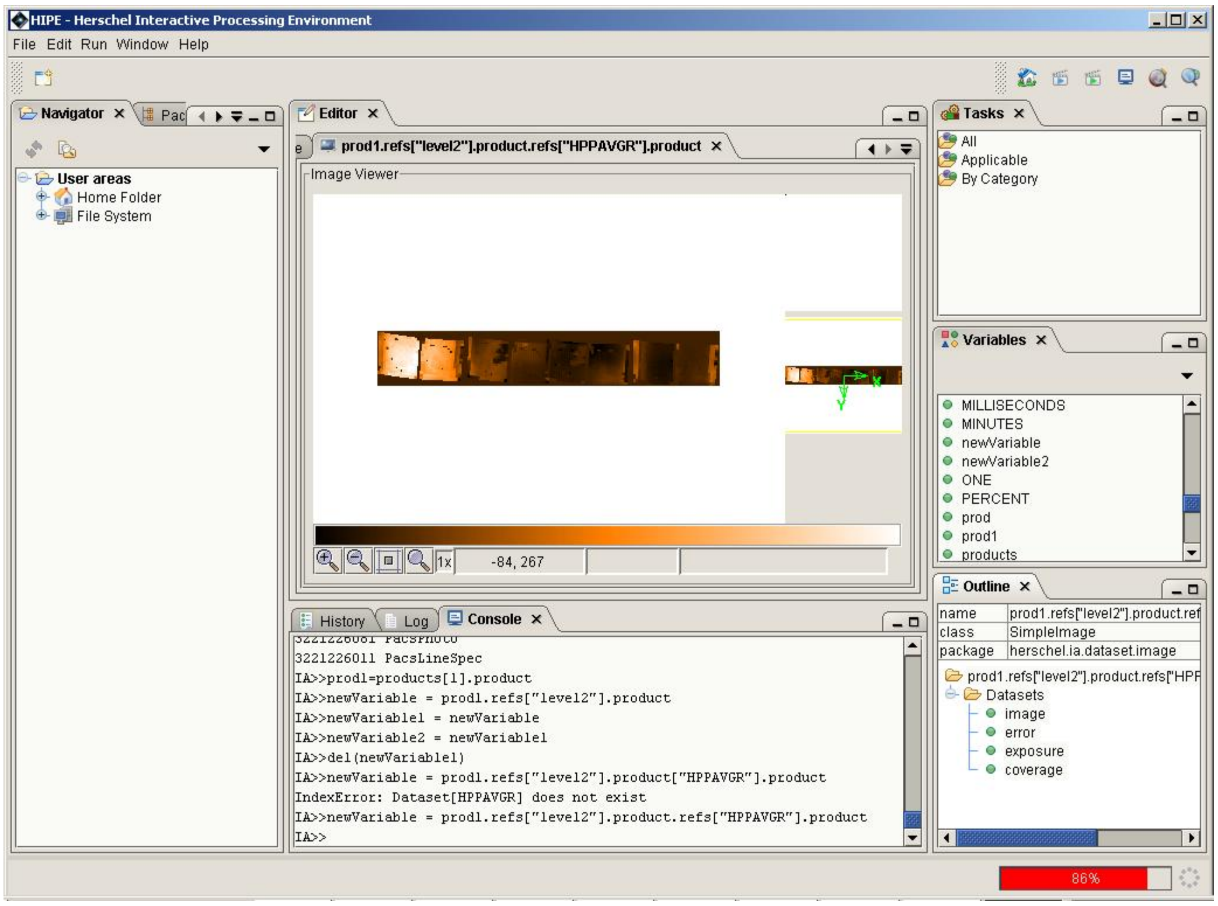
Figure 4.1. Contents of an `ObservationContext`



name	prod1.refs["level2
class	MapContext
package	herschel.ia.pal

prod1.refs["level2"].product
Named Products
HPPAVGR
type: herschel.ia.dataset.im
urn:hsa:herschel.ia.dataset
meta (load)
product (load)
HPPAVGB

Figure 4.2. PACS green channel image access



name	prod1.refs["level2"].product.ref
class	SimpleImage
package	herschel.ia.dataset.image

```
3221226081 PacsPhoto
3221226011 PacsLineSpec
IA>>prod1=products[1].product
IA>>newVariable = prod1.refs["level2"].product
IA>>newVariable1 = newVariable
IA>>newVariable2 = newVariable1
IA>>del(newVariable1)
IA>>newVariable = prod1.refs["level2"].product["HPPAVGR"].product
IndexError: Dataset[HPPAVGR] does not exist
IA>>newVariable = prod1.refs["level2"].product.refs["HPPAVGR"].product
IA>>
```

Figure 4.3. The PACS green channel image displayed in the full work bench of HIPE.

In the "Variables" and "Outline" displayed in Figure 4.1 and Figure 4.2 we see first an `ObservationContext` called "prod1" which is a PACS photometer test observation -- which has been expanded in the "Outline" view. A double click on the "Level2" product will show the outline of the final processed image (which contains two PACS images in two channels of the photometer taken simultaneously, a green channel and a blue channel). This is shown in Figure 4.2. We can also get the `SimpleImage` (e.g., name it "image1") by extracting it from the `ObservationContext`. The line below can do this from the command-line of the "Console" view.

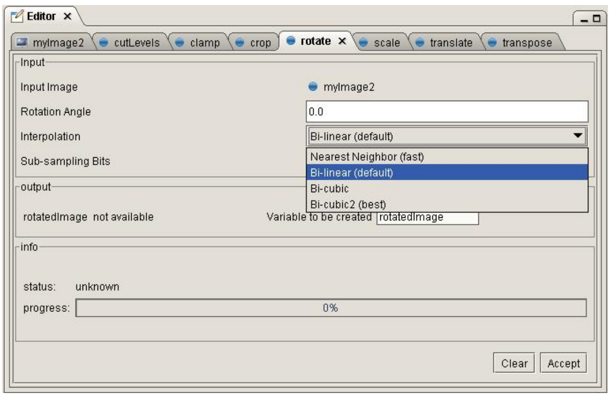
```
image1 = prod1.refs["level2"].product.refs["HPPAVGR"].product
```

A double click on the product automatically opens up an image display of the test image. In the "Outline" window we can actually see that there are several datasets which include an error map, a coverage map and exposure map associated with the image (see Figure 4.3). A right click on any of the associated datasets and going to "Open With..." allows a Dataset viewer to appear which shows metadata and array data for the particular dataset.

4.2.2. Basic image transformations

All image transformations can be applied in the same way. First, select a `SimpleImage` in the *Variables* view, then go to the *Tasks* view and select a transformation from the *Applicable* folder. To select one of the transformation tasks, double-click on its name on the *Tasks* view. A dialogue window for the task appears in the *Editor* view.

Dialogue windows work in a similar fashion for all image transformations. You can set parameters via pull-down menus or text boxes. To run the task, click on the Accept button.



Input
Input Image myImage2
Rotation Angle 0.0
Interpolation Bi-linear (default)
Nearest Neighbor (fast)
Bi-linear (default)
Bi-cubic
Bi-cubic2 (best)
Sub-sampling Bits

output
rotatedImage not available
Variable to be created rotatedImage

info
status: unknown
progress: 0%

Figure 4.4. Example image transformation dialogue window. Rotating an image using the "rotate" task. Several interpolation options are available.

The following basic image transformations are available:

- **Clamping:** also known as *clipping*, sets the floor and ceiling values of an image. Values above the ceiling or below the floor are set to the ceiling or floor, respectively.
- **Cropping:** extracts a section of the image to another `SimpleImage`. The area to extract is defined by two boundary rows and columns.
- **Rotating:** rotates an image by a given angle, with four types of interpolation:
  - **Bi-linear [default]:** interpolates one pixel to the right and one below.
  - **Nearest neighbour [fast]:** direct pixel copying, the fastest option.
  - **Bi-cubic:** uses interpolation via a piecewise bi-cubic polynomial.
  - **Bi-cubic2 [slow]:** variant of bicubic interpolation that can give sharper results.
- **Scaling:** scales the image, allowing for different scaling factors in the X and Y directions. The available interpolation types are as for the *Rotate* task.
- **Translating:** allows either X and Y pixel translations or sky translations (coordinates input as strings of the form `hh:mm:ss.s` and `dd:mm:ss.s`).
- **Transposing:** performs one of the following simple transpositions:
  - Flip vertically (flips top and bottom) and horizontally (flips from side to side).
  - Flip diagonally (bottom left to top right) and antidiagonally (top left to bottom right).
  - Rotate 90, 180 and 270 degrees (clockwise).

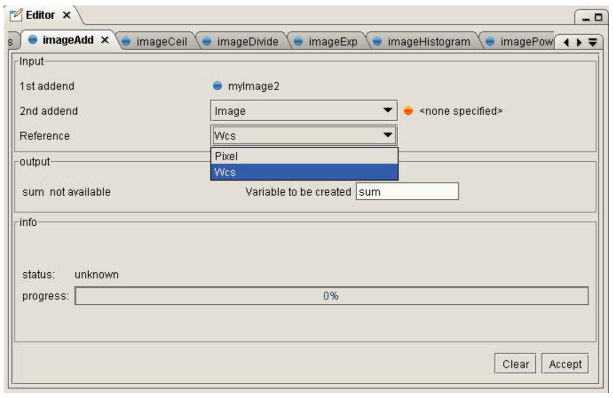
4.2.3. Image arithmetics

Possible arithmetic tasks (available in the *Applicable* folder of the *Task* view when you select an image) are the following:

- Absolute value (imageAbs). To obtain the absolute value image from the input.
- Add/Subtract/Multiply/Divide (imageAdd, imageSubtract, imageMultiply, imageDivide). This allows either a scalar or a second image as the amount to be added/subtracted/multiplied/divided. The second image can be input into the dialog by click-and-dragging of it from the *Variables* view to the small circle next to the corresponding parameter (see Figure 4.5). For images, the combination is by pixels or WCS reference.

- Modulo of an image with respect to another image or a scalar (imageModulo). This is done either pixel-by-pixel or based on the images' WCS.
- Exponent of the image. Including to the power N and 10 (imageExp, imageExpN, imageExp10).
- Log of the image. Including base 10 or N (imageLog, imageLog10, imageLogN).
- Image to the power n (imagePower).
- Image rounding, flooring or ceiling (imageRound, imageFloor, imageCeil).
- Square and square root of the image (imageSquare, imageSqrt).

Most of the above are self-explanatory. One example is shown in Figure 4.5.



Input
1st addend myImage2
2nd addend Image <none specified>
Reference Wcs
Pixel
Wcs
output
sum not available
Variable to be created sum
info
status: unknown
progress: 0%

Figure 4.5. Example image arithmetic dialog.

4.2.4. Smoothing

The following smoothing tasks are available in the *Tasks* view: *meanSmoothing*, *medianSmoothing*, *boxCarSmoothing* and *gaussianSmoothing*.

The dialogue window of each of these tasks has a field where you should enter the value for the `width` parameter, representing the width of the filtering window/boxcar/gaussian. Note that this parameter is called `sigma` for Gaussian smoothing.

For information on smoothing via the command line, see Section 4.3.4.

4.2.5. Flagging saturated pixels

Run the `flagSaturatedPixels` task, which only requires you to enter a cut off value, above which pixels are considered saturated.

The output is another image, called `flaggedImage` by default. It looks like a copy of the input image, except that pixels whose value lies above the cut off value are flagged out with the `SATURATED` flag type. The following figure show an image with flagged saturated pixels:

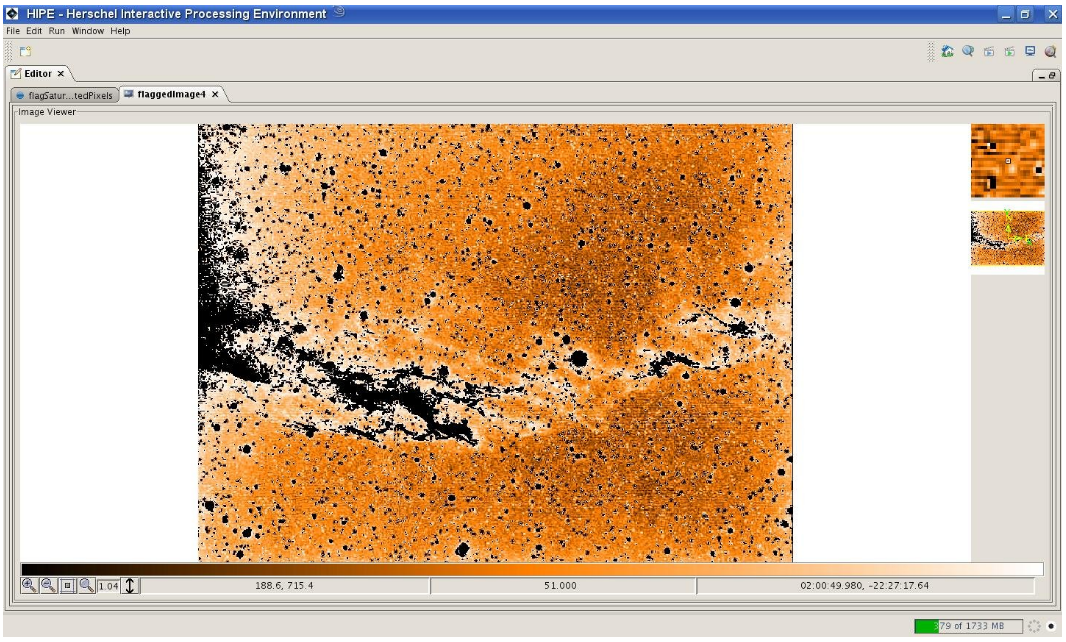


Figure 4.6. Application of the image flagging task.

For information on flagging saturated pixels via the command line, see Section 4.3.5.

4.2.6. Getting cut levels

Using the `cutLevels` task you can determine the cut levels of an image, either using the percentage method or applying a median filter.

Indicate via the *Method* combo box which method you want to use to determine the cut levels. If you select *Percent*, you can change the default percentage value in the *Percent* field.

The result is an array with two elements, the low and high cut. You can find it in the *Variables* view.

For information on determining cut levels via the command line, see Section 4.3.6.

4.2.7. Intensity profiles

With this task you can draw a straight line on an image and plot the intensity along that line.

Select an image in the *Variables* view and double click on the `profile` task in the *Tasks* view. The image appears in a new tab within the *Editor* view.

Click once on the image to define one end of the line. As you move the mouse, the line is updated and the corresponding profile appears below the image (see Figure 4.7). Click a second time to define the other end of the line. The resulting profile is saved into a dataset (see the *Variables* view).

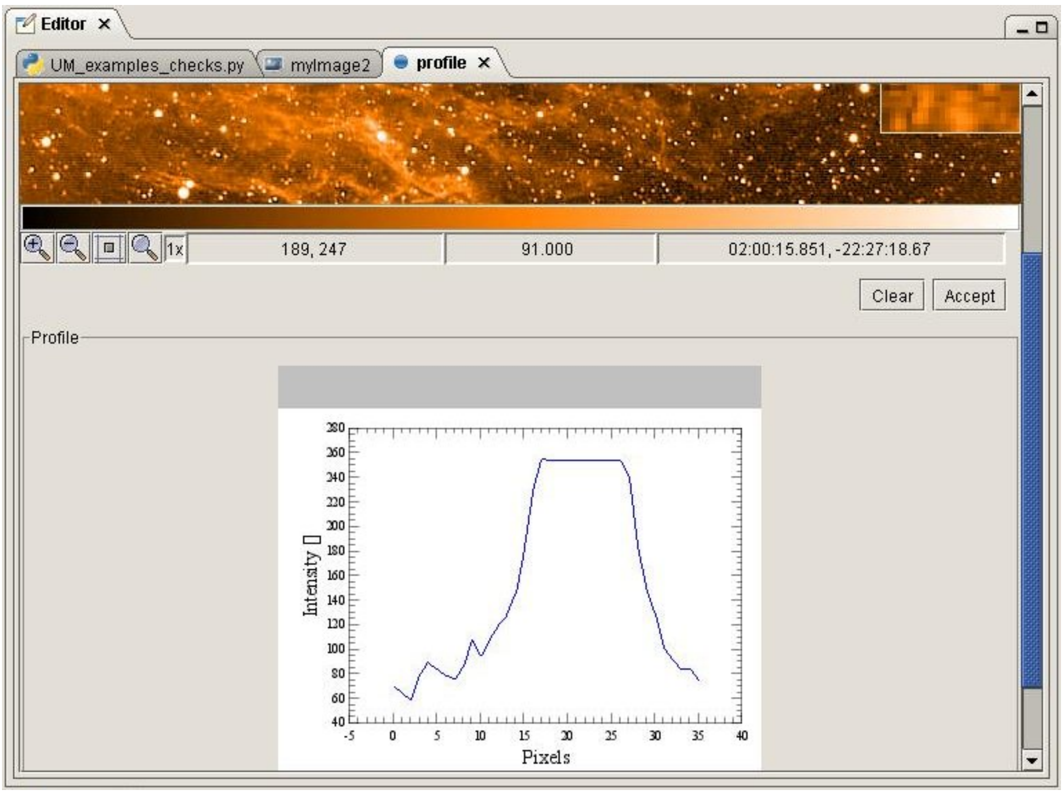


Figure 4.7. The intensity profile below the image.

You can modify the line by clicking on it and dragging the blue handles. Note that, while the plot below the image is updated in real time, the output dataset is not. You have to click the Accept button to obtain a new dataset with the updated result.

Alternatively, you can click on Clear to delete the line and draw a new one. The output dataset will appear as soon as you define the second end of the line, without having to click Accept.

For information on creating intensity profiles via the command line, see Section 4.3.7.

4.2.8. Contour Plotting

A contour plot connects all points in the image with the same intensity, like isobars on a weather map.

You can provide a set of contours in three ways. The first is via the `automaticContour` task, where you select the number of levels and a min and max value, and the intermediate levels are generated automatically with linear or logarithmic intervals of intensity. The second is via the `manualContour` task, which allows you to specify the values of each contour level. The third way is via `contour` to specify a single contour level.

Start by clicking the name of the target image in the *Variables* view. Then double click on `automaticContour`, `manualContour` or `contour` in the *Tasks* list. The corresponding dialogue window appears in the *Editor* view (see Figure 4.8 for example).

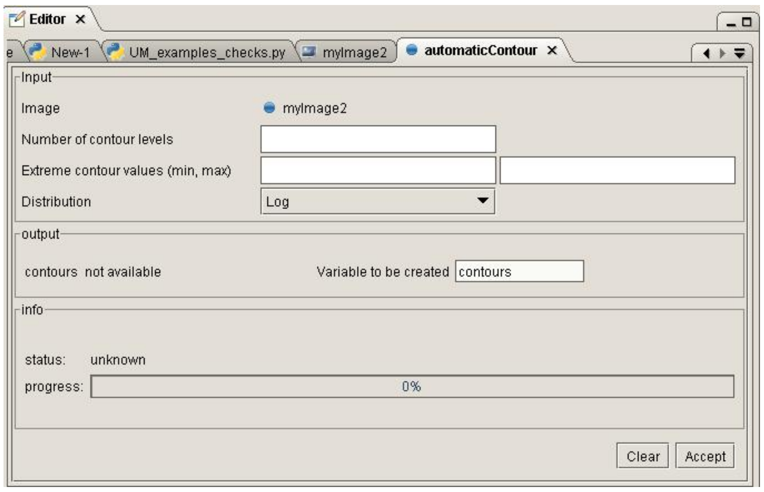


Image: myImage2
Number of contour levels:
Extreme contour values (min, max):
Distribution: Log
contours not available
Variable to be created: contours
status: unknown
progress: 0%

Figure 4.8. Dialogue window for `automaticContour`.

With `manualContour`, enter a contour value and press Add to add it to the list. Remove the last selected value or the whole list by clicking on Remove or Clear respectively.

Clicking on the Accept will execute the task and store the result in a variable (default name is `contours`).

To plot the contours on the image, click on the variable storing the contours in the *Variables* view and drag it onto the image. The result will be as shown in Figure 4.9. If the the contours are calculated for an image with a valid WCS and dragged onto an image with a valid WCS, the plotting will be based on the sky coordinates. In all other cases, the pixel coordinates will be used.

You can also drag your contours over a different image, not just the one you used to compute them. This can be convenient if you want to compare images at different wavelengths.

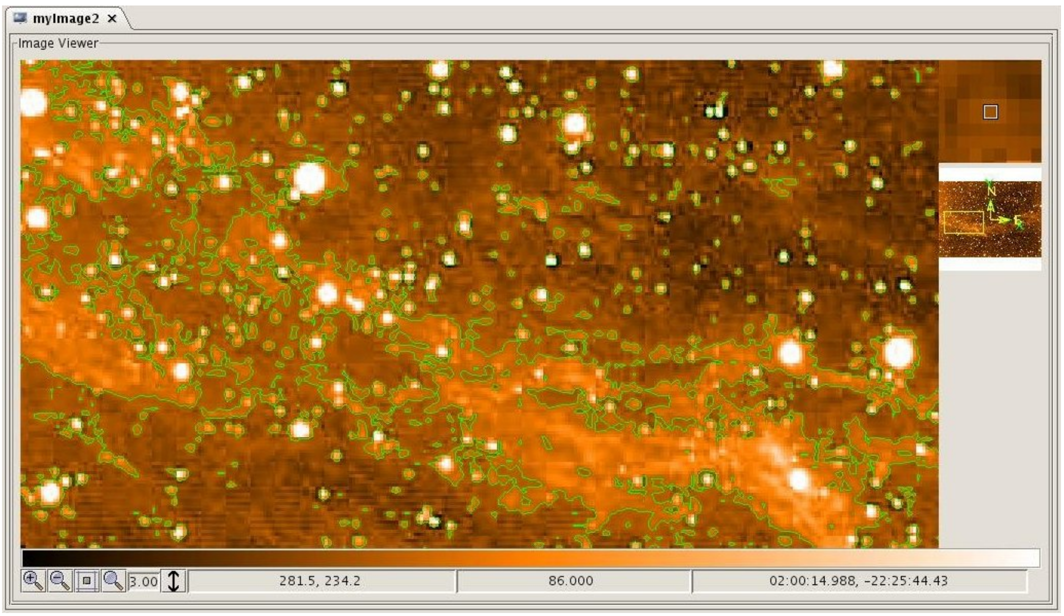


Figure 4.9. Output of the contour task.

For information on contour plotting via the command line, see Section 4.3.8.

4.2.9. Histograms

You can make a histogram of a whole image or of a region bounded by a circle, ellipse, rectangle or polygon.

Select an image from the *Variables* view, then double click on one of the `imageHistogram`, `polygonHistogram`, `circleHistogram`, `ellipseHistogram` or `rectangleHistogram` tasks from the *Tasks* view.

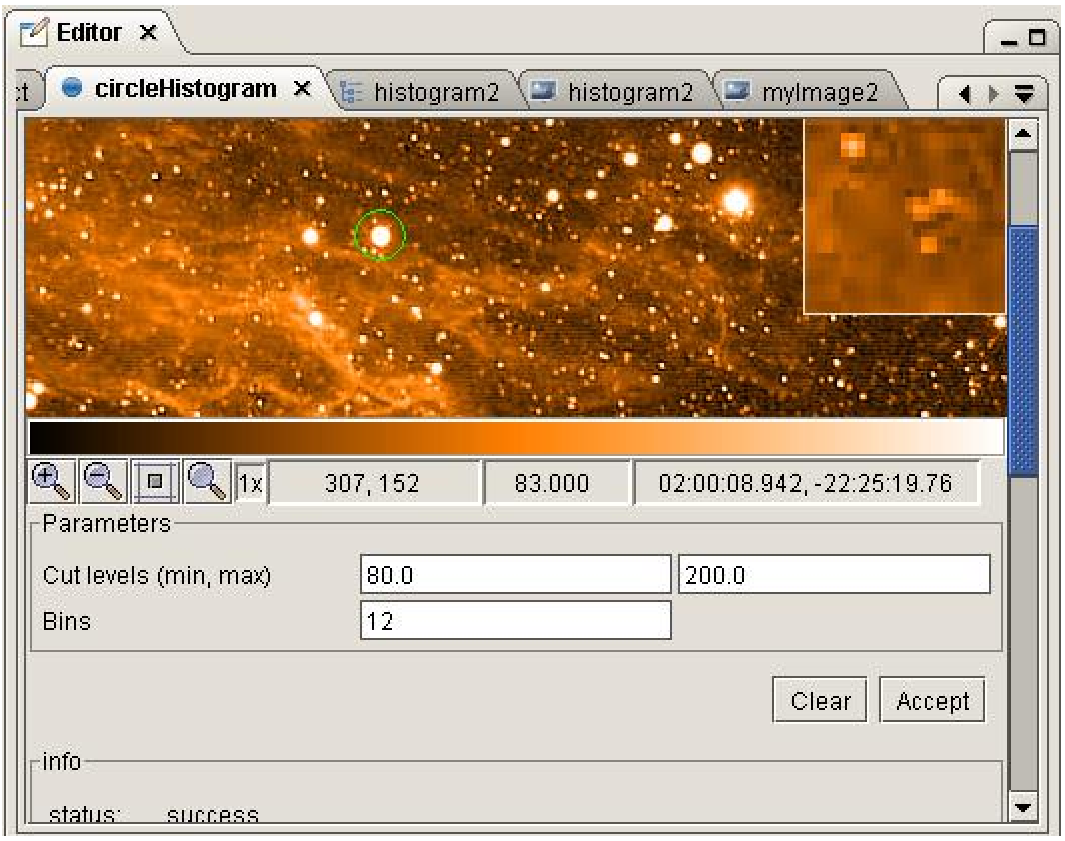
With the exception of `imageHistogram`, which computes the histogram for the entire image, a new copy of the image appears in the *Editor* view. Click and drag the mouse pointer to draw the region. With `polygonHistogram`, a single click adds a vertex, and a double click adds the final vertex.

Once you have created the region, you can move and resize it. To move the region, just click and drag it. To resize the region, click once inside it, then drag the blue resize handles.

Below the image you can enter the cut levels and number of bins for the histogram (see Figure 4.10). Once you press the Accept button, the following happens:

- The histogram appears in the same window (scroll down to see it).
- The equivalent command appears in the *Console* view.
- The histogram values are placed in a dataset that appears in the *Variables* list. Double click the variable name to show more information (see Figure 4.11).

Note that changing the area *after* running the task will modify the histogram shown in the task window, but *not* the one in the dataset. You have to click on Accept again to produce a new dataset with the updated result.



307, 152 | 83.000 | 02:00:08.942, -22:25:19.76

Parameters

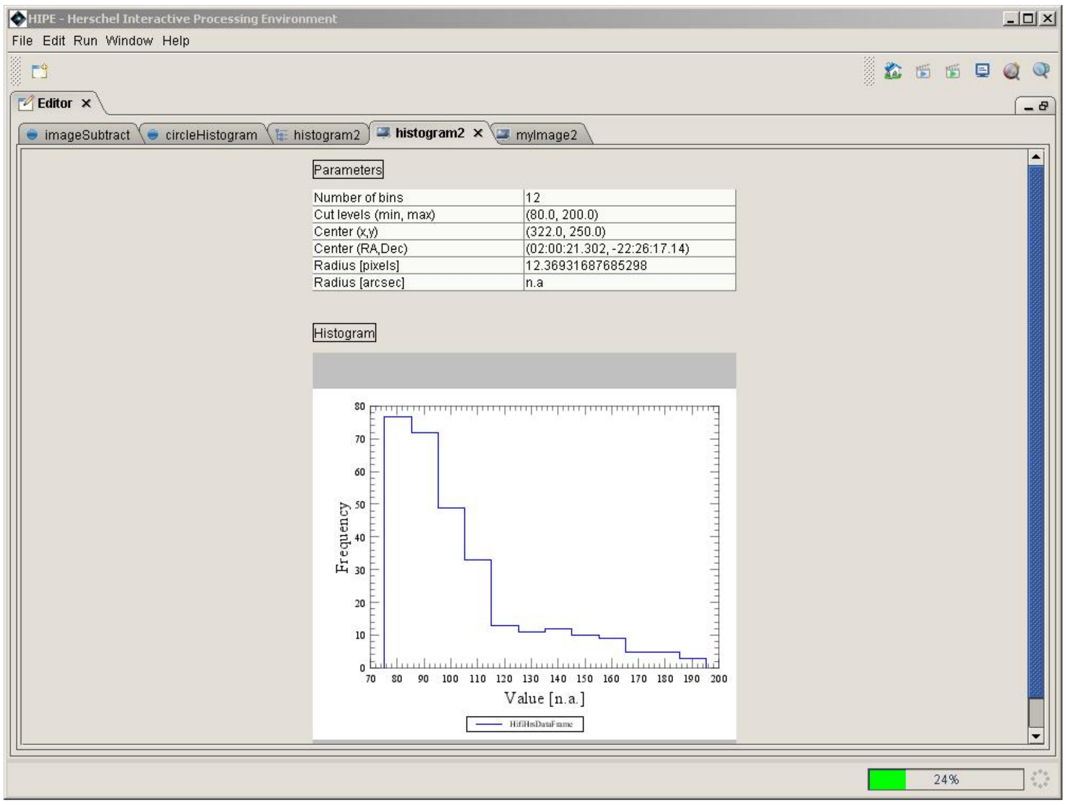
Cut levels (min, max): 80.0, 200.0

Bins: 12

info

status: success

Figure 4.10. Circle histogram area selection and parameter selection.



Parameters

Number of bins	12
Cut levels (min, max)	(80.0, 200.0)
Center (x,y)	(322.0, 250.0)
Center (RA,Dec)	(02:00:21.302, -22:26:17.14)
Radius [pixels]	12.36931687685298
Radius [arcsec]	n.a

Histogram

Figure 4.11. Display of the histogram results held in the histogram output in an expanded Editor view.

For information on creating histogram via the command line, see Section 4.3.9.

4.2.10. Aperture photometry

You can perform aperture photometry on an image using a circular target aperture, and an annular or rectangular sky aperture. You can also provide a fixed sky value. Five algorithms can be used to estimate the sky: average, median, mean-median, the synthetic mode and daophot. The mean-median method gives the average of all the values closer to the median than a specified number standard deviations (for example 1.5). The `daophot` method is a translation of the algorithm used in the IDL `aophot` package.

To perform **annular aperture photometry**, select an image in the *Variables* view and double click on `annularSkyAperturePhotometry` in the *Tasks* view. The image appears in a new tab within the *Editor* view. Below the image you can find the interface to enter the task options.

In the *Target center* pane, a drop-down menu gives you three ways to identify the target:

- By mouse interaction. With this option selected, click once on the image to select the target.
- By entering the X and Y pixel coordinates.
- By entering sky coordinates, if the image has a valid WCS. Use the format `"02:00:39.4"` for RA and `"-22:27:20.6"` for Dec. Note that the quotations are necessary as the input is a string.

The target is identified by a circle. You can drag the circle over a different target.

