# Herschel Data Analysis Guide

## Formerly known as HowTo Documents

# 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.

> **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: Preface.

> **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.

The chapter covers the following topics:

• Retrieving data from the Herschel Science Archive.

• Saving data products into *pools* for easy retrieval and querying.

• Saving images and tables as FITS files, and retrieving data from FITS files.

• Creating and reading ASCII data files.

• Saving and restoring variables from your scripting sessions.

## 1.2. How to

### 1.2.1. Retrieving Data from the Herschel Science Archive User Interface

Data can be accessed directly from the HSA using the Herschel Science Archive User Interface (HUI; see Figure 1.1).
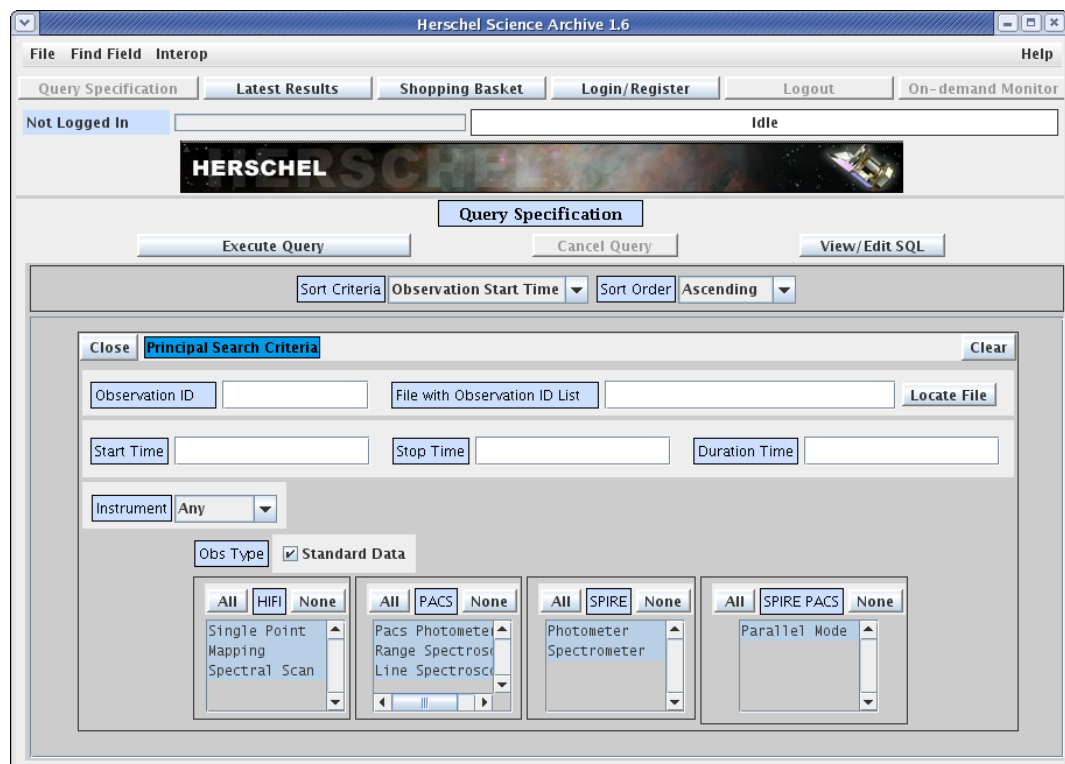


**Figure 1.1. The Herschel Science Archive interface.**

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.

There are to ways of retrieving data from the HUI:

1. Directly downloading data items one by one

2. Via a *Shopping Basket* filled in with a number of observations

Both methods allow the same retrieval options:

• *Auxiliary* to retrieve only auxiliary files (pointing product, event logs, etc...)

• *Calibration* to retrieve only calibration files

• *Level 0* to retrieve only level 0 products

• *Level 0.5* to retrieve only level 0.5 products

• *Level 1* to retrieve only level 1 products

• *Level 2* to retrieve only level 2 products

• *ALL* to retrieve the whole "Observation Context" which includes all the files listed above

• *MULTIPLE* can be used to combine any of the above options in a single retrieval request

| | **Note** |
|---|---|
| | Currently, the option *ALL* should be selected for importing observations into HIPE and for running pipeline scripts. For other options in HUI, like the retrieval of only Level2 products, the provided FITS files can be read with HIPE using the task `fitsReader`. |

In order to retrieve data from the HSA you have to be a registered Herschel user. 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 1.2).



**Figure 1.2. Login/registration in the HSA.**

You do not need to register if you intend solely to browse the content of the archive. Login is, however, a precondition to retrieve data from 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.

After executing a query in HUI, you are automatically moved to the *Latest Result* page (see Figure 1.3) which contains the list of observations matching the query.



**Figure 1.3. Result of query of the HSA**

## 1.2.1.1. Direct data download

A direct download button (Retrieve) is available on this page next to each observation record. This button activates an FTP session, downloading a tar file with the data products corresponding to that observation.

The usage of this button is only recommended for individual observations.

## 1.2.1.2. Retrieving data through 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.
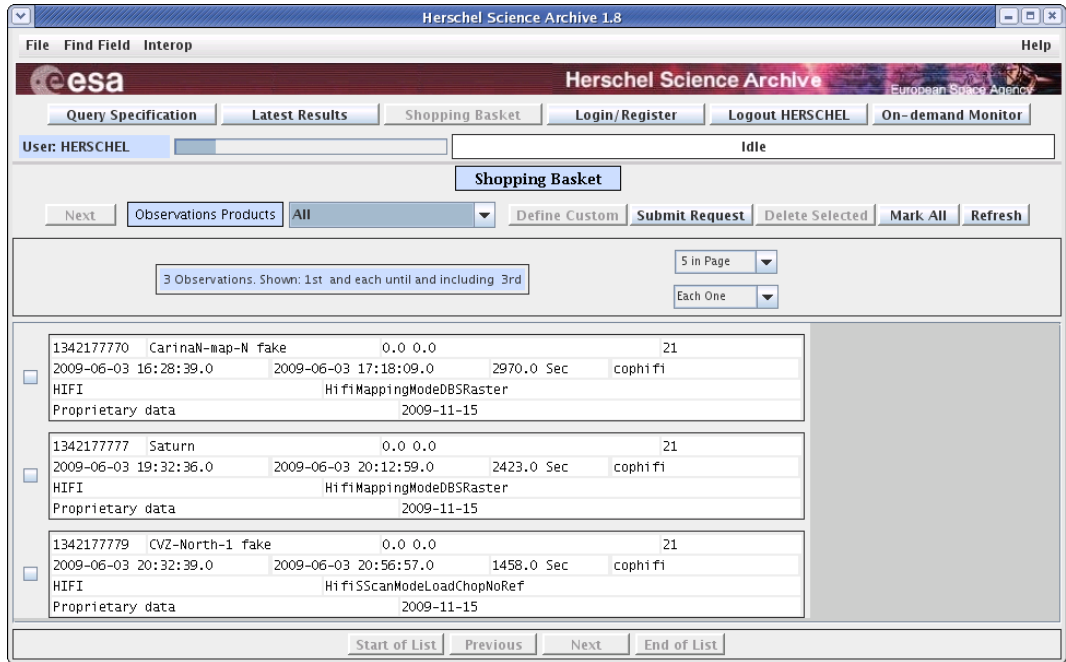
**Figure 1.4. The shopping basket of data to retrieve from the HSA**

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 shall now have disappeared from the *Latest Results* page and appeared in the *Shopping Basket* page.

To request *all* the observations that a query produced click on the *Move All to Basket* button. This will move all the observations into the Shopping Basket and it will also display the Shopping Basket page so that the request can be done straight away.

The shopping basket contains the instantaneous list of all the observation records, whose data are available for retrieval. Items can be deleted 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 the shopping basket has been filled with all the data you intended to retrieve, you are asked to finally confirm the choice, by submitting the data retrieval request. The generation of the dataset to be retrieved is then started (see Figure 1.5).
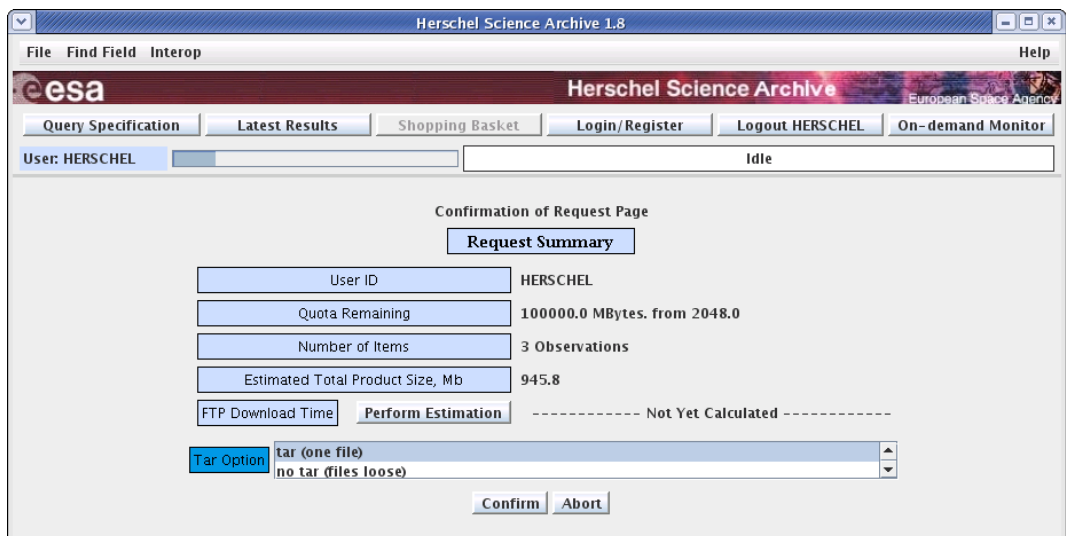


**Figure 1.5. Data retrieval request in the HSA**

The last step prompts the overlay of the Request summary panel. In this panel the number of requested observations, the size of the dataset and the remaining quota are reported. Return back to the Query specification or to the Latest Results page is possible in order 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`). The request should be confirmed with the button at the bottom of the panel and then a request ID is shown. This means that the generation of the dataset has started. An e-mail will be sent to the request originator 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 the next section for details).

> **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 1.3.5.2.

# 1.2.2. Accessing HSA Data within HIPE

There are two methods to get HSA data into HIPE:

1. Importing the HSA tar file into HIPE

2. Directly accessing the HUI from HIPE

## 1.2.2.1. 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 1.2.3 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/
EVERDUGO9525728.tar
```

Open the *Import Herschel data to HIPE* view (see Figure 1.6). 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.
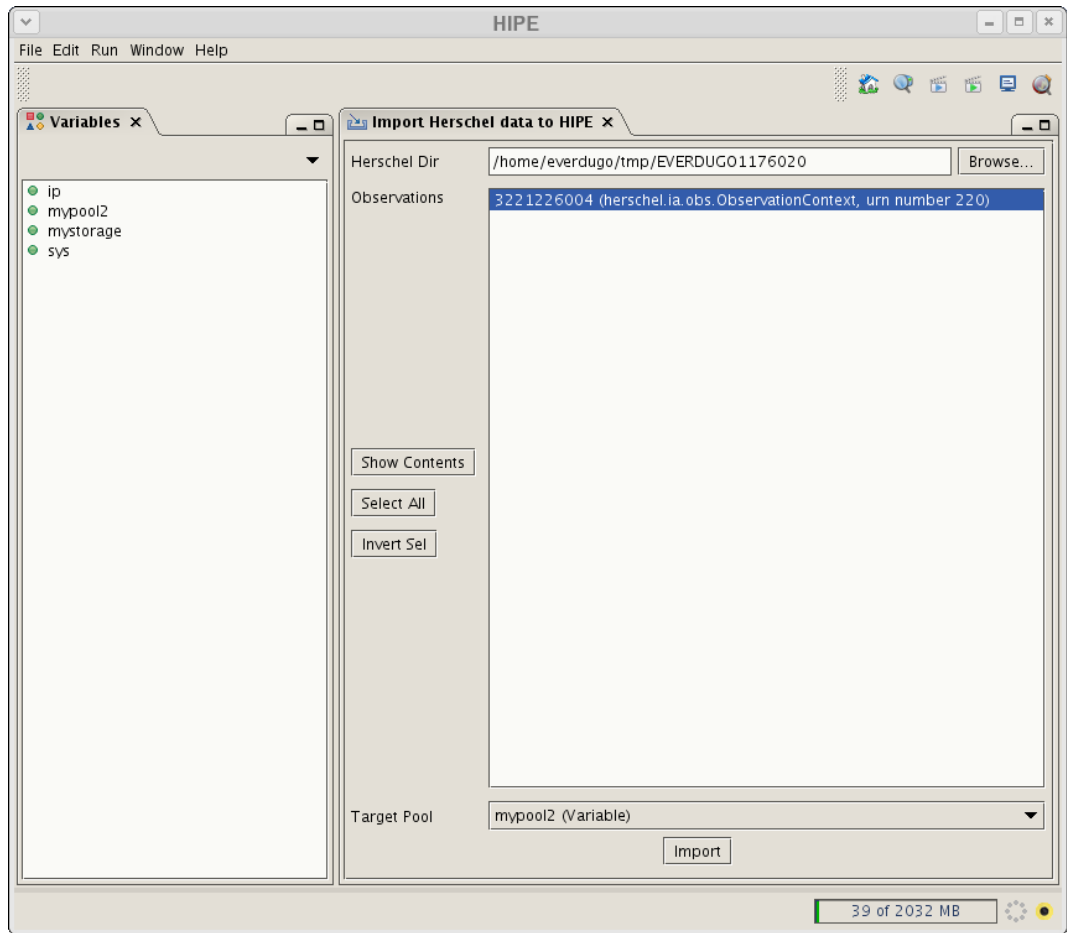
**Figure 1.6. Product loading into HIPE from the HSA tar file.**

This view provides the following:

- A way to navigate and find the directories extracted from the HSA tar files.

- A way to select the pool.

- A way to scan the observations (inside the pool).

- A way to select multiple observations for a bulk import-export.

- A button to generate a sequence of task execution requests: this also provides as a consequence, logging, tracking and *replayability*.

In the same way, pools can be exported 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 1.7).
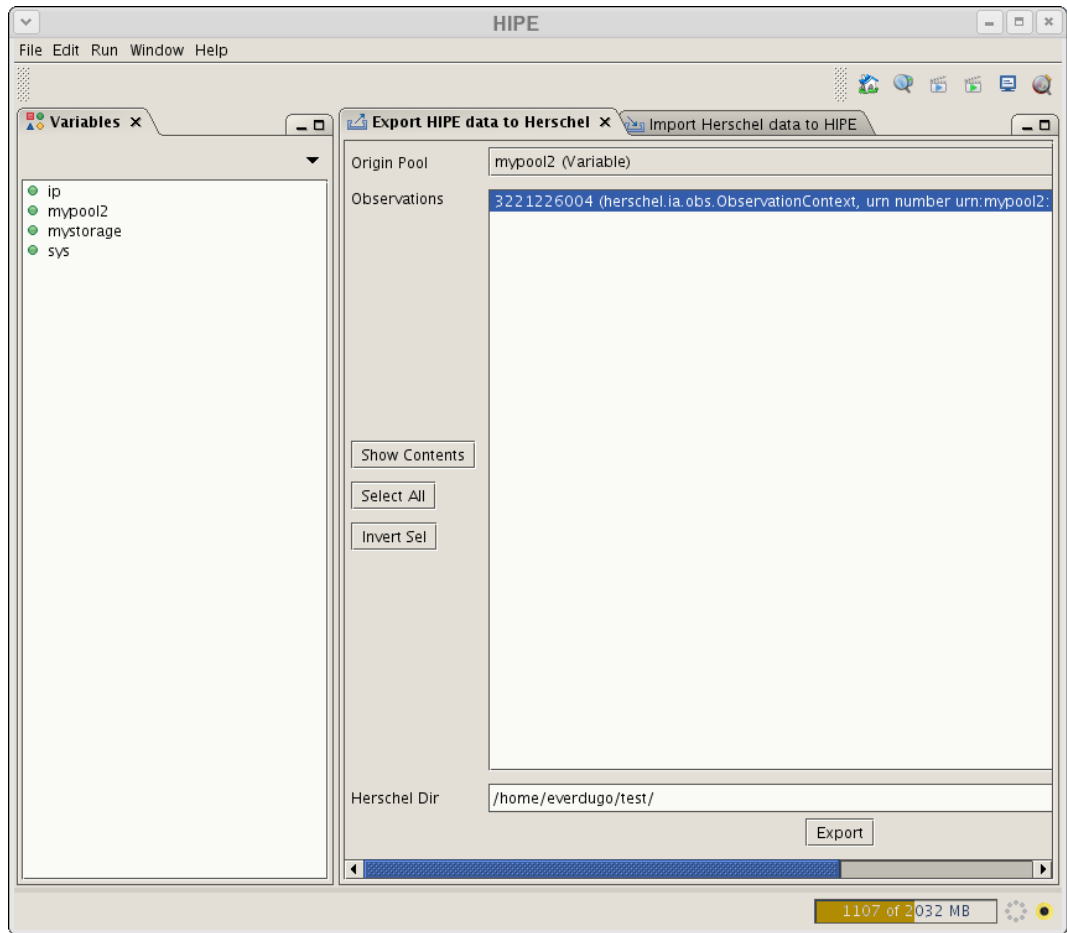
**Figure 1.7. Product export from HIPE into standard Herschel directory structure.**

**Note**

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

- `importUfDirToPal`: Section 2.197.

- `exportPalToUfDir`: Section 2.112.

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

## 1.2.2.2. Direct HUI access from HIPE

This method provides a fast way to visualise products without having to save them first.

To access the HUI from HIPE, click on the Herschel Science Archive icon on the *Data Access* page or select the *Herschel Science Archive* view via the Windows menu (see Figure 1.8).
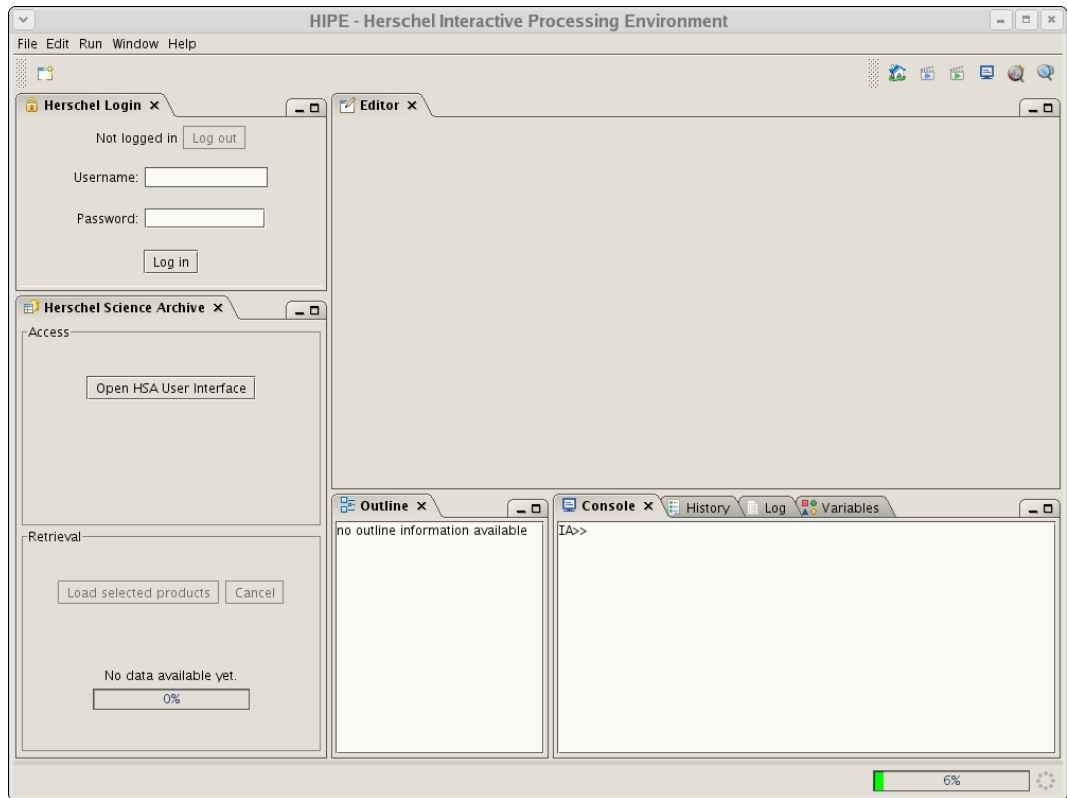
**Figure 1.8. HIPE perspective for the HSA**

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.

Query the HSA for the data to be retrieved and press the button *Send to External Application* next to the observation records. If the connection between the two applications (HUI and HIPE) is well established, you will see in HIPE a message indicating that there is one product ready to be loaded (as in Figure 1.9). Many observations can be sent to HIPE which will be accumulating the products ready to be loaded at once.
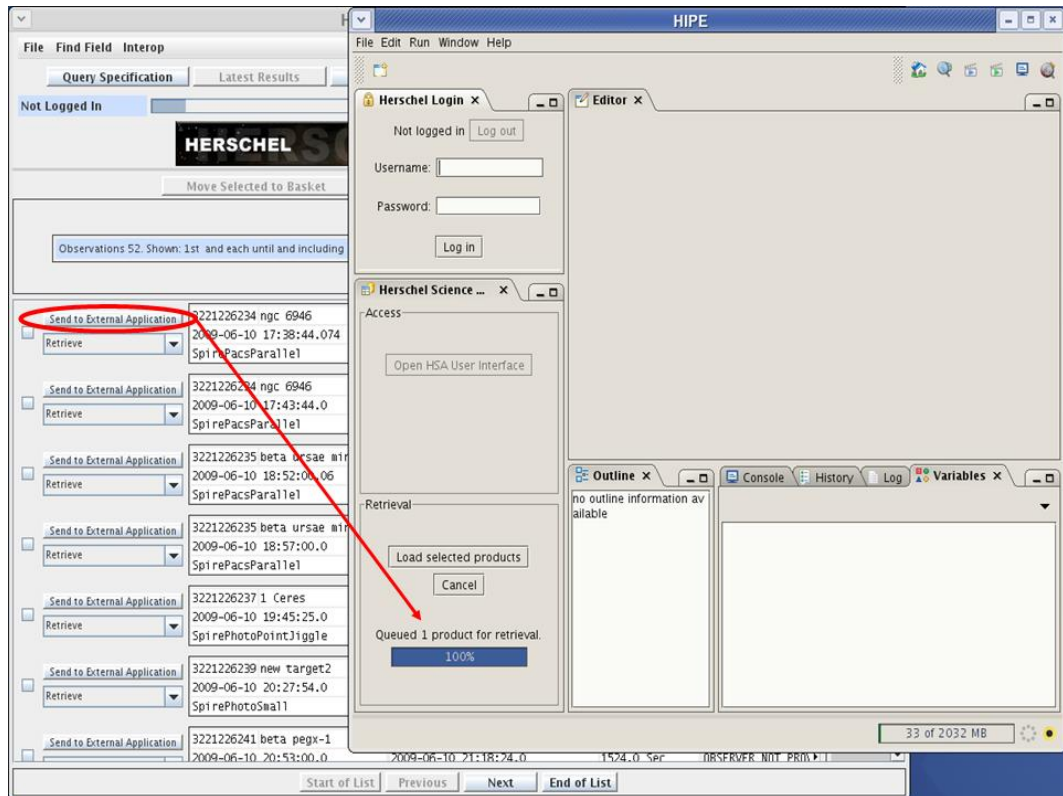
**Figure 1.9. Retrieving observations from the HSA into a HIPE session.**

Press *Load selected products* and the data will start to be loaded into HIPE (see Figure 1.10). During the operation an indicator shows in the HIPE display that loading is taking place and the system is busy.
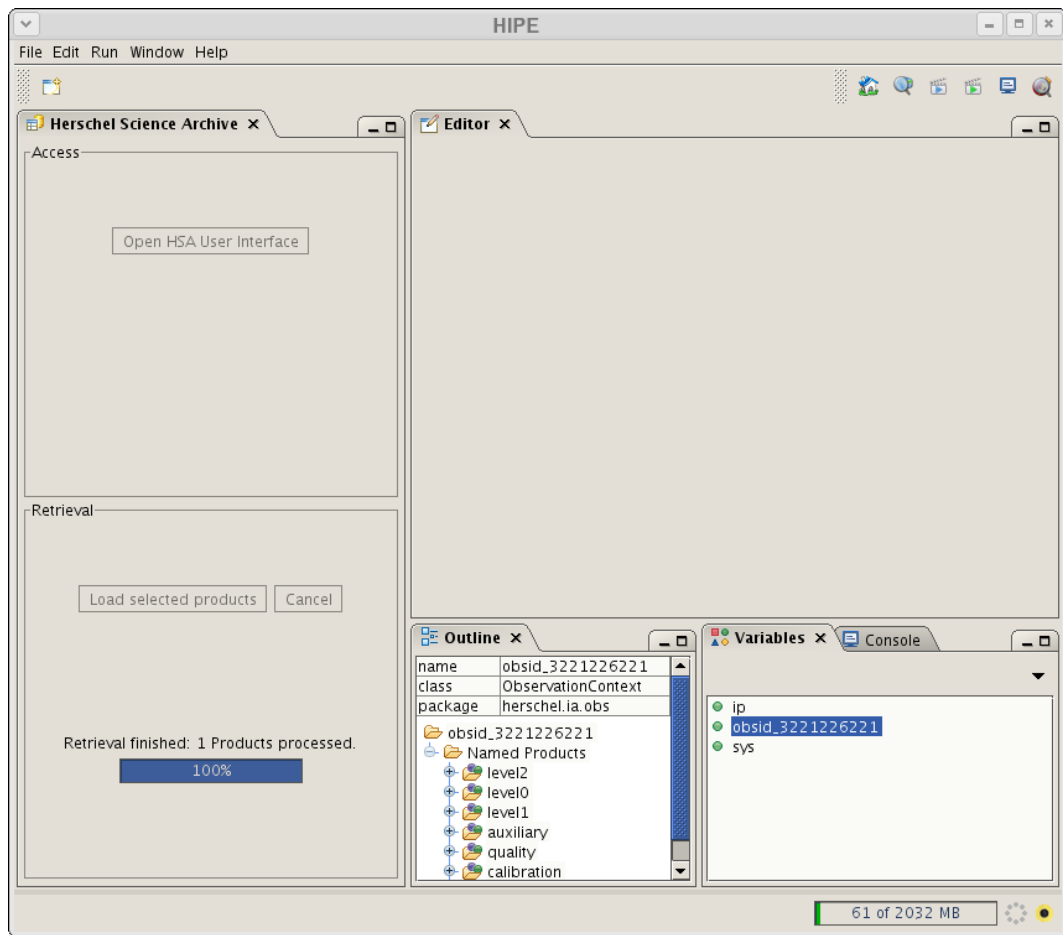
**Figure 1.10. Product loaded into HIPE from the HSA.**

Note 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 using the *PAL Storage Manager* and *Save Products to Storage* views (see Section 1.2.7).

# 1.2.3. Creating and saving products in a pool

Any product (an example being a complete observation in the form of an ObservationContext) can be placed in a pool, or storage area, on the hard disk of the user. You can find advanced information on this topic in the *Scripting and Data Mining* guide: Appendix A. In this section we will simply illustrate how to set up a set of stores (which act a bit like mini databases) in which a user 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.

```
store = ProductStorage() # -"store" is the name (handle) we will use
                         # in our session to place things
myPool = LocalStoreFactory.getStore("myTestPool") # -"myTestPool" is actually
                            # the directory name where the store will be
store.register(myPool)    # now link it/register it under the name of store
store.save(prod1)     # -....and now we merrily add our products called
store.save(prod2)     # -...."prod1", -"prod2" and -"prod3" into the store.
store.save(prod3)     #
```

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 directory on your disk where the data physically resides in the directory ".hcss/lstore" which you will find under your login directory. You will see that the information is actually held as a hierarchical set of FITS files that can be treated like a database, allowing us to query and search for data in similar ways to other databases.

# 1.2.4. 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.

```
#import Configuration components into the environment
from herschel.share.util import Configuration
# 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()  # tell the system it is a store of products
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
```

**Note**

The process of registering/adding pools that the user can use in a session is expected to be made into a simplified tool in the future.

# 1.2.5. Data access via the HIPE GUI

Other than interactions with the Herschel Science Archive (which are discussed elsewhere in this chapter), access to data is via the Product Browser perspective. You can open the Product Browser perspective by clicking on the *P* icon on the HIPE toolbar, or by choosing Window → Show Perspectives → Product Browser.

## 1.2.5.1. Types of stored data

All data is stored in the form of `Products`. These products are kept as FITS files on the local computer system, but are organised into pools of Products/FITS files. This allows querying on the contents of the `Products` (e.g., the metadata or header information). The `Product` wraps information such as images, spectra or tables of data into a storable component. An example is a single Herschel observation (which actually has several products wrapped up into one).

When the user obtains data from the Data Access view it enters the DP session as a `Product` and an overview can be obtained using the "Product Viewer" via a right-click on the name of the product in the session (see information on viewers in the HIPE overview chapter). Datasets such as tables or spectra contained inside the products can be accessed and viewed using Dataset or Spectrum Viewers as described later in this HowTo.

## 1.2.5.2. The Product Browser Perspective

The *Product Browser Perspective* is a graphical HIPE tool to simplify the retrieval and analysis of Products from storages.

> **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 only implemented in prototype stage.

Open the *Product Browser perspective* in HIPE: Window → Show Perspectives → Product Browser.

## A visual tour of the product browser perspective

The following image shows the *product browser perspective* user interface. The screen is divided into four areas:

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.
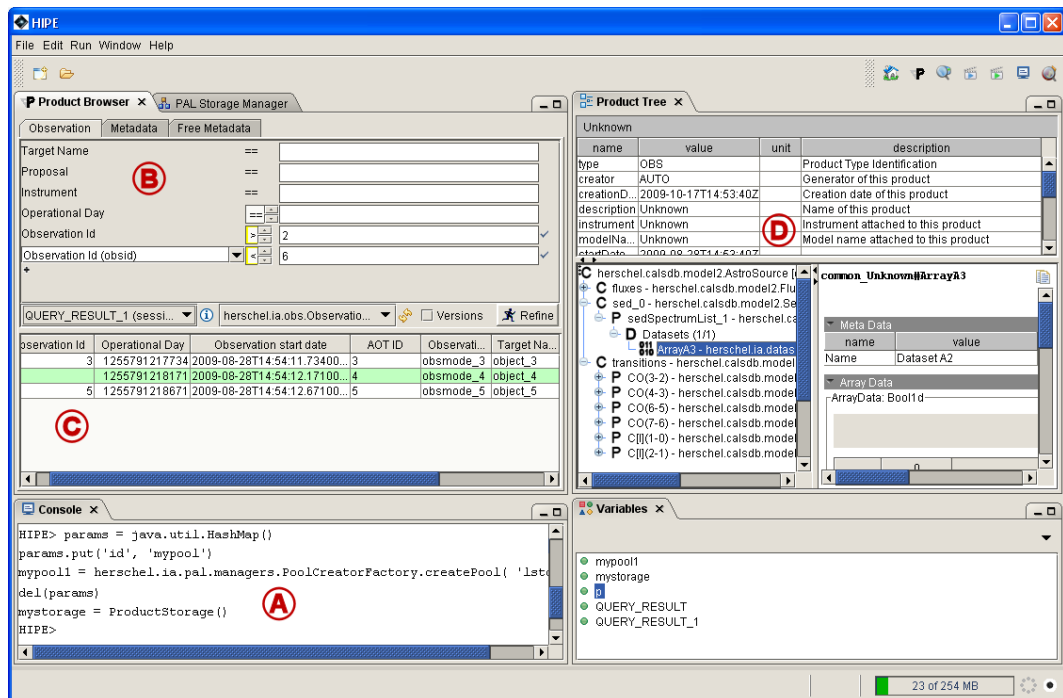


**Figure 1.11. The Product Browser Perspective**

The following sections describe each area in details.

## Cookbook

1. Create a ProductStorage in the Jython Console (A) or in the *PAL Storage Manager* tab in the query area (B).

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:

- 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 products will be returned.

- 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.

- 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.

- 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 select columns. Only checked columns will be shown.

- Right click the column header to add columns. The menu entry "Predefined..." 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 browser 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.

# 1.2.5.3. 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 1.2.4. All pools currently need to be explicitly "registered" to tell the system where to look.

## 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 1.12).

**Figure 1.12. 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 1.13) and it is possible to do a search on values in this given set of attributes -- which are listed in the query interface.



**Figure 1.13. Attributes available for search.**

## Search by metadata

Meta data (like FITS header data) is data more specific to a given observation (See Figure 1.14).
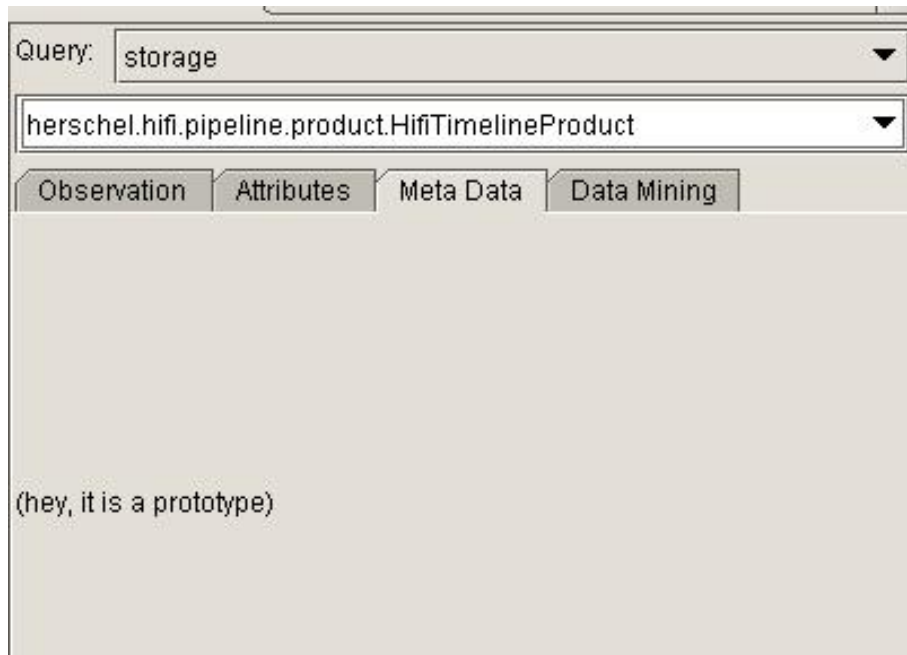
**Figure 1.14. Metadata search.**

## Search by data mining

For data mining it possible to search on specific information contained within the science data itself rather than the meta data. YET TO BE IMPLEMENTED (See Figure 1.15).
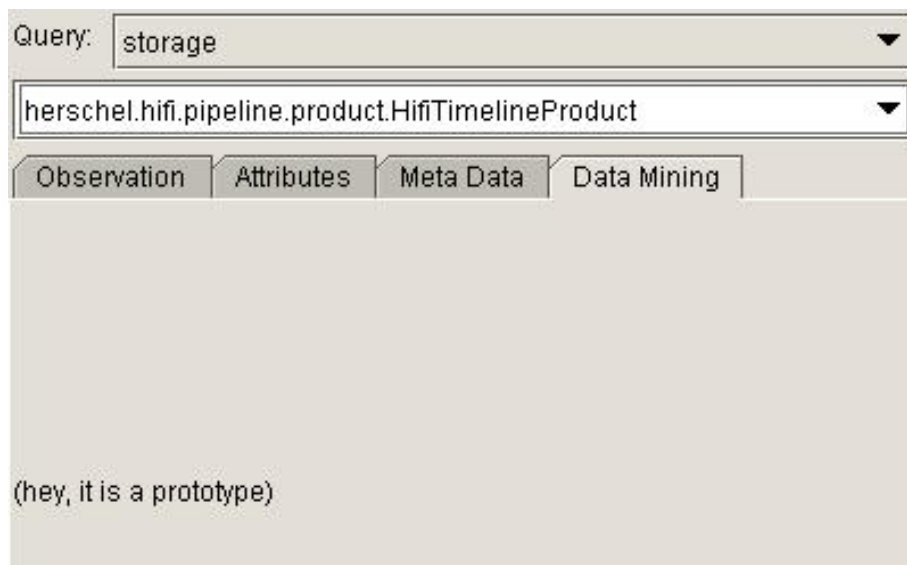


**Figure 1.15. Search via data mining.**

# Output from a query and searching a query result
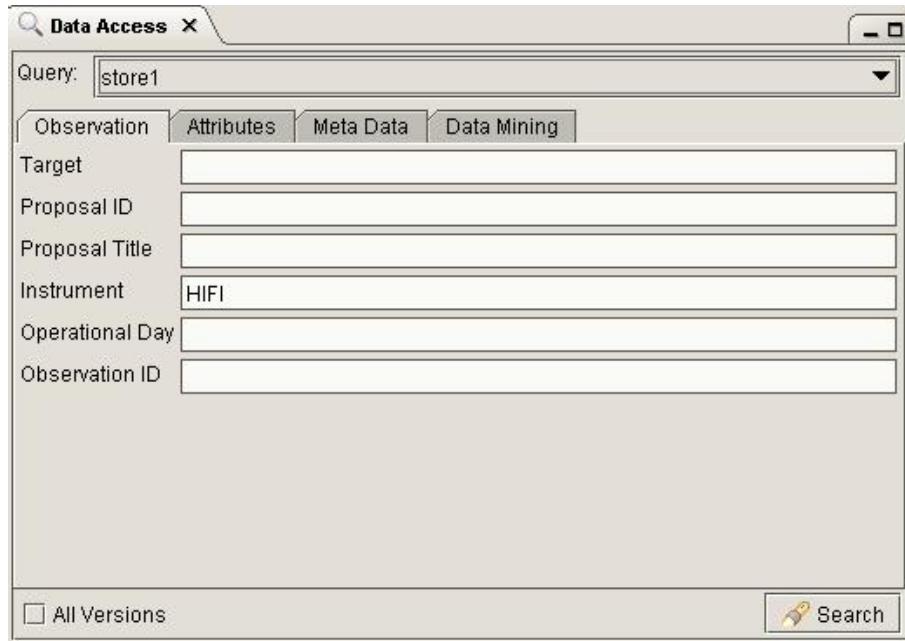
The output from the first query produces a result "QUERY_RESULT". This will be a group of products (e.g., observations) which can then be looked at by the user. The "QUERY_RESULT" name is highlighted in the Variables view (where the name can also be edited to something more appropriate if desired). This 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.

## 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.



**Figure 1.16. Set up of a query for data out of our store.**
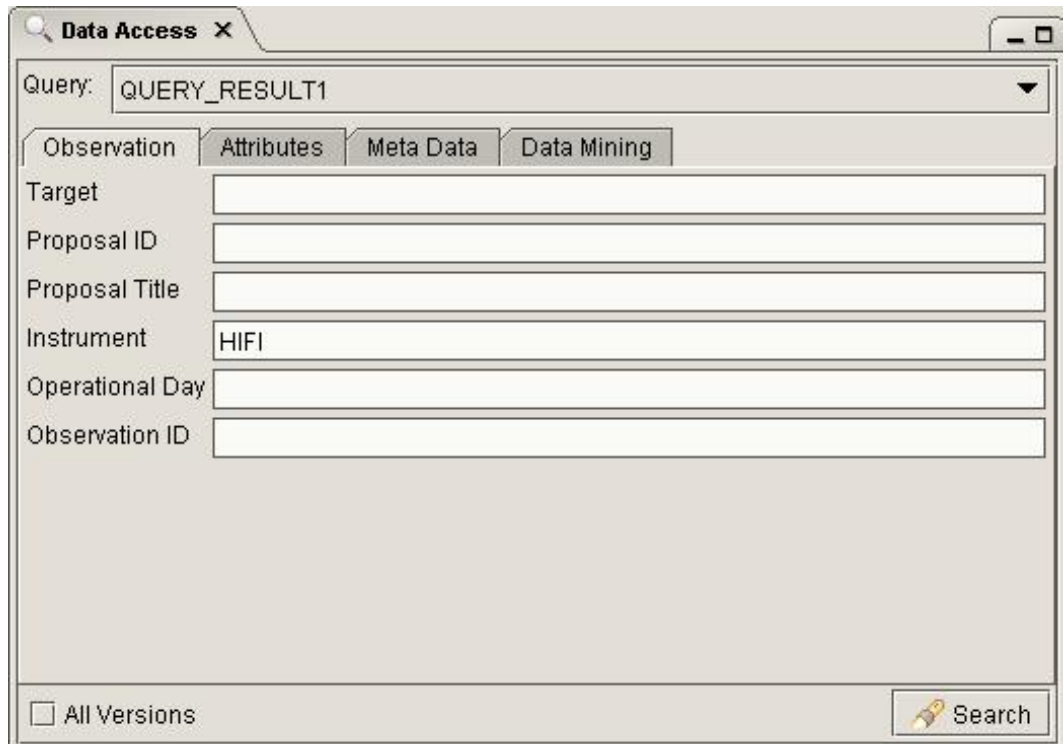
**Figure 1.17. Query result obtained.**



**Figure 1.18. List of query results appear in editor window.**
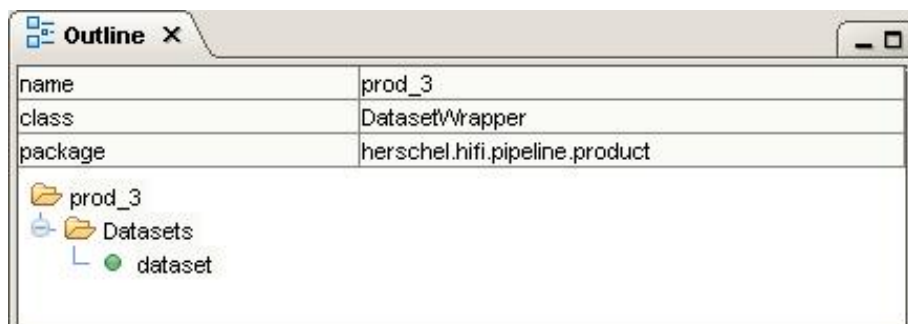


**Figure 1.19. One of the items is selected with outline of contents shown bottom left.**

**Figure 1.20. Metadata (header) display for the extracted spectrum.**
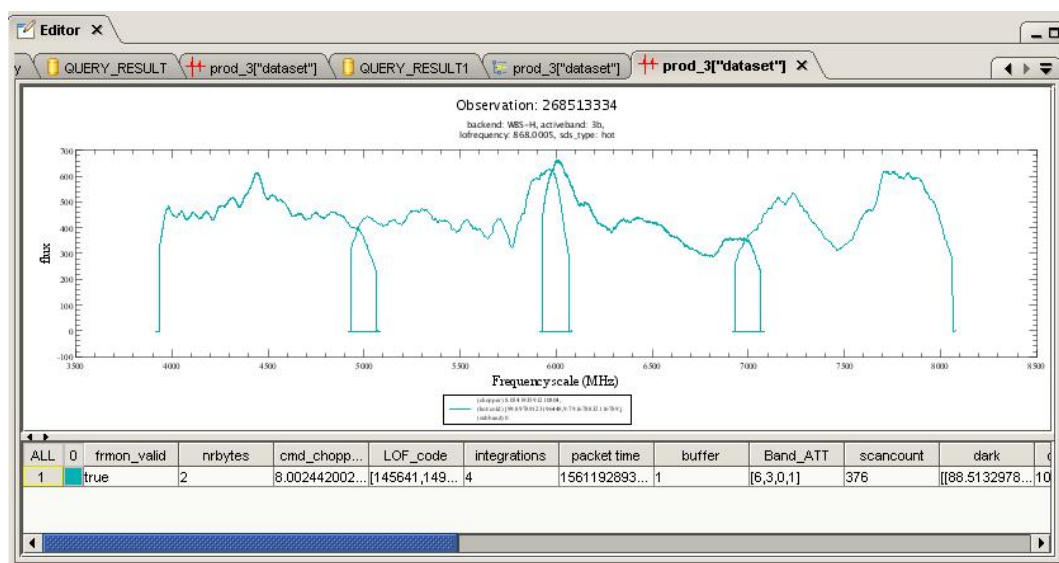


**Figure 1.21. Displaying the extracted spectrum. Note that the view has been expanded using the capabilities of the "Spectrum Explorer" viewer.**

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 1.16.

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 1.17). 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.

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 1.18).

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 1.19).

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 1.19).

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 1.20).

7. Alternately, we can simply view the extracted spectrum dataset by selecting the "Spectrum Explorer" viewer instead (see Figure 1.21 and HowTo on displaying spectra).

# 1.2.6. Data access via the Console View command line

Within the Console View it is also possible to access data directly from the command line. The commands for doing this are actually generated in the Console view when using the dialog interactions noted above.

In the following we show how the information can be extracted into "newVariable1" (as per above example) which is then ready for display, fitting etc.

1. First we do a query on attributes in our data store, which was labeled "storage", looking for HifiTimelineProducts and instrument=HIFI. The following should all be on one line and the easiest way to get it is when it is copied to the Console view when using the Data Access view as noted above.