In the *Apertures* pane you can enter the radii for the target and sky regions. The circular radii are shown on the image (see Figure 4.12). In the *Sky estimation* pane you can specify the algorithm, and whether to use entire pixels or fractional pixels.

You can reset the parameters at any time by clicking Clear. Click Accept to execute the task.

You can display the results by double clicking on the `result` variable shown in the *Variables* view (see Figure 4.13).

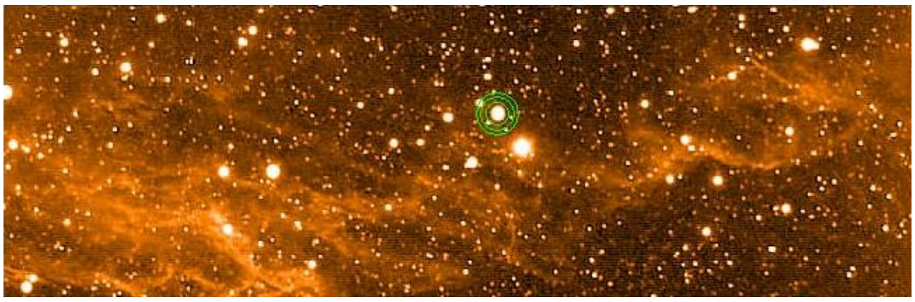
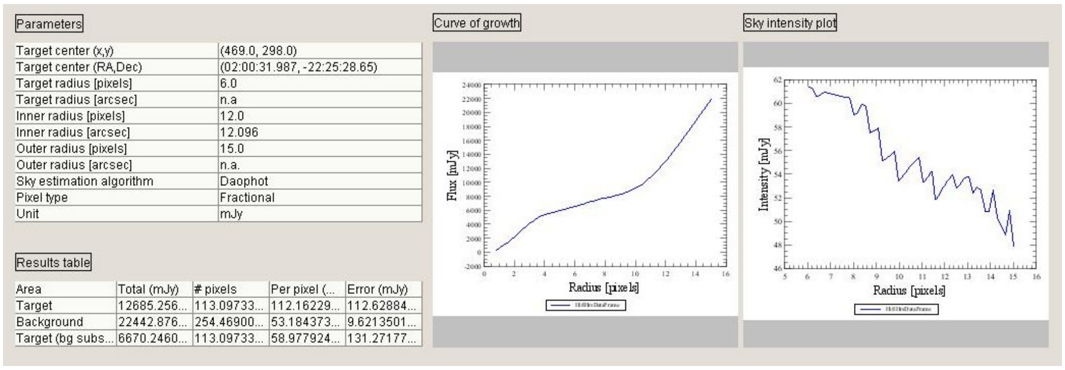


Figure 4.12. Aperture photometry with an annular sky aperture as displayed in HIPE.



Parameters

Target center (x,y)	(469.0, 298.0)
Target center (RA,Dec)	(02:00:31.987, -22:25:28.65)
Target radius [pixels]	6.0
Target radius [arcsec]	n.a
Inner radius [pixels]	12.0
Inner radius [arcsec]	12.096
Outer radius [pixels]	15.0
Outer radius [arcsec]	n.a.
Sky estimation algorithm	Daophot
Pixel type	Fractional
Unit	mJy

Results table

Area	Total (mJy)	# pixels	Per pixel (...	Error (mJy)
Target	12685.256...	113.09733...	112.16229...	112.62884...
Background	22442.876...	254.46900...	53.184373...	9.6213501...
Target (bg subs...	6670.2460...	113.09733...	58.977924...	131.27177...

Figure 4.13. Aperture photometry results plot and tables. Note that n.a. relates to "not applicable" and typically will occur when units are not assigned to the image.

The results include two plots, useful to judge whether your choice of radii was sensible:

- A *curve of growth*, showing the target flux, without the sky, as a function of the radius.
- A *sky intensity plot*, showing the intensity per sky pixel as a function of the inner radius, the outer radius being constant.

You can do **rectangular aperture photometry** by choosing `rectangularSkyAperturePhotometry` item from the *Tasks* view, after selecting an image in the *Variables* view. As for the annular sky aperture photometry, you can select the object with one click or give the coordinate explicitly. Click and drag to select a rectangular aperture. Following the calculation for the first position, you can use the same rectangular box for the sky and choose a new object with a further single click on the image.

The result product has the same structure as for annular photometry, except that the sky intensity plot is missing.

Use `fixedSkyAperturePhotometry` to provide a **fixed sky value**. Executing the task and inspecting the results is done in the same way as for the other types of photometry.

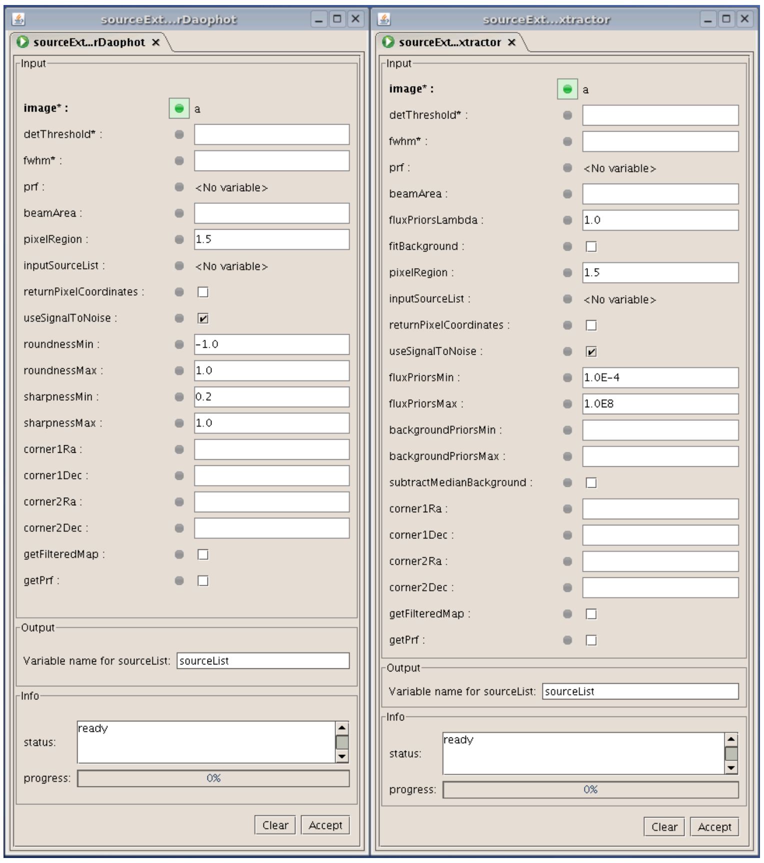
For information on aperture photometry via the command line, see Section 4.3.10.

4.2.11. Source extraction

HIPE includes the `sourceExtractorDaophot` and `sourceExtractorSussextractor` tasks, designed primarily for use on PACS and SPIRE maps. It implements the DAOPHOT (classic) and SUSSEXtractor algorithms for extraction of point sources with a known profile. This section explains how to use the source extractor via the graphical interface; advanced usage is described in the *User Reference Manual*:

- Section 2.381
- Section 2.382

The two tasks are listed in the *Applicable* folder of the *Tasks* view whenever an image is selected in the *Variables* view. The following figure shows the lists of parameters for the two tasks:



sourceExt...rDaophot

Input

image* :	a
detThreshold* :	
fwhm* :	
prf :	<No variable>
beamArea :	
pixelRegion :	1.5
inputSourceList :	<No variable>
returnPixelCoordinates :	☐
useSignalToNoise :	✓
roundnessMin :	-1.0
roundnessMax :	1.0
sharpnessMin :	0.2
sharpnessMax :	1.0
corner1Ra :	
corner1Dec :	
corner2Ra :	
corner2Dec :	
getFilteredMap :	☐
getPrf :	☐

Output

Variable name for sourceList: sourceList

Info

status: ready

progress: 0%

sourceExt...xtractor

Input

image* :	a
detThreshold* :	
fwhm* :	
prf :	<No variable>
beamArea :	
fluxPriorsLambda :	1.0
fitBackground :	☐
pixelRegion :	1.5
inputSourceList :	<No variable>
returnPixelCoordinates :	☐
useSignalToNoise :	✓
fluxPriorsMin :	1.0E-4
fluxPriorsMax :	1.0E8
backgroundPriorsMin :	
backgroundPriorsMax :	
subtractMedianBackground :	☐
corner1Ra :	
corner1Dec :	
corner2Ra :	
corner2Dec :	
getFilteredMap :	☐
getPrf :	☐

Output

Variable name for sourceList: sourceList

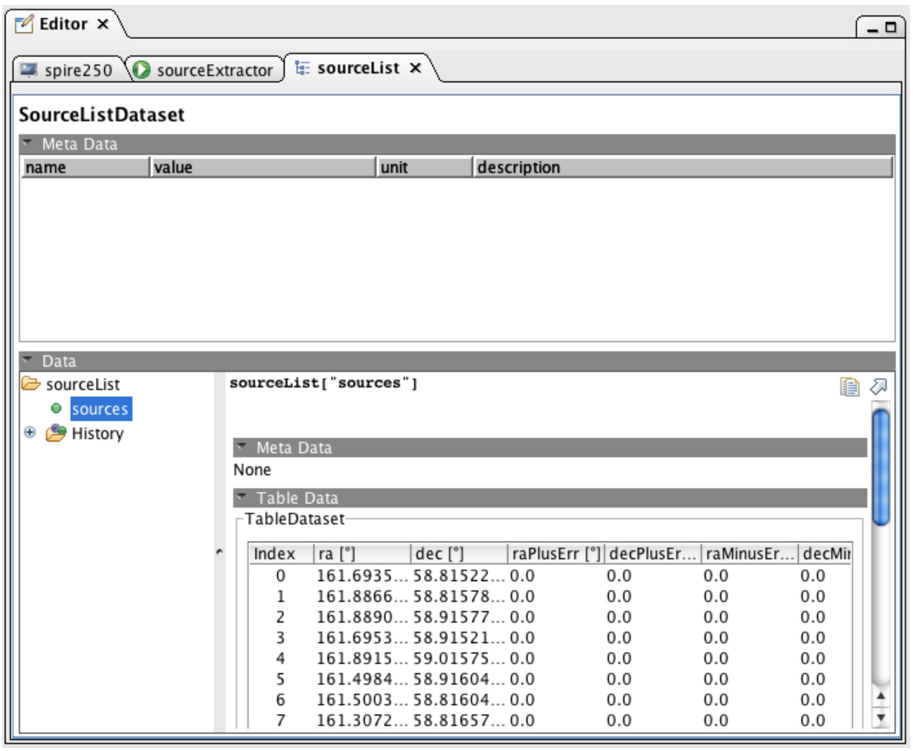
Info

status: ready

progress: 0%

Figure 4.14. List of parameters for the two source extraction tasks.

The output is of type `SourceListProduct` and is called `sourceList` by default. You can inspect it in the *Product Viewer* like any other product, as shown by the figure below:



SourceListDataset

name	value	unit	description

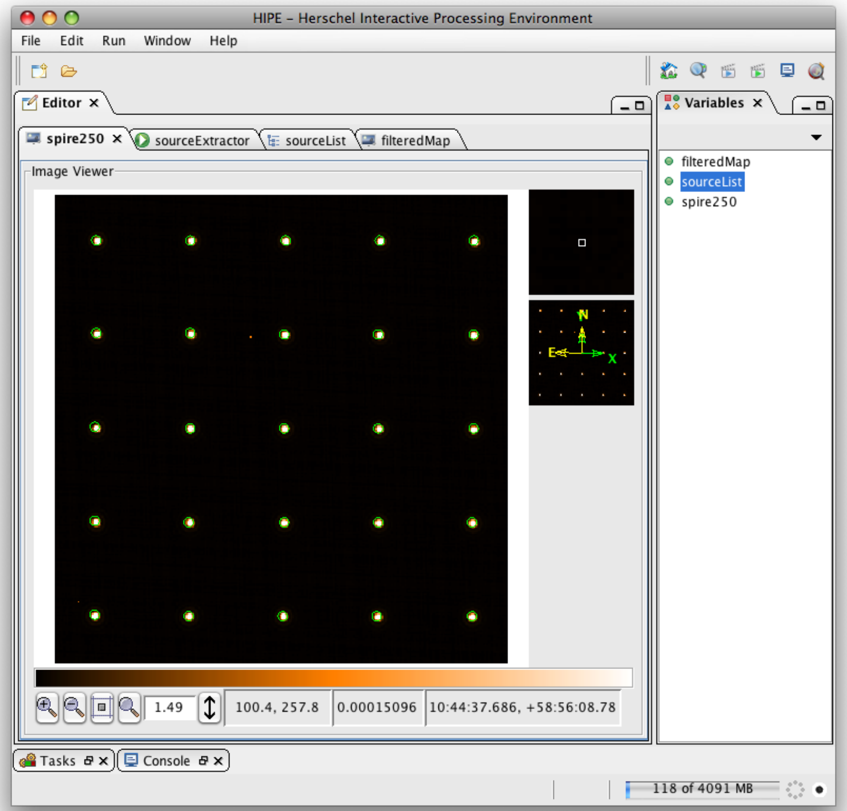
sourceList["sources"]

None

Index	ra [°]	dec [°]	raPlusErr [°]	decPlusEr...	raMinusEr...	decMi
0	161.6935...	58.81522...	0.0	0.0	0.0	0.0
1	161.8866...	58.81578...	0.0	0.0	0.0	0.0
2	161.8890...	58.91577...	0.0	0.0	0.0	0.0
3	161.6953...	58.91521...	0.0	0.0	0.0	0.0
4	161.8915...	59.01575...	0.0	0.0	0.0	0.0
5	161.4984...	58.91604...	0.0	0.0	0.0	0.0
6	161.5003...	58.81604...	0.0	0.0	0.0	0.0
7	161.3072...	58.81657...	0.0	0.0	0.0	0.0

Figure 4.15. The list of sources shown in the Product Viewer, with the internal dataset highlighted.

To display the extracted sources on the image, drag and drop the `sourceList` on the image in the *Editor* view. A circle is overlaid at the location of each source, as shown by the following figure:



1.49 | 100.4, 257.8 | 0.00015096 | 10:44:37.686, +58:56:08.78

Figure 4.16. An image with the locations of the extracted sources overlaid as circles.

Additional outputs

If you check the *getPrf* or *getFilteredMap* checkbox, the output will include the *point response function* and the *filtered map* as additional images. For the SUSSEXtractor algorithm, the filtered map is equal to the input map convolved with the point response function, such that the value at each pixel gives an estimate of the flux of a source, in mJy, assuming there is a source located at the centre of that pixel. For the DAOPHOT algorithm, the filtered map gives the input map convolved with the DAOPHOT kernel.

Warning

If you select one or both of these additional outputs, the result of the task will be an *array* of products (more precisely, a Jython tuple). Double clicking on it in the *Variables* view will not open a viewer. You can extract the individual outputs with the following commands, assuming that the array is called `result`:

```
HIPE> sourceList = result[0]
HIPE> filteredMap = result[1]
HIPE> prf = result[2]
```

Additional actions

These are further actions that may be of interest to you:

- **Specifying the positions of known sources**

  You can use a `SourceListProduct` as an input to the source extractor task to specify the positions of known sources. The task will then give the fluxes of sources at those positions. To provide the list of known sources, drag and drop a variable of type `SourceListProduct` onto the small circle next to the `inputSourceList` parameter.

- **Specifying a custom point response function**

  By default, the point response function (PRF) is assumed to be Gaussian, with full-width-half-maximum (in arcsec) provided by the `fwhm` parameter. Alternatively, you can specify a custom PRF via the `prf` parameter. This should be a variable of type `SimpleImage`. The image should be of odd dimension, with the peak at the centre, normalised such that it gives the (central pixels) of a point source of flux 1 Jy, in the units of the input map.

- **Working with source lists in ASCII files**

  To export the source list to a text file, you can run the `asciiTableWriter` task. First you have to retrieve the source list dataset from the result of the source extraction, like this:

  ```
  HIPE> sourceListDataset = sourceList.default
  ```

  Then click on `sourceListDataset` in the *Variables* view, and you will find `asciiTableWriter` among the applicable tasks.

  To import an ASCII file as a list of sources, use the `asciiTableReader` task. The result of this task is of type `TableDataset`. To obtain a `SourceListProduct`, use a command like the following:

  ```
  HIPE> importedSourceList = SourceListProduct(table)
  ```

  Note that the column names in the imported source list must match the default column names in a `SourceListDataset` ("ra", "dec", "flux" and so on). Column names are case insensitive.

- **Working with source lists in FITS files**

  To export a list of sources of type `SourceListProduct` to a FITS file, select `simpleFitsWriter` from the applicable tasks.

To import a `SourceListProduct` stored in a FITS file, load the file with File → Open File, and HIPE will do the rest. If the FITS file does *not* contain a `SourceListProduct`, the data will be imported as a generic `Product`, with the source list contained in a `Dataset`. You can create a proper `SourceListProduct` with the following command, assuming that the dataset is called `HDU_1`:

```
HIPE> importedSourceList = SourceListProduct(sourceList["HDU_1"])
```

4.3. In depth

4.3.1. Working with the World Coordinates System

The WCS information for an image is stored in its metadata.

With the `Wcs` class you can define a transformation between pixel coordinates and world coordinates. The following illustrates how you can create a WCS and add it to a `SimpleImage`.

```
i = SimpleImage()
i.image=RESHAPE(Double1d.range(200*300), [200,300])
# Create a fake image 200x300 pixels in size

myWcs = Wcs() # Set up the Wcs() object
myWcs.ctype1 = -"LINEAR"  # Start adding things to it...
myWcs.cdelt1 = 5
myWcs.crval1 = 200
myWcs.cunit1 = -"K"
myWcs.crpix1 = 0

myWcs.ctype2 = -"LINEAR"
myWcs.cdelt2 = -.05
myWcs.crval2 = 2.0
myWcs.cunit2 = -"V"
myWcs.crpix2 = 0

i.wcs = myWcs  # Apply the set of WCS information to our image
print i.wcs # To see the WCS of the image
```

Warning

The above code will generate an image with the value 200 assigned to the `NAXIS2` keyword and 300 assigned to `NAXIS1`. In other words, the image size will be 200 pixels along the *y* axis and 300 pixels along the *x* axis. The coordinate values will be displayed in this order (*y*, *x*) in the Image Viewer. For an explanation of why the *y* size comes *before* the *x* size, see the *Scripting and Data Mining* guide: Section 2.6.1.

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 (crval1), uses steps of 5 (cdelt1) and has as unit Kelvin. The y-axis is also LINEAR (ctype2), has the central pixel in row 0 (crpix2, this is the top of the image), has a value of 2 in the central pixel (crval2), uses steps of 0.05 (cdelt2) and has as unit Volts.

Note

Rows and columns start counting from (0,0), pixels from (1,1).

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. It also indicates how we can "set" WCS values in our WCS object :

```
wcs2 = Wcs()  # ❶
```

```
wcs2.setCrpix1(128)
wcs2.setCrpix2(128) # ❷
wcs2.setCrval1(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.

For more information on the WCS see the *Scripting and Data Mining* guide: Section 2.13.

4.3.2. Basic image transformations

Clamping/Clipping

Clamping or clipping an image means that all intensities below a certain value `low` are set to this value, and that all values above another value `high` are set to that value. This means that you need only these parameters for clamping:

- the *image* (`Image image`)
- the *lower value* (`Double low`)
- the *upper value* (`Double high`)

To clamp an `Image` between 20.0 and 100.0, simply type

```
clamped = clamp(image = myImage2, low = 20.0, high = 100.0)
```

By running this task, the clamped `Image` will appear in the *Variables* view.

The result, `clamped`, is a new `Image`, with the same settings as the input `Image`.

Cropping

The size of an Image can be reduced through cropping. The user must only specify these parameters :

- the *image* (`Image image`)
- *from which row* (`Integer row1`) *to which row* (`Integer row2`) the image should be cropped
- *from which column* (`Integer column1`) *to which column* (`Integer column2`) the image should be cropped

To crop an Image for row = 40,..., 120 and column = 30,..., 150 simply type

```
cropped = crop(image = myImage2, row1 = 40, row2 = 120, column1 = 30, \
 column2 = 150)
```

The resulting `Image`, `cropped`, is an `Image` with the same settings (errors, `Flag`, exposure), cut out of the input `Image` between the specified rows and columns. The `Wcs` is adapted, in order to have the same sky coordinates for the same position in the `Images`.

Rotating

An Image can also be rotated over a given angle. If the y-axis points down (up), a positive rotation angle means a clockwise (counterclockwise) rotation. You have to specify three parameters :

- the *image* (`Image image`)
- the *rotation angle in degrees* (`Double angle`)
- the *type of interpolation* (`Integer interpolation`) - optional (per default : linear)

You can choose between four types of interpolation :

- `RotateTask.INTERP_BILINEAR` = 0 : interpolates one pixel to the right and one down (default)
- `RotateTask.INTERP_NEAREST` = 1 : direct pixel copying
- `RotateTask.INTERP_BICUBIC` = 2 : interpolation via a piecewise cubic polynomial
- `RotateTask.INTERP_BICUBIC_2` = 3 : variant of bicubic interpolation that can produce sharper result than bicubic interpolation

In the case you use one of the bicubic interpolation algorithms, you must also specify the *number of bits to use for the interpolation* (`Integer subsampleBits` - optional (per default : 16)).

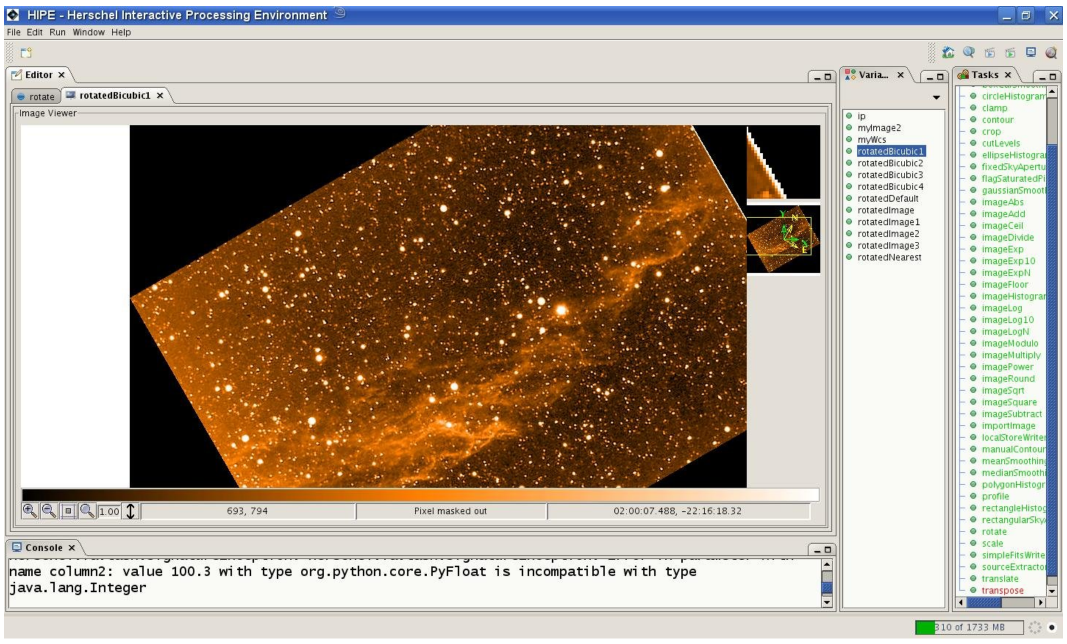
To rotate an image via the command line, just type

```
# Use the default interpolation (linear)
rotatedDefault = rotate(image = myImage2, angle = 30.0)

# Use direct pixel copying
rotatedNearest1 = rotate(image = myImage2, angle = 30.0, \
 interpolation = RotateTask.INTERP_NEAREST)
rotatedNearest2 = rotate(image = myImage2, angle = 30.0, interpolation = 1)

# Use bicubic interpolation
rotatedBicubic1 = rotate(image = myImage2, angle = 30.0, \
 interpolation = RotateTask.INTERP_BICUBIC)
rotatedBicubic2 = rotate(image = myImage2, angle = 30.0, interpolation = 2)
rotatedBicubic3 = rotate(image = myImage2, angle = 30.0, \
 interpolation = RotateTask.INTERP_BICUBIC, subsampleBits = 18)
rotatedBicubic4 = rotate(image = myImage2, angle = 30.0, interpolation = 2, \
 subsampleBits = 18)
```

The result, `rotated`, is an Image with the same settings as the input Image, but rotated over the given angle. The result is shown here :



```
name column2: value 100.3 with type org.python.core.PyFloat is incompatible with type
java.lang.Integer
```

Figure 4.17. Image rotation task.

Scaling

An `Image` can be magnified in the x- and y-directions independently using the `ScaleTask`. Also here interpolation is necessary, just like for rotating, so the input parameters for this task are :

- the image (Image image)
- the magification factor along the x- and y-axes (Doubles x and y)
- the type of interpolation (Integer interpolation) - optional (per default : linear)

The interpolation types are the same as for rotating : `ScaleTask.INTERP_BILINEAR`, `Scale.INTERP_NEAREST`, `ScaleTask.INTERP_BICUBIC` and `ScaleTask.INTERP_BICUBIC_2`. Also here, the number of subsampling bits (`Integer subsampleBits`) must be specified if you choose to use bicubic interpolation.

To perform scaling, you must type

```
scaled = scale(image = myImage2, x = 0.5, y = 2.0, \
 interpolation = ScaleTask.INTERP_BILINEAR)
```

Note

The parameters `interpolation` and `subsampleBits` are to be used exactly the same way as for rotating.

The result, `scaled`, is an `Image` with the same settings as the input `Image`, but stretched independently along both axes. The `Wcs` is adapted in a way that each source has the same sky coordinates in both `Images`.

Translating

You can translate an Image based on pixel or sky coordinates, so the required input parameters are :

- the *image* (`Image image`)
- the *translation vector* in pixel (`Doubles x` and `y`) or sky coordinates (`Strings ra` and `dec`)

To do the translation via the command line, simply type

```
# Translation based on pixel coordinates
translatedPixel = translate(image = myImage2, x = 50.4, y = --5.3)

# Translation based on sky coordinates
translatedSky = translate(image = myImage2, ra = -"00:01:00", dec = -"00:20:00")
```

Note

For the moment you can specify the pixel and sky coordinates at the same time. This should be prohibited in the future.

The result, `translated`, is an `Image` that looks the same as the input `Image`, but has as different `Wcs`, which takes the translation into account.

Transposing

Transposing an `Image` can be done in several ways : flipping horizontally/vertically/(anti)diagonally and rotating over 90, 180 or 270 degrees. This can be done on the command line, or in a GUI in HIPE. The only parameters that need to be specified are :

- the *image* (`Image image`)
- the *transposition type* (`Integer type` - per default : 0)

The possible transposition types are

- `TransposeTask.FLIP_VERTICAL` (0) : flips top and bottom
- `TransposeTask.FLIP_HORIZONTAL` (1) : flips from side to side
- `TransposeTask.FLIP_DIAGONAL` (2) : flips bottom left to top right
- `TransposeTask.FLIP_ANTIDIAGONAL` (3) : flips top left to bottom right
- `TransposeTask.ROTATE_90` (4) : rotates over 90 degrees
- `TransposeTask.ROTATE_180` (5) : rotates over 180 degrees
- `TransposeTask.ROTATE_270` (6) : rotates over 270 degrees

To transpose an Image, type

```
# Flip vertically
flippedVertically1 = transpose(image = myImage2, type = TransposeTask.FLIP_VERTICAL)
flippedVertically2 = transpose(image = myImage2, type = 0)
```

The output, `transposed`, looks exactly the same as the input `Image`, but differently oriented, or flipped. The `Wcs` is adapted, in order to make sure that corresponding points have the same sky coordinates both in the input and the output `Image`.

4.3.3. Image arithmetics

The following arithmetics tasks are available:

- addition/subtraction/multiplication/division of two `Images` pixel-to-pixel, or based on their `Wcs`
- addition/subtraction/multiplication/division of an `Image` and a scalar
- taking the modulus of an `Image` w.r.t. another `Image`, pixel-to-pixel, or based on their `Wcs`
- taking the modulus of an `Image` w.r.t. a scalar

- taking the absolute values of all intensity values
- rounding/flooring/ceiling all intensity values
- changing all intensity values in an Image according to a power/logarithmic/exponential scaling

All these tasks return an `Image` as output.

Addition/Substraction/Multiplication/Division/Modulo

Addition, subtraction, multiplication, division and modulus calculation of two Image can be done pixel-to-pixel, or based on their Wcs. In that case, you need to specify the following parameters:

- the *images* (`Images image1` and `Image2`)
- the *reference frame* for the calculation (`Integer ref`)

The possible values for the ref parameter are

- `ImageArithmeticsTask.PIXEL` = 0 : pixel-to-pixel calculation
- `ImageArithmeticsTask.WCS` = 1 : `Wcs`-based calculation

If you want to use a pixel instead of a second `Image`, omit the `image2` and `ref` parameters and add

- the *scalar* (`Double scalar`)

To do the calculations for two `Images`, `myIm1` and `myIm2`, the commands are

```
# Adding (pixel-to-pixel)
sum = imageAdd(image1 = myIm1, image2 = myIm2, ref = ImageArithmeticsTask.PIXEL)

# Subtracting (pixel-to-pixel)
difference = imageSubtract(image1 = myIm1, image2 = myIm2, ref = 0)

# Multiplying (based on Wcs)
product = imageMultiply(image1 = myIm1, image2 = myIm2, \
 ref = ImageArithmeticsTask.WCS)

# Dividing (based on Wcs)
quotient = imageDivide(image1 = myIm1, image2 = myIm2, ref = 1)

# Modulo
remainder = imageModulo(image1 = myIm1, image2 = myIm2, ref = 0)
```

Note

If added or subtracted images have the same unit, the sum/difference will use that same unit, otherwise the calculation will be done in counts.

The product, quotient and remainder will have the composed unit as unit.

To do the calculations for an Image and a scalar, the commands are

```
# Adding
sum = imageAdd(image1 = myImage2, scalar = 200.0)

# Subtracting
difference = imageSubtract(image1 = myImage2, scalar = 200.0)

# Multiplying
product = imageMultiply(image1 = myImage2, scalar = 1.2)

# Dividing
product = imageDivide(image1 = myImage2, scalar = 0.5)
```

```
# Modulo
remainder = imageModulo(image1 = myImage2, scalar = 200.0)
```

Note

The result has the same unit as the input `Image`.

Absolute values

To take the absolute value of all intensity values in an image, type the following:

```
abs = imageAbs(image = myImage)
```

Rounding/Flooring/Ceiling

To round, floor of ceil all intensity values of an image, type the following:

```
# Rounding
rounded = imageRound(image = myImage2)

# Flooring
floored = imageFloor(image = myImage2)

# Ceiling
ceiled = imageCeil(image = myImage2)
```

Power/Square/Sqrt

You can also change all intensity values according to a power scale. For all three available tasks, you must specify

- the *image* (`Image image`)

For the `ImagePowerTask`, you also have to give

- the *power* (`Double n`)

To run the tasks on the command line, you have to type