```
QUERY_RESULT1 = StorageResult( store1,
 \
herschel.ia.pal.query.MetaQuery(herschel.hifi.pipeline.product.DatasetWrapper,
 \"p", -"(p.meta.containsKey(\"instrument\")
 \ and p.meta[\"instrument\"].value == \"HIFI\")") -)
```

We can always print to the screen the contents using

```
print QUERY_RESULT1
```

Queries on the observation or meta data windows are MetaQuery's rather than AttribQuery's.

2. Now we extract the third in the list of results found.

```
prod_3 = QUERY_RESULT1[2].product
```

Again, we can use the print command to see its contents. Which is actually several products (in this case 5).

3. Get the dataset out of the product.

```
data_3 = prod_3["dataset"]
```

The Outline view of data_3 will show it is of the form WbsSpectrumDataset and that it is a HIFI pipeline product.

A full explanation of how to handle displays and manipulations (arithmetic and fitting) of spectrum (and image) datasets are covered in other HowTos.

When accessing Query results from the command line, it is harder to know which U RN in the list of references is the one we require. To aid us choose the correct one, we can sort the Query result by metadata. For example, to sort by creation time and put the reference to the oldest product first:

```
MetaComparator.sort(QUERY_RESULT1, ["creationDate"])
```

# 1.2.7. How to save and restore products using a Local Store

**Note**

Advanced information on this topic is available in the *Scripting and Data Mining* guide: Appendix A.

The easiest way to save and restore products in a HIPE session is to place them into a "local store". The "local store" is a simple folder structure which contains pools of products in the form of FITS files with corresponding metadata. Pool folders usually reside in the ".hcss/lstore" directory under the user's own home directory.

The "PAL storage manager" view in HIPE is shown in Figure 1.22. This allows the user to easily create pools, open a product storage and register a pool in it. More than one pool (directory of data) can be placed in a storage, if wanted, making many products from different observations or instruments available from the same storage area. The ID given to the storage is the one needed to directly access the products from that storage.
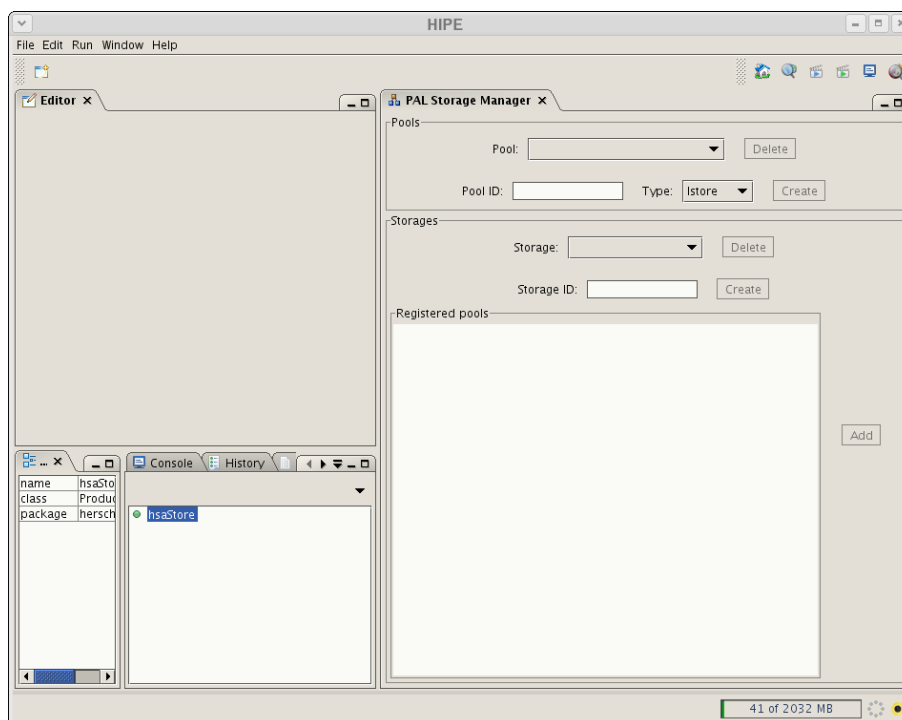


**Figure 1.22. View for creating a pool for storing products**

On the top panel are shown the list of user's pools under ".hcss/lstore" directory. New pools can be created fill-in Pool ID field and clicking the "Create" button. The action performed by the button is echoed in the HIPE Console (see Figure 1.23).
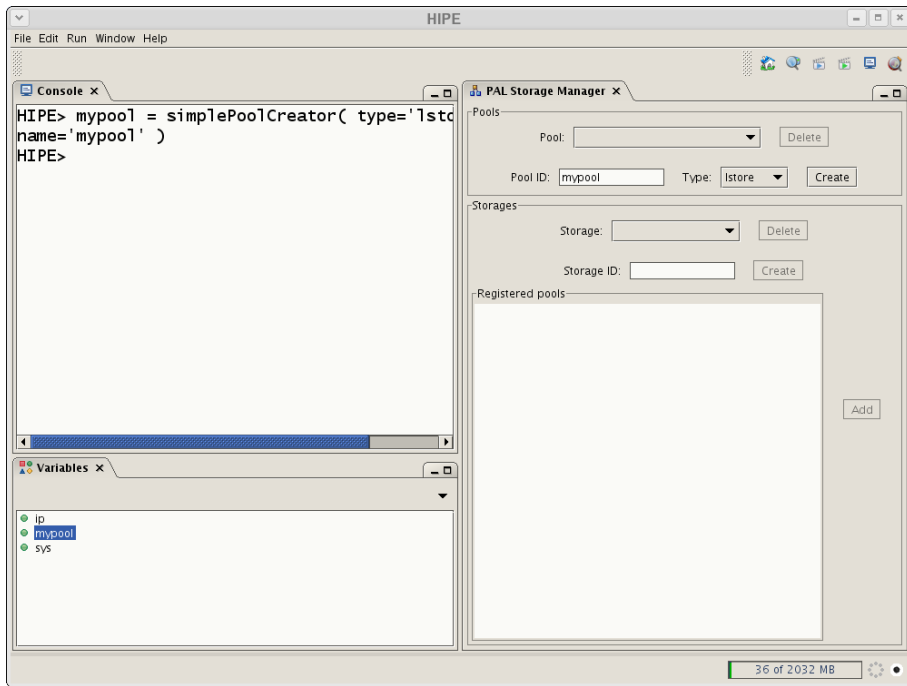
**Figure 1.23. Listing available pools**

Under panel Storages appear a number of pre-defined storages. New ones can be created (Figure 1.24). Moreover, pools can be registered on the storage by selecting the Storage and pressing "Add": A pop-up window will appear with the list of the available pools (Figure 1.25).
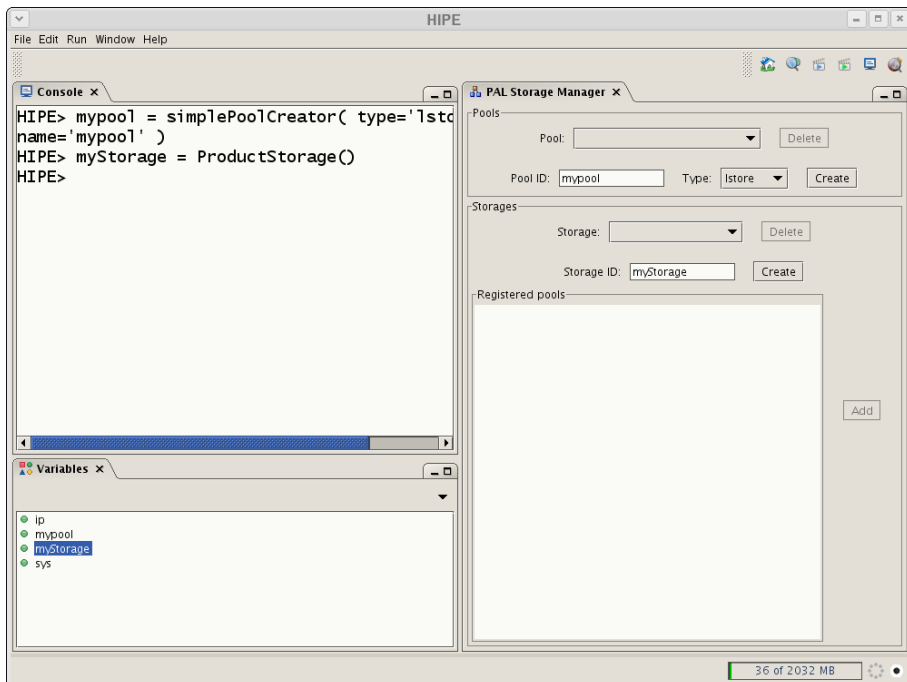


**Figure 1.24. Adding new storages**
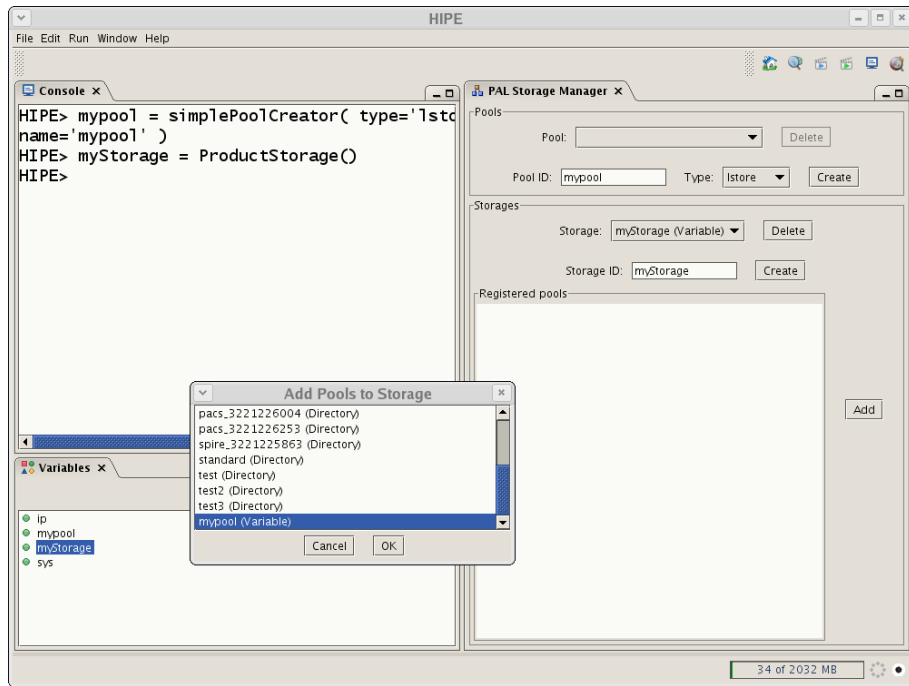
**Figure 1.25. Adding already existing storages to the session**

**Saving data**

In order to save a product into the local store we need to open a product storage, register a pool in it and then save the product into the pool. The first three items are discussed int he previous section. The following example also illustrates how this can be done on the command-line. The last line actually saves the data in the store.

```
store = ProductStorage()
myPool = LocalStoreFactory.getStore("myTestPool")
store.register(myPool)
store.save(myImage)
```

Now the product "myImage" is saved in the local store in the directory on local disk called ${HOME}/.hcss/lstore/myTestPool.
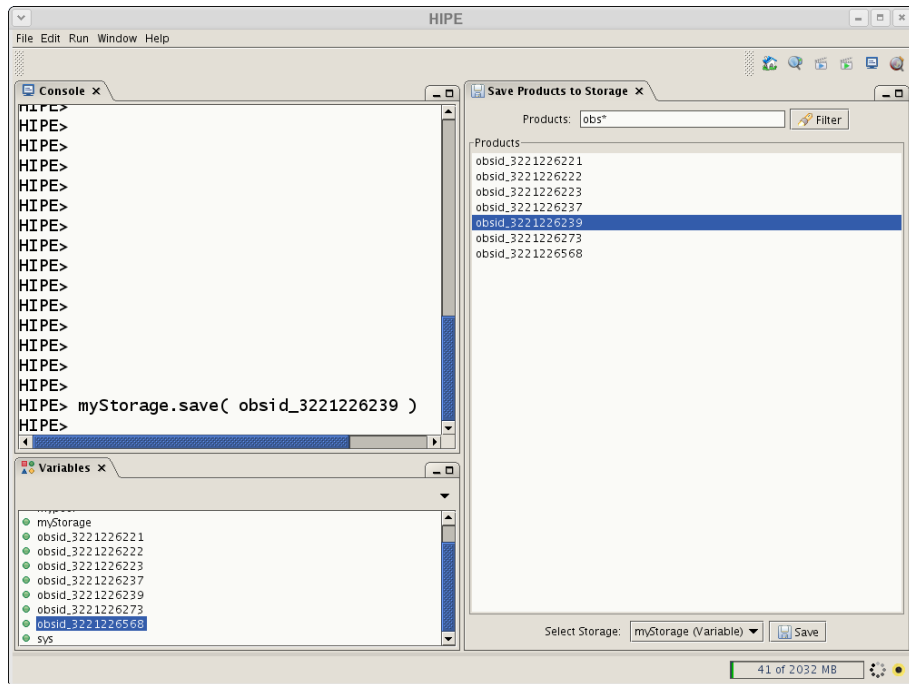
**Figure 1.26. Save products to Storage view**

Saving in HIPE can also simply be done using the "Save Products to Storage" view (see Figure 1.26), which allows to easily save products into a pool by simply selecting those in your DP seesion and the storage to use. Saving them in the storage occurs with the "Save" button at the bottom of the view.

Note that you can only save products, which means that if you want to save a Dataset of any kind - `TableDataset`, `Spectrum1d` or `2d` Dataset etc. 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()))
```

Next we need to put in a product:

```
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 the productBrowser**

Restoring the data back into HCSS is more complicated as it is necessary to know the product URN in the local store in order to retrieve it. However, a simple way to do this is using the productBrowser(). We can also use the Data Access view (*see also HowTo on accessing data* ):

```
store = ProductStorage()
myPool = LocalStoreFactory.getStore("myTestPool")
store.register(myPool)
result = browseProduct(store)
```

You can then query the available products in "myPool", select the one you need, add it to the basket and then exit the `productBrowser`. The steps are shown in the following screenshot:
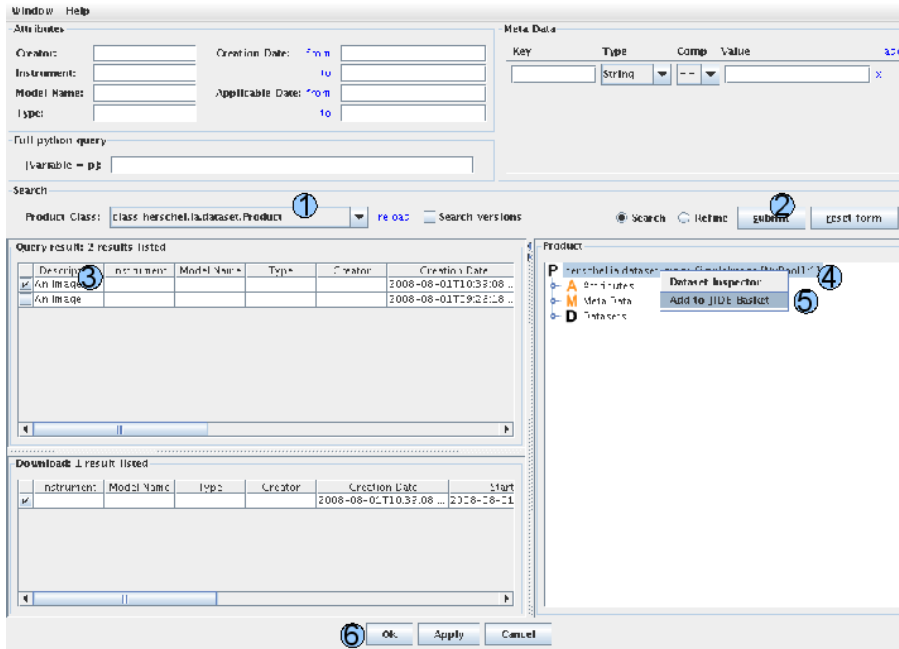
**Figure 1.27. Restoring product from the local store using the `productBrowser`**

These are the steps explained:

1. Select the product class to be of type Product

2. Click on "Submit" button to execute the search

3. Results for the products in myPool of type Product are shown in "Query result" view. Select the one you want.

4. The selected product structure appears in the "Product" view: Attributes, Metadata and Datasets are shown for this particular product.

5. Click with the right-hand mouse button on the product line (with the large "P" in front) opens a menu with "Dataset inspector" and "Add to JIDE Basket". Select the second item. The selected product will appear in the "Downloads" view.

6. Click "OK" to close the productBrowser().

At the end the reference to the product will be stored in the result variable and you can restore the SimpleImage following this example:

```
print result
# [urn:MyPool1:herschel.ia.dataset.image.SimpleImage:0]

image = result[0].product
```

If there is more than one result then we can refer to it with an index ([0] in the previous example).

The same way we can retrieve products which contain datasets (`TableDataset` or `ArrayDataset`) instead of `SimpleImage`.

**Restoring data using command line queries**

We 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.

# 1.2.8. Saving images and tables as FITS files

It is possible to save and read using command-line input or task dialogs in HIPE. For all task dialogs it should be noted that the dialog appears in an Editor view window. To run the task via the dialog always hit the "Accept" button to bottom right in the dialog box.

## 1.2.8.1. Saving with a Task Dialog

The simplest way to save data is using the simpleFitsWriter task. It is required that the data (image/table/set of spectra) are wrapped up as a product. An example product is an observation itself. But we can wrap any dataset into a product so that the appropriate metadata (header information) and history is available. Clicking on the name of any product will show this task available in the Tasks under the Applicable Tasks folder. A double-click on the task (shown in green) brings up the simple dialog shown in Figure 1.28.
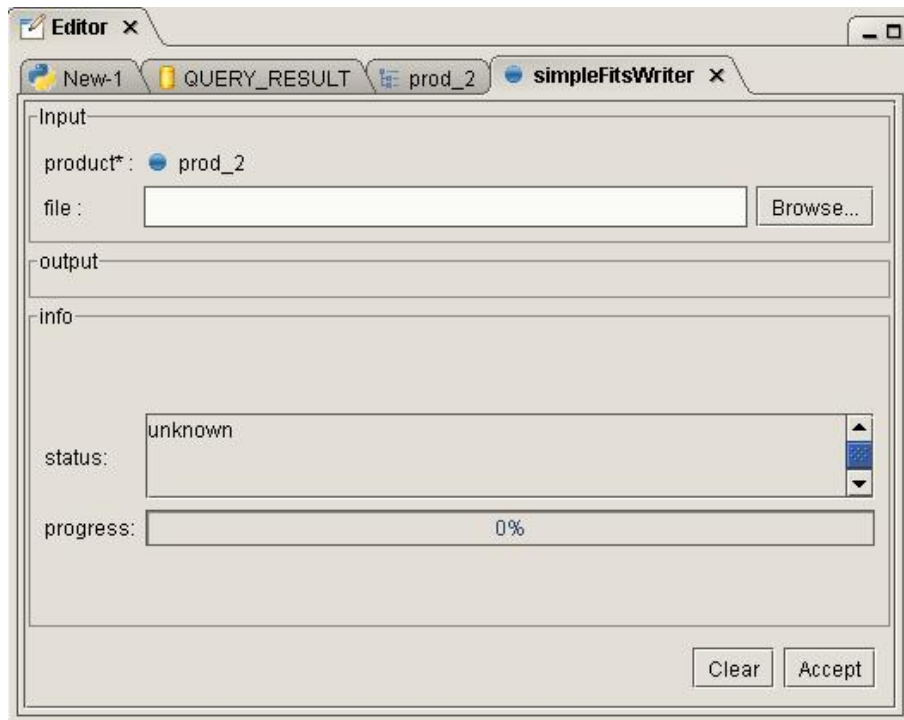


**Figure 1.28. FITS save task dialog.**

The only option the user needs to fill in is the name of the output FITS file. The default directory is the one that hipe was started from, so the full path name is usually required. Hitting the "Accept" button runs the task.

## 1.2.8.2. Saving Using Command-line Inputs

The FITS reader and writer in HCSS is in the FitsArchive package. First, let's store the product "myImage" from our previous example in a FITS file.

```
fits=FitsArchive()
fits.save("testFits-file1.fits",myImage)
```

The file "testFits-file1.fits" will be saved in the folder from where you started up HIPE. Otherwise, the full directory path should be supplied. It is a multi-extension FITS file with all the content of the SimpleImage product. Here is the structure of the saved FITS file:

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

  0  PRIMARY                 32     0
  1  IMAGE     image        --64     100 100
  2  IMAGE     flag          16     100 100
  3  IMAGE     error        --64     100 100
  4  IMAGE     exposure     --64     100 100
```

## 1.2.8.3. How to Save TableDatasets as FITS Files

Once we have the TableDataset wrapped in a Product we can save it like all other products. We can use the same FITS writing task from HIPE as noted above, or we can use a command-line method. For example:

```
fits=FitsArchive()
myTable = TableDataset() # create an empty table
myTable["X values"] = Column(Double1d([2,3.4,4])) # create fake column
myTable["Y values"] = Column(Double1d([2,4.5,4.8])) # create 2nd column
tProduct = Product(description="This is a table") # create the product
tProduct["firstTable"] = myTable  # add in the table and give it a label
fits.save("testFits-file2.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
```

We can see that the column names, which we named as "X values" and "Y values" are in the file.

## 1.2.8.4. How to Read FITS Files

You can use two tasks to read FITS files: fitsReader and simpleFitsReader. The fitsReader task (see Figure 1.29) 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. This in turns determines what kind of other tasks are applicable to the data. If fitsReader does not recognize the file contents, it defaults to the simpleFitsReader task. This task is optimized 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.
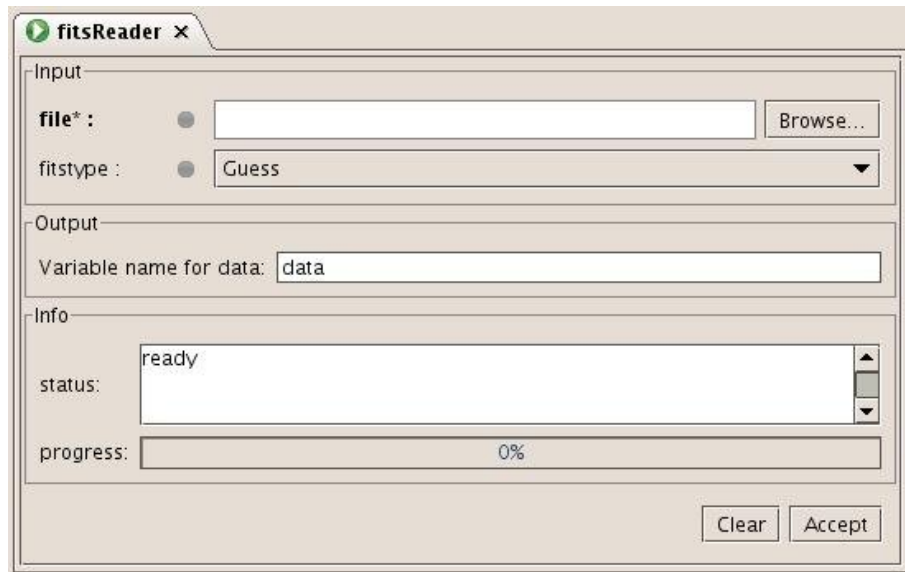
**Figure 1.29. FITS read task dialog.**

# 1.2.9. Creating and reading ASCII table files

In this case it is not necessary to put the TableDataset in a Product and we can directly save the Dataset to an ASCII file. As for FITS writing, we can do this from within a HIPE task dialog or from the command-line.

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 brings up a dialog 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 1.30).
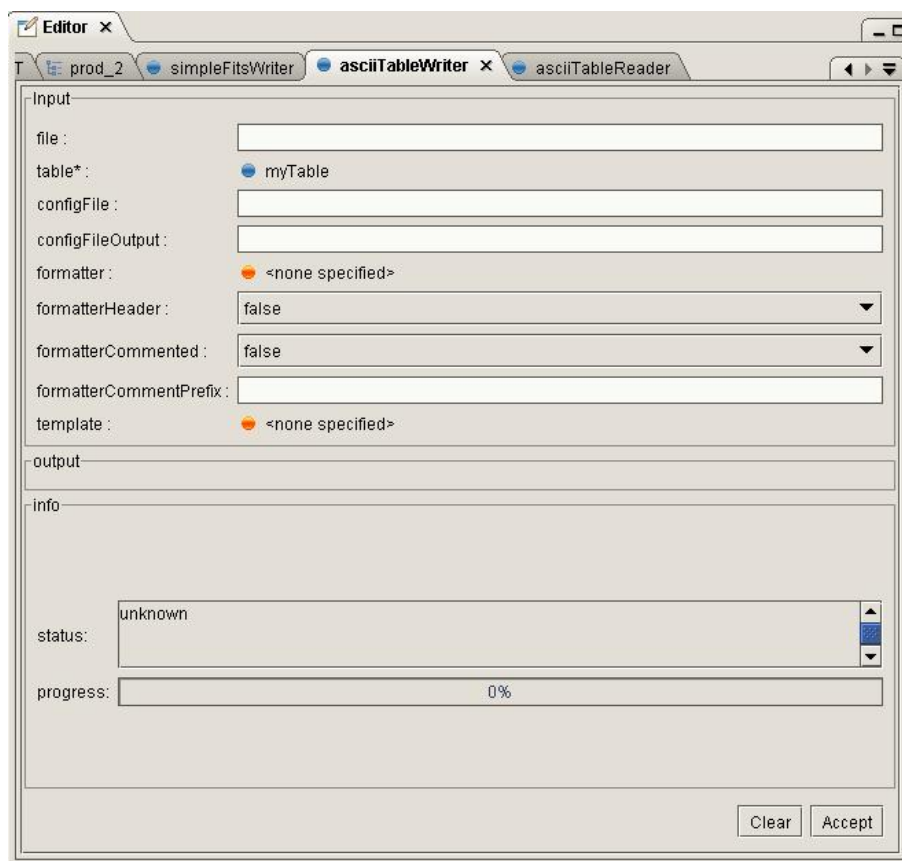
**Figure 1.30. FITS save task dialog.**

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 dialog).

```
* 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 Advanced
User's Manual for more details).
```

Clicking on Accept at the bottom of the task dialog window runs the task and creates an ASCII table.

Reading an ASCII table into HIPE can be done using the `asciiTableReader`. Go to the "Tasks" view and open the folder "All". Double-click on the word `asciiTableReader`. This provides a dialog. For standard CVS tables the only thing that needs to be filled in is the file name of the ASCII table to be read in. More

```
* 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.
```
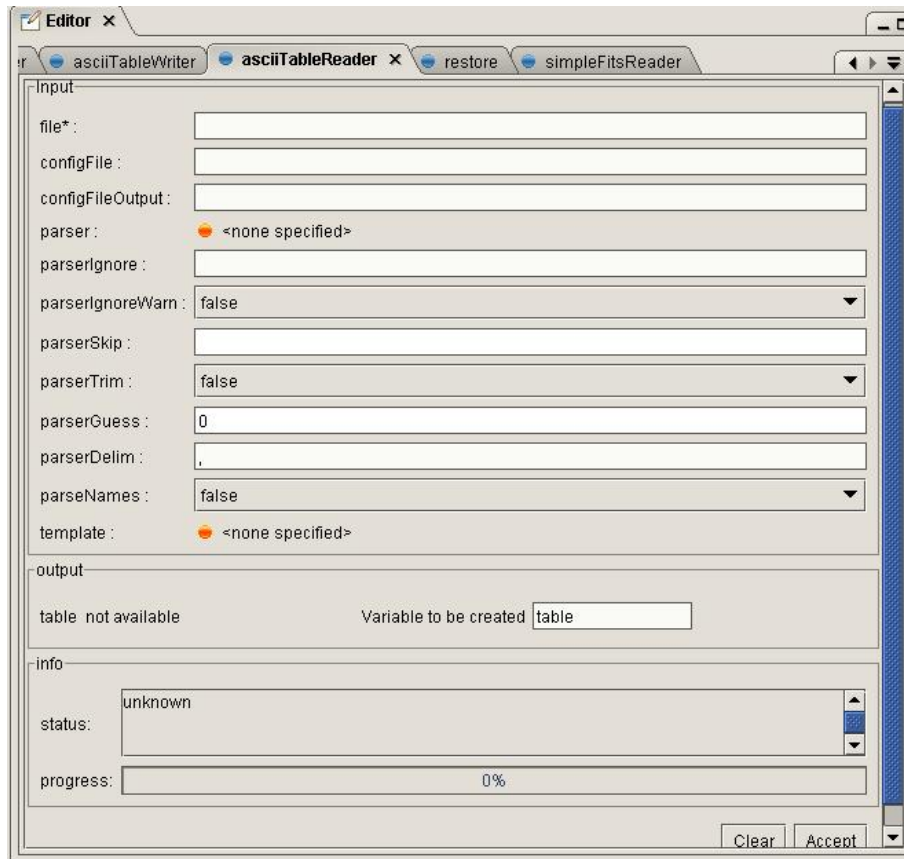
**Figure 1.31. ASCII read task dialog.**

Information on saving and reading tables from the command line is available in <u>Section 1.3.2</u>.

# 1.3. In depth

## 1.3.1. 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 1.1. Using save and restore**

## 1.3.2. ASCII table import/export

For the following examples to work you need the herschel.ia.io.ascii and herschel.ia.dataset packages, which should be imported automatically into your HIPE session.

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).

## 1.3.2.1. 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*")
```

## 1.3.2.2. 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 1.3.2.1](#)) 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 1.3.2.1](#)) 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])
  ```

## 1.3.2.3. 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"])
```

# 1.3.2.4. Examples of how to import/export ASCII tables

Section 1.3.2 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 1.2. ASCII demo data**

Finally, we also present an example of the use of the RegexParser for importing tables.