```
# Power
powered = imagePower(image = myImage2, power = 1.5)

# Square
squared = imageSquared(image = myImage2)

# Sqrt
sqrt = imageSqrt(image = myImage2)
```

Logarithmic/Exponential

Instead of using a power scaling to adapt the intensity value, you can also use a logarithmic or exponential scaling. For all these tasks (`ImageLogTask`, `ImageLog10Task`, `ImageLogNTask`, `ImageExpTask`, `ImageExp10Task` and `ImageExpNTask`), you must give

- the *image* (`Image image`)

For the `ImageLogNTask` and `ImageExpNTask`, you also have to give

- *n* (`Double n`)

The commands are

```
# Log
log = imageLog(image = myImage2)
# Log10
log10 = imageLog10(image = myImage2)
# LogN
logN = imageLogN(image = myImage2, n = 8.0)

# Exp
exp = imageExp(image = myImage2)
# Exp10
exp10 = imageExp10(image = myImage2)
# ExpN
expN = imageExpN(image = myImage2, n = 8.0)
```

4.3.4. Smoothing

Four different smoothing algorithms are available: average, median, boxcar and gaussian smoothing. They all take the following parameters as input:

- the *image* (`Image image`)
- the *width of the filtering window/boxcar/gaussian* (`width`, or `sigma` for Gaussian smoothing)

The parameter `width` must be an odd positive `Integer` for mean and median smoothing and a positive `Integer` for boxcar smoothing. The parameter `sigma` must be a positive `Double` for Gaussian smoothing.

The commands for the four different tasks are very alike:

```
# Mean smoothing
smoothedMean = meanSmoothing(image = myImage, width = 3)

# Median smoothing
smoothedMedian = meanSmoothing(image = myImage, width = 3)

# Boxcar smoothing
boxCarSmoothed = boxCarSmoothing(image = myImage, width = 4)

# Gaussian smoothing
gaussianSmoothed = gaussianSmoothing(image = myImage, sigma = 2.5)
```

For information on smoothing via the HIPE graphical interface, see Section 4.2.4.

All these tasks have an `Image` as output. This has the same settings (Wcs, errors, flag, exposure) as the input image. You can explore it using `Display`, or by double-clicking on it and thus opening an image explorer.

4.3.5. Flagging saturated pixels

You can flag out pixels with their intensity above a certain value, with the `SATURATED` flag type. This can be done with the task `FlagSaturatedPixelsTask`, by specifying these parameters:

- the *image* (`Image image`)
- the *cut off value* (`Double value`)

To flag the saturated pixels, type the following:

```
flagged = flagSaturatedPixels(image = myImage2, value = 100.0)
```

The resulting image will appear in the *Variables* view.

For information on flagging saturated pixels via the HIPE graphical interface, see Section 4.2.5.

4.3.6. Getting cut levels

Using the task `CutLevelsTask` you can determine the cut levels of an image. You have to specify the following parameters:

- the *image* (`Image image`)
- the *method* used for determining the cut levels (`Integer method`)

Two methods are available:

- `CutLevelsTask.PERCENT` = 0: percentage method
- `CutLevelsTask.MEDIAN_FILTER` = 1: median filter

If you choose the percentage method, you must define one extra parameter:

- the *percentage* (`Double percent`), default 99.5

To execute the task, type the following:

```
# Percentage method
percentCutLevels1 = cutLevels(image = myImage2, method = CutLevelsTask.PERCENT)
percentCutLevels2 = cutLevels(image = myImage2, method = 0, percent  = 98.0)

# Median filter
median1 = cutLevels(image = myImage2, method = CutLevelsTask.MEDIAN_FILTER)
median2 = cutLevels(image = myImage2, method = 1)
```

The result, `percentCutLevels`, is a double array. To gain access to the low and high cut, type

```
# The low cut
low = percentCutLevels[0]

# The high cut
high = percentCutLevels[1]
```

4.3.7. Intensity profiles

`ProfileTask` allows you to determine the intensity of the pixels along a straight line on a given image. This can be convenient to see whether there is a gradient in intensity in your image.

The only input parameters are

- the *image* (`Image image`)
- the *beginning and end of the straight line* either in pixel (`Doubles beginX`, `beginY`, `endX` and `endY`) or in sky coordinates (`Strings beginRA`, `beginDec`, `endRA` and `endDec`)

To make a profile plot, type the following:

```
profilePixel = profile(image = myImage2, beginX = 236.0, beginY = 378.0, \
   endX = 557.0, endY = 232.0)
profileSky = profile(image = myImage2, beginRA = -"02:00:15.119", \
  beginDec = -"-22:24:07.16", endRA = -"02:00:38.462", endDec = -"-22:26:34.08")
```

Both output products (`profilePixel` and `profileSky`) appear in the *Variables* view in HIPE and can be inspected as follows:

```
# Returns a Double1d with the pixel coordinates of begin and
# end of the straight line
profile.getBeginPixelCoordinates()
profile.getEndPixelCoordinates()

# Returns a String1d with the sky coordinates of begin and
# end of the straight line
profile.getBeginSkyCoordinates()
profile.getEndSkyCoordinates()

# Returns the intensity plot as a Double1d
profile.getProfile()

# Returns the unit of the intensity
profile.getIntensity()
```

4.3.8. Contour plotting

You can make contour plots by specifying one (`ContourTask`) or several (`ManualContourTask`) contour values, or to let them be calculated automatically (`AutomaticContourTask`).

If you want to plot only one contour value, use `ContourTask`. The only input parameters are the *image* (`Image image`) and the *contour value* (`Double value`).

To run this task, type the following:

```
contours = contour(image = myImage2, value = 100.0)
```

If you want to specify multiple contour values, use `ManualContourTask`. The input parameters are the *image* (`Image image`) and the list of *contour values* (`Double1d values`).

Run the task as follows:

```
# Construction of the list of contour values
values = Double1d()
values.append(100.0)
values.append(120.0)

# Calculating the contours
contours = manualContourTask(image = myImage2, values = values)
```

Another option is to specify the minimum and maximum contour value, the number of contour levels and the distribution (linear or logarithmic, either natural or base 10), using the `AutomaticContourTask`. The task will then determine the corresponding contour values and calculate the contours. The input parameters are the following:

- the *image* (`Image image`)
- the *extreme contour values* (`Doubles min` and `max`)
- the *number of contour levels* (`Integer levels`)
- the *distribution of the contour levels* (`Integer distribution`)

Run the task as follows:

```

# For a linear distribution of the contour levels
contoursLin = automaticContour(image = myImage2, levels = 2, min = 0.0, \
 max = 255.0, distribution = 0)

# For a logarithmic distribution of the contour levels
contoursLog = automaticContour(image = myImage2, levels = 2, min = 0.0, \
 max = 255.0, distribution = 1)

# For a ln distribution of the contour levels
contourLn = automaticContour(image = myImage2, levels = 2, min = 0.0, \
 max = 255.0, distribution = 2)
```

All the results of these tasks will appear in the *Variables* view. For information on running these tasks via the HIPE graphical interface, see Section 4.2.8.

4.3.9. Histograms

You can use several tasks to create a histogram of the values of an image, or of a region within an image. Such region can be bounded by a circle, an ellipse, a rectangle or a polygon.

For all these tasks, the following input parameters must be specified:

- the *image* (`Image image`)
- the *cut levels* (`Doubles lowCut` and `highCut`)
- the *number of bins* (`Integer bins`)

For the tasks with a region of interest, the appropriate parameters must be specified:

- bounded by a *circle*:
  - the *center of the circle* in pixel (`Doubles centerX` and `centerY`) or sky coordinates (`Strings centerRA` and `centerDec`)
  - the *radius of the circle* in pixels (`Double radiusPixels`) or arcsec (`Double radiusArcsec`)
- bounded by an *ellipse*:
  - the *center of the ellipse* in pixel (`Doubles centerX` and `centerY`) or sky coordinates (`Strings centerRA` and `centerDec`)
  - the *dimensions of the ellipse* in pixels (`Doubles widthPixels` and `heightPixels`) or arcsec (`Doubles widthArcsec` and `heightArcsec`)
- bounded by a *rectangle*:
  - the *position of the corner of the rectangle with the minimal row and column* in pixel (`Doubles minX` and `minY`) or sky coordinates (`Strings minRA` and `minDec`)
  - the *dimensions of the rectangle* in pixels (`Doubles widthPixels` and `heightPixels`) or arcsec (`Doubles widthArcsec` and `heightArcsec`)
- bounded by a *polygon*:
  - the *vertices of the polygon* in pixel (`Double1d edgesPixel`, stored as `x1`, `y1`, `x2`, `y2`,...) or sky coordinates (`String1d edgesSky`, stored as `RA1`, `Dec1`, `RA2`, `Dec2`,...)

To make a histogram, follow this example:

```
# Making a histogram of an image
histogram = imageHistogram(image = myImage2, lowCut = 0.0, \
 highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by a circle
circleHistogramPixel = circleHistogram(image = myImage2, centerX = 417.5, \
 centerY = 240.0, radiusPixels = 217.6, lowCut = 9.0, highCut = 255.0, bins = 10)
circleHistogramSky = circleHistogram(image = myImage2, centerRA = -"02:00:28.319", \
 centerDec = -"-22:26:26.15", radiusArcsec = 219.3, lowCut = 9.0, \
 highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by an ellipse
```

```
ellipseHistogramPixel = ellipseHistogram(image = myImage2, centerX = 360.0, \
 centerY = 237.0, widthPixels = 642.0, heightPixels = 229.1, lowCut = 9.0, \
 highCut = 255.0, bins = 10)
ellipseHistogramSky = ellipseHistogram(image = myImage2, centerRA
= -"02:00:24.138", \
 centerDec = -"-22:26:29.22", widthArcsec = 647.136, heightArcsec = 230.9, \
 lowCut = 9.0, highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by a rectangle
rectangleHistogramPixel = rectangleHistogram(image = myImage2, minX = 211.0, \
 minY = 127.0, widthPixels = 471.0, heightPixels = 175.0, lowCut = 9.0, \
 highCut = 255.0, bins = 10)
rectangleHistogramSky = rectangleHistogram(image = myImage2, minRA
= -"02:00:13.308", \
 minDec = -"-22:28:20.17", heightArcsec = 474.8, widthArcsec = 176.4, \
 lowCut = 9.0, highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by a polygon
pyEdgesPixel = Double1d([133.0, 206.0, 247.0, 333.0, 620.0, 233.0, 487.0, 112.01])
polygonHistogramPixel = polygonHistogram(image = myImage2, \
 edgesPixel = pyEdgesPixel, lowCut = 9.0, highCut = 255.0, bins = 10)
pyEdgesSky = String1d([])
polygonHistogramSky = polygonHistogram(image = myImage2, \
 edgesSky = pyEdgesSky, lowCut = 9.0, highCut = 255.0, bins = 10)
```

Note

For each task, all dimensions must be specified in the same unit.

The dimensions can only be specified in arcsec if the `Image` has a valid `Wcs` and the pixel scaling is the same in both directions.

The following examples show how to explore the output variable, assuming it is called `histogram`:

```
# Returns the number of bins as an integer (int)
histogram.getNbOfBins()

# Returns the cut levels as a double
histogram.getLowCut()
histogram.getHighCut()


# Returns the histogram as a TableDataset
histogram.getHistogram()
# Returns the values and frequencies of the histogram as a Double1d
histogram.getValues()
histogram.getFrequencies()
# Returns the unit for the intensity
histogram.getUnit()
```

For the `CircleHistogramTask` you can also use

```
# Returns the center  of the circle in pixel (Double1d) and
# sky coordinates (String1d)
histogram.getCenterPixelCoordinates()
histogram.getCenterSkyCoordinates()

# Returns the radius of the circle in pixels and arcsec as double
histogram.getRadiusPixels()
histogram.getRadiusArcsec()
```

For the `EllipseHistogramTask` you can use

```
# Returns the center of the ellipse in pixel (Double1d)
# and sky coordinates (String1d)
histogram.getCenterPixelCoordinates()
histogram.getCenterSkyCoordinates()
```

```
# Returns the dimensions of the ellipse in pixels as double
histogram.getWidthPixels()
histogram.getHeightPixels()

# Returns the dimensions of the ellipse in arcsec as double
histogram.getWidthArcsec()
histogram.getHeightArcsec()
```

For the `RectangleHistogramTask` you can use the following:

```
# Returns the corner of the rectangle with minimal row and column in
# pixel (Double1d) or sky coordinates (String1d)
histogram.getUpperLeftCornerPixelCoordinates()
histogram.getUpperLeftCornerSkyCoordinates()

# Returns the dimensions in pixels
histogram.getWidthPixels()
histogram.getHeightPixels()

# Returns the dimensions in arcsec
histogram.getWidthArcsec()
histogram.getHeightArcsec()
```

For the `PolygonHistogramTask` you can use

```
# Returns the vertices of the polygon as a TableDataset
histogram.getEdges()

# Returns the vertices of the polygon in pixel coordinates
# as a TableDatset and Double2d
histogram.getEdgesPixelCoordinates()
histogram.getEdgesPixelCoordinatesDouble2d()

# Returns the vertices of the polygon in sky coordinates as a TableDataset
histogram.getEdgesSkyCoordinates()
```

For information on creating histogram via the HIPE graphical interface, see Section 4.2.9.

4.3.10. Aperture photometry

You can do aperture photometry in two ways:

- With a circular target aperture and an annular or a rectangular sky aperture
- With a circular target aperture and a fixed value for the sky intensity

Circular target aperture and annular sky aperture

Use the `AnnularSkyAperturePhotometryTask` task.

The input parameters you need are :

- the *image* (`Image image`)
- the *target center* either in pixel (`Doubles centerX` and `centerY`) or sky coordinates (`Strings centerRA` and `centerDec`)
- the *target radius* either in pixels (`Double radiusPixels`) or in arcsec (`Double radiusArcsec`)
- the *inner and outer radii of the annular sky aperture* either in pixels (`Doubles innerPixels` and `outerPixels`) or arcsec (`Doubles innerArcsec` and `outerArcsec`)

- the *kind of pixels* (entire/fractional) used (`Boolean fractional` (optional - per default : `True`))
- the *sky estimation algorithm* (`Integer algorithm`)

To perform aperture photometry, type the following:

```
# The target center specified in pixel coordinates, the radii in pixels
# and using fractional pixels
photPixels = annularSkyAperturePhotometry(image = myImage2, centerX = 430.0, \
 centerY = 467.0, radiusPixels = 5.0, innerPixels = 20.0, outerPixels = 40.0, \
 fractional = 1, algorithm = 4)

# The target center specified in sky coordinates, the radii in arcsec
# and using entire pixels
photSky = annularSkyAperturePhotometry(image = myImage2, \
 centerRA = -"02:00:29.214", centerDec = -"-22:33:37.32", radiusArcsec = 5.04, \
 innerArcsec = 20.16, outerArcsec = 40.32, fractional = 0, algorithm = 4)
```

Note

You can only specify distances in arcsec (here `radiusArcsec`, `innerArcsec` and `outerArcsec`, if the pixel scaling is the same in both directions (**myImage2.getCdelt1() = myImage2.getCdelt2()**). Moreover, the `Image` must have a valid `Wcs`.

All distances must be specified in the same unit, either pixels or arcsec.

You can choose between five sky estimation algorithms: average, median, mean-median, synthetic mode and the algorithm used by Daophot. Here is how:

```
# Using the average sky estimation algoritm
photAverage = annularSkyAperturePhotometry(image = myImage2, centerX = 430.0, \
 centerY = 467.0, radiusPixels = 5.0, innerPixels = 20.0, outerPixels = 40.0, \
 algorithm = 0)

# Using the median sky estimation algorithm
photMedian = annularSkyAperturePhotometry(image = myImage2, centerX = 430.0, \
 centerY = 467.1, radiusPixels = 5.0, innerPixels = 20.0, outerPixels = 40.0, \
 algorithm = 1)

# Using the mean-median sky estimation algorithm
photMeanMedian = annularSkyAperturePhotometry(image = myImage2, centerX = 430.0,\
 centerY = 467.0, radiusPixels = 5.0, innerPixels = 20.0, \
 outerPixels = 40.0, algorithm = 2)

# Using the synthetic mode sky estimation algorithm
photSyntheticMode = annularSkyAperturePhotometry(image = myImage2, \
 centerX = 430.0, centerY = 467.0, radiusPixels = 5.0, innerPixels = 20.0, \
 outerPixels = 40.0, algorithm = 3)

# Using the Daophot sky estimation algorithm
photDaophot = annularSkyAperturePhotometry(image = myImage2, centerX = 430.0, \
 centerY = 467.0, radiusPixels = 5.0, innerPixels = 20.0, outerPixels = 40.0, \
 algorithm = 4)
```

All these output products will appear in the *Variables* view in HIPE.

You can inspect the output product with the following commands:

```
# Returns target center in pixel (as Double1d) and sky coordinates (as String1d)
phot.getTargetCenterPixelCoordinates()
phot.getTargetCenterSkyCoordinates()

# Returns the radii in pixels as Doubles
phot.getTargetRadiusPixels()
phot.getInnerRadiusPixels()
phot.getOuterRadiusPixels()

# Returns the radii in arcsec as Doubles
phot.getTargetRadiusArcsec()
phot.getInnerRadiusArcsec()
phot.getOuterRadiusArcsec()

# Returns the sky estimation algorithm
phot.getAlgorithm()

# Returns the kind of pixels used as a String
phot.getPixels()

# Returns the results table as a TableDataset and as a Double2d
phot.getTable()
phot.getDouble2dTable()

# Returns the total flux (Double1d), number of pixels (Double),
# intensity per pixel (Double) and error on the flux (Double) for the target,
# including the sky
phot.getTargetPlusSkyTotal()
phot.getTargetPluxSkyPixels()
phot.getIntensityPerTargetPlusSkyPixel()

# To return the same for the sky and the target without the sky, simply replace
# -"TargetPlusSky" with -"Sky" or -"Target"

# Returns the curve of growth as a TableDataset and the corresponding radius
# and flux as Double1ds
phot.getCurveOfGrowth()
phot.getGrowthRadius()
phot.getGrowthFlux()

# Returns the sky intensity plot as a TableDataset and the corresponding radius
# and intensity as Double1ds
phot.getSkyIntensityPlot()
phot.getSkyIntensityRadius()
phot.getSkyIntensity()
```

Circular target aperture and rectangular sky aperture

The immediate neighbourhood of the target is not always the best location to estimate the sky. Then you better take a rectangular region a bit further away from the target. This can be done with the `RectangularSkyAperturePhotometryTask`.

The input parameters are :

- the *image* (`Image image`)
- the *target center* either in pixel (`Doubles centerX` and `centerY`) or sky coordinates (`Strings centerRA` and `centerDec`)
- the *target radius* either in pixels (`Double radiusPixels`) or arcsec (`Double radiusArcsec`)
- the *position of the corner of the rectangle with minimal row and column*, either in pixel (`Doubles minX` and `minY`) or in sky coordinates (`Strings minRA` and `minDec`
- the *dimensions of the rectangle* either in pixels (`Doubles widthPixels` and `heightPixels`) or arcsec (`Doubles widthArcsec` and `heightArcsec`)
- the *kind of pixels* (entire/fractional) used (`Boolean fractional` (optional - per default : `True`))
- the *sky estimation algorithm* (`Integer algorithm`)

To perform aperture photometry, just type

```
# The target center is specified in pixel coordinates, the target radius in pixels
photPixel = rectangularSkyAperturePhotometry(image = myImage2, centerX = 501.0,\
 centerY = 266.0, radiusPixels = 5.0, minX = 553.0, minY = 132.0, \
 widthPixels = 120.0, heightPixels = 47.0, algorithm = 4)

# The target center is specified in sky coordinates, the target radius in arcsec
photSky = rectangularSkyAperturePhotometry(image = myImages2, \
 centerRA = -"02:00:34.388", centerDec = -"-22:25:59.87", radiusArcsec = 5.04, \
 minRA = -"02:00:38.179", minDec = -"-22:28:14.89", widthArcsec = 120.96, \
 heightArcsec = 47.376)
```

Note

The same remarks hold as for `AnnularSkyAperturePhotometryTask`.

The target center and the corner of the rectangle with minimal row and column must be specified in the same coordinates (pixel/sky).

Choosing the kind of pixels and the sky estimation algorithm can be done as for the `AnnularSkyAperturePhotometryTask`.

All these output products will appear in the *Variables* view in HIPE.

To inspect the output product via the command line, you can use the same commands as for the `AnnularSkyAperturePhotometryTask`, except for those referring to the annular sky aperture. To obtain information about the rectangular sky aperture, use these commands:

```
# Returns the dimensions of the rectangle in pixels
phot.getWidthPixels()
phot.getHeightPixels()

# Returns the dimensions of the rectangle in arcsec
phot.getWidthArcsec()
phot.getHeightArcsec()

# Returns the corner of the rectangle with minimal row and
# column in pixel and sky coordinates
phot.getUpperLeftCornerPixelCoordinates()
phot.getUpperLeftCornerSkyCoordinates()
```

Circular target aperture and a fixed sky value

Sometimes you might have already determined a good value for the sky, so you want to use that. This can be done with the `FixedSkyAperturePhotometryTask`.

On the command line

The input parameters are :

- the *image* (`Image image`)
- the *target center* in pixel (`Doubles centerX` and `centerY`) or sky coordinates (`Strings centerRA` and `centerDec`)
- the *target radius* in pixels (`Double radiusPixels`) or arcsec (`Double radiusArcsec`)
- the *sky intensity value* (`Double sky`)
- the kind of pixels (entire/fractional) used (`Boolean fractional` (optional - per default : `True`))

To perform aperture photometry, just type

```
# The target center is specified in pixel coordinates, the target
# radius in pixels
photPixels = fixedSkyAperturePhotometry(image = myImage2, centerX = 499.0, \
 centerY = 566.0, radiusPixels = 5.0, sky = 48.0)
# The target center is specified in sky coordinates, the target radius in arcsec
photSky = fixedSkyAperturePhotometry(image = myImage2, centerRA = -"02:00:34.242",\
 centerDec = -"-22:25:59.87", radiusArcsec = 5.04, sky = 48.0)
```

Note

The target radius can only be specified if the `Image` has a valid `Wcs` and the pixel scaling is the same in both directions.

The following are two additional methods to inspect the result product, here called `phot`:

```
phot.getSkyValue()
phot.getIntensityPerSkyPixel()
```

4.3.11. Mosaicking

This task is not yet integrated in HIPE, so it is only available from the command line. The input parameters are:

- a *list with images you want to combine* (`ArrayList<Image> images`)
- *oversampling* (`Boolean oversampling`) - optional (`True` by default)

To combine `n Images`, say `image_1` to `image_n`, to a mosaic, type the following:

```
# Imports
from java.util import ArrayList
from herschel.ia.toolbox.image import MosaicTask

# Making an ArrayList with the Images
images = ArrayList()
images.add(image_1)
...
images.add(image_n)


# Making an oversampled mosaic
mosaicOversampled1 = MosaicTask()(images = images, oversample = 1)
mosaicOversampled2 = MosaicTask()(images = images)

# Making a non-oversampled mosaic
mosaicNonOversampled = MosaicTask()(images = images, oversample = 0)
```

The result, `mosaic`, is a `SimpleImage` and can be treated like any other `Image`.

Chapter 5. Spectral analysis

5.1. Summary

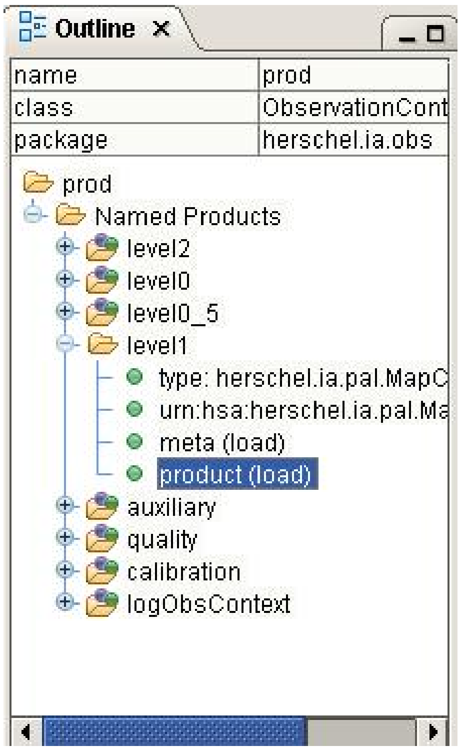
This chapter describes the following topics:

- Tasks for spectral arithmetics.
- The SpectrumFitter toolbox.
- The standing wave removal tool.
- The baseline smoothing and line masking tool.
- Creating a spectral cube.
- The cube spectrum analysis toolbox.
- Fitting spectra from the command line.

5.2. How to

5.2.1. Starting example: dataset of HIFI spectra

It is assumed that an observation product containing spectral data is available and active within your HIPE session. For this chapter, we will have an active variable called `prod` which is a HIFI observation downloaded from the HSA (see Section 1.3). This contains several levels of data processing. We will be dealing with level1 data: double-click on the highlighted "product(load)" in Figure 5.1. The results appear in a new Editor window and include some metadata on the product plus (scrolling down) a set of associated products (see Figure 5.2). Clicking on the highlighted "summary" will provide a list of what datasets are contained for apid=1030 (the WBS spectrometer H polarization). In the particular case (a Double Beam Switch observation) we are using we see that there a comb (frequency calibration measurement), a hot-cold internal calibrator measurement (hc), a tuning measurement (other) and two science measurements datasets for ON and OFF target (datasets 4 and 5). We will pick out dataset 4 for our purposes (double-click highlighted "product(load)" gives Figure 5.3). This produces a list of metadata for the selected product and a dataset (with green dot beside it) at the bottom of another Editor window. Drag-and-drop the dataset to the "Variables" view and this dataset is automatically given a name in the session -- typically "newVariable."

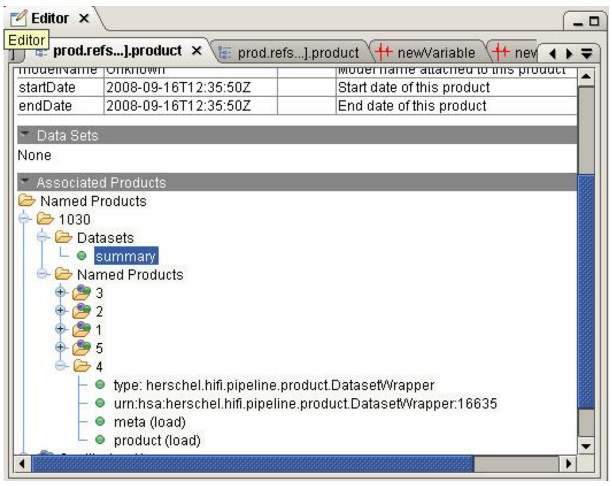


Outline

name	prod
class	ObservationCont
package	herschel.ia.obs

```
prod
  Named Products
    level2
    level0
    level0_5
    level1
      type: herschel.ia.pal.MapC
      urn:hsa:herschel.ia.pal.Ma
      meta (load)
      product (load)
    auxiliary
    quality
    calibration
    logObsContext
```

Figure 5.1. Selecting Level 1 data from a downloaded archive observation done by HIFI.



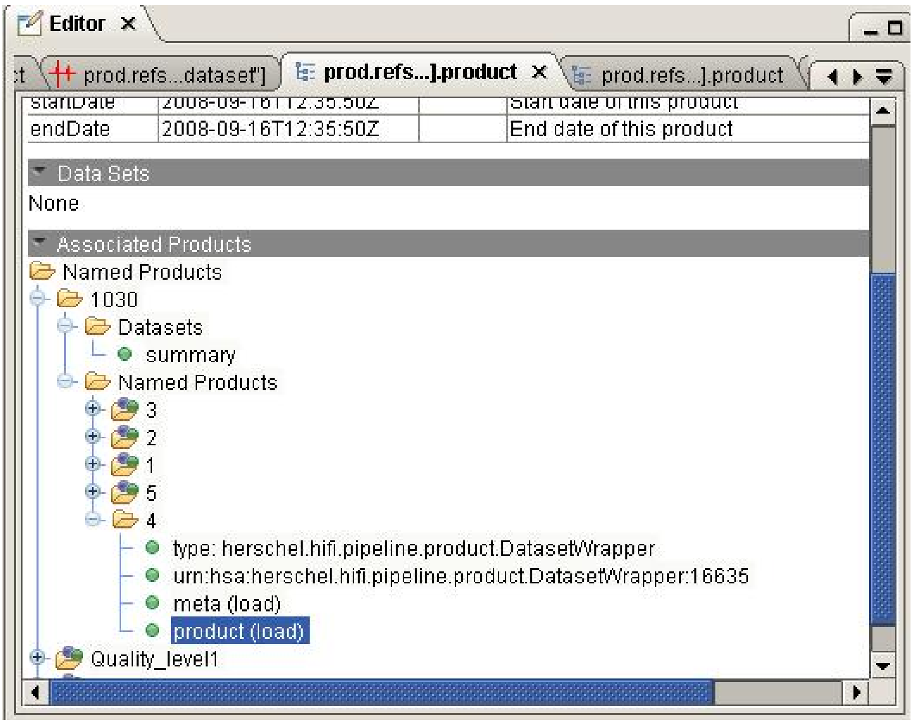
modelName	Unknown		Model name attached to this product
startDate	2008-09-16T12:35:50Z		Start date of this product
endDate	2008-09-16T12:35:50Z		End date of this product

Data Sets
None

Associated Products