```
from herschel.ia.io.ascii import *
```

```
#instantiate the table tool
ascii = AsciiTableTool()
# regular expression looks for vertical slash between spaces
ascii.parser=RegexParser(delimiter="\s*\|\s*")
#6 columns will be read
ascii.template = TableTemplate(6)
# now load it
cat = ascii.load("test_ascii_space.dat")
#get the number of data elements in the first column
n = len(cat["Column0"].data)

#Now print out the columns we have read into -"cat"
for i in range(n):
    print cat["Column0"].data[i],cat["Column1"].data[i],\
        cat["Column2"].data[i],cat["Column3"].data[i],\
        cat["Column4"].data[i],cat["Column5"].data[i]

############

The data file for the above script is the following which should
be called -"test_ascii_space.dat":
########
1 -| 2     -| 3 -| 4
2 -| 3 -|    4 -| 5
3 -| 4 -| 5   -| 6
4 -| 5 -| 6 -|   7 -|

| 5 -| 6 -| 7 -| 8   -|
6    -| 7 -| 8 -| 9 -|
a    -| b -| 8 -| 9 -|
#########

The result from above script should look like this:
#######
1 2 3 4 None None
2 3 4 5 None None
3 4 5 6 None None
4 5 6 7 None None
None None None None None None
None 5 6 7 8 None
6 7 8 9 None None
a b 8 9 None None
#####
```

# 1.3.3. 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 1.3. 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` in FITS format:

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

**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.

# 1.3.4. Parameter name conversion and FITS header

The current implementation of the FITS archive converts long, mixed-case parameter name, defined in the meta data of your product, into a FITS compliant notation. The latter dictates that parameter names must be uppercase, with a maximum length of eight characters. Clearly, we do not want to force all our parameters to have names that fit within such a FITS specific restriction.

The FITS Archive uses lookup dictionaries that convert well known FITS parameter names into a convenient and human readable name. 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 of FITS IO is available in the example below. The script creates a product with several (nested) datasets, stores it into a FITS file, and then retrieves it again.

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

# ---- construction of a product. note this is only for demonstration
# purposes. For more infomation, please see the dataset and numeric
# chapters of the manual (chapters 5 and 6).
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
```

**Example 1.4. FITS IO example**

```
#and look at a specific piece of metadata
print n.meta["startDate"]
```

# 1.3.5. Caveats

For more information see the [FITS IO](#) general documentation.

## 1.3.5.1. 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.

## 1.3.5.2. 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).
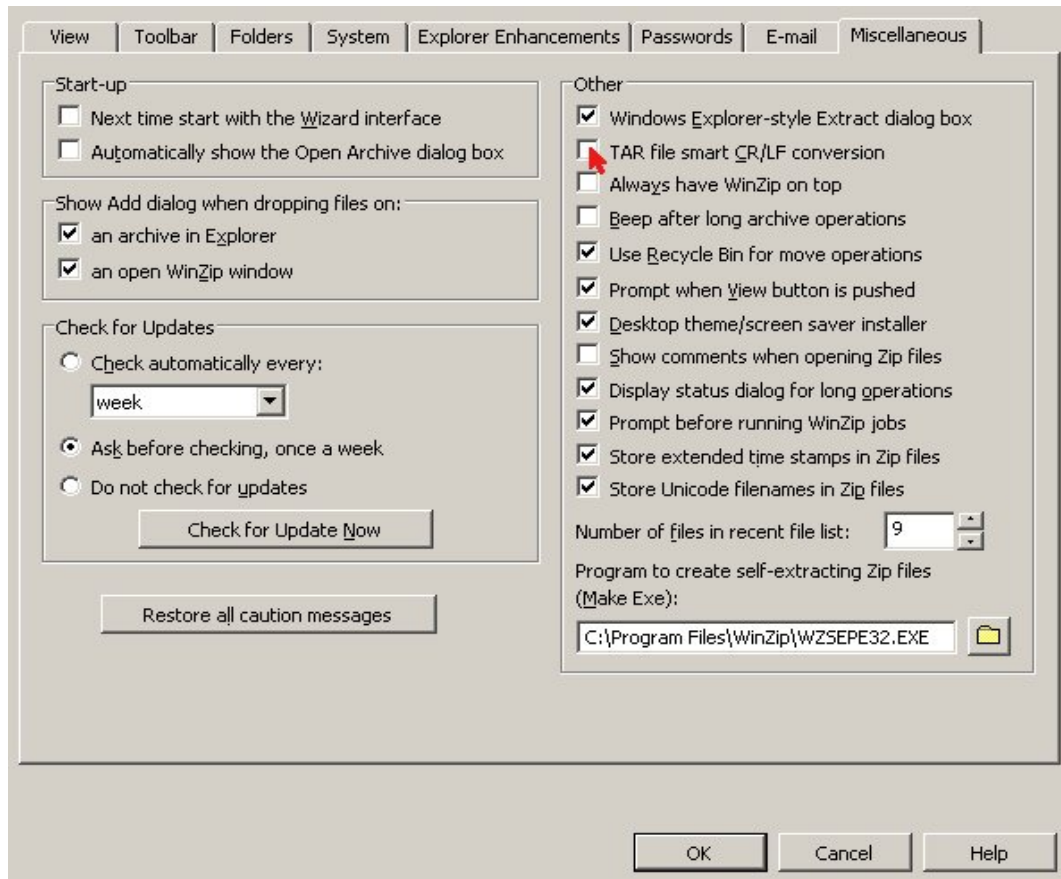
**Figure 1.32. The Configuration dialogue window in WinZip.**

**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.

# 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.

> **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. 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 2.3.1.1.

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 .

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

# 2.2.2. 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.
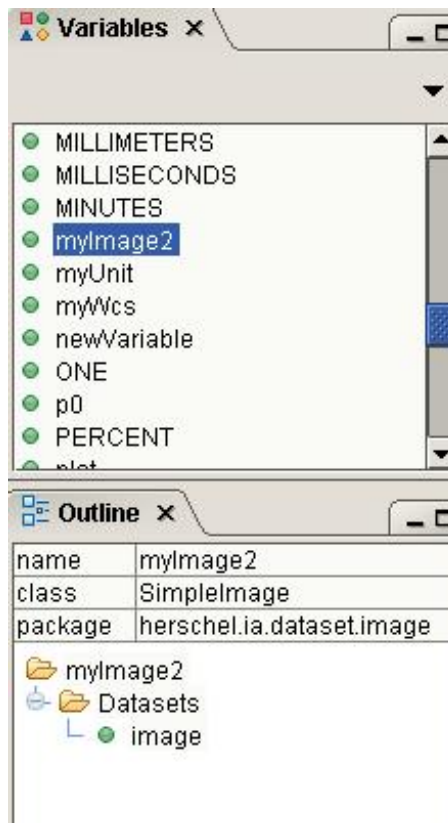


**Figure 2.1. The Variables view shows the "myImage2" highlighted. A double click on this automatically brings up the image in a new Editor window (top left). In the Tasks view the folder "Applicable", when opened, shows the tasks that can be applied to this image.Variables view with `SimpleImage` variable 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.

# 2.2.3. Viewing the Metadata and Array Data Associated with an Image Dataset

An image can have several datasets. For example, we can include a flag image dataset for flagging bad pixels (see the *Advanced User's Manual* 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 short menu with "Open With...." and the dataset viewer selectable.

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 2.2).
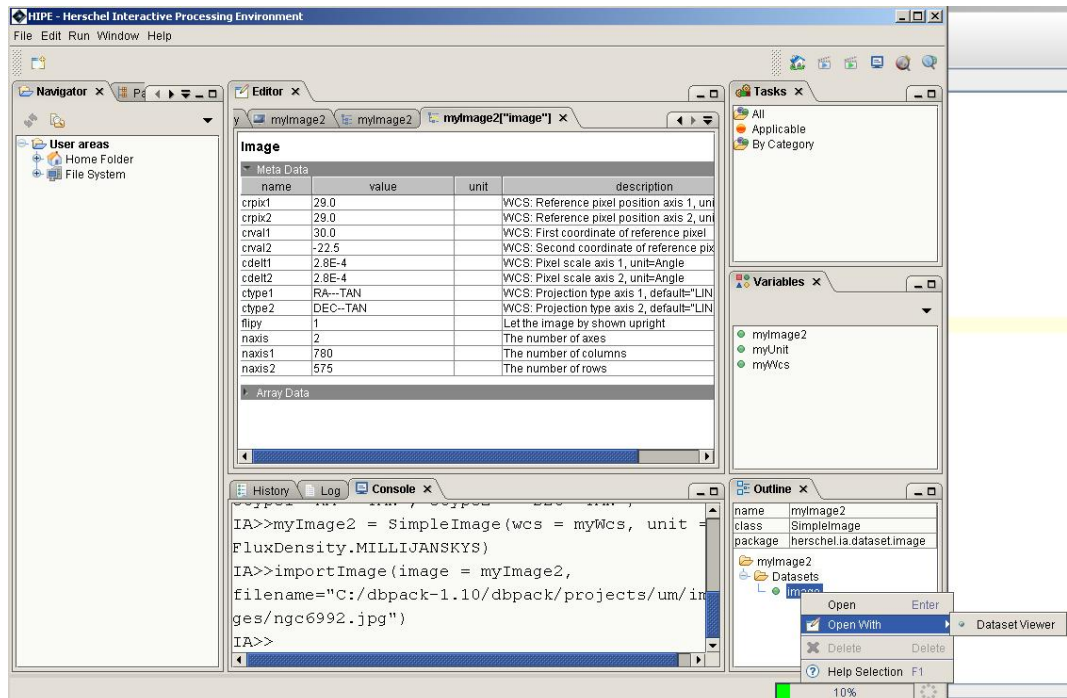


**Figure 2.2. Metadata and Array data view using the Dataset viewer with an image.**

# 2.2.4. Viewing an image

The simplest way to display an image in HIPE is to double-click the image name (for instance in the *Variables* list). A display will appear in the *Editor* view (see Figure 2.3).
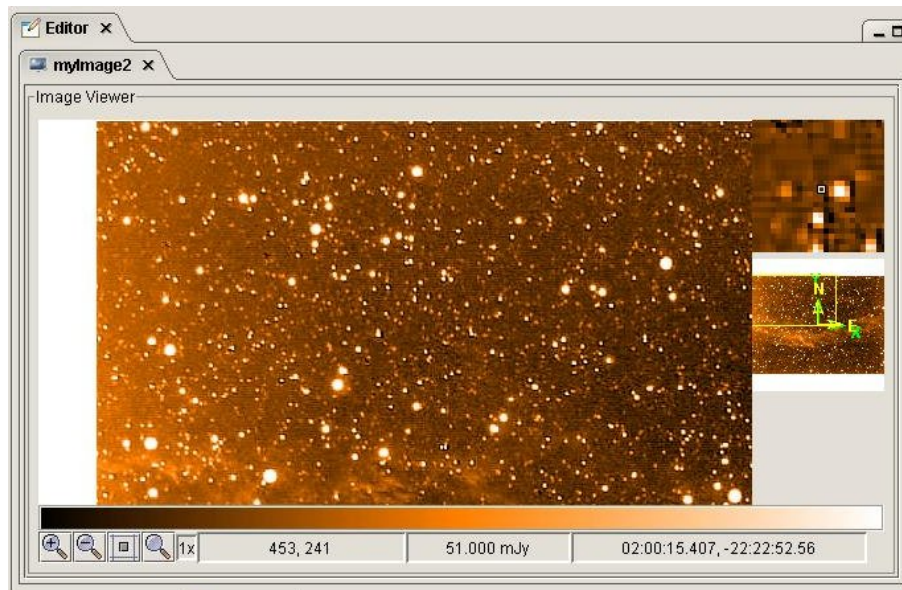
**Figure 2.3. Viewing an image in HIPE.**

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. You can change the position of the zoom/pan region by dragging it.

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 four boxes right of the icon show the following information (left to right):

- Zoom level

- Pixel coordinates at mouse pointer

- Pixel intensity value at mouse pointer

- WCS coordinates (if defined) at mouse pointer

## 2.2.5. Editing and printing an image

Right-click on an image to activate a menu with editing functions (see Figure 2.4).
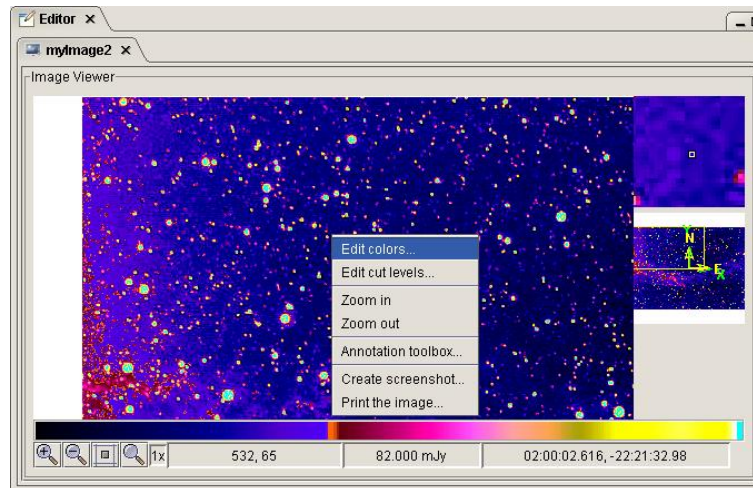
**Figure 2.4. Image editing functions.**

- Edit colours: change the colour lookup table to a number of different types plus linear/log/ exponential scalings.

- Edit cut levels: adjust the cut levels for which the colour lookup table is to be applied.

- Zoom in/out: same effect as the zoom icons.

- Annotation toolbox: add lines, simple drawings and text to the image.

- Create screenshot/Print the image: print or save to JPG, PNG or BMP format. You can also decide whether to keep all overlays and annotations.

## 2.2.5.1. Editing the Colour Look Up Table (LUT)

The standard colour scheme for image display is for "Real" colours shown in a "Ramp" intensity with a "linearScale". Selection of the "Edit colors..." from Figure 2.4 displays the colour menu Figure 2.5. Hitting the "Reset" button always enables the default colour display.

To select any other colour scheme simply click on the colour type and/or intensity or algorithm to create a new colour scheme. The scheme is applied to the image immediately. The window (Figure 2.5) can be dragged away from the image.
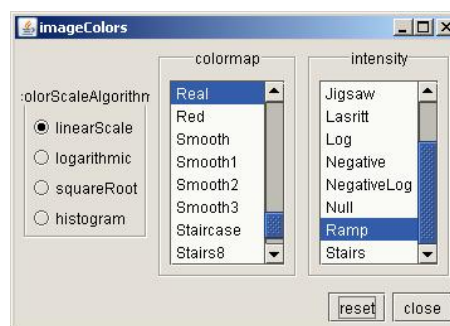


**Figure 2.5. Colour table selection menu. Hitting "Reset" takes you back to the original colour table.**

## 2.2.5.2. Editing the Cut Levels

The default cut levels for images is 99.5 per cent of pixel values. Selection of the "Edit cut levels..." from Figure 2.4 displays the a cut level selection including a histogram of the current pixel intensity values Figure 2.6. Hitting the "Reset" button always enables the default cut levels.

To select any other cut level the user can do one of two things.

- A button selection of cut levels (90, 95, 98, 99, 99.5 or 100 per cent). Note that selection of any of these will adjust the histogram display above.

- Adjustment of the upper and/or lower-level cutoffs of the histogram by click-and-drag of the yellow arrows (left or right) shown at either end of the histogram view.
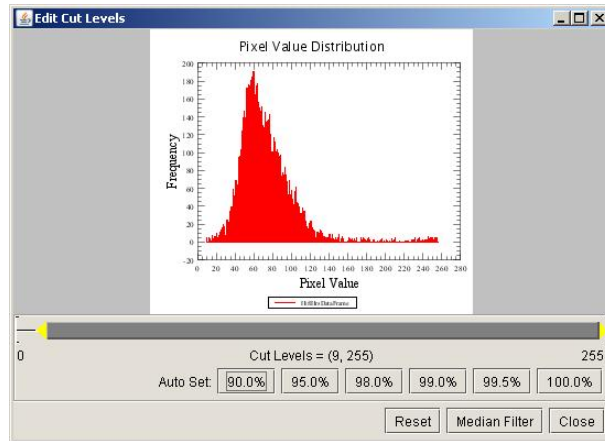


**Figure 2.6. Cut level selection window. Hitting "Reset" takes you back to the original cut levels of 99.5 per cent.**

## 2.2.5.3. The annotation toolbox

The annotation toolbox is shown in Figure 2.7.
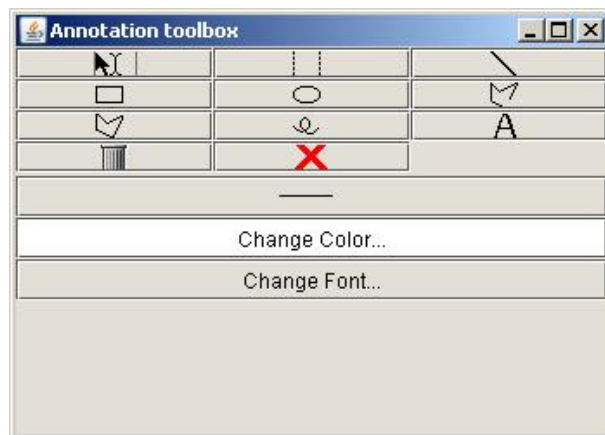


**Figure 2.7. The annotation toolbox.**

The icons in the annotation toolbox appearing in Figure 2.7 have the following usage (from left to right and from top to bottom):

- Select annotation

- Select all annotations in a region

- Draw a line

- Draw a rectangle

- Draw an ellipse

- Draw a polyline

- Draw a polygon

- Draw with the free hand on the image

- Add a text annotation

- Remove the selected annotation(s)

- Remove all annotations

Letting the mouse linger over an icon also displays its function.

The polygon and polyline methods will enable you to select points on the image which should be used as a corner of the polygon using the mouse. Double-clicking the mouse will end the selection procedure.

The three buttons below the ones already described change the view of the annotation. 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.

## 2.2.6. Obtaining a Spectrum from an `ObservationContext`

Most users will obtain spectra from downloading observations from the Herschel Science Archive (HSA). The main product form of an observation is referred to as an `ObservationContext`. An `ObservationContext` contains all the components of an observation, including all calibrations needed for repeating pipeline processing data. The full observation download from the HSA includes all levels of processed data from level 0 (raw data) to level 2 (final pipelined product).

An observation context can contain many spectral products (e.g., from all 4 spectrometers of HIFI) at each of the different levels of processing.

We can consider a HIFI `ObservationContext` called "prod" which has been downloaded from the HSA. A double-click on the variable name "prod" in the "Variables" view of HIPE provides an outline view of its contents in the "Outline" view (see Figure 2.8). This shows the containers of spectra at the different processed levels (also quality and calibration information associated with the observation). If we now open the level 2 folder and click on the product (highlighted in Figure 2.8), we get a view in a new "Editor" window like the one shown in Figure 2.9) -- after expanding out the folder labeled 1030.

At present, the HIFI spectrometers are identified by the values 1028 = HRS H polarization, 1029 = HRS V polarization, 1030 = WBS H polarization and 1031 = WBS V polarization. In this case the 1030 folder under level 2 is the final, processed data product from Standard Processing at the Herschel Science Centre of WBS H polarization data. The folder labeled "1" contains the first (and in this case only) final product. A double-click on product(load) -- highlighted in Figure 2.9) -- provides access to

a listing of the metadata and the final dataset (scroll to bottom of "Editor" window) which is marked with a green dot beside it.

We can display this final spectrum via a viewer called Spectrum Explorer. As with all viewers in HIPE -- click with right mouse button on the dataset word, choose "Open With..." from the menu that appears and then click on the words "Spectrum Explorer". This will display a new "Editor" window with a blank spectrum display (initially). To fill in the spectrum the user needs to click on the four boxes (label above shows 1, 2, 3 and 4) to the left under the plot or simply clicking on the box under "ALL". This fills in the 4 sub-bands of 1GHz wide CCDs that make up the full backend spectrum from a WBS spectrometer (see Figure 2.10). An appropriate legend is automatically created.
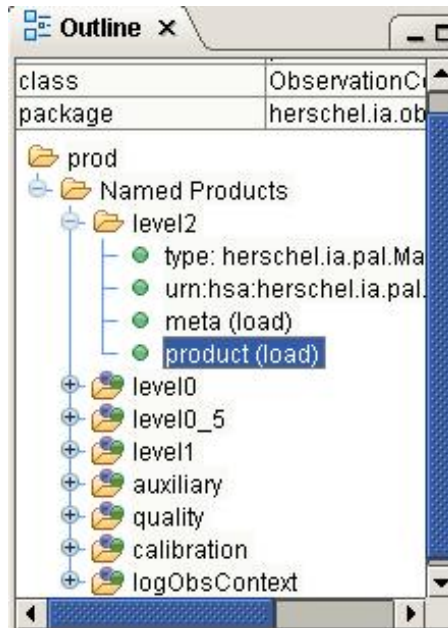


**Figure 2.8. Accessing the level 2 (final) processed product from an observation.**
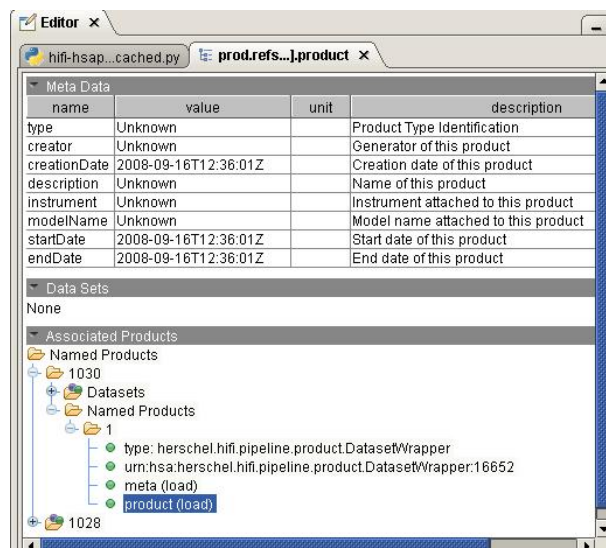


**Figure 2.9. Editor view showing access to the final WBS H polarization spectrum from within the full observation tree.**
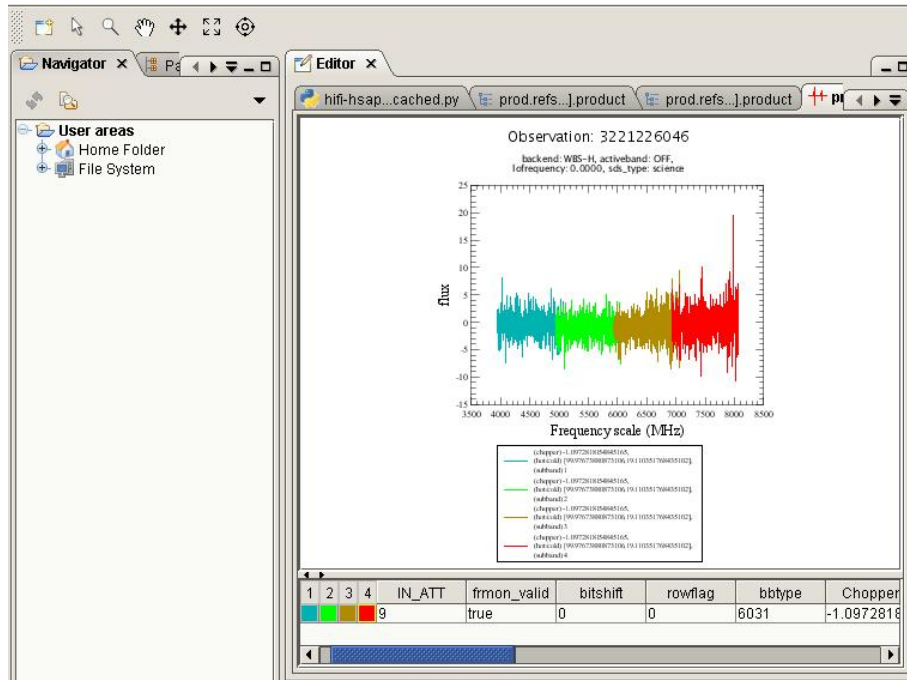
**Figure 2.10. Display of a test data (no source) produced by the HIFI pipeline using the `SpectrumExplorer`.**

An example of a SPIRE spectrum is provided in [Figure 2.13](#). Here we can input (say) a FITS file using the "simpleFitsReader" available in the Tasks view (see [Figure 2.11](#)). It is also possible to input a FITS file by simply double-clicking on it in the directory display of the Navigator view of HIPE. The variable created ("product1" in this case) can be seen in the Variables menu. Click on this and the Outline is shown and it can also be opened by double-clicking. This provides access to the metadata and datasets (see [Figure 2.12](#)). Click on the spectrum dataset (with the green dot next to it) and the SpectrumExplorer is started with a display of the spectrum (click the "1" symbol to bottom left). This will then show a spectrum similar to that shown in [Figure 2.13](#).
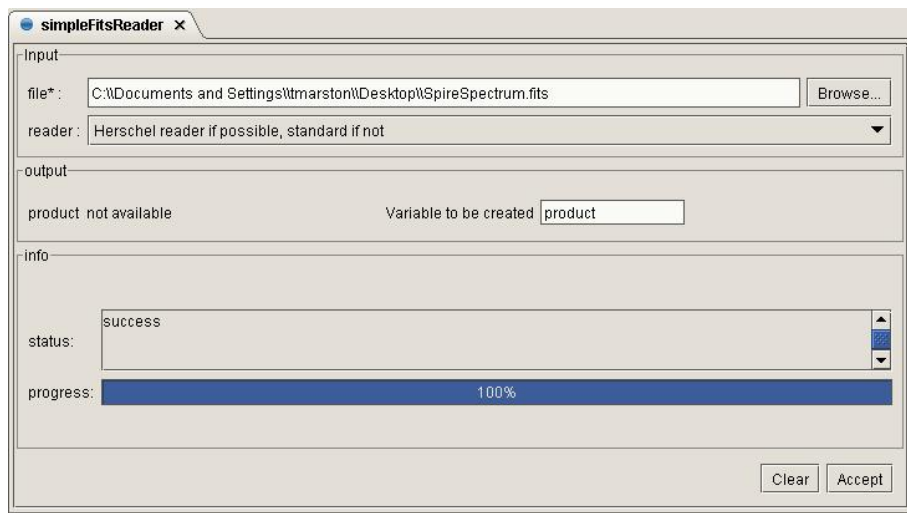


**Figure 2.11. Using the simpleFitsReader task for reading in a FITS file with a SPIRE spectrum.**
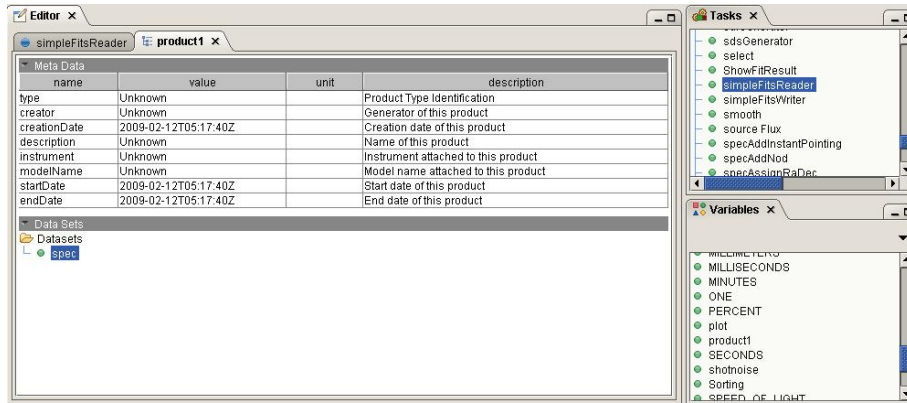
**Figure 2.12. Display of the metadata and datasets in the FITS file when the variable is double-clicked.**
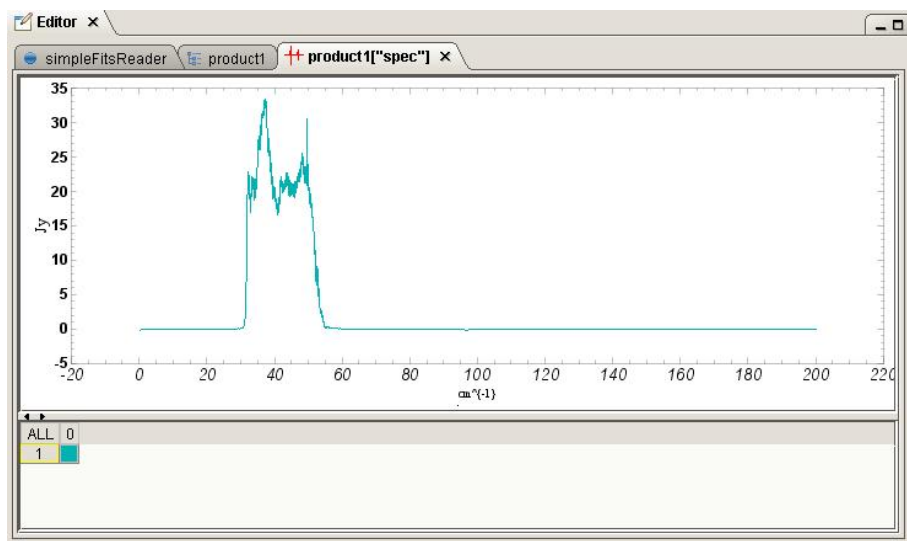


**Figure 2.13. Display of the SPIRE spectrum using the `SpectrumExplorer`.**

# 2.2.7. The `SpectrumExplorer` Package

The `SpectrumExplorer` package is based on the PlotXY package, but allows the user to visualize `SpectrumDatasets` in a friendlier, interactive way.

In the example from the previous section, the different colors indicate different WBS sub-bands. Individual sub-bands or individual scans can be plotted by clicking on the appropriate boxes in the bottom panel and removed by double-clicking. Any plot parameter (plot range, titles, colors etc.) can be modified using the right mouse bottom in the same way as for the PlotXY package (see "HowTo Plot Data"). For example, a right-click on the plotted spectrum allows changes in the axes and plot properties (e.g., labels fonts etc.). It is also possible to save and print the plot from the right-click menu.

In addition the plot can be modified interactively after clicking the appropriate action button which `Spectrum Explorer` places in the top left of the HIPE display (see top left of Figure 2.10). Hovering the mouse over the icons allows provides the user with a tooltip for what the icon allows you to do. From left to right:

- button 1: highlight/select a spectrum (or WBS sub-band) by moving the mouse over it and click the right mouse button to change its color, description, or remove it.

- button 2: change the horizontal and vertical plot ranges by drawing a rectangular box using the left mouse button. Also, one can scroll the spectrum along the horizontal and vertical axes by clicking

on an axis with the left mouse button and then moving the mouse or using the mouse wheel. The mouse wheel can also be used to (un)zoom the spectrum.

- button 3: pan through the spectrum by clicking the left mouse button and moving the mouse.

- button 4: click on a spectrum (or WBS sub-band) and drag it to right or down to another or a new panel which is automatically generated on release (however, dragging to the left or top of the first panel is not possible).

- button 5: click on this button to auto-range the displayed spectra (after zoom).

- button 6: only show the active plot panel, and change the axis ratio in order to fit the screen.

This allows images such as the Figure 2.14 to be constructed from the displayed data.
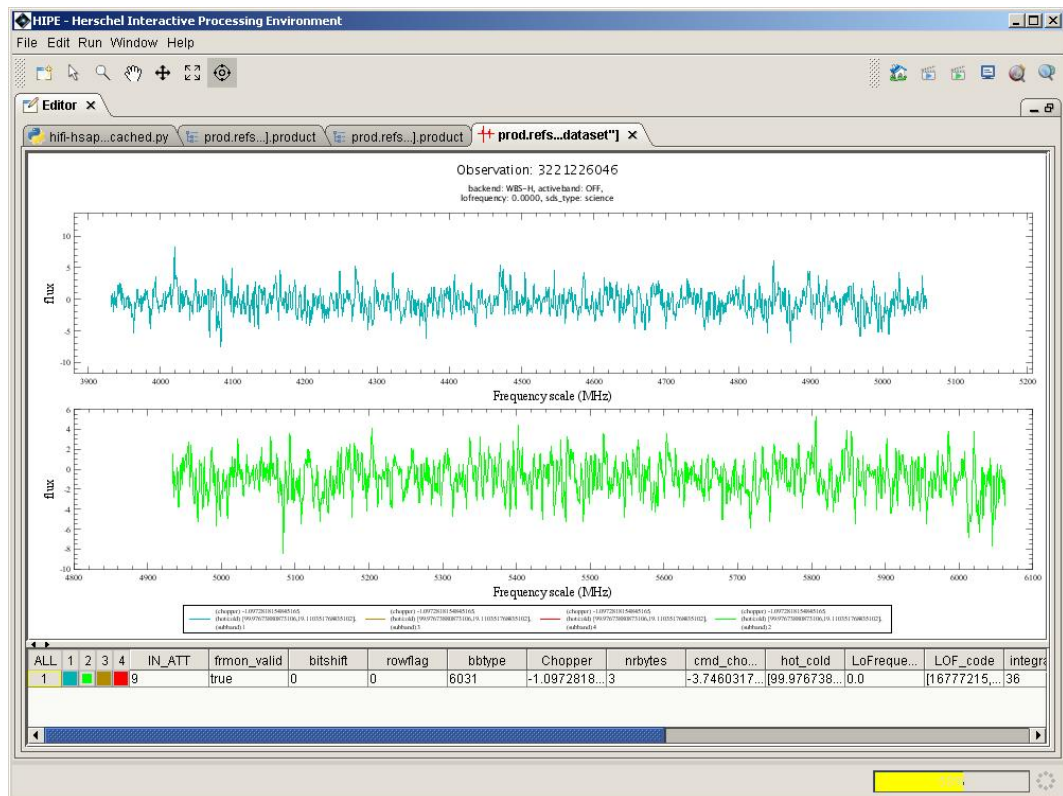


**Figure 2.14. Two sub-bands extracted interactively (button 4) using `SpectrumExplorer` shown with the full resolution of the screen (button 6).**

# 2.3. In depth

## 2.3.1. The Dataset Inspector

This section describes in detail a graphical tool to inspect table datasets, the *Dataset Inspector*.

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

[Figure 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.
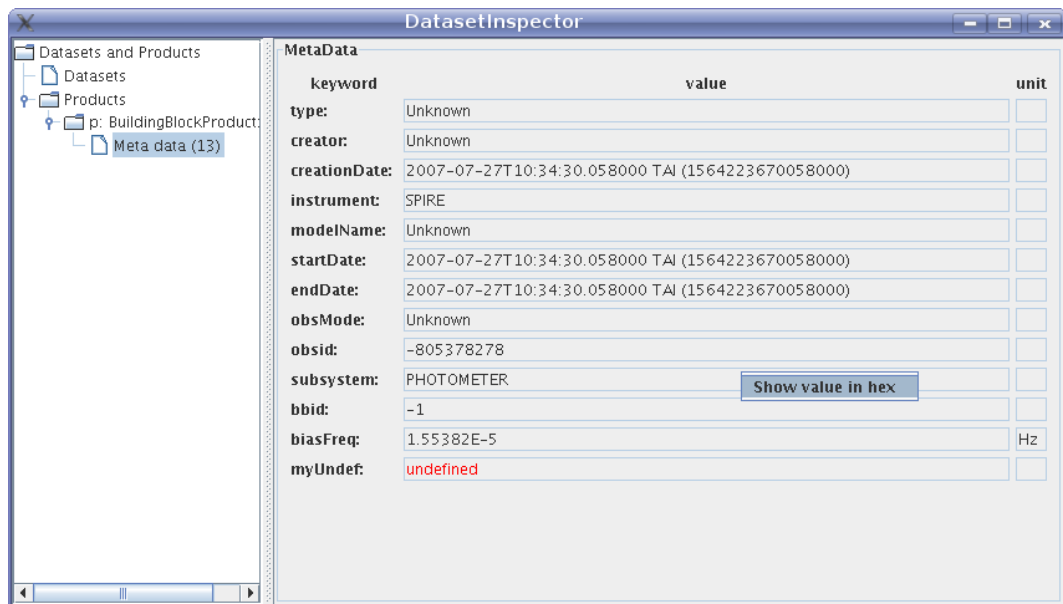


**Figure 2.15. The Dataset Inspector showing product metadata.**

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 we can switch between decimal and hexadecimal representations.

Dates and times are shown by default in UTC (Coordinated Universal Time), with their `FineTime` representation in brackets (for more information on how time is represented see the *Scripting and Data Mining* guide: [Chapter 6](#)). By right-clicking on the parameter values we can switch between UTC and TAI (International Atomic Time).