```
Named Products
  1030
    Datasets
      summary
    Named Products
      3
      2
      1
      5
      4
        type: herschel.hifi.pipeline.product.DatasetWrapper
        urn:hsa:herschel.hifi.pipeline.product.DatasetWrapper:16635
        meta (load)
        product (load)
```

Figure 5.2. Display of product set.



startDate	2008-09-16T12:35:50Z		Start date of this product
endDate	2008-09-16T12:35:50Z		End date of this product

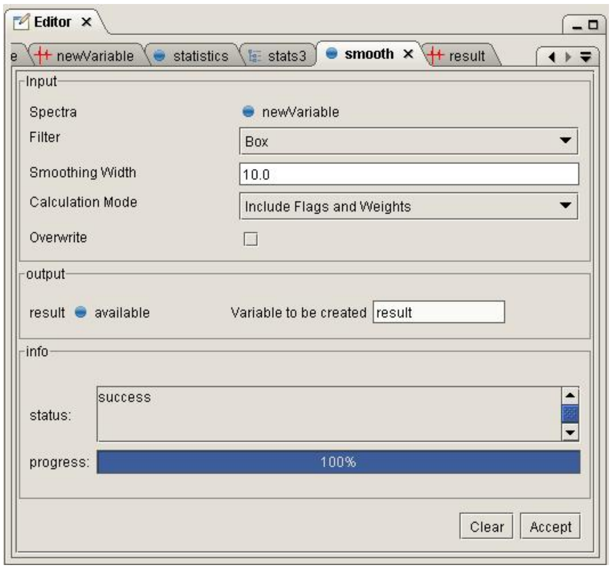
```
Data Sets
None
Associated Products
Named Products
  1030
    Datasets
      summary
    Named Products
      3
      2
      1
      5
      4
        type: herschel.hifi.pipeline.product.DatasetWrapper
        urn:hsa:herschel.hifi.pipeline.product.DatasetWrapper:16635
        meta (load)
        product (load)
Quality_level1
```

Figure 5.3. Choosing the product with the dataset we want.

A double-click on newVariable in the "Variables" view will open the dataset using the `SpectrumExplorer` (see HowTo on Spectral Display for information on how to manipulate the visualization). In the example dataset used here there are 18 spectra.

5.2.2. Spectrum arithmetics

You can open the tasks described in this section by clicking on a spectrum in the *Variables* view and opening the *Applicable* folder in the *Tasks* view.



Input

Spectra	newVariable
Filter	Box
Smoothing Width	10.0
Calculation Mode	Include Flags and Weights
Overwrite	☐

output

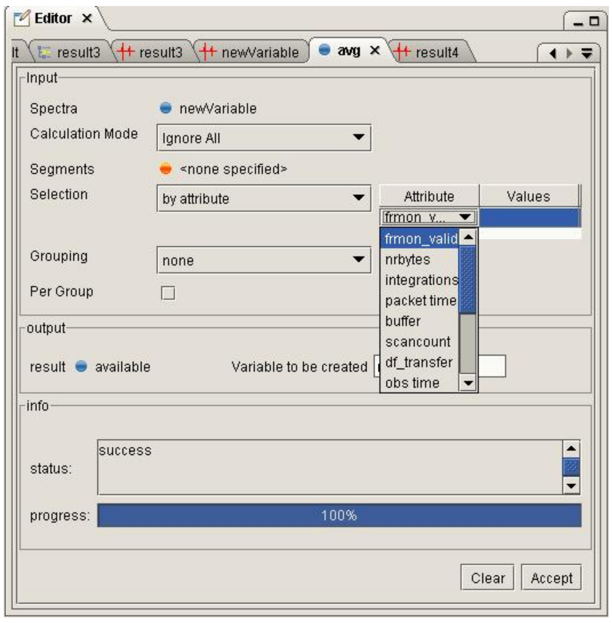
result available Variable to be created result

info

status: success

progress: 100%

Figure 5.6. Using the `smooth` task



Input

Spectra ● newVariable

Calculation Mode: Ignore All

Segments ● <none specified>

Selection: by attribute

Attribute	Values
frmon_v...	

Grouping: none

Per Group ☐

output

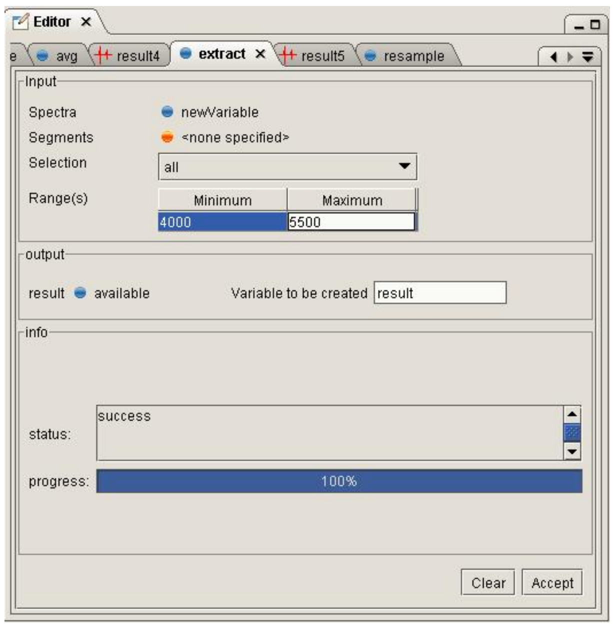
result ● available Variable to be created

info

status: success

progress: 100%

Figure 5.7. Using the `avg` task



Input

Spectra ● newVariable

Segments ● <none specified>

Selection: all

Range(s)

Minimum	Maximum
4000	5500

output

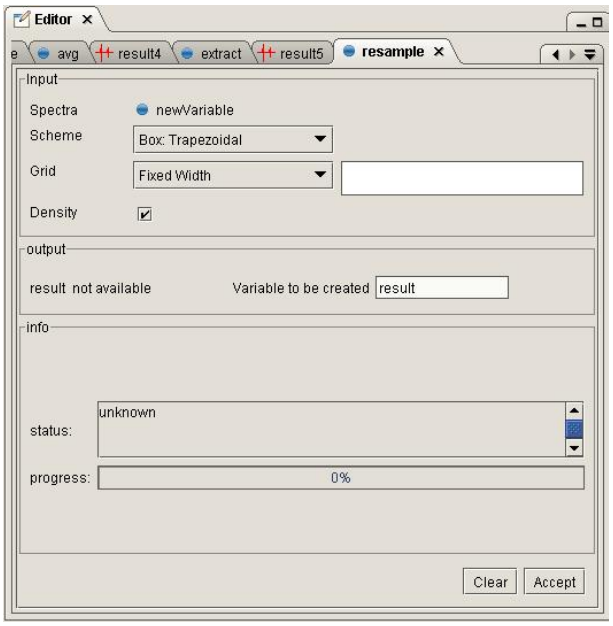
result ● available Variable to be created: result

info

status: success

progress: 100%

Figure 5.8. Using the `extract` task

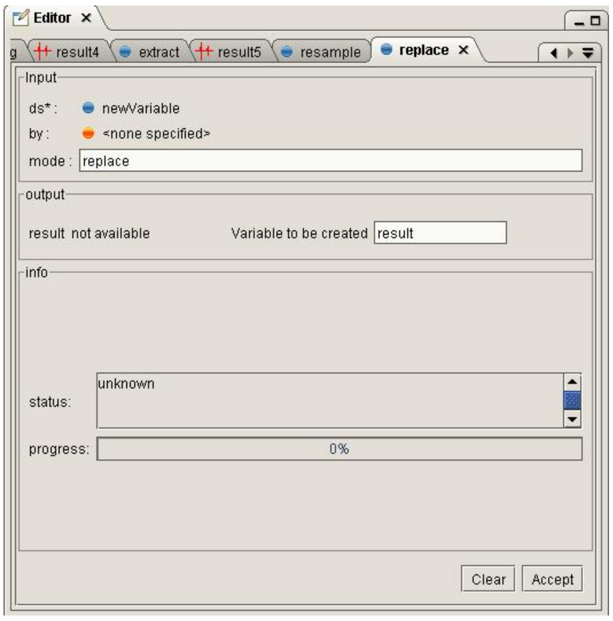


Input
Spectra newVariable
Scheme Box: Trapezoidal
Grid Fixed Width
Density ✓

output
result not available
Variable to be created result

info
status: unknown
progress: 0%

Figure 5.9. Using the `resample` task



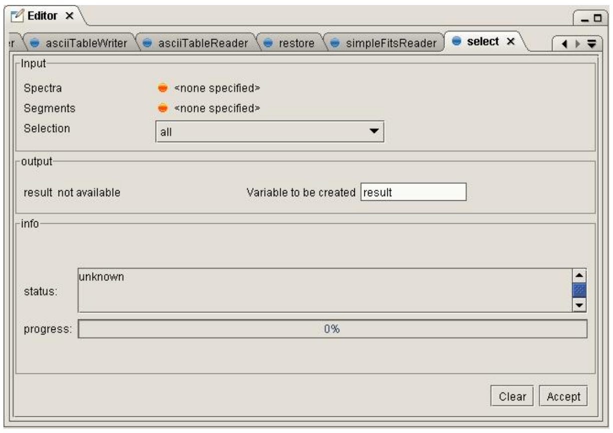
Input
ds* : newVariable
by : <none specified>
mode : replace

output
result not available
Variable to be created result

info
status: unknown
progress: 0%

Figure 5.10. Using the `replace` task

- *select*: Provides a means of selecting those spectra that can be combined. A given attribute value or range of values can be used or simply the index number of the spectrum within the group (see Figure 5.4).



Input

Spectra <none specified>
Segments <none specified>
Selection all

output

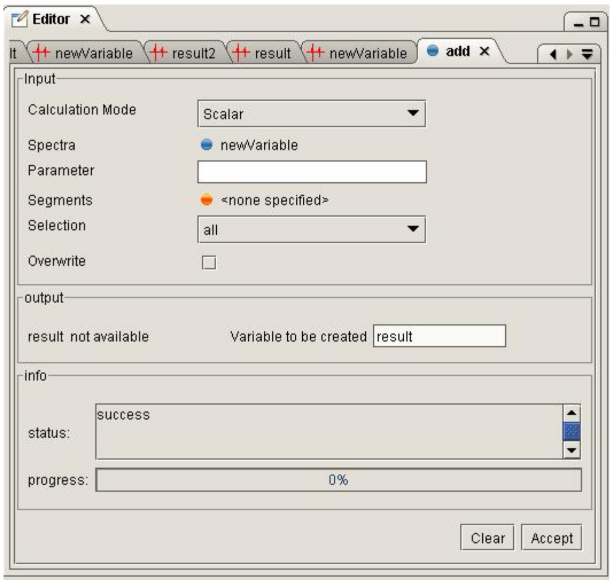
result not available
Variable to be created result

info

status: unknown
progress: 0%

Figure 5.4. Using the `select` task

- *add/subtract/multiply/divide*: Provide means of adding/subtracting/multiplying/dividing groups of spectra or single spectra together (pair-wise), or adding/subtracting/multiplying/dividing a scalar value to/from all spectra in the selected dataset. Numbered segments, e.g., subbands, can be selected for addition if available within the dataset (see Figure 5.5 for adding the scalar value 200.0 to all spectra in our dataset)



Input

Calculation Mode Scalar
Spectra newVariable
Parameter
Segments <none specified>
Selection all
Overwrite ☐

output

result not available
Variable to be created result

info

status: success
progress: 0%

Figure 5.5. Using the `add` task

- *statistics* This allows for statistical operations to be performed on the datasets (it automatically works on individual sub-bands presently). It provide as mean, median, variance, standard deviation or percentiles for samples / selections of spectra from a dataset that can contain many datasets (spectra) when the "Accept" button is clicked. The result is an output that contains a number of datasets holding statistical information on the datasets. The main output is the "summary" table that is typically the last dataset listed of the set (double-clisk on output variable, e.g., "stats", in the Variables view. Use an appropriate viewer (Dataset viewer or Tableplotter to see the results).

- *smooth*This allows a transformation of the data via a box or gaussian (of user-selected width) smooth of the spectra in a dataset. Flags and weights for the different spectral points can be added in the future. To run this tool, click on the dataset, e.g., "newVariable", in the "Variables" view to highlight. The Applicable Tasks in the "Tasks" view include `smooth`. Double-click on this to get the self-explanatory dialog shown in Figure 5.6. The task runs by hitting the "Accept" button.

- *avg*This allows the average of a selection of spectra from a dataset. Flags and weights for individual channels/pixels can be used if available. Spectra can be selected by their index number in the dataset or by attributes (such as buffer number -- a pull-down selection list is available.). To run this tool, click on the dataset, e.g., "newVariable", in the "Variables" view to highlight. The Applicable Tasks in the "Tasks" view include `avg`. Double-click on this to get the self-explanatory dialog shown in Figure 5.7. The task runs by hitting the "Accept" button.

- *extract*This allows the extraction of a data from a minimum to a maximum frequency/wavelength range for the complete set of spectra in a dataset. Flags and weights for individual channels/pixels can be used if available. Spectra can also be selected by their index number in the dataset or by attributes (such as buffer number -- a pull-down selection list is available.). To run this tool, click on the dataset, e.g., "newVariable", in the "Variables" view to highlight. The Applicable Tasks in the "Tasks" view include `extract`. Double-click on this to get the self-explanatory dialog shown in Figure 5.8, where the channels with frequencies 4000 to 5500 MHz have been selected. The task runs by hitting the "Accept" button.

- *resample*This allows the resampling of data using a Trapezoidal or Euler box, with a choice of variable or fixed width. Flags and weights for individual channels/pixels can be used if available. Spectra can also be selected by their index number in the dataset or by attributes (such as buffer number -- a pull-down selection list is available.). To run this tool, click on the dataset, e.g., "newVariable", in the "Variables" view to highlight. The Applicable Tasks in the "Tasks" view include `resample`. Double-click on this to get the self-explanatory dialog shown in Figure 5.9. The task runs by hitting the "Accept" button.

- *replace*This allows the replacement of certain frequency/wavelength channels. To run this tool, click on the dataset, e.g., "newVariable", in the "Variables" view to highlight. The Applicable Tasks in the "Tasks" view include `replace`. Double-click on this to get the dialog shown in Figure 5.10. The task runs by hitting the "Accept" button.

5.2.3. The SpectrumFitter Toolbox

The SpectrumFitter Toolbox (or SFTool), is used to fit spectral features in the data of all three Herschel instruments. The Toolbox is launched from the Tasks menu, or the Applicable Tasks menu if you have already selected a SpectrumDataset, or by right-clicking on the SpectrumDataset and selecting "Open with..." SFTool. The latter two will open the SFTool with the data loaded in.

Note that NaNs must be removed from the spectra before fitting. In HCSS 2.0 this is done by SFTool.

5.2.3.1. Viewing Spectra in the SpectrumFitter Toolbox GUI

The SFTool GUI is most easily handled if you drag it out of the editor pane and expand it so you can see the whole GUI.

Select the spectrum from the Container environment (lower-left pane of SFTool - middle tab) by right-clicking on the spectrum of your choice and clicking "Select for Fitting".

The spectrum to be fit then appears in the upper panel of the plotter and in the Spectra tab, the default name for the first spectrum selected is `S1`. This name also appears in the text box above the plotter.

After `S1` is fit, the residual will be plot in the panel below and a temporary name for the residual appears in the text box below.

5.2.3.2. Fitting models to spectra

Single model fits

- Press the *InitModel* button in the right pane of SFTool to initialise the models toolbox.

- The default model is a first order polynomial fit, which is based on the first and last 10% of the data points of the spectrum.
- You can improve on this first guess by selecting a higher order polynomial via the buttons (0-4) in the upper right pane or entering a value in the box. The polynomial coefficients are specified as the first column in the text boxes below the choice of polynomial degree. The second column is used to report the standard deviations on these coefficients.
- You can also select a different model to fit (e.g. Gaussian) from the drop down menu.
- Windows to fit in can be selected via manually entering values in the windows boxes, or by clicking first in a window box (it should turn yellow) and drawing a window on the spectrum with the cursor.
- Once a satisfactory model is found, you can apply it by clicking "OK" in the model pane (upper right) and then "DoFit" in the fitting pane (lower right).
- If the fit is good, click "AcceptFit" (in the lower right panel) and the residual spectrum, S1_R01, is passed to the Spectra tab (lower left). This tab will now hold two spectra: S1 and S1_R01.
- To apply another fit model on S1, click InitModel again. To apply a fit model to the residual spectrum S1_R01, select this spectrum in the Spectra tab and click UseSpectrum in the lower right panel.

Multiple Model Fits

- To fit another model to S1, press the InitModel button again. A new default 1st degree polynomial model will be proposed. Same strategy as above: select the type you want, specify the parameters, specify the windows, click OK and DoFit.
- Adding this second model will return a simultaneous fit of S1 using both models you applied to this spectrum. Assuming you first fit a polynomial and afterwards you fit a Gaussian profile to your spectrum (S1), this means that the coefficients of the polynomial will be updated. You can retrieve these new coefficients by clicking on the appropriate model name in the Models tab (lower left).
- Fitting e.g. a Gaussian to S1_R01 will NOT update the model coefficients used to get from S1 to S1_R01.
- **Global fit.** A global fit is more mathematically precise than separate model fits. To apply a global fit using all the models applied to S1, click OK, DoFit and AcceptFit after specifying your last fitting model. Another spectrum will appear in the Spectra tab, called S1_R01_R01. Right-click on this spectrum and select GlobalFit to simultaneously fit all models to S1. This will create a new spectrum, called S1+1, in the Spectra tab.
- Selecting S1+1 will show you the spectrum and all applied models over-plotted in the upper plot panel. The residuals are shown in the lower panel.

5.2.3.3. Sending results back to HIPE

To export the spectrum of your choice (and the applied models) as a HIPE variable, right-click on the spectrum and click Save to HIPE. A pop-up window will ask you for the name of the variable.

After applying models to your spectrum, you can retrieve your GUI actions as a Jython-script, by right-clicking on the end result (AFTER AcceptFit), and selecting Write Script. A pop-up window will ask you where to save the script. (Remember to put .py at the end of the file name.)

5.2.4. General Standing Wave Removal Tool

5.2.4.1. Introduction FitFringe

FitFringe is a general sine-wave fitting task that can be used to remove periodic signals in spectra, such as standing waves. A description of the method and history of the code can be found in Kester et al. ("The Calibration Legacy of the ISO Mission", 2003, ESASP 481, 375).

Briefly, FitFringe does the following:

1. A baseline for the signal to be fitted is determined by using the SmoothBaseline task. Sharp spectral features are flagged using a sigma clipping algorithm. The user can control the baseline shape by indicating a typical period ('midcycle') that is being searched for.

2. Single sine waves are fitted to the baseline-subtracted spectrum, over a wide range of periods. Best-fitting periods are determined from local or absolute minimum Chi-square points.

3. The sine-wave amplitudes and phases are determined by solving a set of linear equations using the 'LU' matrix decomposition method.

4. The solution is subtracted from the data and the baseline is added back in.

The user can fit any number of sine waves to the data. In the future an option will be added to determine the minimum amount of sine waves needed to fit the data, taking into account the spectral noise using Bayesian statistics. As this is not an instrument-specific task, the input data have to be in the general SpectralSegment format. The assumed wavelength units are micron. See the HIFI Standing Wave Removal Tool chapter in the HIFI User Manual for using FitFringe with HIFI data.

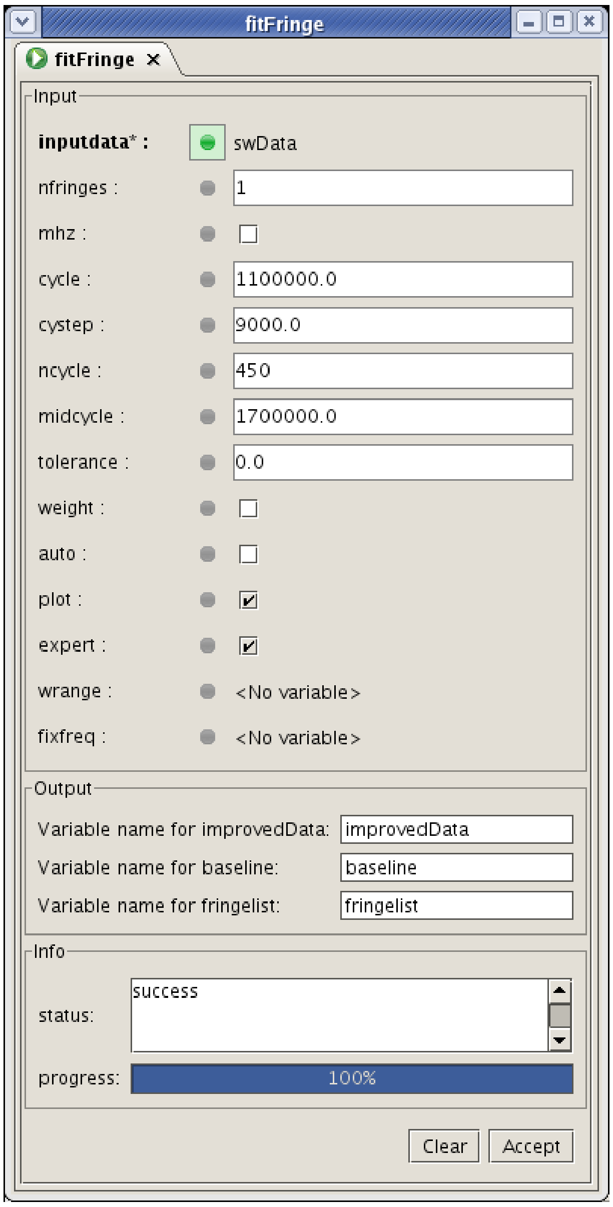
5.2.4.2. Running FitFringe

FitFringe only accepts data in the SpectralSegment format. Required are frequency in micron (Double1d), flux (Double1d), flags (Int1d), and weights (double1d). The input data for FitFringe is then created as follows:

```
swData = FitFringeData(myFreq,myFlux,myFlag,myWeight)
```

FitFringe can be run on the command line and with a GUI.

The latter looks as follows:



fitFringe

fitFringe

Input

inputdata* :	swData
nfringes :	1
mhz :	☐
cycle :	1100000.0
cystep :	9000.0
ncycle :	450
midcycle :	1700000.0
tolerance :	0.0
weight :	☐
auto :	☐
plot :	✓
expert :	✓
wrange :	<No variable>
fixfreq :	<No variable>

Output

Variable name for improvedData:	improvedData
Variable name for baseline:	baseline
Variable name for fringelist:	fringelist

Info

status:	success
progress:	100%

Figure 5.11. The FitFringe task interface

Clicking on 'Accept' assumes the defaults further explained below. It is equivalent to the command line statements:

```
hf = FitFringe()
```

```
improvedData = hf(swData)
```

In the process, two plots are created by default. The following plots were created using the script listed in the box below. The first one shows the sine wave period as a function of Chi^2. Selected dips with minimum Chi^2 are indicated with vertical red lines.

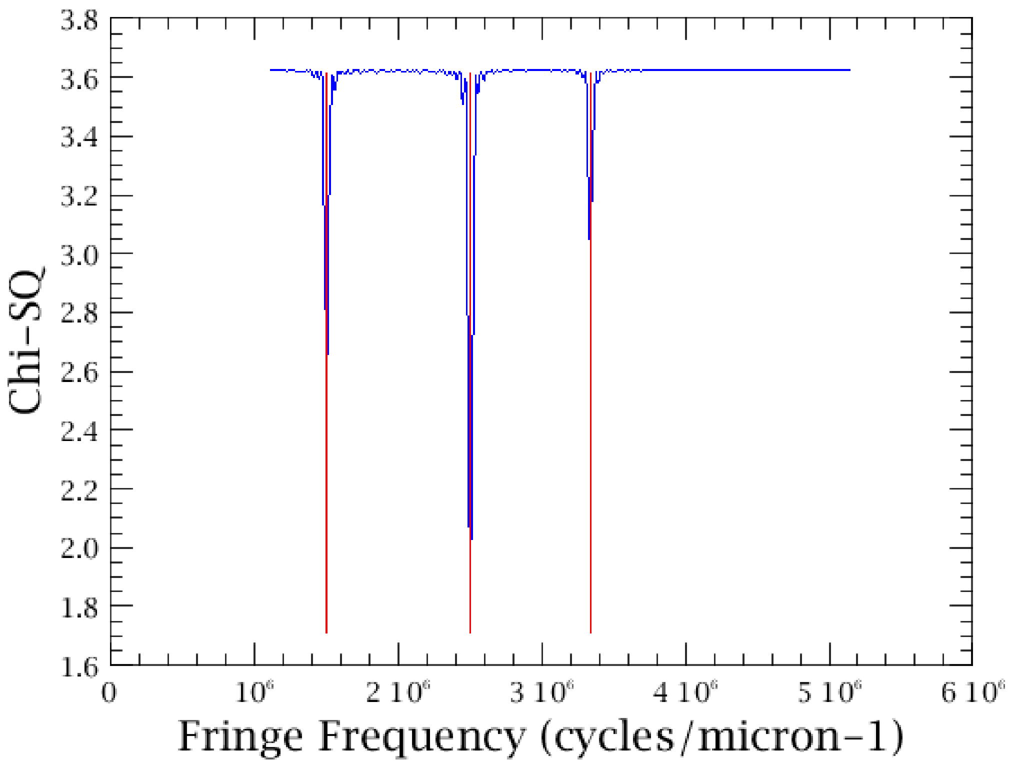


Figure 5.12. Sine wave period plot

The second plot shows the original data, the baseline, the sine-wave subtracted data, and the mask:

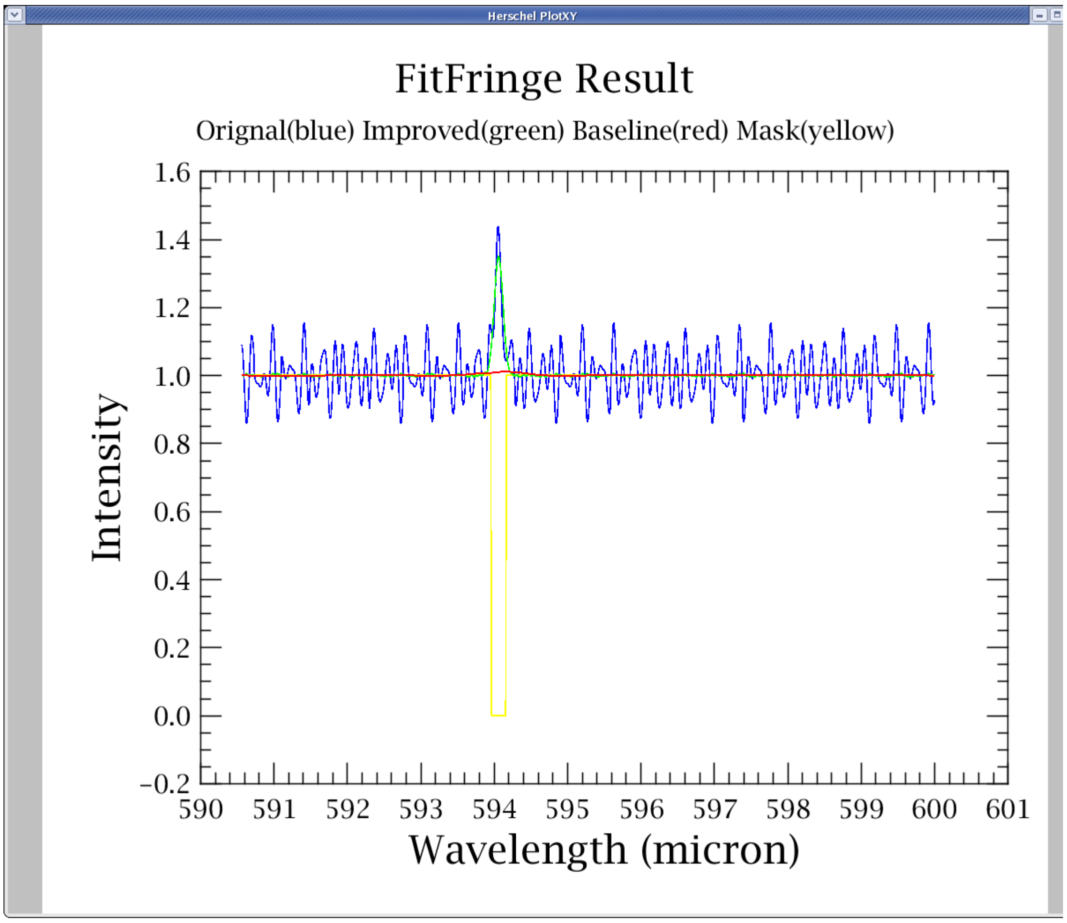


Figure 5.13. FitFringe result plot

The output data with the sine waves subtracted can be retrieved as follows:

`wave=improvedData.wave`

`flux=improvedData.flux`

`flag=improvedData.flag`

`weight=improvedData.weight`

The applied baseline is stored in a similar way.

The fitted parameters are stored in a TableDataset, which contains a list of the fitted sine waves:

```
fringeNum: fringe number
cycle: period [per inverse wavenumber in micron]
cycle_In_MHz: period [in MHz]
sinAmp:  amplitude of sine component
cosAmp:  amplitude of cosine component
chisq:   chi^2
chiRed:  total chi^2 reduction
```

The list can be viewed as

`f=hf.fringelist`

`print f`

For example, the sine wave periods in MHz are retrieved as

```
print f.getColumn("cycle_In_MHz")
```

As the GUI shows, several parameters can be controlled by the user. Periods are by default expressed in units of cycles per inverse wavenumber in micron, unless the 'mhz' boxed is checked. In order of importance:

- nfringes: number of sine waves to be fitted [DEFAULT: 1]
- midcycle: typical cycle frequency used for smoothing in order to determine the baseline [DEFAULT: 1.7E6 cycles/micron^-1=176 MHz]
- cycle: start of sine wave period search range [DEFAULT: 1.1E6 cycles/micron^-1=2727 MHz]
- plot: show results in plots [DEFAULT: a period versus Chi^2 plot and a before/after plot]
- expert: show more plots of intermediate steps [DEFAULT: not]
- fixfreq: fix periods to these values, i.e. do not search for them. Has to be same number as nfringes [DEFAULT: search for periods].
- ncycle: number of cycles to check [DEFAULT: 450]
- cystep: step between cycles, i.e. resolution of the frequency space to search for standing waves [DEFAULT: 9000 cycles/micron^-1--unlikely to be modified by the user]
- weight: set all weights to 1 [DEFAULT: assign smaller weights to outliers]
- wrange: limit operations to wavelengths within this range (in micron). [not yet implemented]
- tolerance: reduce chi^2 until reduction is less than tol (0.01 == 1 percent) [not yet implemented]
- auto: automatically determine the maximum number of fringes needed within the noise, using Bayesian statistics [not yet implemented]

This example shows how FitFringe can be used. It is the script used to produce the plots shown above.

```
#frequency in GHz (FitFringe assumes the periods are
#constant in frequency space)
myFreq=Double1d.range(800)/100.+500

#flag and weights
myFlag=Int1d(800)
myWeight=Double1d(800)+1.