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

## 2.3.1.1. The TablePlotter

### Introduction

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

### Invoke TablePlotter

• **Invoke TablePlotter as a Viewer in HIPE**

The TablePlotter works with Table Datasets and products that contain TableDatasets. 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](#)).
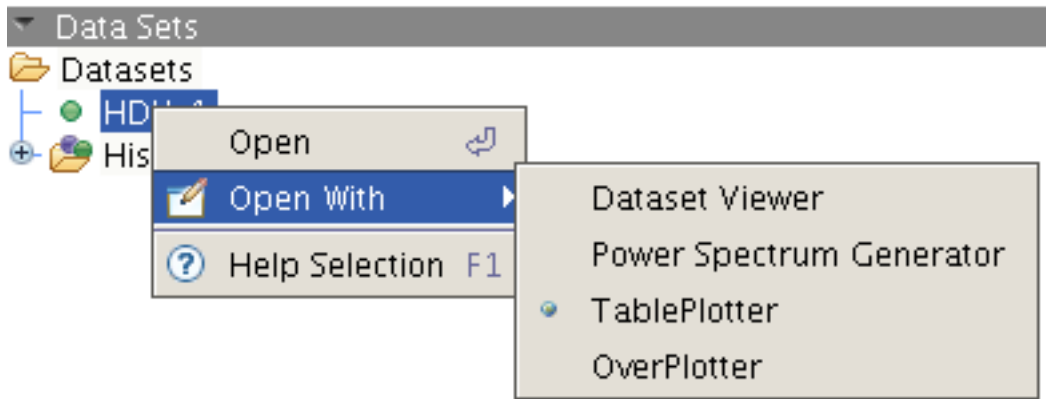
**Figure 2.16. Options for different viewers appearing upon right click on a table dataset in the product viewer, among them TablePlotter and OverPlotter.**

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

- *Invoke 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)
```

- *Limitation on Datasets*

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

## Layout of the TablePlotter

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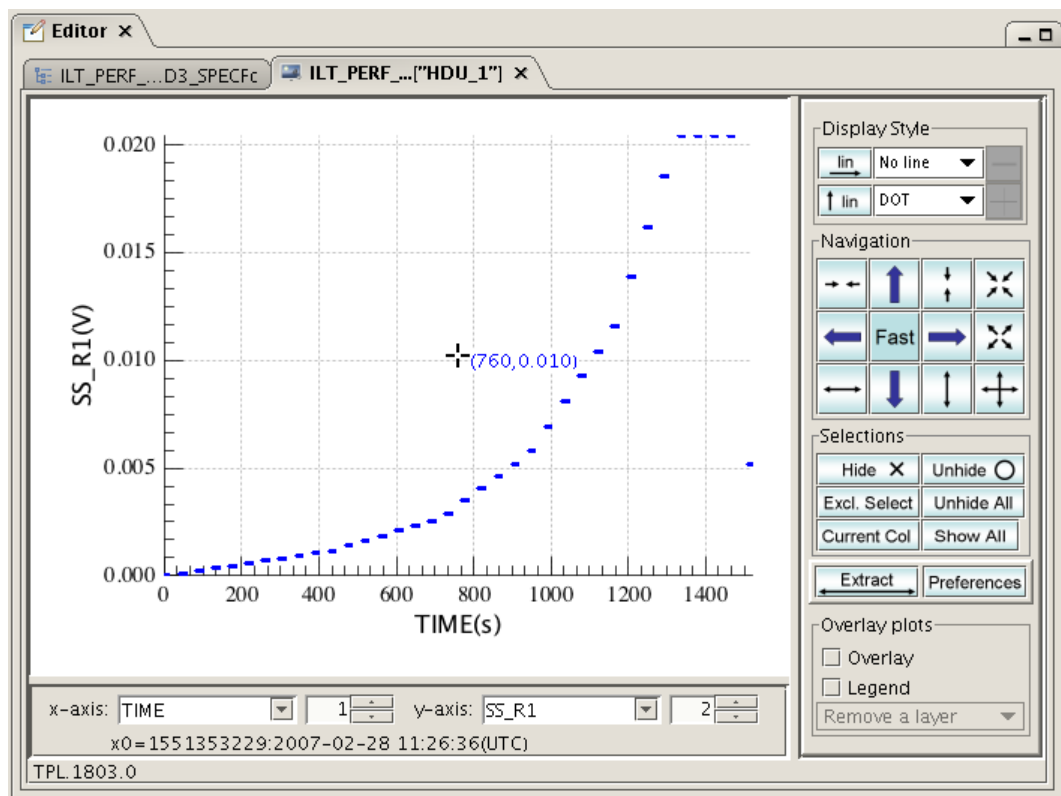


**Figure 2.17. Layout of the TablePlotter GUI in a HIPE editor view.**

## 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 coumn 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. The colum is selected by left clicking on the respective name. 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.

- **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.
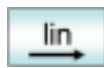
   This button signifies that the linear scale is selected for the X-axis. Clicking on the button will switch to logarithmic scale.

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

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

   This button signifies that 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).**

   This button zooms out simultaneously in X- and Y-axis.

This button zooms in simultaneously in X- and Y-direction.

This button zooms out along the X-axis only.

This button zooms out along the Y-axis only.

Pan the view towards the left/right.

Pan the view up/down.

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

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

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

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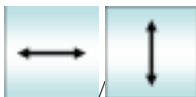
Switch the X/Y axis into free-scale mode.

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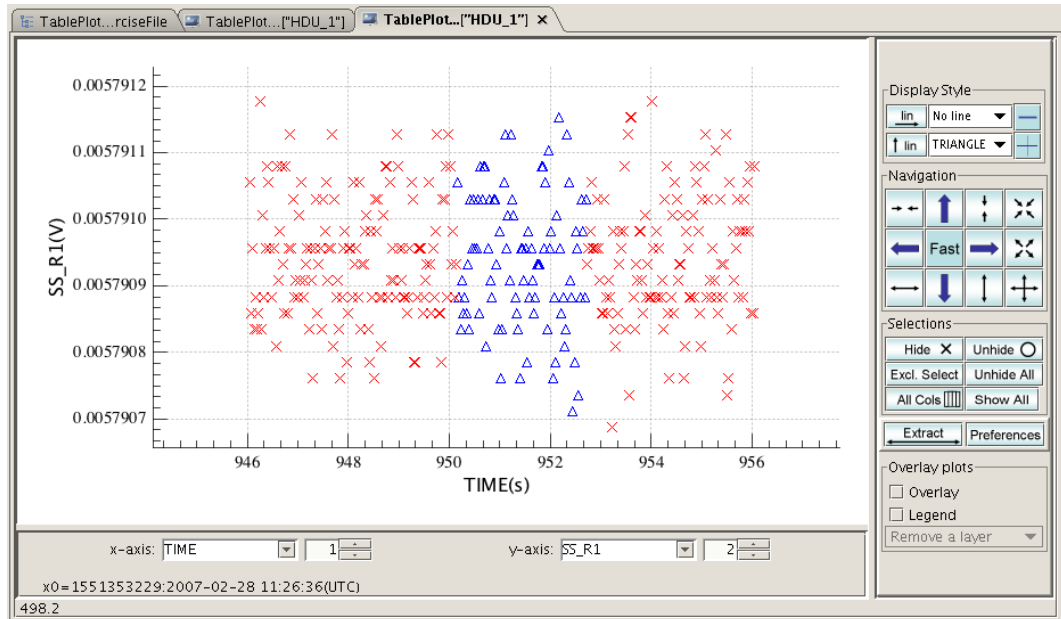
**Figure 2.18. The plot with selected (blue) and hidden (red crosses) data points.**

- **Printing and Saving the Plot:**

  Right click into the plot area of Table Plotter brings up a small window with the options "Save as" and "Print". The first one brings up a file selection dialog, that allows to save the current display as PDF, PNG, JPEG. or EPS image file. The second choice brings up a printer dialog including general selection of print service, page setup, and print appearance.

- **Dataset Extraction:**

  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.

   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  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) will pop 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).
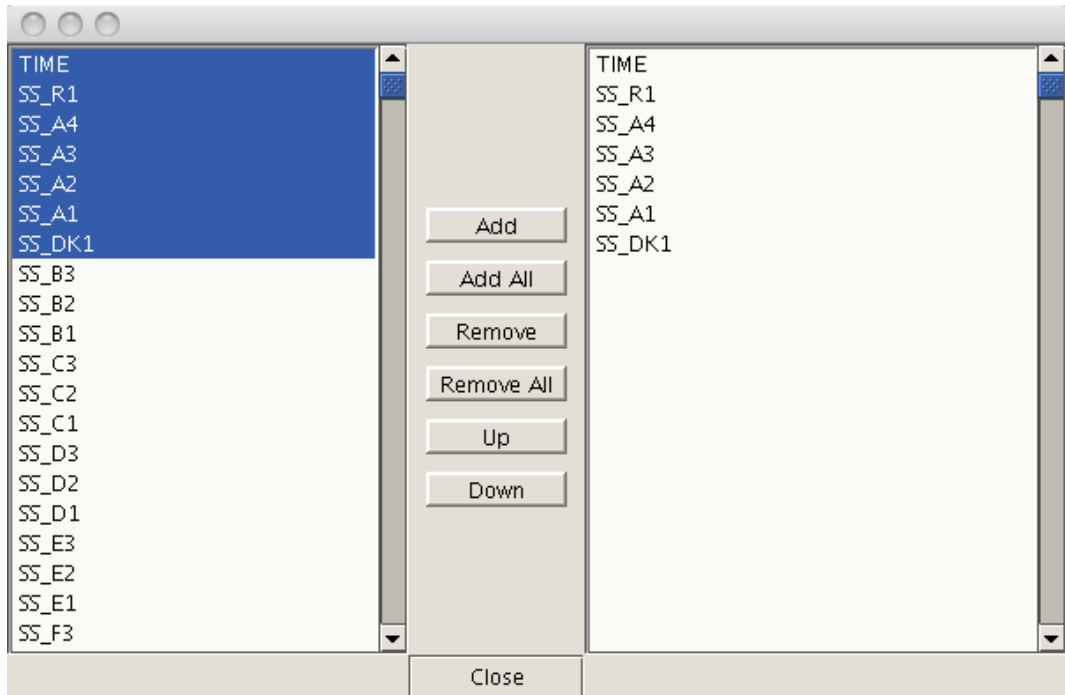


**Figure 2.19. Extract Selected Data from Multi Columns to a New DataSet.**

Hitting the **Close** button will complete the extraction and an option is provided to change the default name of the new dataset. A default name is given too (see Figure 2.20).
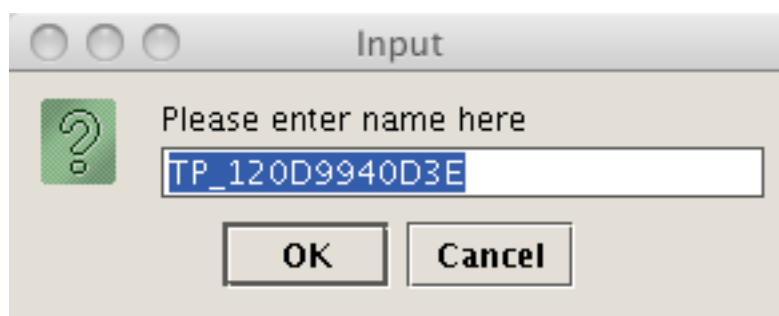


**Figure 2.20. Rename panel for new extracted table dataset. A default name is present that will be taken with the OK button.**

After OK in the rename panel, the newly created table dataset appears as a new variable in the Variables view of HIPE and can be worked with in the same way as any other table dataset in the session. In particular it can be displayed again with the Table Plotter. Other tools like the Over Plotter (see below), the Power spectrum tool, and the simple Dataset Viewer are available as well and can be applied.
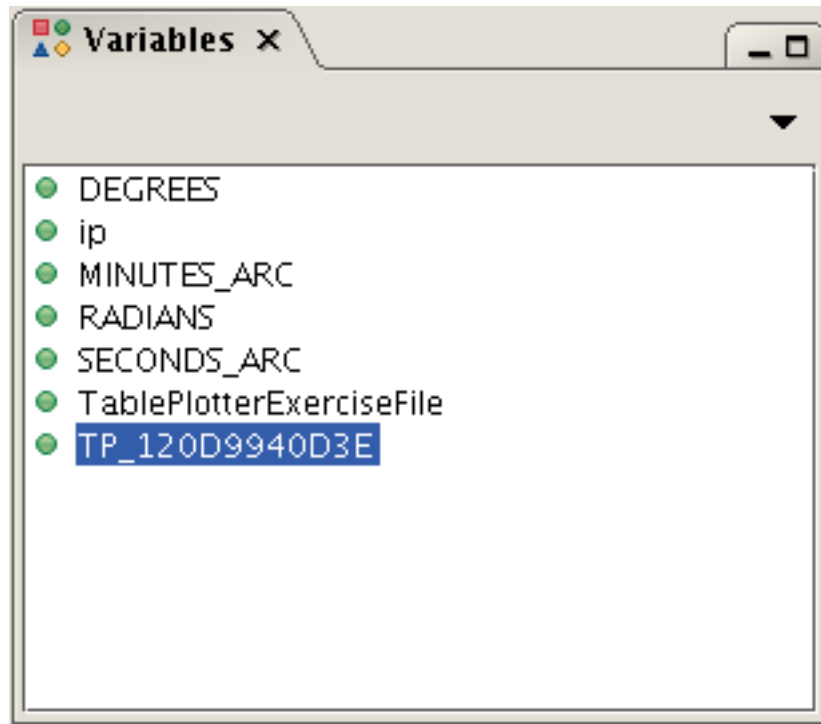
**Figure 2.21. The newly created table dataset appears as a new entry in the Variable view of HIPE. Double clicking opens it in its default application, right clicking opens a menu with all available applications for this type of dataset.**

- *Overlay Plots:*

  Even though the TablePlotter was primarily designed for single X-Y scattergram display, there is limited overlay capability available, simply because it was easy to do from a technical point of view. 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 2.22 ).
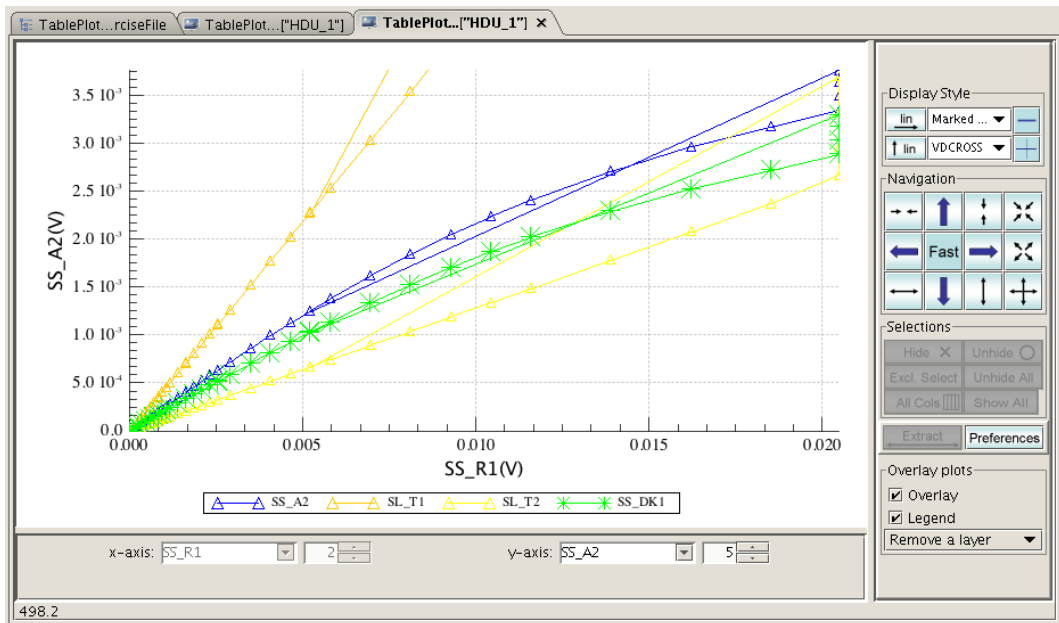
**Figure 2.22. Simple overlay plots of different columns plotted against the same X-axis are created by marking the Overlay field.**

- *Preferences:*

There are several parameters in Table Plotter, that control behavior of certain functions. These are accessible with the Preferences button.

 This button provides a drop-down menu, giving access to the zoom- and pan-properties, the display rules for complex data, and the control over time offset subtraction on the X-axis of the plots (see Figure 2.23).
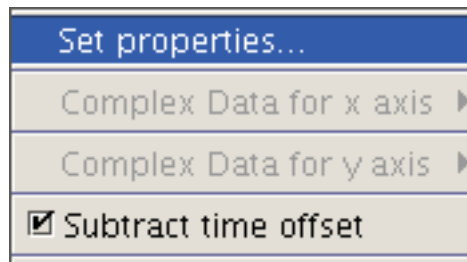


**Figure 2.23. Preferences: This drop-down menu gives access to the zoom- and pan-properties, the display rules for complex data, and the control over time offest subtraction on the X-axis.**

Choosing the **Set properties** entry brings up the respective panel, where individual percentages for panning and zooming in fast and slow mode can be set. **Reset** and **cancel** buttons are available for convenience (see Figure 2.24).
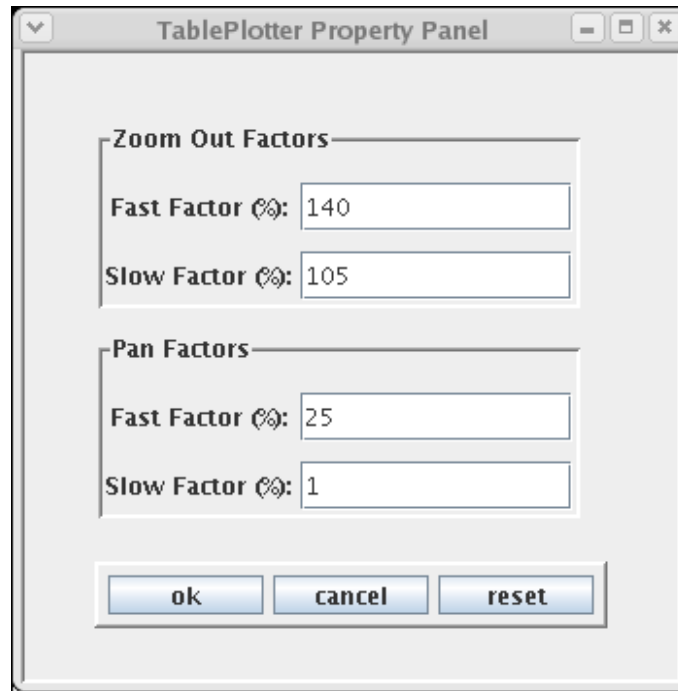
**Figure 2.24. Preferences: The properties panel allows for selection of zoom- and pan-factors individually for fast and slow modes.**

The Table Plotter is able to show complex data in 4 different representations, the modulus, the real part, the imaginary part, or the phase. This is individually controlled for each axis through the preferences manu.
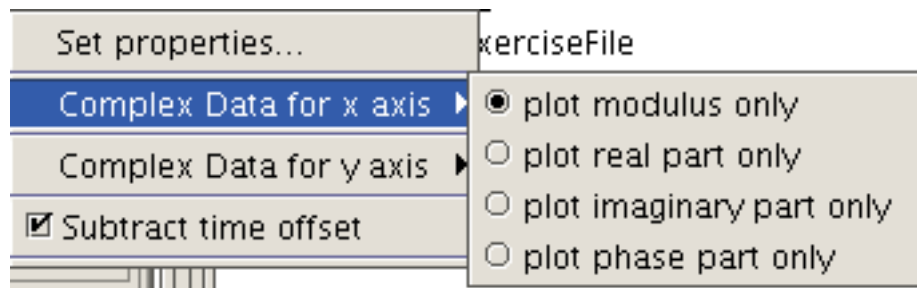


**Figure 2.25. Preferences: Complex data can be displayed in 4 different ways as shown in this properties menu.**

The last entry in the Preferences menu is a flag for subtracting time offsets from the data chosen for the X-axis. This is useful for 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. If this flag is activcated, and if the column bears a time unit, the first time in the X-axis column is subtracted from all other values in this column for display only. The subracted value is displayed below the selector for the X-axis and also converted to a start date date in UTC.

The selected preferences are stored in a properties file and will be "remembered" in the next call to 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.

## 2.3.1.2. The Over Plotter

### Introduction

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.

### 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.

### Layout of Over Plotter

The Over Plotter main view looks very similar to the Table Plotter, but also shows a few important differences.(seeFigure 2.26). The main differences are the "Layer Controls" panel, which replaces the "Overlay Plots" panel, and the addition of four synchronization buttons. The plot area now contains obviously more graphs and a second pair of axes to the top and right sides.
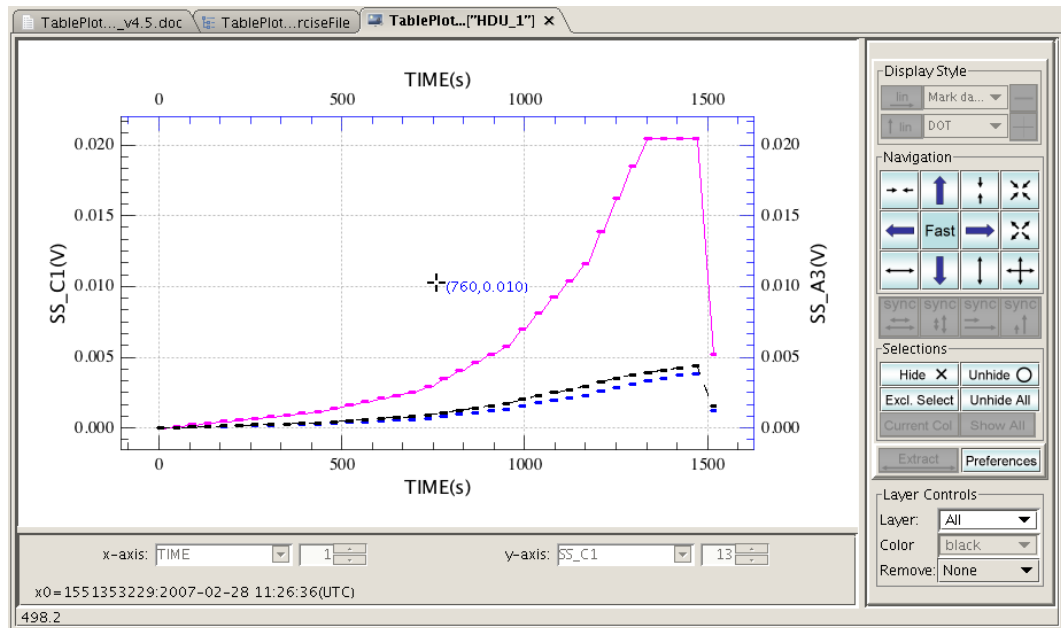
**Figure 2.26. The main panel of Over Plotter is very similar to that of the Table PLotter. New features include the Layer Controls panel and the synchronization buttons. This Over Plotter is in "All Layers" mode.**

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 2.26 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.
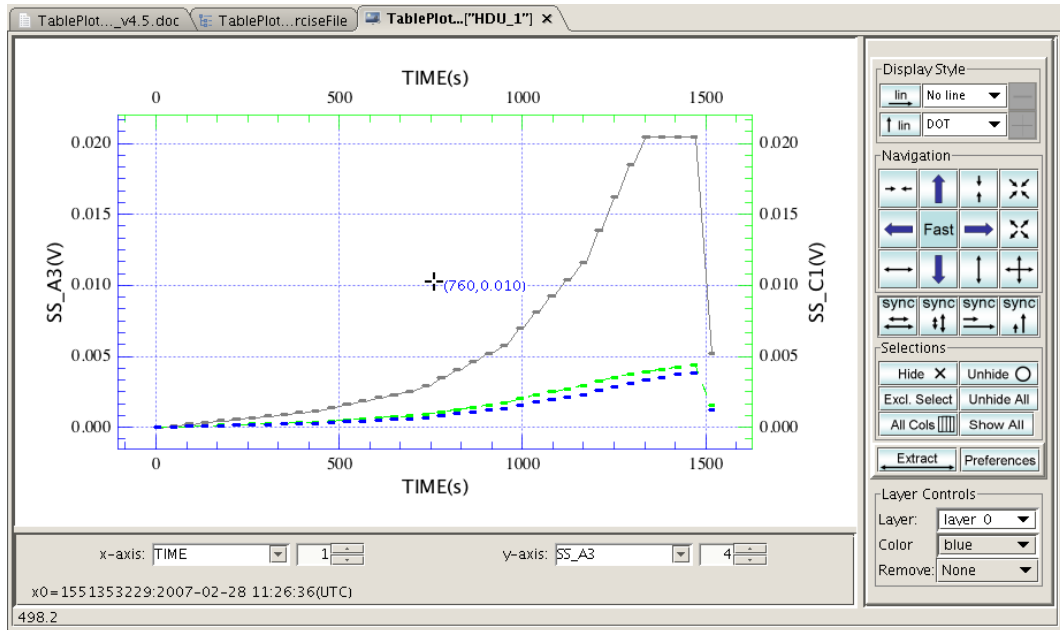
**Figure 2.27. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected color and the secondary layer is displayed in green. All other layers are displayed in grey color.**

In Figure 2.27 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 2.27, where the third layer that was gray in the previous example is now chosen to be prime, and the colors change accordingly.
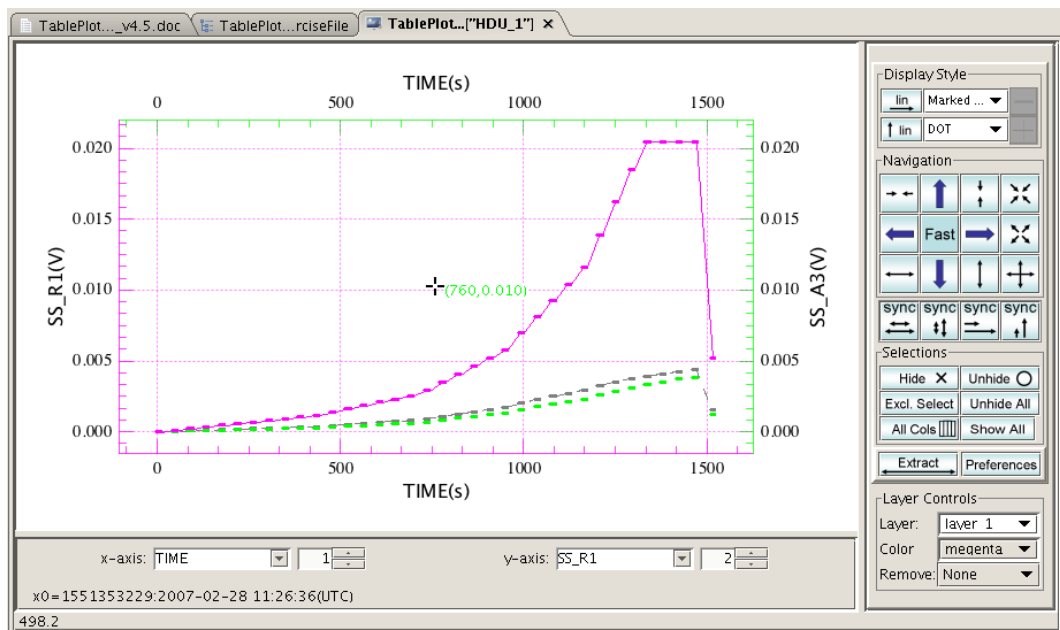


**Figure 2.28. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected color and the secondary layer is displayed in green. All other layers are displayed in grey color. These are the same layers as in the previous figure, but after selecting Layer 1 to become prime.**

## 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.

**Color [blue ▼]** This drop down menu button shows the currently selected color. This menu is only available in single layer mode and selects the color of the currently active layer. In this mode by default the secondary layer appears in green, while the primary layer appears in blue or another color manually selected by this menu. In All Layer mode all datasets appear in their selected colors.
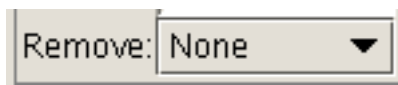
**Remove: [None ▼]** This drop down menu button allows to remove specific layers. This menu is available in any mode.

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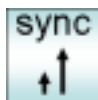
**sync ↑** This button synchronizes the offset of the Y-axis of the primary layer to the offset of the secondary layer, i.e. the primary layer is shifted in Y-direction such that the values where the bottom X-axis cuts the primary and secondary Y-axes become the same.

With all the possibilities of Table Plotter, except for the overlay function, available for each layer, many combinations are possible. In Figure 2.29 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.
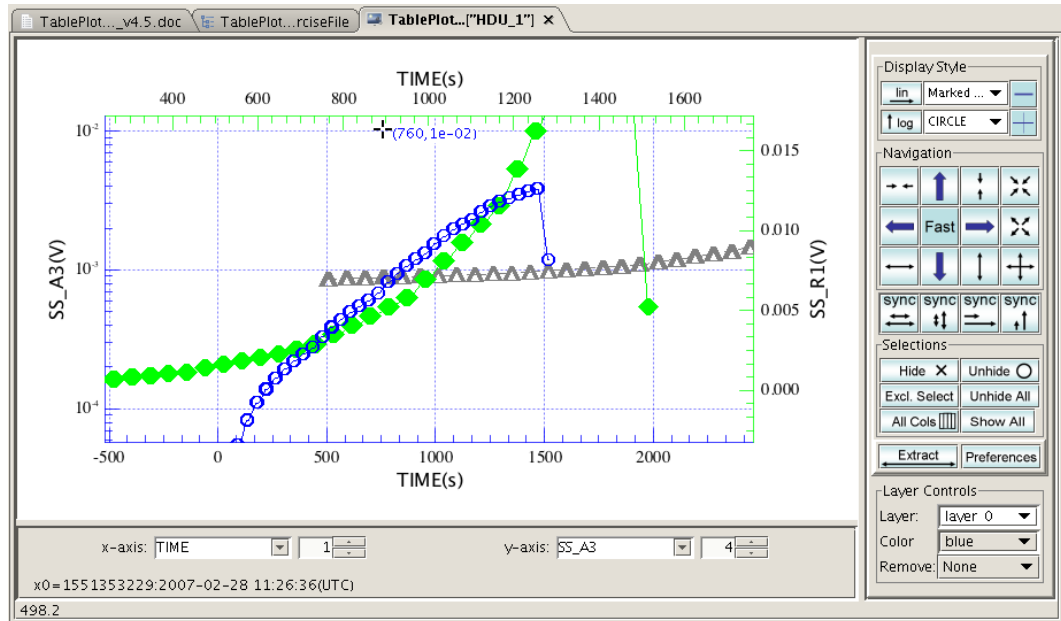
**Figure 2.29. A complex example for illustration. The Over Potter is in "Single Layer" mode. The primary layer is displayed in blue with large symbols and connected by a line. The Y-axis is set to logarithmic mode. The secondary layer is displayed in green with large filled diamonds. The third layer 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.

## 2.3.1.3. 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 2.30, below).
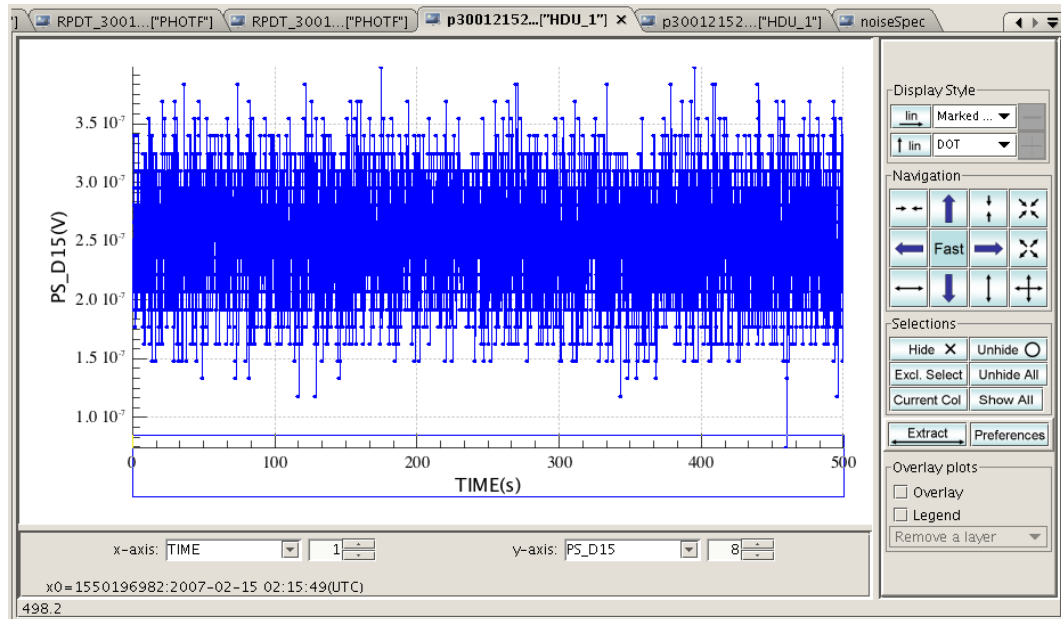
**Figure 2.30. A signal timeline displayed in Table Plotter that the Power Spectrum generator can be applied to.**

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 2.31, below).



**Figure 2.31. 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 2.32.
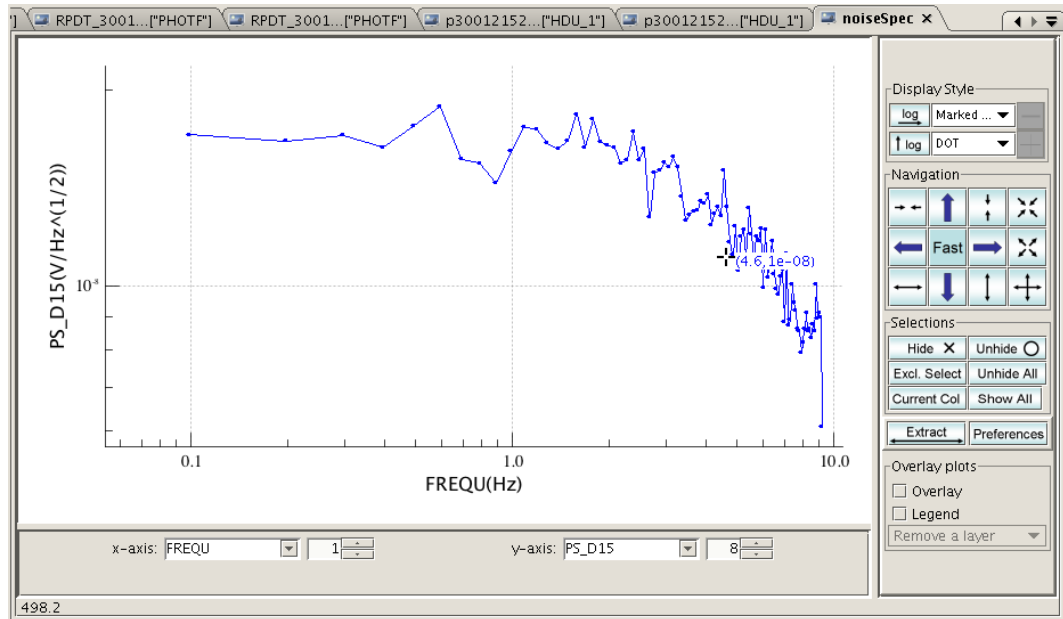
**Figure 2.32. Displaying the newly created power spectra in the Table Plotter.**

# 2.3.2. 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.

## 2.3.2.1. 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*.

## 2.3.2.2. Coordinate Conversions: the `Wcs` Class

A `Wcs` makes sure you have all the information to do coordinate conversions. This only holds if the `Wcs` is valid. In the following example is shown how you can construct such a `Wcs`.