#sum of 90, 120, and 200 MHz standing waves
#and a Gaussian emission line
sw_freq1=90
myFlux=SIN(2*Math.PI*myFreq/(sw_freq1*1.e-3))*0.04+1.0
sw_freq2=120.
myFlux=myFlux*(1.+SIN(2*Math.PI*myFreq/(sw_freq2*1.e-3))*0.07)
sw_freq3=200.
myFlux=myFlux*(1.+SIN(2*Math.PI*myFreq/(sw_freq3*1.e-3))*0.05)
myFlux=myFlux+0.35*EXP(-0.5 * (( myFreq -- 505. -) -/ 0.05 -)**2 -)

#fitFringe expects wavelength in micron
myFreq=(3.e14/(((myFreq))*1.e9))

# Make the input standingwave data
swData = FitFringeData(myFreq,myFlux,myFlag,myWeight)

# Run FitFringe
```

```
hf = FitFringe()
improvedData = hf(swData,nfringes=3)

#output data will be in
# improvedData.wave
# improvedData.flux
# improvedData.flag
# improvedData.weight

# Check the fringe list (a TableDataset)
f=hf.fringelist
print f
print f.getColumn("cycle_In_MHz")
```

5.2.5. Baseline Smoothing and Line Masking Tool

5.2.5.1. Introduction SmoothBaseline

The SmoothBaseline task produces a smooth baseline and a mask of spectral features with no (or very little) user interaction. It works by smoothing, median filtering, and clipping the spectrum a number of times. Spectral lines are masked and any standing waves are smoothed over. Both the smooth baseline and the mask are returned to the user. Although SmoothBaseline was originally developed for use with the FitFringe sine wave fitting routine, it can be used on its own as well, for example for automated baseline and line detection purposes.

5.2.5.2. Running SmoothBaseline

SmoothBaseline accepts SpectralSegments as input. See the example in the box below for how to construct those. The box also shows how SmoothBaseline can be run from the command line. A GUI can also be started by double-clicking on the SpectralSegment in the Variables window.

The key input parameter is 'midcycle', which is essentially the typical scale to which the baseline is to be smoothed. It's unit is the number of cycles per wavenumber unit, where wavenumber is defined as 1/wavelength. Any structure in the spectrum that has a much longer period than 'midcycle' is considered baseline structure and will not be smoothed or masked.

After applying median filter with width 'midcycle', a boxcar smoothing with 10 times the width of midcycle is done to determine outliers larger 4 times the difference between the smoothed and input spectrum. The default box-car value of 10 can be overruled by the user, although this is likely rarely needed.

The user can also mask spectral regions a priori, by using the 'suppressBegin' and 'suppressEnd' parameters.

In the box below, the output from SmoothBaseline is explained. By default two plots are generated, but this can be avoided by entering 'plot=False'.

Example script:

```


#make a test spectrum

#wavelength in micron
myWave=Double1d.range(800)/100.+500

#flag and weights
myFlag=Int1d(800)
myWeight=Double1d(800)+1.
```

```
#a standing wave with a wavelength of 0.5 micron
#and a Gaussian emission line
sw_wl=0.5
myFlux=SIN(2*Math.PI*myWave/(sw_wl))*0.04+1.0
myFlux=myFlux+0.35*EXP(-0.5 * (( myWave -- 505. -) -/ 0.05 -)**2 -)

# Prepare spectrum data to be processed
swData = FitFringeData(myWave,myFlux,myFlag,myWeight)

# Run SmoothBaseline. Note that the exact value of midcyc is not
# very important, though it should be of the same order of magnitude
# as the waves in the spectrum. Here, 7.e5 cyc/micron^-1
# corresponds to waves with lengths of lambda^2/midcyc=0.35 micron
baseline = smoothBaseline(data=swData,midcycle=7.e5, plot=True)

#obtain mask of found spectral lines
mask = smoothBaseline.mask

#smooth baseline will be in
# baseline.wave
# baseline.flux
# baseline.flag
# baseline.weight
```

The script generates the following plots:

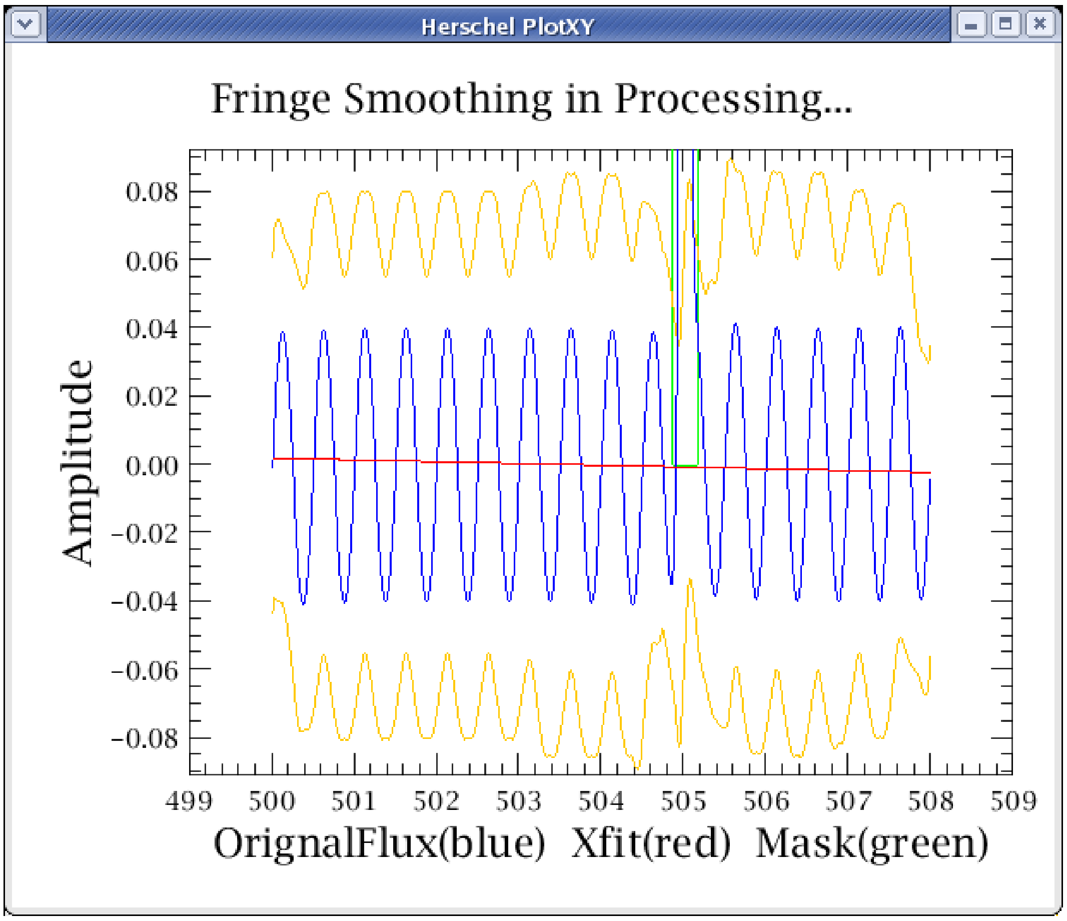


Figure 5.14. The first plot generated by SmoothBaseline shows the initial baseline (red) and the limits above and below which signal will be masked (yellow). Clearly the emission line is masked, as indicated by the green line.

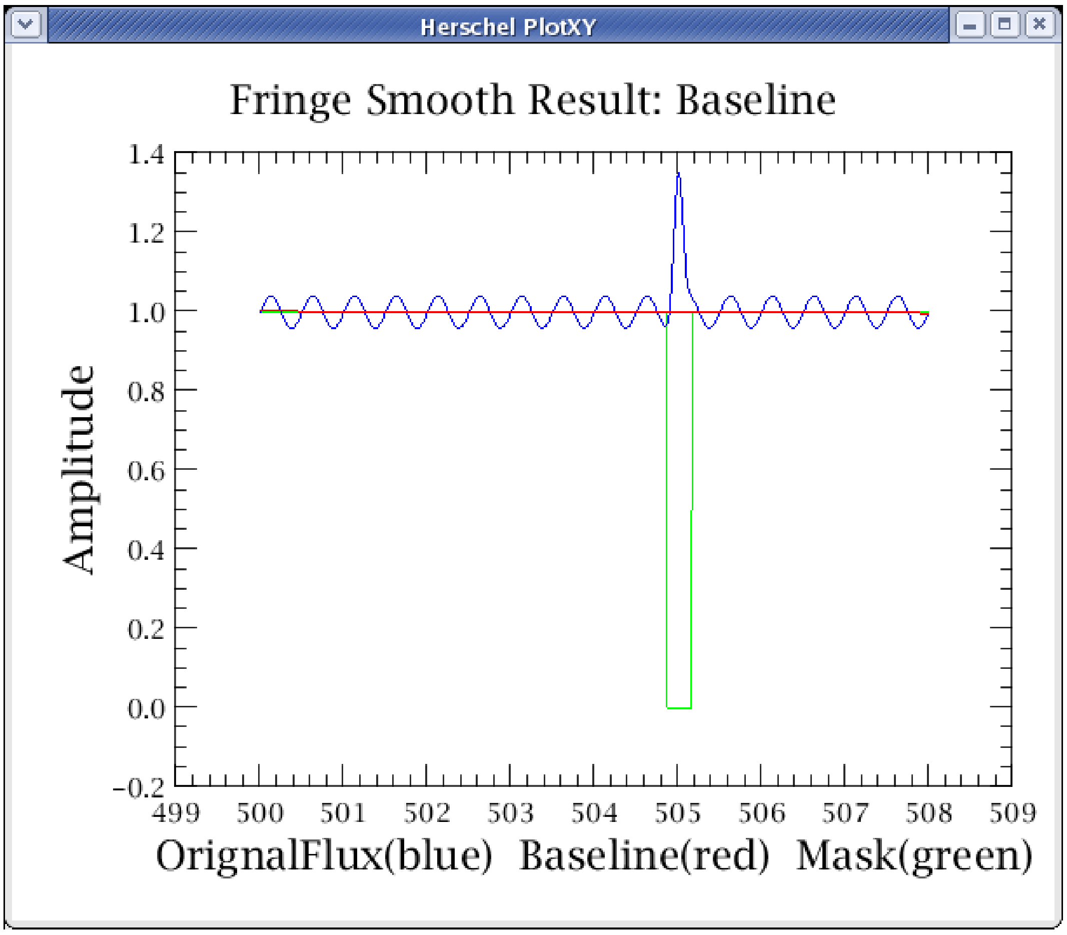


Figure 5.15. The second plot generated by SmoothBaseline shows the baseline in red and the masked regions indicated in green.

5.2.6. Creating a Spectral Cube

A spectral cube is a three-dimensional data structure in which two dimensions represent spatial dimensions (e.g. right ascension and declination) and the third dimension represents a spectral axis (see Section 2.11.1). `SpectralSimpleCube` data products can contain spectral cubes from all three science instruments on-board Herschel. They can be inspected by the spectrum toolbox (Section 5.2.7) and the spectrum explorer (Section 5.2.11).

Several projection tasks are capable of creating `SpectralSimpleCube` objects, but they all provide a common subset of methods. The following subsections explain the input data, the common methods to create cubes, and the output data.

5.2.6.1. Input Data

Unprojected Cubes

Spectral projection tasks require three-dimensional arrays of double precision floating point values for flux, right ascension, and declination, and a one-dimensional array of wave data. Conceptually, these three-dimensional arrays (of type `Double3d`) are unprojected spectral cubes, i.e. the sky positions for each flux element in the three dimensional cube are independent of all other sky positions.

The wave scale must be provided as a one-dimensional array of double precision floating point values (of type `Double1d`). It is assumed that the wave scale applies to each sky position. The three-dimensional cubes must have identical dimensions, and their spectral axis must have the same length as the wave scale.

Data from all three instruments can be stored in a `SpectralSimpleCube`; the cube projection tasks are generic and applicable to any data in the correct format. There may be preprocessing tasks, such as `SpirePreprocessCubeTask`, capable of transforming the data of other instruments into the required format.

Organising SPIRE Data as Unprojected Cubes

Any data in the correct format can be used as input for a `SpectralSimpleCube` projection task, but, for an example, we will show how to transform data from the SPIRE instrument into the required format. The `SpirePreprocessCubeTask` can transform SPIRE `SpectrometerDetectorSpectrum` objects into the three-dimensional arrays needed to create a `SpectralSimpleCube`. Shown below is an example use of the task:

```
spc = SpirePreprocessCubeTask([sds1, sds2])
```

This produces an object called a `SpirePreprocessedCube` (SPC), which is simply a container for the unprojected data. SPC objects will also contain unit information and metadata, if available from the input `SpectrometerDetectorSpectrum`. The contents of the SPC can be used to create a spectral cube. Shown below are some examples of how to retrieve the contents of an SPC.

```
# Get data for the SPIRE Spectrometer Long WaveLength array.
flux = spc.getFlux("SLW")
fluxUnit = spc.getFluxUnit()
ra = spc.getRa("SLW")
raUnit = ra.getUnit()
# etc.
```

`SpirePreprocessCubeTask` has some preconditions that must be met. For all channels of all scans of all input `SpectrometerDetectorSpectrum` objects, the following must be consistent:

- wave scale length and values.
- units.

The PACS pipeline employs a similar processing step when it calls `SpecWaveRebinTask` to create a `PacsRebinnedCube`.

Target Grids

In addition to three-dimensional arrays of floating-point values, projection tasks require a `TargetGrid` object. A target grid specifies the dimensions of the spectral cube to be created, in particular the two-dimensional spatial grid onto which the data are to be projected.

You can define your own target grid or use a method to create a default target grid. Every projection task includes several methods capable of creating default target grids from three-dimensional arrays of right ascension and declination data.

```
# Create the task.
pt = NearestNeighbourProjectionTask() # It could be a different projection task.
# Create a default target grid.
grid = pt.targetGrid(ra, dec, wave)
# Alternatively, specify the pixel size.
grid = pt.targetGrid(ra, dec, dra, ddec, wave)
```

Metadata and Units

Although not always required, metadata and units can be provided to projection tasks using the appropriate setter methods. Some tasks may assume default units if none are provided, but that behaviour is task-specific. Some preprocessing tasks, such as `SpirePreprocessCubeTask`, will conveniently extract units and metadata from the input data.

5.2.6.2. Cube Projection

By this point, the necessary arrays and a target grid should be available, and the projection task can be executed. All projection tasks provide a set of methods for creating spectral cubes. To create a cube, some interpolation scheme is used, but the kind of interpolation is task-dependent. All cube projection tasks support only spatial interpolation, not spectral resampling. See the *Data Processing User's Manual* for information about spectral resampling: Section 3.12.7.

```
# Operates on unprojected data.
project(flux, error, flag, ra, dec, targetGrid, detectorNames, allowExtrapolation)
# Operates on a single cube.
project(ssc, targetGrid, allowExtrapolation)
# Operates on a list of cubes.
project([ssc1, ssc2], targetGrid, allowExtrapolation)
```

(Note: the "detectorNames" parameter is a `String1d` object used to distinguish between HIFI, PACS, and SPIRE detectors. It can be set to None if unused by the task.)

Unprojected Cubes

A `SpectralSimpleCube` can be created using a target grid and the unprojected three-dimensional arrays of flux, right ascension, and declination.

```
# Create the projection task.
pt = NearestNeighbourProjectionTask() # It could be a different projection task.
# Project a cube.
ssc = pt.project(flux, error, flag, ra, dec, grid, detectorNames, Boolean.TRUE)
```

Cube Regridding

After a cube has been created, it can be regridded. In order to regrid a cube, specify a new target grid with a different spatial grid. Cube projection tasks provide, at minimum, two methods for regridding cubes: (1) a method that takes a single cube as input, and (2) a method that takes a list of cubes (making it possible to regrid several cubes into one).

```
# Create the projection task.
pt = NearestNeighbourProjectionTask() # It could be a different projection task.
# Regrid a cube.
ssc = pt.project(ssc2, grid, Boolean.TRUE)
```

Extrapolation

A `SpectralSimpleCube` projection task provides three projection methods. Each of these methods have a boolean parameter, "allowExtrapolation", which specifies whether extrapolation is allowed. Extrapolation is task-specific as it depends on the interpolation method used by the task.

NearestNeighbourProjectionTask

`NearestNeighbourProjectionTask` is a very basic but robust projection task. The methods described in this example are common to all projection tasks.

Algorithm

The `NearestNeighbourProjectionTask` employs the following algorithm for each sky position in the target grid:

1. Convert the sky position in the target grid (row, column) to world coordinates (RA/DEC).
2. Determine which spectrum from the three-dimensional cubes is closest to the world coordinates of the pixel.
3. Copy the flux, error, and flag data from the input spectrum to row and column of the `SpectralSimpleCube`.

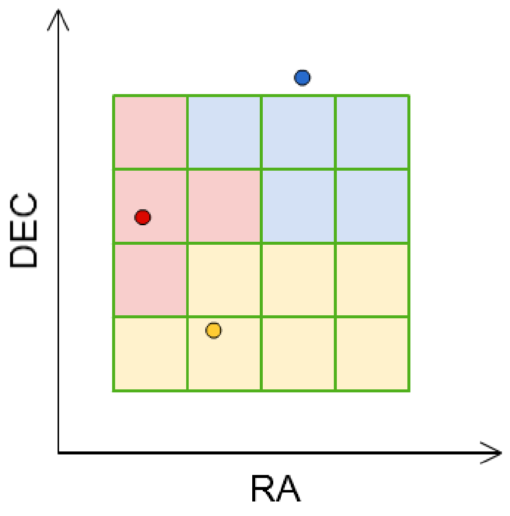


Figure 5.16. Spectra at initial sky positions (coloured dots) to be projected onto the green target grid. The colour of the target grid pixel indicates which spectra the algorithm determines is closest.

Extrapolation

If the target grid specifies a pixel on the sky which is not within the rectangle defined by the extreme right ascension and declination values of the input data, then the `NearestNeighbourProjectionTask` will thrown an exception unless the "allowExtrapolation" boolean parameter is true. This is true of both projecting a cube from arrays of data or regridding one or several cubes. In the case of regridding a list of cubes into a single cube, extrapolation is necessary only if the target grid specifies a pixel that is not within the boundaries of any of the input cubes.

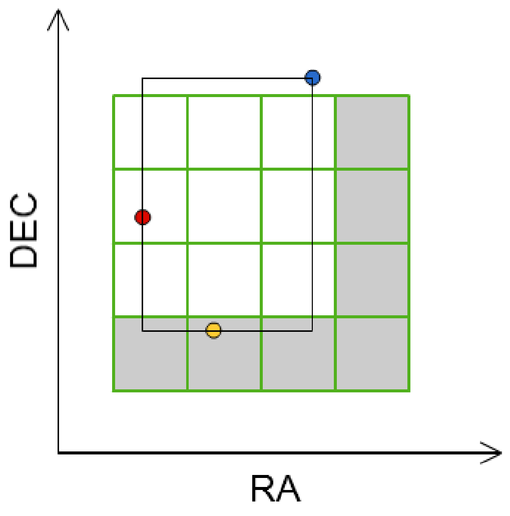


Figure 5.17. Spectra at initial sky positions (coloured dots) to be projected onto the green target grid. The black rectangle shows the extreme right ascension and declination values of the input data. Because the input data rectangle doesn't cover the grey pixels of the target grid, extrapolation is necessary to project a cube.

5.2.6.3. Output

Projection tasks return a `SpectralSimpleCube` product for which HIPE provides dedicated visualisation and analysis tools.

5.2.7. The Cube Spectrum Analysis Toolbox

The Cube Spectrum Analysis Toolbox (CSAT) is an interactive, user friendly toolbox which provides navigation, quick access, manipulation and analysis of spectral cubes in HIPE.

The navigation and quick access parts offer the following features:

- Navigation through the cube along the spectral axis
- Quick look and extraction of individual spectra
- Quick look and extraction of spectral regions

The cube manipulation tools offer the following features:

- Extraction of sub-cubes in the spectral or spatial domain
- Creation of monochromatic images in the spectral domain.

The analysis tools offer the following features:

- Creation of PV diagrams (position-velocity maps)
- Creation of velocity maps creation

The following sections will cover these topics:

- The graphical user interface of the CSAT and the features it offers
- Accessing these features from the command line

The CSAT works with cubes of type `SimpleCube` and `SpectralSimpleCube`. Please see the *Scripting and Data Mining* guide for details on how to create and manipulate these cubes: Section 2.11.1.

Note

To be usable by the CSAT, a cube must have a valid WCS (World Coordinate System). See the *Scripting and Data Mining* guide for details: Section 2.13.

5.2.8. The CubeSpectrumAnalysisToolbox GUI

The cube tool graphical user interface offers you a more user-friendly way to run the various functions (*Tasks*) of the spectral cube analysis toolbox. Each of the functions can also be run straight from the HIPE command line.

The cube tool is available in the package herschel.ia.gui.cube, while the individual tasks are located in the package herschel.ia.toolbox.cube. The cube tool accepts data of type `SimpleCube` or `SpectralSimpleCube`; you can find out what type your cube is either by hovering over it in the *Variables* panel, or with the following command (assuming `myCube` is the name of your cube):

```
print myCube.class
```

For example, cubes produced by HIFI and SPIRE pipelines are in `SimpleCube` format. For PACS, the cube coming out of the task `specProject` is also in `SimpleCube` format. All these can immediately be ingested in the GUI.

The cube tool can be launched in the following ways:

- By right clicking on a `SimpleCube` or `SpectralSimpleCube` product in the *Variables* panel, where the cube tool (along with other tools) is offered as a viewing option (`CubeAnalysisToolbox`).

- From the command line, with a command like the following:

```
myCubeResults = CubeSpectrumAnalysisToolbox(myCube)
```

  This command opens the cube tool with `myCube` in it. Alternatively, you can do the same in two steps:

```

myCubeResults = CubeSpectrumAnalysisToolbox()  # Opens an empty window
myCubeResults.setCube(myCube)
```

With the command line method, anything you create via the cube tool will be put in the `myCubeResults` variable, from where it can be accessed later. Both usages will return the result of all the operations done with this tool as a new variable in HIPE. The command line allows in addition to access the results in a second way and, for some features, in a different format. All this will be described later.

5.2.9. Using the GUI

When you launch the CSAT the following window will appear:

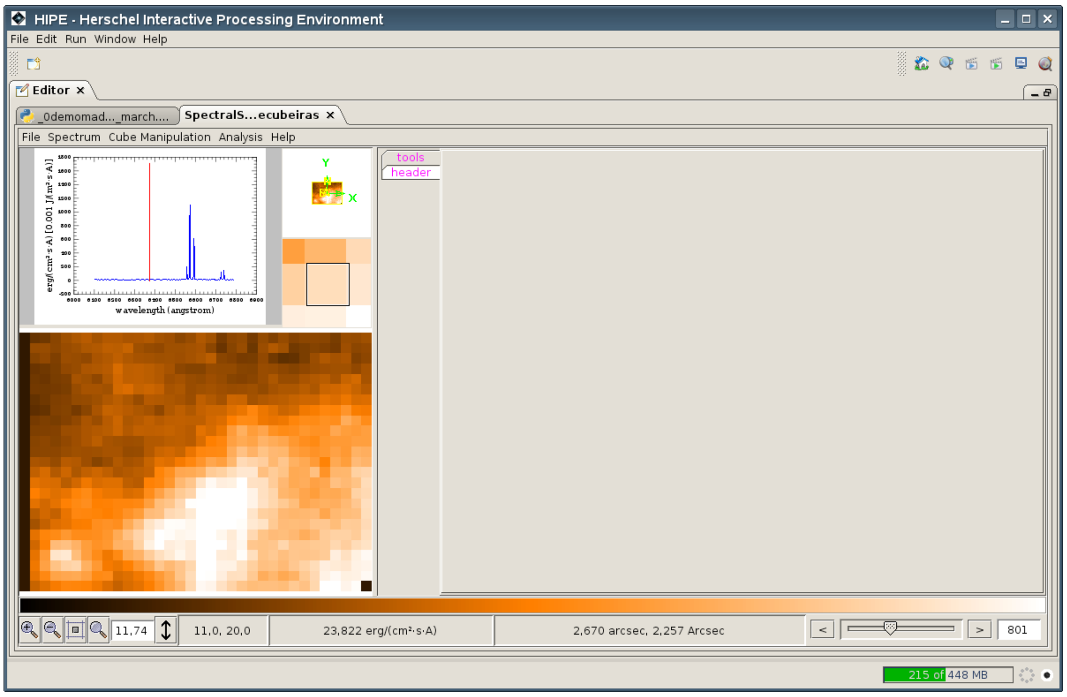

HIPE - Herschel Interactive Processing Environment
File Edit Run Window Help
Editor
_0demomad..._march....
SpectralS...ecubeiras
File Spectrum Cube Manipulation Analysis Help
tools
header
erg/(cm²·s·A) [0.001 J/(m²·s·A)]
wavelength (angstrom)
Y
X
11,74
11,0, 20,0
23,822 erg/(cm²·s·A)
2,670 arcsec, 2,257 Arcsec
801
215 of 448 MB

Figure 5.18. The first view you will have of the cube tool GUI

The CSAT will appear within the *Editor* HIPE view. We recommend you maximise the view or take it out of the main HIPE window (see the *HIPE Owner's Guide* for details). You may then want to zoom-to-fit-window on the cube image, and resize the whole GUI so all the information boxes fit the information in them.

Note that the CSAT is still under construction, so some features may be different from what is written or shown here.

Note

If your spectrum displays with odd ranges, it is possible that previous plot settings are still in effect. Try selecting the spectrum plot's Properties and choose Auto range → First Layer → Both Axes.

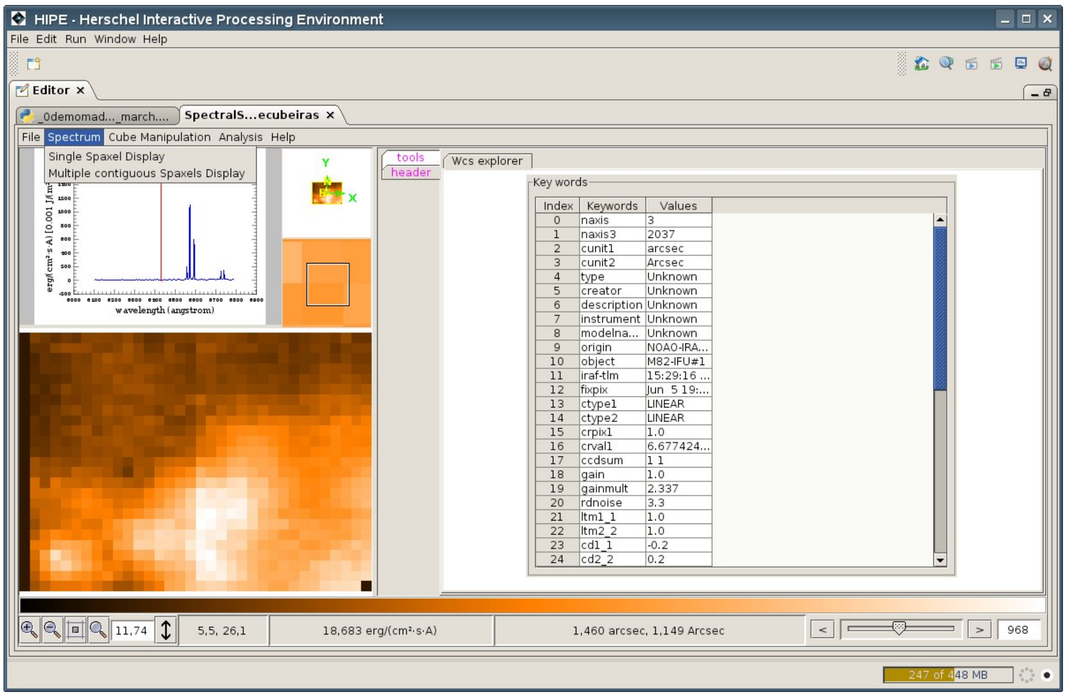
5.2.9.1. Design

The CSAT interface is split into two main regions:

- On the left side (the *image side*) the imported cube is displayed as a large image. The spectral slice of the image currently shown is adjustable with a slide bar to the lower right of the interface. Above the image you can see the following:
  - A real-time display of the spectrum in the *spaxel* (spatial pixel) that is under the mouse in the image, with a red vertical line corresponding to the layer (spectral cut) currently selected.
  - A *zoom and navigate* section. In the upper part you can adjust the view of the total cube plane that is shown in the large image. You can also see the N-S and E-W axes, if the WCS is included in the imported cube. The lower is a zoom of the spaxels around the mouse pointer.
- On the right side (the *working side*) are found, located in sub-tabs of the *cube* tab, the results of various selections you will have done on the cube (this is explained later). You can also find a *header* tab with the cube's header information (see figure below).

At the bottom of the interface you can find the following:

- Buttons for zoom, pan and adaptive zoom.
- Pixel coordinates, intensity value and sky coordinates (if present in the cube) of the spaxel under the mouse pointer.
- An adjustable colour bar, and a slide bar for navigating along the spectral dimension of the cube. The units of the slide bar are not spectral, but indicate the position in the spectral dimension (the array position). Spectral units are shown on the plots.


HIPE - Herschel Interactive Processing Environment
File Edit Run Window Help
Editor
_0demomad..._march....
SpectralS...ecubeiras
File Spectrum Cube Manipulation Analysis Help
Single Spaxel Display
Multiple contiguous Spaxels Display
wavelength (angstrom)
Y
X
tools
header
Wcs explorer
11,74
5,5, 26,1
18,683 erg/(cm²·s·A)
1,460 arcsec, 1,149 Arcsec
968
247 of 448 MB

Key words

Index	Keywords	Values
0	naxis	3
1	naxis3	2037
2	cunit1	arcsec
3	cunit2	Arcsec
4	type	Unknown
5	creator	Unknown
6	description	Unknown
7	instrument	Unknown
8	modelna...	Unknown
9	origin	NOAO-IRA...
10	object	M82-IFU#1
11	iraf-tlm	15:29:16 ...
12	fixpix	Jun 5 19:...
13	ctype1	LINEAR
14	ctype2	LINEAR
15	crpix1	1.0
16	crval1	6.677424...
17	ccdsum	1 1
18	gain	1.0
19	gainmult	2.337
20	rdnoise	3.3
21	ltm1_1	1.0
22	ltm2_2	1.0
23	cd1_1	-0.2
24	cd2_2	0.2

Figure 5.19. The header explorer

5.2.9.2. The Spectrum menu

The Spectrum menu is dedicated to spectrum extraction and allows you to select spectra out of the cube on the image side, from single spaxels or from a spaxel region. The result is displayed in the working side and can be manipulated, printed, and saved.

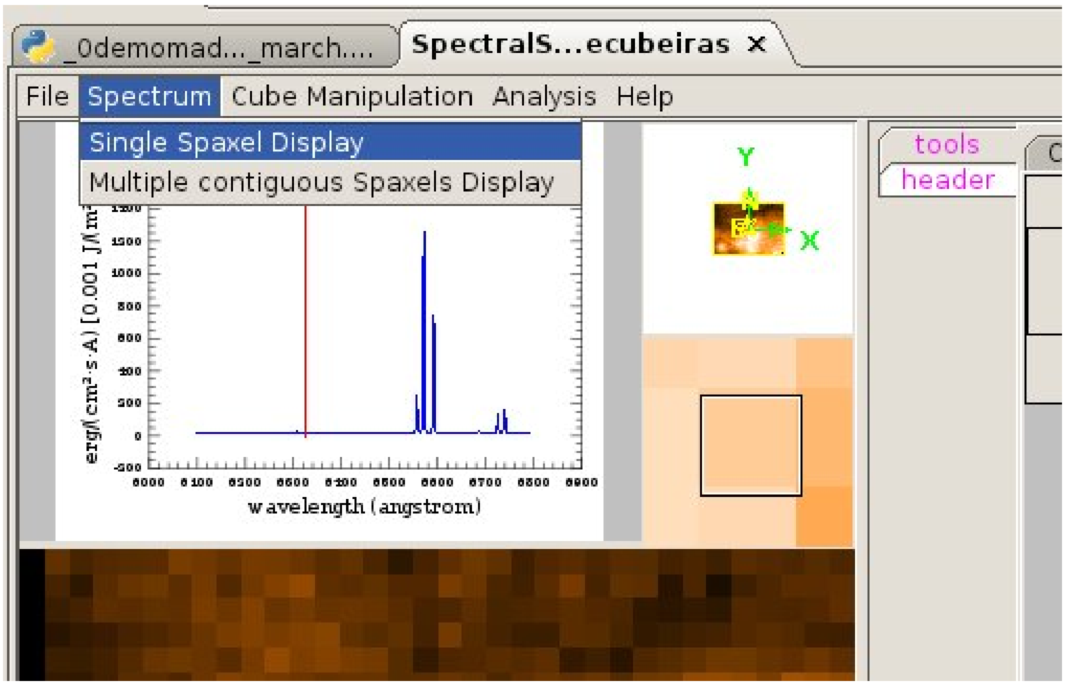


Figure 5.20. The Spectrum menu

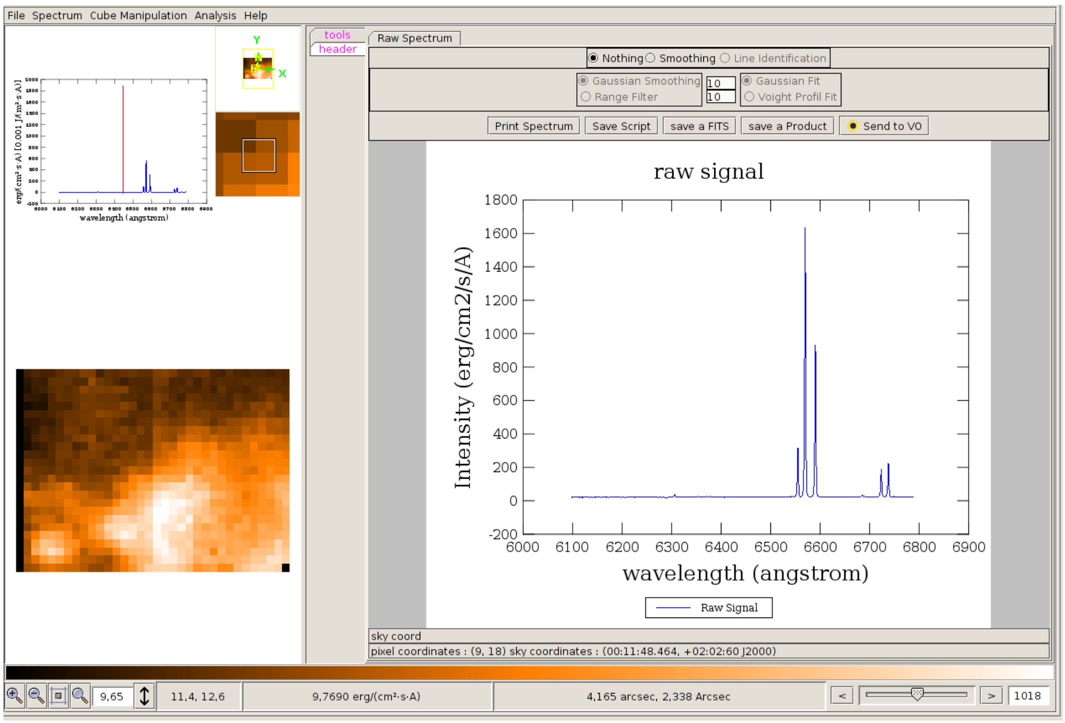
Single Spaxel Display

Selecting the first Spectrum menu item `Single Spaxel Display` brings up

- A new graphical panel on the right part of the window
- A blue rectangle following the mouse cursor when this one is over the cube.

The panel of the right panel contains 2 sections:

- The upper section contains radio buttons and text fields. This includes: to define the optional filtered spectrum to be extracted and buttons to print and to save a script that reproduces the last executed commands within this tab
- The lower section contain a plot section (a PlotXY component) showing the single spectrum.
- Under the plot section the spaxel coordinates and sky coordinates of the current spectrum are displayed



pixel coordinates : (9, 18) sky coordinates : (00:11:48.464, +02:02:60 J2000)

Figure 5.21. The single spaxel spectrum extraction window

The spectrum corresponding to the spaxel under the mouse cursor is displayed in real time in the plot section. Clicking on the left button of the mouse stop this real time display and freezes it on the spectrum of the "clicked" spaxel. At the same time, the spectrum is sent to HIPE with a default name (see below).

The plot in the tab can be manipulated in the usual way that PlotXY plots can be in HIPE.

In the upper part of the tab we find additional functionalities offered via radio buttons and text fields.

- Smoothing: selecting this you can perform a Gaussian smoothing or a boxcar filter, for both of which you can chose the width of the filter, in units of channels (i.e. not spectral units but rather the number in the spectral axes array positions). A red smoothed spectrum is now superimposed on the blue original in the plot (currently to see this on the frozen spectrum you need to select smoothing before you select the spaxel to freeze, or you need to go back to the image and select a spaxel again)

- Print spectrum; open a print dialog box to print the plotted spectrum

- Save script: save a python script containing the sequence of commands you have just executed (corresponding to the buttons you have just pressed) within this tab, including selecting the spaxel in the first place

- Save as fits: save the selected spectrum, in a multi-extension FITS file in a Spectrum1d product format. This format can be read by standard software and can also be reloaded into HIPE

- Save product: This button sends the current spectrum as a Spectrum1d variable to the HIPE session, from where it can be read into other spectral fitting/viewing tools provided by HIPE. The naming convention for the variable saved is: variable name of the cube +"**singlepixspectr_**" + spaxel coordinates from which it was extracted

- Send to VO button: a button to use the conection between HIPE and the Virtual Observatory

To extract a new spectrum from another spaxel or to extract a new smoothed spectrum, the interface needs to be "reactivated": this can be done by clicking on one of the radio buttons in the upper part of the tab or by re-selecting the menu.

Note that the GUI is based on the "PlotXY" tool of HIPE; at present you can chose to zoom on the plot on the working side but if you go back to the image side to select a new spaxel to display, you lose the zoom you just defined and a complete new spectrum is shown.

If the CubeSpectrumAnalysisToolbox was opened from a command line the last spectrum extracted can be accessed with the method getSinglePixelSpectrum():

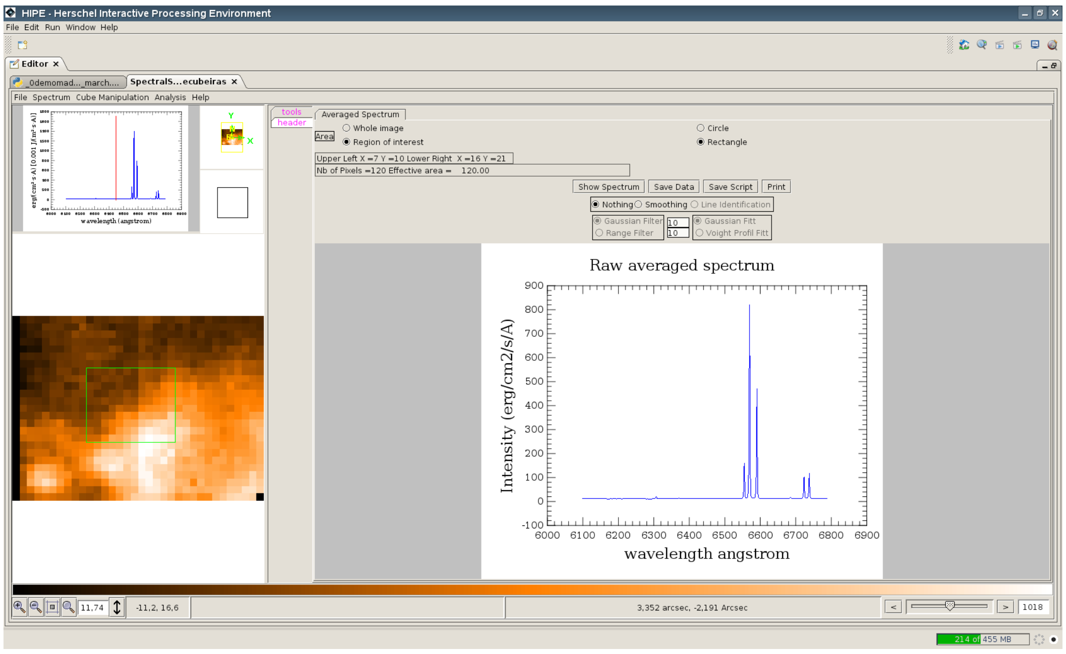
```
# the CSAT was called up with
mycsat = CubeSpectrumAnalysisToolbox()
# and then
spec = mycsat.getSinglePixelSpectrum()
```

which returns a Spectrum1d variable.

Multiple Contiguous Spaxel Display

The second menu item allows you to create a spectrum that is the average of a spaxel region. The selection of this menu brings up a panel in a new tab in the right part of the CSAT. This new panel contains:

- An upper part with
  - an area to define the kind of region on which to extract the spectrum, using radio buttons
  - an information section about the area selected
  - a button to execute the extraction, save the data, save the script, or print the current spectrum
  - a section to define an optional filter
- A plot section where the resulting spectrum is shown



Whole image [✓ Region of interest]
Circle [✓ Rectangle]

Upper Left X =7 Y =10 Lower Right X =16 Y =21
Nb of Pixels =120 Effective area = 120.00

[✓ Nothing] Smoothing Line Identification
[✓ Gaussian Filter] Range Filter 10 10
[✓ Gaussian Fitt] Voight Profil Fitt

11.74 -11.2, 16.6 3.352 arcsec, -2.191 Arcsec 1018

The spectrum shown is the average from the region drawn by the user on the cube image (which is in the left part of the window).

Different kinds of regions can be defined via the radio buttons:

- The whole image

- A rectangle, taking only integer pixels and with the border being along the spaxels selected
- A circle, for which the center and the radius are float values

  All pixels inside the circle are taken in account with a weight of 1, all pixels on the border of the circle are taken in account with a weight corresponding to the part inside the circle (weight < 1)

When you select the "region" mode you define the region to select on the cube image by a clicking the left button on the cube, drag the mouse with the button pressed, and a green contour will follow the movement of the mouse. When you have the region you want, release the button. You can move the region after having drawn it by clicking on it to select it and drag it to an other place, you can modify its shape by selecting one of the blue corner dots and dragging that.

Remark: at this date the CIRCLE mode is not an ELLIPSE mode (even thought it is probably possible to make the shape elliptical). When modifying a circle make sure you keep it as a circle, otherwise the resulting spectrum will be slightly wrong.

To see the result of your selection click on the "Show Spectrum" button. The spectrum is displayed in the plot section and is also sent to HIPE with a default name (see below).

You can adjust the properties of the plot in the same way as with most plots in HIPE.

if you want to see a new result after having—moved the selection; changed the method of selection (rectangle#circle, or Region#whole image...); modified a region (resized it for example)—then you must click on "Show Spectrum" button again.

Also possible within this tab are:

- The same Save Script, Print as found on the single spectrum tab
- An information bar
- A Save Data button which saves the spectrum in a multi-extension FITS file
- The same smoothing options as offered with the single spaxel section. The smoothing information must be defined before you click on the computation button

Remark: All single spectra extracted in this interface are sent to HIPE as as class **Spectrum1D** with a default name: variable name of the cube + "**regionspectrum_**" the type of region selected ("whole" , "Rectangle" or "Circle"). The information about the location of the region selected are stored in the metadata of the Spectrum1d.

If the CubeSpectrumAnalysisToolbox was opened from a command line the last average spectrum extracted can be accessed with the method getAvgspectrum() (note: this will at some point change to getAvgSpectrum()), for example:

```
# the CSAT was called up with
mycsat = CubeSpectrumAnalysisToolbox()
# and then
spec = mycsat.getAvgspectrum()
```

which returns a Spectrum1d.

5.2.9.3. Cube Manipulation

The Cube Manipulation menu has two entries:

- Spectral Range Extraction, for extracting smaller parts of the original cube.
- Integrated Map, for integrating over a selected spectral domain.

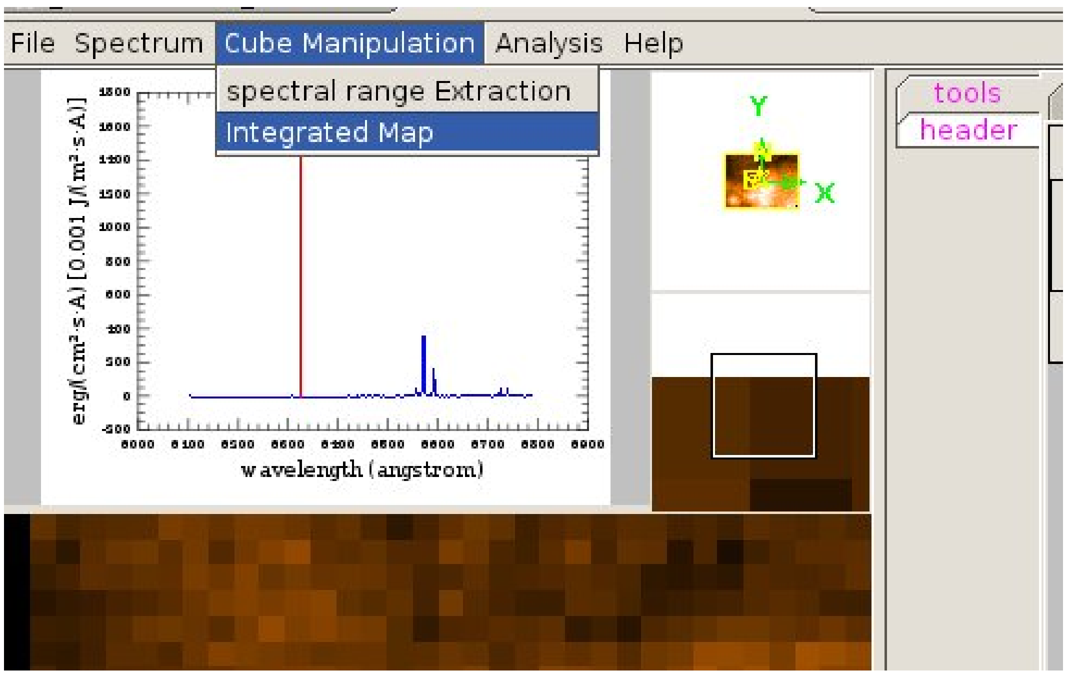
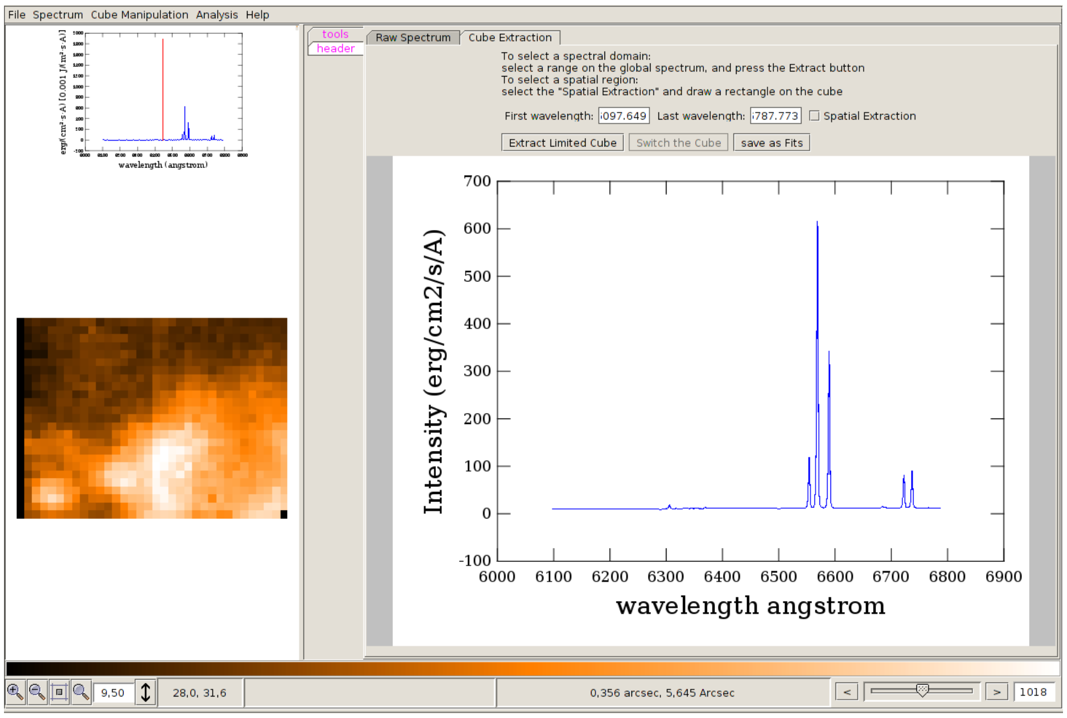


Figure 5.22. The Cube Manipulation menu

Spectral Range Extraction

This options allows you to select a particular spectral range and then save the cube over this spectral range only. This menu also allows you to proceed to a spatial extraction. *Note that this functionality is still under construction.*

After selecting this entry a new tab will open on the working side of the GUI.



To select a spectral domain:
select a range on the global spectrum, and press the Extract button
To select a spatial region:
select the "Spatial Extraction" and draw a rectangle on the cube

First wavelength: i097.649 Last wavelength: i787.773 ☐ Spatial Extraction

Extract Limited Cube | Switch the Cube | save as Fits

Figure 5.23. The Range extraction window

This new tab contains a plot with the global spectrum of the cube and a parameter and information section which shows the following:

- two fields with the first and last spectral values of the cube
- Two buttons, one of which (Switch the Cube) is currently not functional

The wavelength limits shown on the plot are of the current cube, i.e. the original cube on first start-up, and then updated each time you extract a cube on a smaller range.

Extracting a smaller cube with spectral limits

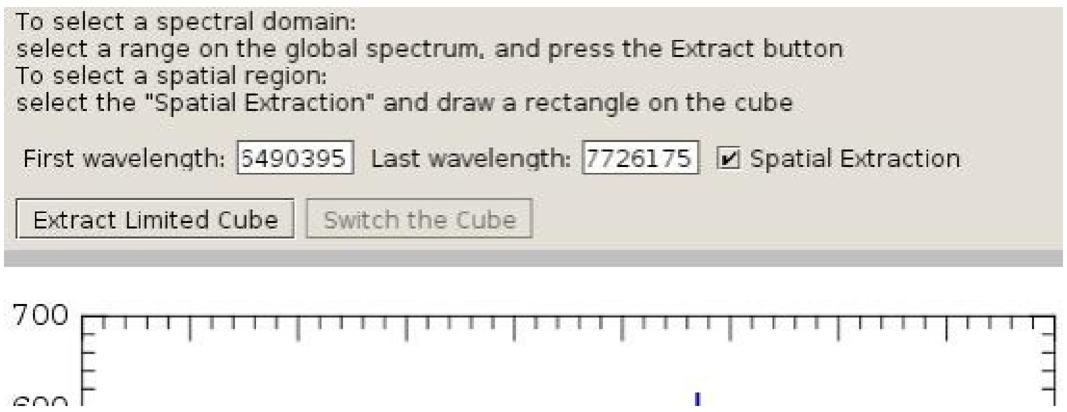
The main purpose of the cube extraction tool is the extraction of a smaller spectral range. To extract a cube on a smaller spectral domain do the following:

- Zoom in on the spectral range you are interested in with the usual mouse box selection on the plot in the right part of the window (do not type numbers in the boxes). You can zoom out by right-clicking on the plot and choosing Zoom Out from the contextual menu).
- Click on the Extract Limited Cube button, which will open a standard dialogue window where you can save your selection in FITS format. The behaviour will be the same if you choose to extract a spatially limited cube (see below for details).

If you do not specify a file name, the cube is saved with the default name `extractedCube_` + date + hour + `.fits`).The cube is also sent back to HIPE, with the default name `rangecube`.

Extracting a cube with spatial limits

You can crop the cube spatially (in fact you can crop the cube on both the spatial *and* the spectral dimension). To crop on the spatial dimension, select the Spatial Extraction checkbox (see figure below) located above the plot, next to the wavelength/frequency limits.You can then select a spatial region of the cube by dragging the mouse pointer over the image (in the image side) to draw a rectangle. This feature comes in addition to the spectral selection i.e the resulting cube will cover the rectangle drawn on the left *and* the spectral domain selected on the right (if you did not do a spectral zoom it will retain the original range).


To select a spectral domain:
select a range on the global spectrum, and press the Extract button
To select a spatial region:
select the "Spatial Extraction" and draw a rectangle on the cube

First wavelength: 5490395 Last wavelength: 7726175 ☑ Spatial Extraction

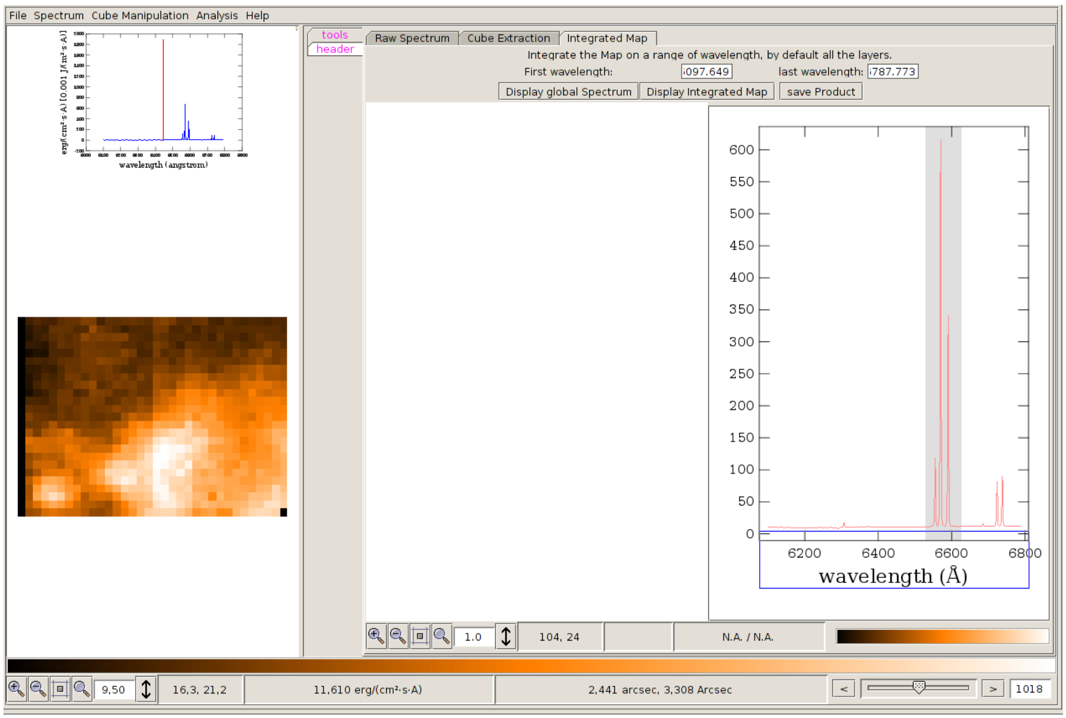
Figure 5.24. The range of extraction and the radio button for the spatial extraction

After clicking on the Extract button the process is the same as explained previously: the same windows appear and the cube is returned to HIPE. The default name is now `rangespatialCube`, of type `SpectralSimpleCube`.

If you opened the CSAT from the command line, you can access the last extracted cube via the `getRangeExtractedCube()` method.

Integrated Map

The Integrated Map menu item allows you to define a spectral range over which to integrate the individual *layers* of the cube (or *image slices*). The result is a `SimpleImage` (that is, a two-dimensional product) and contains the sum of the individual layers for the given range.



Integrate the Map on a range of wavelength, by default all the layers.

First wavelength: ¡097.649 last wavelength: ¡787.773

Figure 5.25. The integrated map interface

The right part of the window is divided in three:

- An Info & Parameters section, displaying the values and units of the spatial axis of the original cube and two buttons, Display Global Spectrum and Display Integrated Map. This will be modified in the next release of the CSAT.
- A Display section, which receives the result of the integration as one `SimpleImage` per specified range.
- A Spectrum Plotter section, which shows the global spectrum of the cube (average of all the spectra).

To integrate one or more ranges (or spectral domains) do the following:

1. Display the global spectrum in the Spectrum Plotter by clicking on the Display Global Spectrum.
2. On this spectrum choose the Select Range tool by right clicking and choosing Tools → Select Range. Then with the mouse select one or more ranges to integrate.
3. Launch the integration by clicking on the Display Integrated Map button.

Note

This tool only allows you to select separate ranges, i.e one range must end before the next one starts, and there is no possibility to extract overlapping ranges in one step. However it is possible to clean the selection after having integrated a first set of integrated maps and therefore to create a new set of images to integrate.

The resulting images are displayed in the same graphical component; if you selected many ranges the images are "stacked" and can be selected by using the slide bar.

Every time you select a range and click on the button to integrate and display the resulting image, these images are sent to HIPE as a set of new variables of type `SimpleImage`.

If you opened the CSAT from the command line, you can retrieve the resulting images as a list of images with the following command, which returns a list of images:

```
a = cat.getIntegratedMaps()
```

The images can then be accessed like this:

```
img1 = a[1]  # A SimpleImage
```

5.2.9.4. Analysis menu

The Analysis menu is dedicated to analysis of the cube itself. All operations take the cube as primary input and return a 'scientific' result.

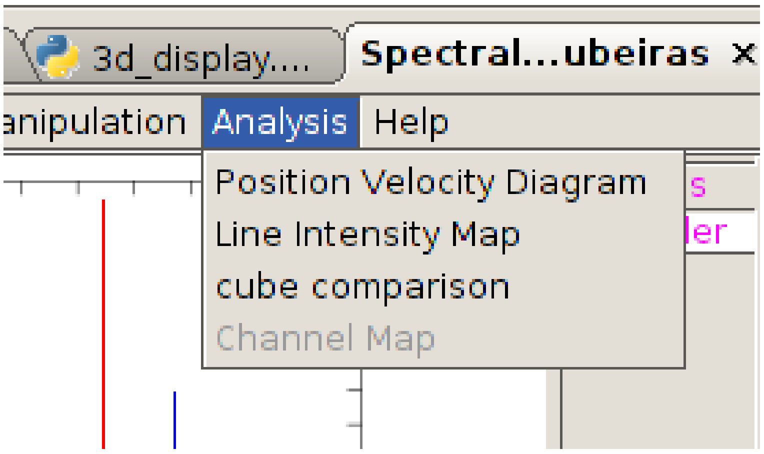


Figure 5.26. The analysis menu

Position-Velocity Diagrams and Maps

This item allows you to make Position-Velocity (PV) diagrams from your cube. Selecting this will open a new tab on the working side. From the radio buttons there you can chose between two modes:

- Axis, which works by allowing you to select a slit along which the PV diagram is computed
- Map, which makes a 2D map where the intensity scale is the velocity values. This functionality should only be used on "line scan" cubes (i.e. one spectral line in your spectrum). If you have a range scan cube you should first select out a line scan (i.e. a single spectral line) to make a new cube, and then on this cube open a new CSAT. *This menu option is still under construction*

For both modes you need to define, using the slide bar at the bottom of the working side, a reference layer. This layer will define the spectral value corresponding to the velocity "0" (zero) for the PV diagram. You will notice that as you move this slide bar the red line on the plot on the image side of the GUI moves (at the moment you release the mouse button)#in this way you can "translate" reference layer units into spectral units. As you move this slide bar the numbers in white boxes next to the Map radio button will change, but you cannot change the numbers by typing directly in the boxes.

Input Data

The data for these velocity features can come with various units and physical meanings on the 3rd axis: frequency, wavelength, or velocity.

When the data are in frequency or wavelength the velocities are computed using the non-relativistic Doppler effect. When the data are in velocity the selection of the reference layer just shifts the zero and the velocities are directly from in the cube.

Position-Velocity Diagrams

The "Axis mode" produces "position velocity maps" or "PV diagrams". To compute such a map, the user needs to provide:

- An axis along which a spectrum is extracted

  This is done by draging the mouse on the cube on the left part of the window, you must click on the starting point of the line, release the mouse, drag it to the end point and click again. The line appear in green on the cube.

- A reference layer in the cube to define the velocity 0

  This is done by choosing the displayed layer with the slide bar.

- The line you draw defines a slit with a default width of 1 pixel. This width can be modified by filling the dedicated field in the parameter section of the panel.

  What the task actually does is create an averaged spectrum for the spaxels along the slit you set: for a slit width of 1 the spaxels selected are those that the green line you set actually goes through; for a width of 2, and additional 1/2 of the spaxels either side are selected, this meaning that the spectra from these spaxels are selected but the intensities are weighted by 0.5; etc. for widths of 3 and greater. It is then from this averaged spectrum that the PV diagram is created. Hence, a wider slit will increase the signal-to-noise ratio of the spectrum the PV diagram is constructed from.

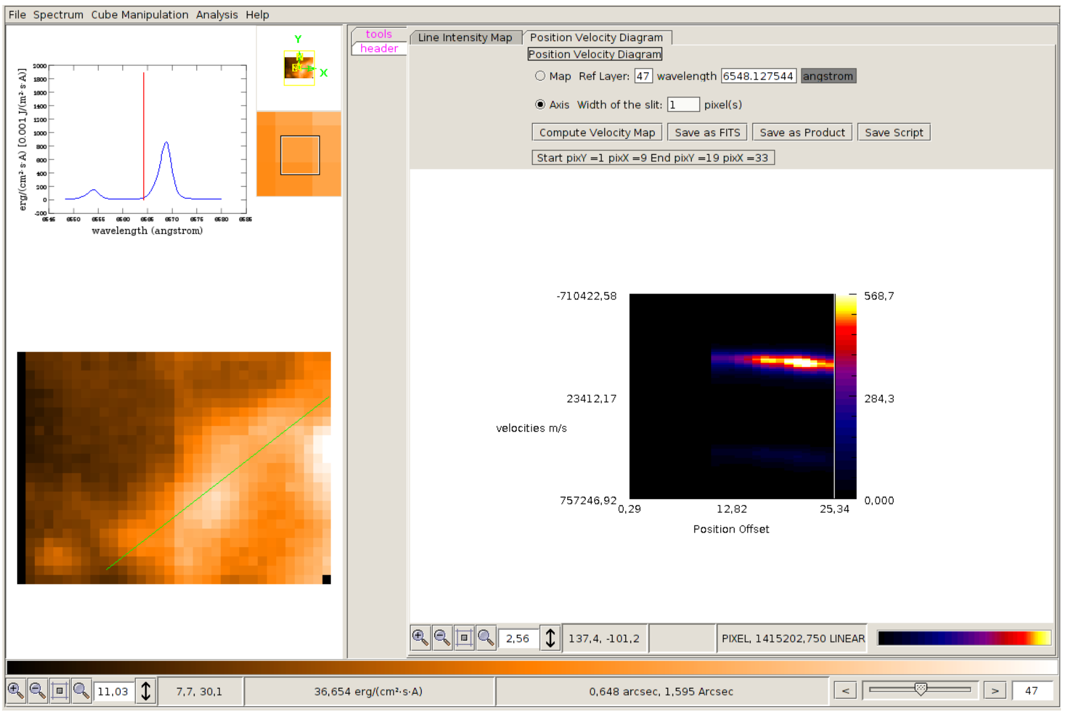
For data given in wavelength the velocities are computed using the usual **v=c # #wavelength/ wavelength(ref).** For data in frequency the velocities are computed using **v=c # #frequency/ frequency(ref)** and are given in **m/s.**

For both modes you actually compute the PV diagram by clicking the "Compute Velocity Map" button. This will produce an image such as show below. For Axis mode the horizontal axis is offset from the left side of the slit you drew (or offset from the top if the slit is directly up-down) and the vertical axis is velocity on the left and colour scale on the right. For Map mode the two axes are the spatial axes of the cube (in WCS units) and colour indicates velocity, with a colour bar on the right showing the scale, which runs from maximum to minimum velocity.

Note

If the PV diagram you see looks to be too small it is possible you are on a super-zoom; try panning out or zooming-to-fit using the magnifying-lens tabs below the PV diagram. If it is too dark, edit the cut levels (right-click on the PV diagram itself).

Note that currently one can only zoom on both axes at the same time; later we will allow for a zoom on the axes independently. Also note that the PV diagram is constructed from the whole cube that is currently show on the image side, i.e. over the whole spectral range you have. Therefore, it is likely you have a lot of velocities that are very large numbers! We are redesigning the GUI to allow you to select a smaller spectral region from which to make a PV diagram, but currently if you want to do this, you need to first create a new cube from a small spectral region (the Spectral Range Extraction menu item) and run the GUI on that cube.



Position Velocity Diagram

○ Map Ref Layer: 47 wavelength 6548.127544 angstrom

● Axis Width of the slit: 1 pixel(s)

Start pixY =1 pixX =9 End pixY =19 pixX =33

2,56 137,4, -101,2 PIXEL, 1415202,750 LINEAR

11,03 7,7, 30,1 36,654 erg/(cm²·s·A) 0,648 arcsec, 1,595 Arcsec 47

Figure 5.27. Position-velocity diagram

Note

If the PV diagram you see looks to be too small it is possible you are on a super-zoom; try panning out or zooming-to-fit using the magnifying-lens tabs below the PV diagram. If it is too dark, edit the cut levels (right-click on the PV diagram itself).

Once you have created the PV diagram, you may wish to adjust the properties of the diagram. You do this in the familiar way, that is right-click on the diagram where you are offered options: edit cut levels, edit colours, zoom, annotate, create screen shot (jpg), print, flip Y-axis.

As with all other tabs on the working side, you can Save Data in FITS format, and the data are returned to HIPE as a SimpleImage with a WCS containing velocities and offset along the axis.

Remarks:

- The "Save Script" button saves a jython script containing the sequence of commands that produce the PV diagram, but it save also a fits file with the list of spaxels defining the "slit" to be read (i.e. the slit you chose).

- The graphical interface uses a task which reshapes the result of the PV diagram so that the aspect ratio is compatible with the display geometry. In the script that "Save Script" products, the call of this task will be commented out, so that you can have the actual image geometry as result of the script.

  This reshaping task is not part of the cube spectrum analysis toolbox and so it is not explained in this document.

Velocity Map

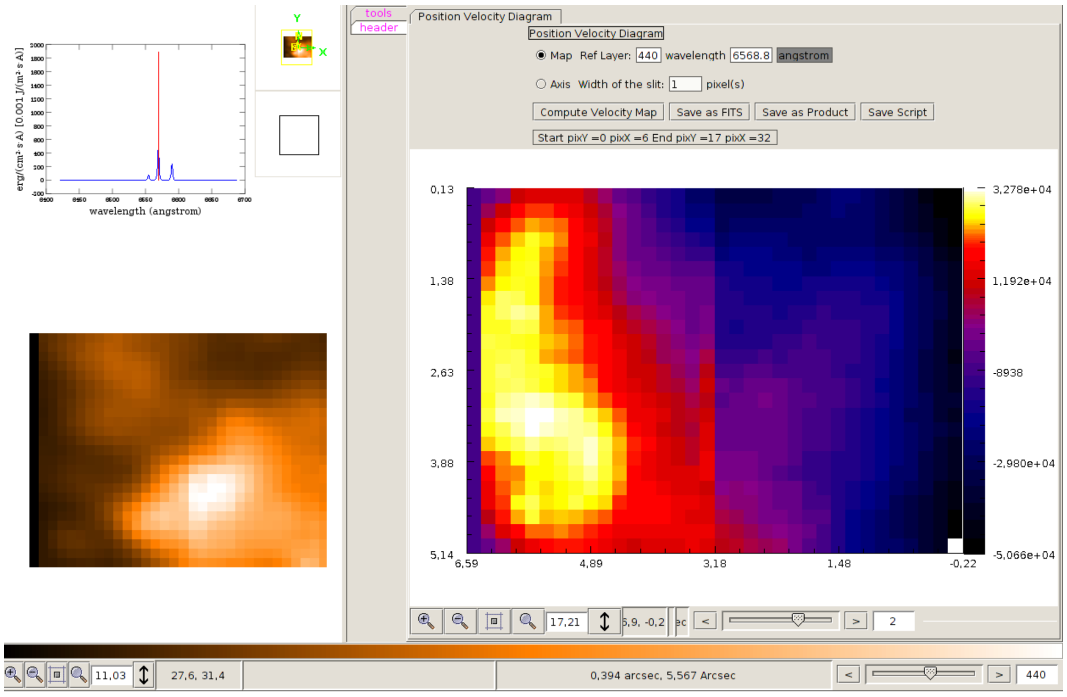
The velocity map is a 2D image in which the intensity value of the pixels is the velocity of the maximum of emission of the spectrum in the corresponding spaxels. To compute such a map the user need to provide :

- A reference layer, this is done by choosing the displayed layer with the slide bar.

For data given in wavelength the velocities are computed using the usual **v=c # #wavelength/ wavelength(ref)**, for data in frequency they are computed using **v=c # #frequency/frequency(ref)** and are given in **m/s.**

The maximum of the spectrum, i.e. the peak of the spectral line, which correspond to the radial velocity of the spectral line, is found by **fitting a gaussian** to the line. It is therefore important that the cube contains only one line of emission, otherwise the fitting will not work. If the original cube contains many line it as to be "cropped" with the *range extraction* tool of the CSAT.

The actual computation of the velocity map is done by clicking on the "**Compute Velocity Map**" button.



Position Velocity Diagram

(●) Map Ref Layer: 440 wavelength 6568.8 angstrom

( ) Axis Width of the slit: 1 pixel(s)

Start pixY =0 pixX =6 End pixY =17 pixX =32

Figure 5.28. Velocity map

A set of images is returned by the task. These are displayed in the GUI as stack of images in the following order: velocity image, dispersion, layer maximum, flux maximum. The results can be sent back to HIPE as new products (of class SimpleImage) by clicking on the button "Save as Product". The products can also be saved as FITS using the button "Save as Fits"

A script to reproduce the same computation can be created with the button "Save Script". This script could then be launched from the HIPE command line by you.

Line Intensity Map

This GUI provides an user-friendly interface to create a 2D map of the integrated flux for one given emission line.

The flux correspond to the integration of the fitted line **after subtraction of the continuum (see below). The following must be done:**

- Optional selection of the continuum to remove; define the degree of the polynomial and graphically select the region(s) to use for the fit.

  If you select "none" for the continuum, it is assumed that there is no continuum in your data, i.e. that the base-level is at 0 (zero). If you try to fit a line with continuum that is offset from 0 but select "none" for the continuum, the resulting fit will be wrong. You should in this case select a polynomial of degree 0.

- Selection of the profile model to fit (Gaussian, Voight, sinc)
- Optional defintion of the initial guess for the fit parameters

Depending of the choices in the selection panel, the fit is done in the following way:

- Step by step - steps 1 and 2 are skipped if you selected no continuum to fit:

  1) fit the continuum

  2) subtraction of the continuum

  3) fit the spectral line

  4) computation of the integrated flux using the resulting formula

  5) creation of the flux map, dispersion map and "position of the max" map

  This "position of the max" map stores for each pixel the position of the maximum of emission as channel number (i.e. not in spectral units, but as array position)

  At each of these steps various cubes are constructed: a continuum cube, a cube without continuum, a fitted line cube, a residual cube. These cubes are all stored as SpectralSimpleCubes, and by inspecting them you can check on the quality of the fit and therefore the quality of the line intensity map

- In only one step

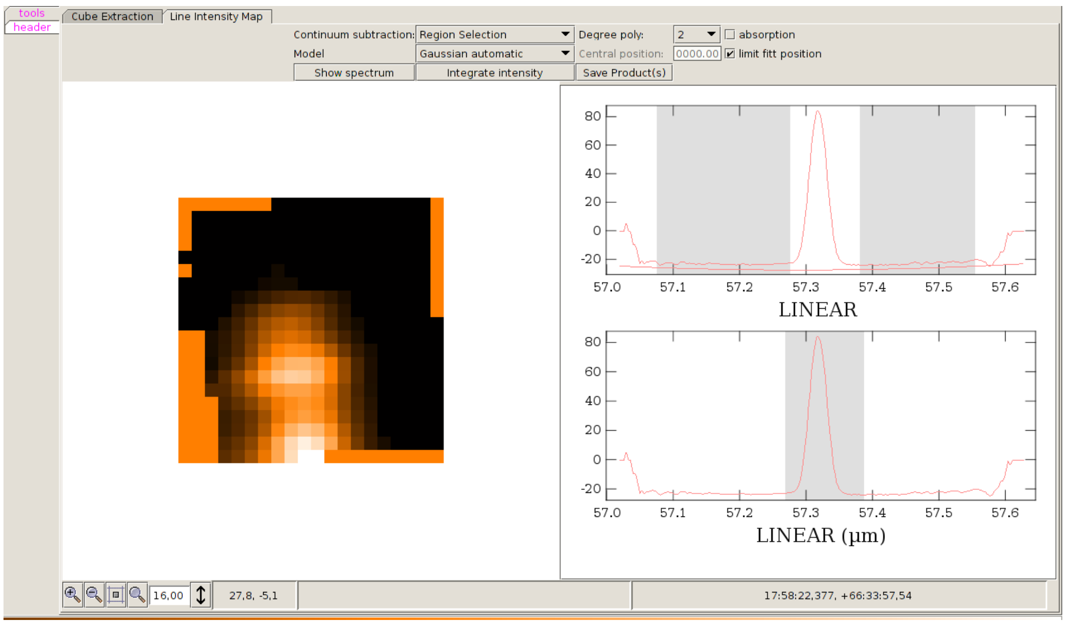
  A fit model is constructed by adding the continuum polynomial and the profile (including any initial guesses). This summed model is used and the fit process is done for each spaxel. At the end of the fit the various cubes previously mentioned are constructed

All the spaxels are processed in a loop following the same recipe.

Operation

At the selection of this feature (Line Intensity Map) in the menu the user must do the following:

- fill the spectrum viewer by clicking on the button "Show Spectrum"
- choose the continuum mode: no subtraction, polynomial based on selected region
- choose the degree of the pylonomial
- choose the model (only Gaussian at this date) and the mode "automatic" or "guess on the central position"
- (later it will be possible to do this on absortion lines by checking the absorption check box)
- you can also select a limited area for the result of the fit



Continuum subtraction: Region Selection
Degree poly: 2
absorption
Model: Gaussian automatic
Central position: 0000.00
limit fitt position

The computation is done by clicking on the "Integrate Intensity" button, and the integrated image is displayed in the left part of the GUI.

The Button "Save Product" sends to hipe a set of products:

- continuum cube
- cube after subtraction of the continuum
- cube of the fitted line
- SimpleImage of the integrated map

Cube Comparison

This GUI allow one to compare directly spectra from two cubes for a spaxel at the same sky position.

The user can choose a new SimpleCube or a SpectralSimpleCube to compare to the one she is current looking at with the CSAT, either from the disc by loading a FITS file or, if the new cube to view has already been loaded into HIPE, you can select it from a drop-down menu at the top of the cube comparison tab. After selection the cube is opened in the CSAT.

When moving the mouse over the second cube image, the real time spectrum viewer displays the spectrum from both cubes when the sky position is covered by both cubes.

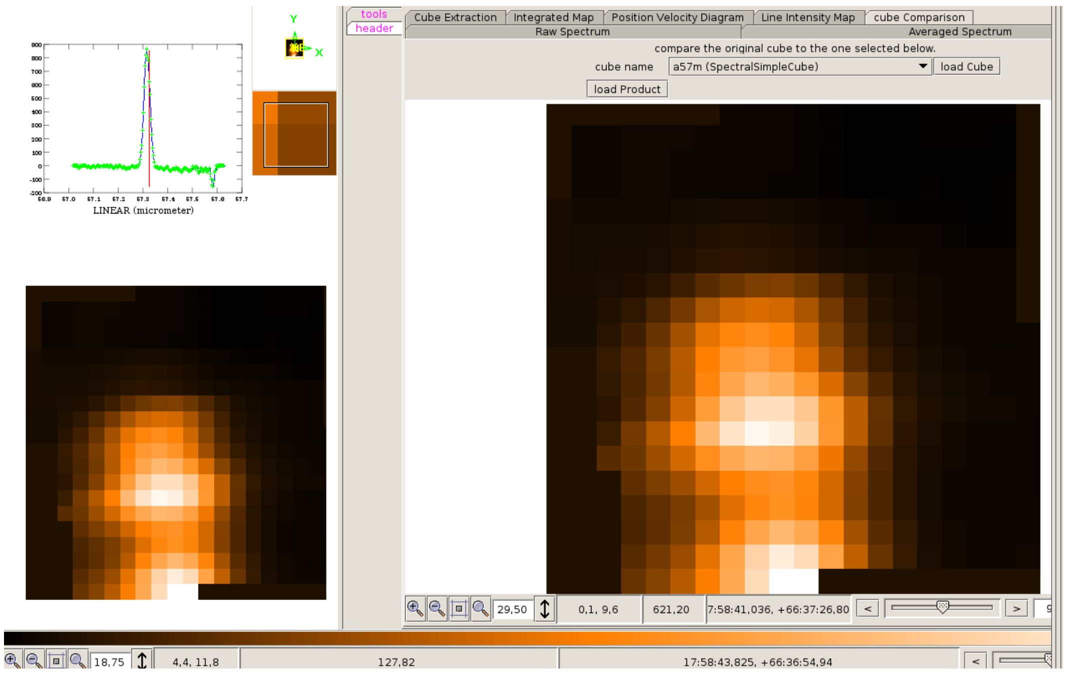


Figure 5.29. The cube comparison GUI

Since the two spectra are displayed on the same plot, the real time plot in the upper left part of the CSAT, if the range the spectra cover on either plot axis differ significantly, it will be difficult to see both spectra well. If possible, the system tries to show the layer corresponding to the same spectral value for the two cubes. If the cube are too different in terms of their spatial coordinates the user is warned via a popup window.

5.2.10. Running the tasks outside of the cubetool GUI

It is possible to run the tasks that the CSAT calls upon outside of the GUI, and here we will tell you how. This is something that you should do only when you are comfortable scripting in HIPE, because in the instructions that follow we assume you have already done some scripting. It is also possible to access on the HIPE command line the products that the cube tool creates, as has been explained in previous sections, e.g. when you select the spectrum from a single spaxel.

5.2.10.1. Accessing the individual products

As you perform activities in the CSAT (a.k.a. the cube tool), e.g. select out spectral or spatial regions, the results are held in tabs on the working side; but they are also held in new products that you can access from the HIPE command line or GUI.

- As you do things with the cube tool new products are created and are listed in the HIPE Variables panel, with names similar to "singlepixspectrum". (And then singlepixspectrum1 for the next selection, then 2...). These should appear no matter how you started the cube tool, *although it is possible that with the still-under-construction version this will not work if you started via right-click on your cube in the HIPE Variables panel*.
- In addition, your creations are (supposed to be) stored in one of two separate products, also listed in the HIPE Variables panel, that were created when you started up the cube tool. If you started with click-selection then the product is currently called "cat", if you started from the command-line (using the syntax of Sec. 2) then it is called "mycuberesults". *However, currently "cat" containing nothing useful and should be ignored.*

The advantage of this is that you can access your cube tool creations outside of its GUI. As with anything listed in the Variables panel in HIPE (and which we assume you are familiar with), you can inspect these new products by right-clicking and choosing one of the viewers offered. You can also use PlotXY; documentation for PlotXY is provided in he HIPE help system.

You can access these products from the command line with the syntax:

```
singlespectrum = mycuberesults.getSinglePixelSpectrum()
```

Here you are extracting into "singlespectrum" the result of the last single spaxel spectral selection that you did in the cube tool. The data type of "singlespectrum" is Spectrum1d. For each of the cube tool products the python syntax for the "get" differs, this is explained in the table below: on the left is the "get" part of the command (the one after "mycuberesults."), the middle is the data type this product will be, the right is the cube tool command that created the product.

Table 5.1. Syntax for extracting cubetool-products from the command line

getSinglePixelSpectrum	Spectrum1d	single spaxel spectrum display
getAvgspectrum	Spectrum1d	region spectrum display
getRangeExtractedCube	SimpleCube	range extraction
getIntegratedMapImages	ArrayList of SimpleImage	Integrated Maps
getVelocityAxisImage	SimpleImage	position-velocity diagram
getVelocityMapCube	SimpleCube	position-velocity diagram

5.2.10.2. Details for specific tasks

This section is for those who may wish to incorporate the cube spectral analysis toolbox in their own scripts or call up individual tasks that the GUI otherwise runs for you. Here we show you the calling syntax and I/O structure. Note that in almost all cases the units of the spectral dimensions are not wavelength or velocity but layer/channel (i.e. array location).

Single spaxel selection

The task to extract a spectrum from a single spaxel is **extractSinglePixelSpectrumTask** and is registered in HIPE as **extractSinglePixelSpectrum**, The task need 3 input parameters:

- **simplecube**, the cube (SimpleCube or SpectralSimpleCube)
- **posX**, the X coordinate of the spaxel we want the spectrum of
- **posY**, the Y coordinate of the spaxel we want the spectrum of

The coordinate must be spaxel, not in sky, coordinates.

The output parameters are

- **finalspectrum**, the spectrum stored as a Spectrum1d
- **spectrum**, an array (1d) of the fluxes stored as a Double1d

So to extract the spectrum from spaxel (4,5) use:

```
myspectrum=extractSinglePixelSpectrum(simplecube=mycube,posX=4,posY=5)
# or
myspectrum=extractSinglePixelSpectrum(simplecube=mycube,posX=4,posY=5).spectrum
```

where

- **mycube** is in SimpleCube format

- **PosX**,**Y** are the X,Y coordinates of the spaxel

and the first command creates output in Spectrum1d format, with meta data (taken from the input cube) and the second creates a spectrum of Double1d without meta data. The spectrum has columns of flux, weight, flag, segment (segment for now is just a placeholder, its value everywhere here is 1) and wavelength, this latter being in the same unit that your SimpleCube had.

Note that if you typed the first command and then realised you wanted in fact the second output format, then a way to do this faster than running the second command is to rather type now

```
myspectrum1=extractSinglePixelSpectrum.spectrum
```

because as long as you have not run "extractSinglePixelSpectrum" since running it the first time, this method does not re-run the task on your cube but simply extracts out the result in a different format.

Multiple contiguous spaxel selection

The task to extract a spectrum averaged over a set of spaxels is **extractRegionPixelSpectrumTask** and is registered in HIPE as **extractRegionPixelSpectrum**. The task needs 2 or 3 input parameters, depending of the type of extraction we want.

- **simplecube**, the cube (SimpleCube or SpectralSimpleCube)
- **wholeImg**, a boolean which defines the type of extraction we need: the whole image (True) or a set of spaxels (False)
- **posArray** a Double2d array, of dimensions [somenumber, 3] which contains the X,Y coordinates of the pixel to read and the weight of these spaxels in the averaging process

  Each line contain the information in this order: X Y weight

The output parameters are:

- **finalspectrum**, the spectrum stored as a Spectrum1d
- **spectrum**, an array (1d) of the fluxes stored as a Double1d

To extract the average spectrum from the whole cube use:

```
# output as a Double1d containing the flux
myspectrum=extractRegionPixelSpectrum(simplecube=mycube,wholeImg=True)
# for Spectrum1D format for the output type then after that
myspectrum1d=extractRegionPixelSpectrum.finalspectrum
# or just type (on a single line)
myspectrum1d =
  extractRegionPixelSpectrum(simplecube=mycube,wholeImg=True).finalspectrum
```

Note that you can create a Double2d with wavelength and flux from the Spectrum1d output:

```
flux=myspectrum1d.flux # in Double1d format
wave=myspectrum1d.wave  # in Double1d format
spectrum=Double2d()
spectrum[0,:]=wave
spectrum[1,:]=flux
```

To extract an average spectrum from a region you need to make a Double2d array with columns of [X Y weight], to indicate which spaxels to select, and starting with entry [0 0 0]. Weight will determine by what fraction the spectrum from each X,Y will be multiplied in the average, i.e. can be considered to be an area-weight. Assuming that this array has the name "foo":

```
# output as Double1d
```

```
myspectrum=extractRegionPixelSpectrum(simplecube=mycube,wholeImg=False,posArray=foo)
# or Spectrum1d, type just after that
myspectrum1d=extractRegionPixelSpectrum.finalspectrum
# or only type (on a single line)
myspectrum1d =
  extractRegionPixelSpectrum(simplecube = mycube, wholeImg = False,
  posArray = foo).finalspectrum
# and you can also see the effective area in spaxels you have extracted
totalWeight=extractRegionPixelSpectrum.totalWeight
```

Smoothing filters

This is a long sequence of commands:

```
filt=FilterSpectrumTask()
filt.rawSpectrum = myspectrum
filt.spectralDimension = -”Physical meaning of the spectral axis”
# is a string
filt.spectralUnit = -”unit”
filt.sizeOfSpectrum = sizeOfSpectrum
# sizeOfSpectrum is just an integer that is the length of the spectrum
filt.specIndex = specIndex
# specIndex is a Double1d, previously created, containing the spectral
#   values for the flux
filt.modelFilter = -"GAUSSIAN"
# model to use
filt.widthFilter = 10
# width of the filter in units of array/channel, not spectral units
filt.execute()

# and then
FilteredSpectrum = filt.filteredSpectrum
# is the Double1d array containing the filtered flux
MaxValueFitSpectr = filt.maxValue
# is a double
PosMaxFitSpectr = filt.maxPosition
# is an integer
```

The input is a Double1d, here called "my spectrum" containing the fluxes of the spectrum, i.e. something you created before. For example, if you extracted it using the single/region extraction commands given above and put it in "final spectrum", you then just need to type:

```
mspectrum=finalspectrum.getFlux()
```

Spectral range selection

The task doing the spectral (and/or spatial) extraction from an original cube is **RangeExtractionTask()**. It creates smaller cubes.

The input parameters are :

- **simplecube**, Cube. This is the original cube from which we want to extract a part
- **startIndex**, Integer. The index of the first layer to put in the output cube (i.e. the wavelength/ frequence range to select over). This value is the index, not the spectral value of the layer; this allows one to use this task on all kind of SimpleCubes, even if the 3rd axis is not spectral
- **endindex**, Integer. The index of the last layer in the output cube. This value is also index
- **Crop**, Boolean. This parameter activates the "spatial extraction" when it is True. When False you are doing a spatial extraction of the whole cube but over a limited wavelegth/frequency range.
- **Xmin**, Integer. Minimum x spaxel coordinate for the spatial extraction
- **Xmax**, Integer. Maximum x spaxel coordinate for the spatial extraction

- **Ymin**, Integer. Minimum y spaxel coordinate for the spatial extraction
- **Ymax**, Integer. Maximum y spaxel coordinate for the spatial extraction

There is a coherency check between the crop value and the Xmin Xmax Ymin Ymax parameters. If crop is true the coordinates parameters must be filled.

The output parameters are:

- **rangeCube**, a cube containing the result of a simple spectral extraction
- **specRangeCube**, a cube containing the result of a **spectral and spatial extraction**
- **error**, an integer containing the error code value if a PB occurred: only for internal use and not explained here
- **log**, a string to store the description of the error

This is currently rather awkward to run in HIPE, but if you really want to, next we provide some developer-oriented example scripts.

Since this task can be used to selecting a sub-spectral range for the whole cube or to extract on a sub-spatial domain AND a sub-spectral domain, we give here an example of each.

Only spectral range extraction:

```
rangeextraction=RangeExtractionTask()
rangeextraction.simplecube=mycube
rangeextraction.startIndex=200
rangeextraction.endIndex=600
rangeextraction.Crop = False
rangeextraction.perform()
# access the results
res1=rangeextraction.rangeCube   #result is stored in a SimpleCube
errcode=rangeextraction.error
logmssg=rangeextraction.log
```

Spatial and/or spectral domain extraction:

```
rangeextraction=RangeExtractionTask()
rangeextraction.simplecube=mycube
rangeextraction.startIndex=200
rangeextraction.endIndex=600
rangeextraction.Crop = True
rangeextraction.Xmin = 1
rangeextraction.Xmax = 8
rangeextraction.Ymin = 3
rangeextraction.Ymax = 9
rangeextraction.perform()
# access the results
res1=rangeextraction.rangeCube   #result stored in a SimpleCube
res2=rangeextraction.specRangeCube # result stored in a SpectralSimpleCube
errcode=rangeextraction.error
logmssg=rangeextraction.log
```

Integration Map

The task making an integrated map from an cube is **IntegrateMapFromCubeTask()**. To run on the command line the integration map you need to create 2 arrays of spectral values. The graphical interface helps you to do this but if you know well the structure of your cube you can prepare these arrays yourself. The first array stores the starting indices of the integration domains, the second stores the ending indices of these domains. For integrating only on one domain the arrays contain only one element each.

The input parameters are:

- **cube**, Cube. The cube from which we wish integrate
- **nbStack**, Integer. The dimension over which the integration is taken
- **startArray**, Double1d. The array of the first indices of each stack of images to integrate
- **endArray**, Double1d. The array of the last indices of each stack of images to integrate

For example, if startArray is (10,40,80) and endArray is (20,50,120) and the 3rd dimension is the wavelength/frequency dimension you want to integrate over, then nbStack will be 3. The integration must always be done on the spectral dimension.

The output is

- **images**, an arrayList of SimpleImages (i.e. the class of images is arrayList, which contains products of class SimpleImages)

Here is an example of a call to this integration map task for the extraction of 3 integrated maps:

```
integr=IntegrateMapFromCubeTask()
integr.cube=mycube
#startarray and endarray store the indices of the layers, not their spectral values

startarray = Double1d(3)
startarray.set(0,10)
startarray.set(1,400)
startarray.set(2,800)

endarray = Double1d(3)
endarray.set(0,60)
endarray.set(1,600)
endarray.set(2,860)   #i.e. here assuming that the depth of the cube is >  860

integr.startArray =startarray
integr.endArray =endarray
integr.nbStack = 3
lstimages = integr.perform()

#results
print lstimages.size()  #returns 3 if everything went well
#access the results:
image1= lstimages[0]   #image1 is a SimpleImage
```

The returned images of this task contain in their header information on the central spectral value and the bandwidth over which the integration was made.

- The bandwidth or size of integration is stored in the keyword BANDWIDTH
- Depending of the details of the original cube the central spectral value is stored in the keywords:
  - WAVELENGTH if the spectral dimension was wavelength
  - FREQUENCY if the spectral dimension was frequency
  - VELOCITY if the spectral dimension was velocity
  - SPECTRAL VALUE if there was no physical dimension available in the cube

Velocities

The operation on velocities are all done in the Task **positionVelocityDiagram().** This task can be use for the PV diagram and the velocity map, therefore there is 2 way to use it and each usage will be described separately.

PV Diagram

To run on the command line for Axis mode you need to make a list of the spaxels to be read into the task, with columns [index, X, Y, weight]. This list of spaxels define a "slit" so there is an axis and a width. The "Index" is the offset along the slit from the beginning, and if the X and Y are in order this will simply be 0,1,2,3..... For slit widths >1 all the spaxels of one "column" have the same index. For example, your "list" can be: [0;4;0;0.5] on the first line, [0;5;0;1] on the second, [0;6;0;0.5] on the third.....

The input parameters of the task are:

- **simplecube**, SpectralSimpleCube, the original cube; as this task needs spectral information it work only with a cube of this class
- **axis**, a boolean that defines the usage of the task. For the PV diagrams (for the "axis" mode) it must be True
- **coordSlitArray**, Double2d. The array listing the spaxels to be read
- **nbpixelsAxis**, Integer. The length of the slit
- **referencelayer**, Integer. The array layer for the velocity 0 (the array being the dimension that contains the spectral values, e.g. frequencies)

The output parameters are

- **velocityMap**, Double3d. Used only for the "map" mode (so see next section)
- **velocityMapAxis**, Double2d. A 2d array containing the PV diagram without meta data: but should not be used
- **velocityMapAxisProd**, SimpleImage. This is the PV diagram stored as an image with an adapted WCS (first axis offset along the "slit", second axis is the spectral units)
- **cubeVelocityMap,** SimpleCube. A cube which contain the velocity map, the dispersion map, the "layer map" (map of the layer index corresponding to the velocity maximum) and the flux map (the flux corresponding to the velocity maximum. This parameter is not output if using the axis mode
- **VelocityMapList**, listArray of SimpleImages. The contents of the previous parameter split into separate SimpleImages, with compatible WCS for each image

The command for the computation of a PV diagram is therefore:

```
# for output of type Double3d
velocityMap =
positionVelocityDiagram(simplecube=mycube,axis=True,coordSlitArray=list,
   widthSlit=1,nbpixelsAxis=15,referenceLayer=200)
# for output of type SimpleCube you then type
cubevelocitymap=positionVelocityDiagram.cubeVelocityMap
# and to access other parts of the creation
velocityMapAxis = positionVelocityDiagram.velocityMapAxis # Double2d
velocityMapAxisProd = positionVelocityDiagram.velocityMapAxisProd # simpleImage
```

The only interesting result is velocityMapAxisProd since this one contain the metadata which are very useful for you to understand what you have created.

Velocity Map

The Velocity Map mode computes a 2D image of the same dimensions as the spatial dimensions of the cube. The values of the pixels are the velocities at each spaxel; other maps are also computed at the same tme. The velocity map and its associated products are obtained via the task **positionVelocityDiagram().**

The Velocity map is created by fitting to every spectrum a Gaussian line; the velocity map corresponds to the peak position of the resulting fit.

A dispersion map is computed using the sigma output parameter of the Gaussian fit.

A "layer of maximum" map is also created, which allows the user to check quality of the result.

Finally a map of the flux at the maximum is created, which also allows for quality checks.

These results are also gathered together in a List of SimpleImages, this list being the output parameter **velocityMapAxisProd**.

Line intensity map

The line intensity map computation is done in the task **LineIntensityMapTask().** The input parameters for this task are:

- **cube**, Cube. The cube from which we wish to compute a line intensity map
- **continuumSubtraction**, Integer. The continuum subtraction mode: 0=no subtraction; 1=fit on an region of the spectrum (this mode needs the parameter regionStartArray and regionEndArray to be filled); 2=continuum subtraction combined with the line fit in one model
- **regionStartArray**, Double1d. The array of the first indices of the regions to be used for the continuum subtraction
- **regionEndArray**, Double1d. The array of the last indices of the regions to be used for the continuum subtraction
- **PolyDegree**, Integer. The degree of the polynomial for the continuum fit
- **fittmodel**, String. The name of the model to be used for the line fitting. In the first release only "Gaussian" is available but in the future Voight profile and sinc will be added. *The name of this parameter will soon change to "fitModel" (one "t")*
- **mode**, String. At present is just a placeholder
- **centralPos**, Double. The guess at the central position, in layer (array position) value, not wavelength/frequency
- **RangeForFitt**, Double1d. An array containing the limit on the validity domain for the fit: is used to reject false detections. Must contain 2 values, the lower and the higher limit, in layer units (array position). *The name of this parameter will soon change to "RangeForFit" (one "t")*
- **typeOfLine**, String. A string which says if the line to fit is an emission line (value="emission") or an absorption line (value=currently anything other than "emission", but later this may change to "absorption")

The OUTPUT parameters for this tasks are

- **lineIntMap**, SimpleImage. The main output, the map of the integrated flux on the fitted line without continuum
- **fittedLineCube**, SimpleCube. The cube containing the final fitted line, without the continuum
- **continuumCube**, SimpleCube.The cube with the fitted continuum
- **cubeWithoutContinuum**, Cube. Contains the original cube after subtraction of the continuum cube: each spectrum has been rectified

- **residualCube**, Cube. The residual cube which contains the "cube minus continuum minus fitted line"

This task was not designed to be used directly from the command line but if you really want to use it, it can be used with a script such as this:

```
lineintensity=LineIntensityMapTask()
lineintensity.cube=mycube
lineintensity.continuumSubtraction=1
startarray=Double1d(2)
endarray=Double1d(2)
#selesct a first range from 10 to 40  and one other from  80 to 140
startarray.set(0,10)
startarray.set(1,80)
endarray.set(0,40)
endarray.set(1,140)
lineintensity.regionStartArray=startarray
lineintensity.regionEndArray=endarray
lineintensity.PolyDegree=2
lineintensity.fittmodel="gaussian"
lineintensity.execute()
linemap = lineintensity.lineIntMap.copy()
fittedLineCube         =lineintensity.fittedLineCube.copy()
continuumCube          =lineintensity.continuumCube.copy()
cubeWithoutContinuum   =lineintensity.cubeWithoutContinuum.copy()
residualcube           =lineintensity.residualcube.copy()
```

Importation Task

ImportSpectralCubeTask and **ImportCubeTask** are two tasks which allow on to load into HIPE cubes stored as FITS fits. These cubes can be SimpleCube or SpectralSimpleCube class, or can be a "standard cube (i.e. cubes from other instruments).

These tasks are automatically used when the user click on "Open File" menu, because of the automatic task interface in HIPE. When the user creates an empty SimpleCube or SpectralSimpleCube and select the import task in the "Applicable tasks" panel of HIPE, this is what happens.

These task can also be used from the command line to manually fill a cube. Both tasks have the same number of parameters, the script to use them are almost the same, so we will explain here the usage of importCubeTask.

The input parameters are :

- **SimpleCube** or **SpectralCube. The empty cube to fill, also the output of the task, as this cube is modified by the task**
- **filename** String. The complete filename (path+name) of the FITS file to open

The importcube task can be use via a set of commands:

```
d=SimpleCube()
importCube(simplecube=d,filename="/home/agueguen/_WorkHipe/fitsfiles/
simpleCube_sinfonie.fits")
```

The task importspectralcube must be called in the same way.

Remark on Gui without specific tasks (cube Comparison)

The cube comparison tool is a graphical feature which doesn't have a specific associated task. This GUI uses various tasks already explained here (**ExtractSinglePixelSpectrumTask** and **ImportSpectralCubeTask**) and tools coming from other packages of HIPE. It is anyway more viewer than a selection, extraction or computation tool and it doesn't create or save anything.

5.2.11. SpectralSimpleCube panel for the spectrum explorer

5.2.11.1. The spectrum explorer

The spectrum explorer is a powerful tool to display and analyse Herschel spectra. All *spectrum container* data can be show in this toolbox.

Spectrum containers

A *spectrum container* can be seen as a set of one or more spectra.

Spectrum containers containing many spectra may have spatial coherency, cover a large spectral domain for the same sky position, have many *segments* and so on.

Graphical interface

The graphical interface of the spectrum explorer is split into two areas:

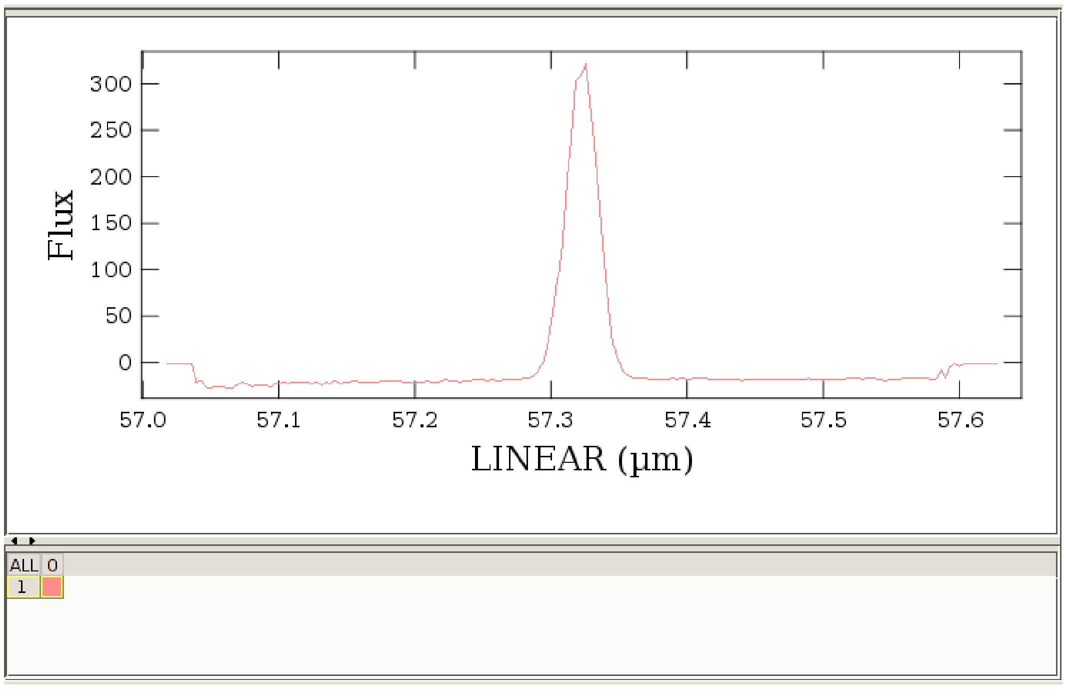


Figure 5.30. The graphical interface of the spectrum explorer

The upper part allows you to zoom, select spectra and so on (see the spectrum explorer section).

The lower part in its default mode show the list of the spectra in the spectrum container, with one line per spectrum. This list can become extremely long for large cubes.

For spectra in a cube corresponding to different sky positions it is not easy to identify which spectrum correspond to which position.

SpectralSimpleCube panel

It is possible to define panels for the lower part of the spectrum explorer. These panel can be registered for a given type of spectrum container. The SpectralSimpleCube is a spectrum container, so it is possible to define a panel for this type of spectrum container.

The following figure shows the panel for SpectralSimpleCube.

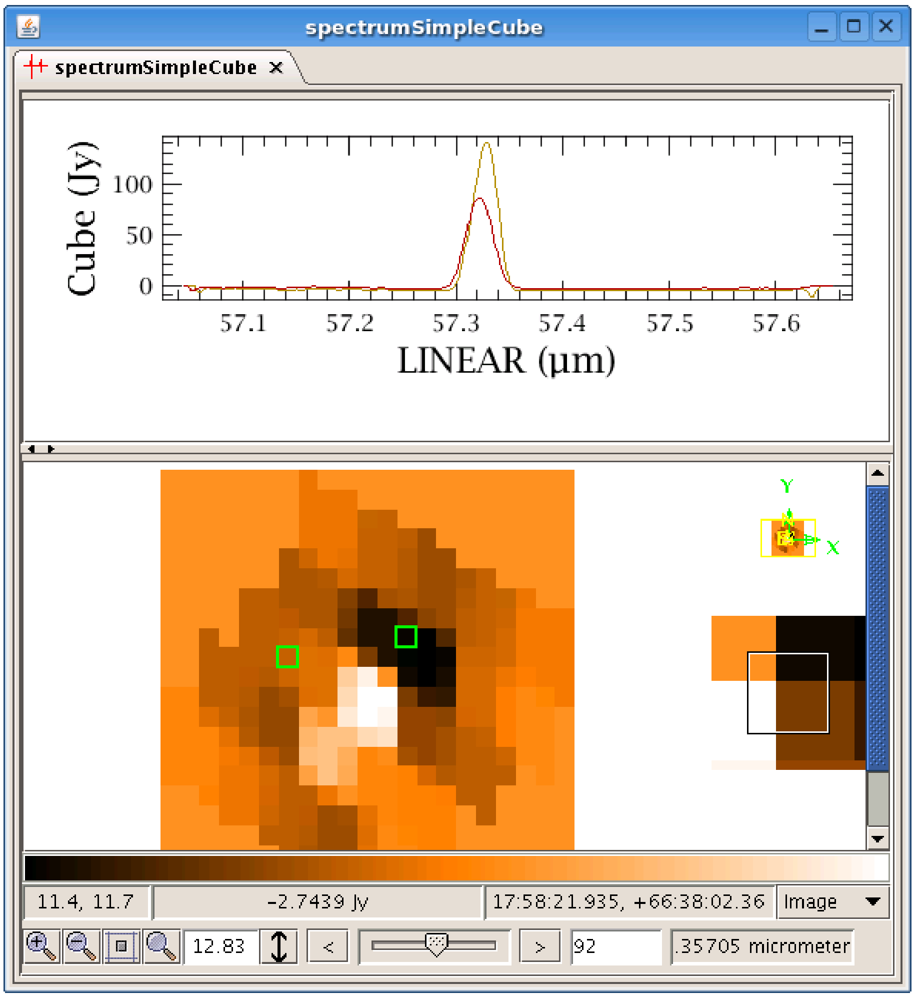


Figure 5.31. The panel for SpectralSimpleCube

From this display its possible to select one or more spectra to be shown in the upper part (spectrum plotter) by right-clicking on the display and choosing the selection mode, either rectangle region or single pixel.

- In the single pixel mode, every click selects a new spaxel. Click on a selected spaxel to unselect it.
- Currently the region mode offers only rectangular shapes. Click and drag the mouse to define the rectangular area. A green rectangle appears after you release the mouse button. Note that the rectangle does not appear *while* you are dragging.

With these two modes you can select a rectangle and then remove pixels from inside it, thus creating a selection with *holes*.

A cube of type `SpectralSimpleCube` is opened by default in this viewer. We recommend the following steps to use the viewer in the most efficient way:

1. After opening the cube, undock the viewer by dragging its tab away from the main HIPE window.
2. Resize the viewer and move the central divider bar so that you can see the image in the lower pane *and* the plot in the upper pane.
3. Click the *zoom to fit* button at the bottom of the window.
4. Drag the slider at the bottom of the window to move away from the last wavelength, which is shown by default.

5.3. In depth

5.3.1. Fitting spectra from the command line

1. Create an instance of the fitter. Let us suppose that your spectrum is assigned to a variable called `data`.

   ```
   from herschel.ia.toolbox.spectrum.fit import SpectrumFitter
   sf=SpectrumFitter(data)
   ```

   Assume that the data looks as shown in Figure 5.32.

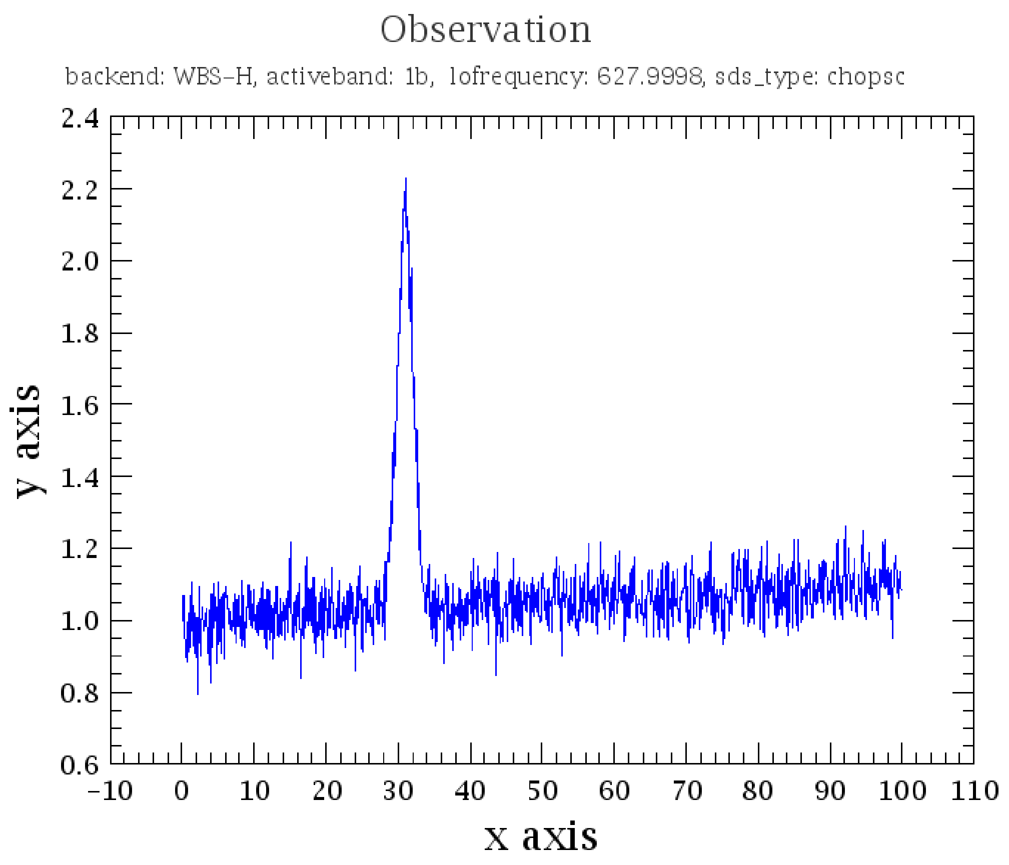
   

   Figure 5.32. Test data to fit. Start the SpectrumFitter

2. The SpectrumFitter is an interactive tool and is best used in conjunction with the SpectrumModel tool, which allows you to select (and change) models and fitting parameters. The three models you are most likely to use are Gaussian, Lorentzian and Polynomial; the model fits, their parameters, and their usage in the SpectrumFitter tool are summarized in Table 5.2:

   Table 5.2. Model fits, their parameters and usage in the SpectrumFitter tool

Model	Mathematical fit	Parameters	Usage
Gaussian	$f(x) = a_0 \, exp\left\{\frac{-(x-x_0)^2}{2s_0^2}\right\}$	a_0 = amplitude of line	`sf.addModel ('gauss', [a0,x0,s0])`
		x_0 = location of line peak	

Model	**Mathematical fit**	**Parameters**	**Usage**
		s_0 = width of line (sigma)	
Lorentzian	$f(x) = p_0 \left[\frac{p_2}{(x-p_1)^2 + p_2^2} \right]$	p_0 = amplitude of line	**`sf.addModel ('lorentz', [p0,p1,p2])`**
		p_1 = location of line peak	
		p_2 = half width at half maximum of line	
Polynomial	f(x) = $c_0 + c_1 x + ... + c_n x^n$	n = order of polynomial	**`sf.addModel ('poly', [n], [c0,c1, ..., cn])`**
		c_0 .. c_n = polynomial coefficients	

Note that you must know (roughly) where you expect a spectral feature in your data to be, in addition to its expected shape and approximate 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.

Now, fit first the baseline with a polynomial and then fit the line with a Gaussian.

```
# First the baseline
# Apply the model
model=sf.addModel('poly', [2],[0,0,0])
# Do the fit
sf.doFit()
# Inspect the residual after the baseline is removed
sf.residual()
# Keep the fit
sf.fitOK()
# Now the line
sf.addModel('gauss', [1.0,30,0.1])
sf.doFit()
sf.residual()
sf.fitOK()
```

These steps result in the plot below. A black line (not seen here) displays the model and is replaced by a green line showing the fit (the Gaussian model here). The red line is the final fit for the entire spectrum. The residual is shown in a separate plot.

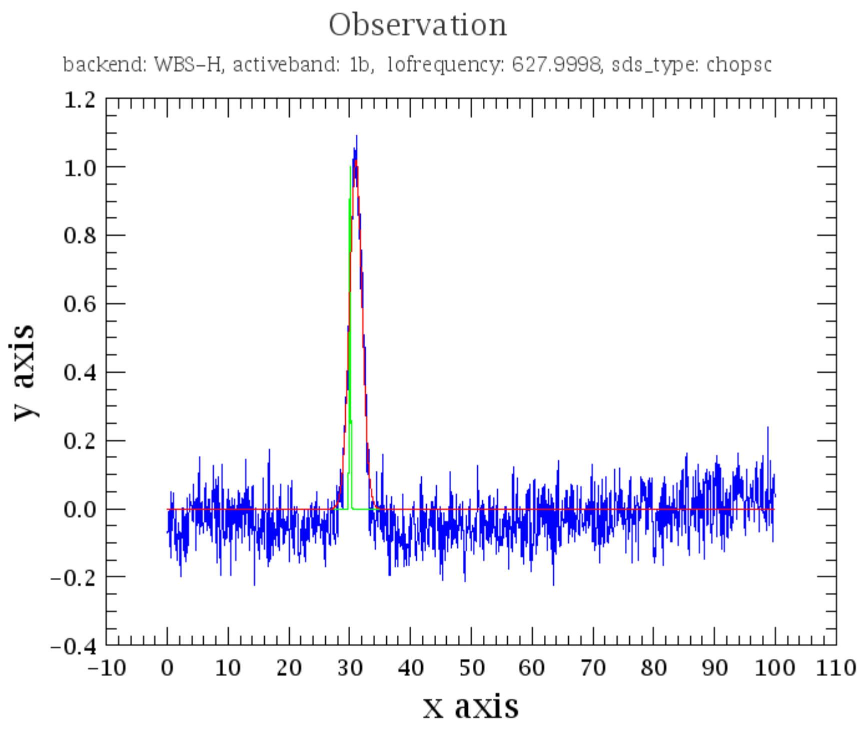


Figure 5.33. Fit result. Fit results for spectrum

3. It is possible to do both fits at the same time, globally, since the instance of our SpectrumFitter remembers what it has done so far.

```
sf.doGlobalFit()
```

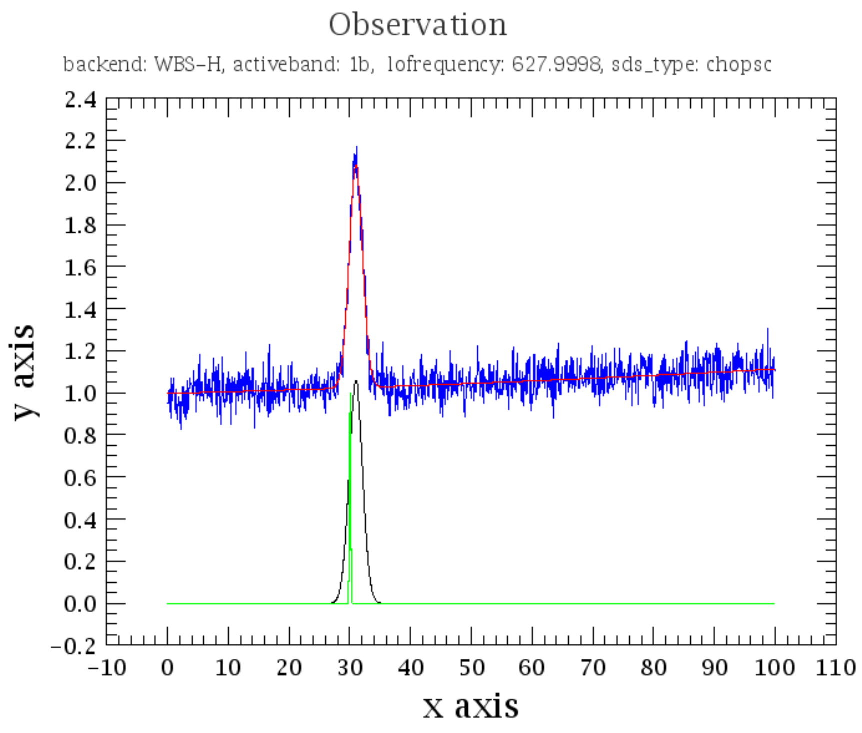


Figure 5.34. Global fit. Use the models together in a global fit

4. It is also possible to mask data. The following will do a polynomial fit only using data from 0 to 20 and from 40 to 100.

```
model=sf.addModel('poly', [2],[0,0,0])
# After you've created the model, add the masks.
model.setMask(0, 20)
model.setMask(40, 100)
```

   To best see how this works, include this masking in the example given above.

   After you have added a mask, you can also *remove* and *invert* it. You do this with the `unsetMask` and `invertMask` methods, respectively, as shown in the following code:

```
# Remove the first mask set in the previous example
model.unsetMask(0, 20)
# Invert the remaining mask
model.invertMask()
```

   After the above code, the mask will cover the whole spectrum *except* the (40, 100) interval.

5. The fitted model parameters and their standard deviations are printed to screen with:

   `print sf`

6. It is possible to manipulate the models produced by SpectrumFitter in various ways:

   • If you wish to change the initial parameters of any of the models (`model = sf.addModel(...)`), use setParameters:

     `model.setParameters([...])`

A new fit will be made on the fly.

- There are two ways to remove models:

  ```
  sf.removeModel(m)
  ```

  Or:

  ```
  m.remove()
  ```

- Subtract the model from the dataset:

  ```
  sf.subtractModel(m)
  ```

  This also removes the model from the fitter tool.

- 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.

- To apply them to a different dataset:

  ```
  sf.setData(otherData)
  ```

  Note that 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.

Chapter 6. External tools

6.1. Summary

This chapter explains how HIPE can communicate and exchange data with external applications:

- In Section 6.2 you will learn how to exchange data with applications implementing the *Virtual Observatory* standards.
- In Section 6.3 you will learn how to communicate with a wider range to external applications by using FITS files as a common standard for data exchange.

6.2. How to

6.2.1. Interoperating with the Virtual Observatory

The Virtual Observatory is a community-based initiative. A number of national and international projects are organized in the International Virtual Observatory Alliance, whose mission it is to *"facilitate the international coordination and collaboration necessary for the development and deployment of the tools, systems and organizational structures necessary to enable the international utilization of astronomical archives as an integrated and interoperating virtual observatory"*. This initiative has led to the definition and implementation of technologies on different topics: Integration of applications (such as HIPE and spectral analysis tools), integration of general data analysis tools and archives of different missions, and more.

The most relevant technology at this point is the integration of applications. In practical terms, the integration means being able to view and manipulate data in one application, send it to another application with the click of a button, view and manipulate the data there, and send it back to the original application. The technology currently used for this interaction is *Plastic*, to be replaced in 2009 by *SAMP*. Plastic and SAMP are very similar, but Plastic was intended as a prototype and SAMP consolidates the protocol, resolving various minor issues. HIPE version 2.0 supports both Plastic and SAMP. Support for Plastic will be dropped in a future version of HIPE.

Plastic and SAMP work using a so-called *message hub*: an independent, very light-weight application (the hub) is started on your desktop and all applications interested in communicating register on the hub. The communication works by sending a message to the hub, which will deliver it to the intended target application (broadcasting to all applications is also an option). The message can contain the data that is sent, but generally the data will be written to a temporary file, and the message is used to pass the location of the temporary file, plus additional information, such as the units of the data.

All ESA archives are VO-aware already, but access to VO-aware archives in HIPE is not available yet. Aladin provides an interface already to many data sources (such as the ESA archives). So it is possible to access the ESA archives by retrieving the data using Aladin and sending it to HIPE from there.

6.2.1.1. Getting practical: Sending products to other applications from HIPE

The HIPE main window provides the File → Interop, with options to register with a hub (a hub will be started automatically if none is found to be running already), unregister from the hub and send products to other applications.

To send a product from HIPE to another application, such as VOSpec, launch the other application. The button *External Tools* on the HIPE Welcome page lists a number of VO applications, and provides buttons to launch the applications (among these are Aladin and VOSpec). In HIPE, choose File →

Interop → Connect to the VO to connect to the Plastic and SAMP hubs. An icon at the lower right corner of the HIPE window shows whether the VO connection is active (●) or inactive (●).

Note

If the above command gives an error, you have found a bug in the software. Please report it.

The File → Interop → Send Product To menu will now be filled with all applications that have registered with the hubs. Normally this will already include the other application we just launched, because many applications connect to Plastic at start-up. If the other application is not listed in the Send Product To menu, make sure the application is registered with Plastic and/or SAMP.

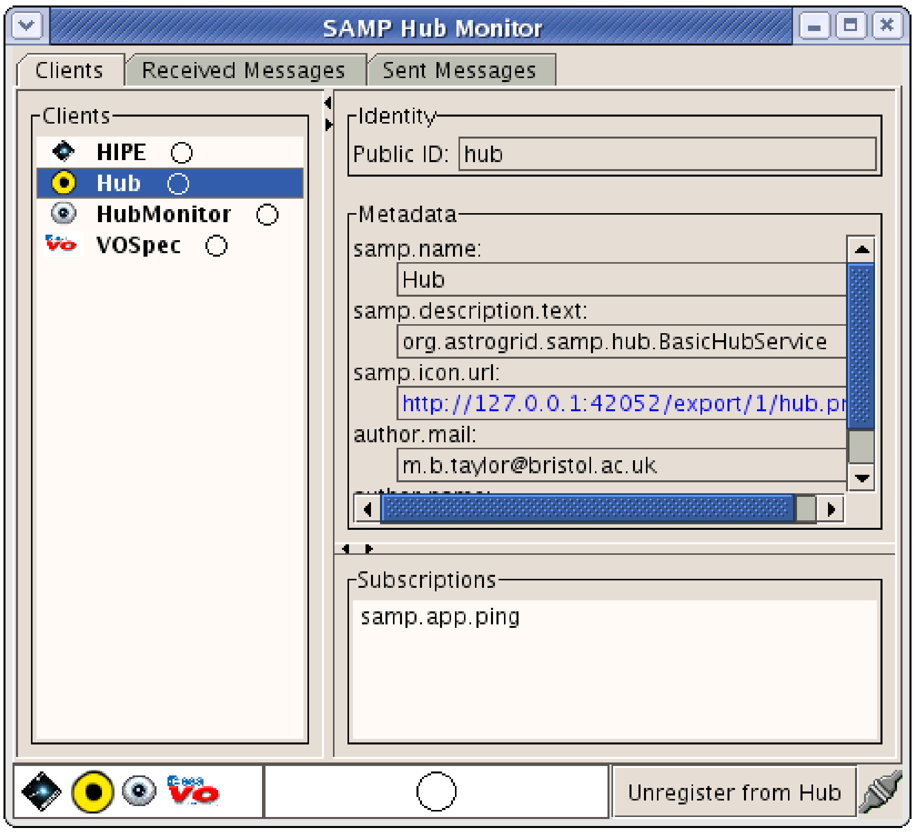
To send a product you first have to select it, for example by selecting its corresponding variable in the Variables View. Then simply choose the desired application from the File → Interop → Send Product To menu, and the product should appear in the chosen application. Note that this can only be done if there is an overlap between the VO interfaces supported by HIPE and the other application. If the applications have no supported interface in common, no data can be exchanged. This is indicated by the external application appearing in grey in the Send Product To menu.

An application may appear with two entries, one for Plastic and one for SAMP. You can choose either option to send a product.

Note that only *sending* of data is supported. To receive the product back into HIPE, it must be sent from the other application. Refer to the documentation of that application to find out how this is done.

You can disconnect from the hubs by selecting File → Interop → Disconnect from the VO.

When you select File → Interop → SAMP Hub Status the following windows appears:



Public ID: hub

samp.name: Hub
samp.description.text: org.astrogrid.samp.hub.BasicHubService
samp.icon.url: http://127.0.0.1:42052/export/1/hub.p
author.mail: m.b.taylor@bristol.ac.uk

samp.app.ping

Figure 6.1. The SAMP Hub Monitor window.

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

6.3. In depth

6.3.1. Interoperating with external software

HIPE offers a complete solution for reducing, visualising and analysing your data. However, for a variety of reasons you may want to do some processing with other astronomical or data analysis software, such as IDL or IRAF. This section explains how to do that.

Any data processing, whether done through an official pipeline or a custom script, is a series of *Tasks* applied on *Products*, like in the code fragment below. For more information on Tasks, see Chapter 4; for more information on Products, see Section 2.14. Both are in the *Scripting and Data Mining* guide.

```
...
product_2 = TaskA()(product_1)
product_3 = TaskB()(product_2)
...
```

Any Task can output a Product representing the state of processing up to that point. For example, `product_2` is the result of processing by `TaskA`, before `TaskB` is applied.

To continue processing outside HIPE, you only have to export a Product to FITS format, as explained in Section 1.4.3. See also the `simpleFitsWriter` entry in the *User's Reference Manual*: Section 2.359.

You can start processing outside HIPE with the `system` instruction. For example, to launch the **myCommand** command insert the following in your script:

```
os.system('myCommand')
```

For this to work you need to import the `os` module first:

```
import os
```

The **myCommand** executable could be, for instance, a shell script with further processing instructions. Whatever your external processing, it *must* accept as input the FITS file produced by HIPE, and *must* output another FITS file that can then be loaded into HIPE again. For more information on how to load a FITS file into HIPE, see the `fitsReader` entry in the *User's Reference Manual*: Section 2.128.

A script with part of the processing carried out outside HIPE would look something like this:

```
import os
aProduct = aTask()(inputProduct)
simpleFitsWriter(product = aProduct, file = -"aProduct.fits")
os.system('myCommand') # Reads aProduct.fits, produces output.fits
outputProduct = fitsReader(file = -"output.fits")
```

Warning

The HCSS system is specifically written and optimised for Herschel data and should be your first choice for data analysis. If you have to resort to external software because of a shortcoming in the HCSS, please raise a ticket so this can be corrected.