```
myImage = SimpleImage()
myImage.setImage(RESHAPE(Double1d.range(200*300), [200,300]))

myWcs = Wcs()

myWcs.setCtype1("LINEAR")
myWcs.setCdelt1(5)
myWcs.setCrval1(200)
myWcs.setCunit1("K")
myWcs.setCrpix1(0)

myWcs.setCtype2("LINEAR")
myWcs.setCdelt2(0.05)
myWcs.setCrval2(2.0)
```

```
myWcs.setCunit2("V")
myWcs.setCrpix2(0)

myImage.setWcs(myWcs)

print myImage.getWcs() # To see the Wcs of the image
print myImage.getWcs().isValid() # To see whether the Wcs is valid
```

**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 creates a coordinate system, where temperature and current are set for the axes. The x-axis is linear (*ctype1*), has the reference pixel in pixel 0 (*crpix1*), which corresponds to column = -1, has a value of 200 in the reference pixel (*crval1*), uses steps of 5 degrees (*cdelt1*) and has Kelvin as unit. The y-axis on the other hand, is also linear (*ctype2*), has the reference pixel in row 0 (*crpix2*), which corresponds to row = -1, has value 2 in the reference pixel (*crval2*), uses steps of 0.05 degrees (*cdelt2*) and has Volts as unit.

**Note**

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

To ensure the possibilty to convert pixel coordinates to sky coordinates and vice versa, you can make use of the `Wcs` class. This can be done using the standard `Wcs` parameters, as shown in the example below.

```
myWcs = Wcs() # (1)
myWcs.setCrpix1(128)
myWcs.setCrpix2(128) # (2)
myWcs.setCrval1(101.676612741936)
myWcs.setCrval2(0.829427624677429) # (3)
myWcs.setCtype1("RA---TAN")
myWcs.setCtype2("DEC--TAN") # (4)
myWcs.setRadesys("ICRS")
myWcs.setEquinox(2000.0) # (5)
myWcs.setCd1_1(-1.9064468150235E-6)
myWcs.setCd1_2(3.39797311269006E-4)
myWcs.setCd2_1(3.39811958581193E-4)
myWcs.setCd2_2(1.580446989748E-6) # (6)
```

1. The creation of a `Wcs`.

2. The reference pixel pixel is set to pixel (128,128).

3. The reference is set at the position with right ascension 6h45'42.387" and declination 0d49'45.94".

4. The projection is set. The first axis defines the right ascension, the second axis the declination. Both use the gnomonic projection.

5. We use the standard ICRS type and set the equinox to 2000.0.

6. The linear transformation matrix is set. This defines the pixel scaling and the rotation of the `Image`.

## 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. In HIPE, you can explore your image just by double-clicking on it in the *Variables* view, thus opening

the *image explorer*. The image explorer and `Display` have the same graphical interface, but from the command line you can only access `Display`.

To display an image use the following command:

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

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.

**Note**

When you create a Double2d(height, width), you create an array of pixels where height is the number of rows and width is the number of columns.
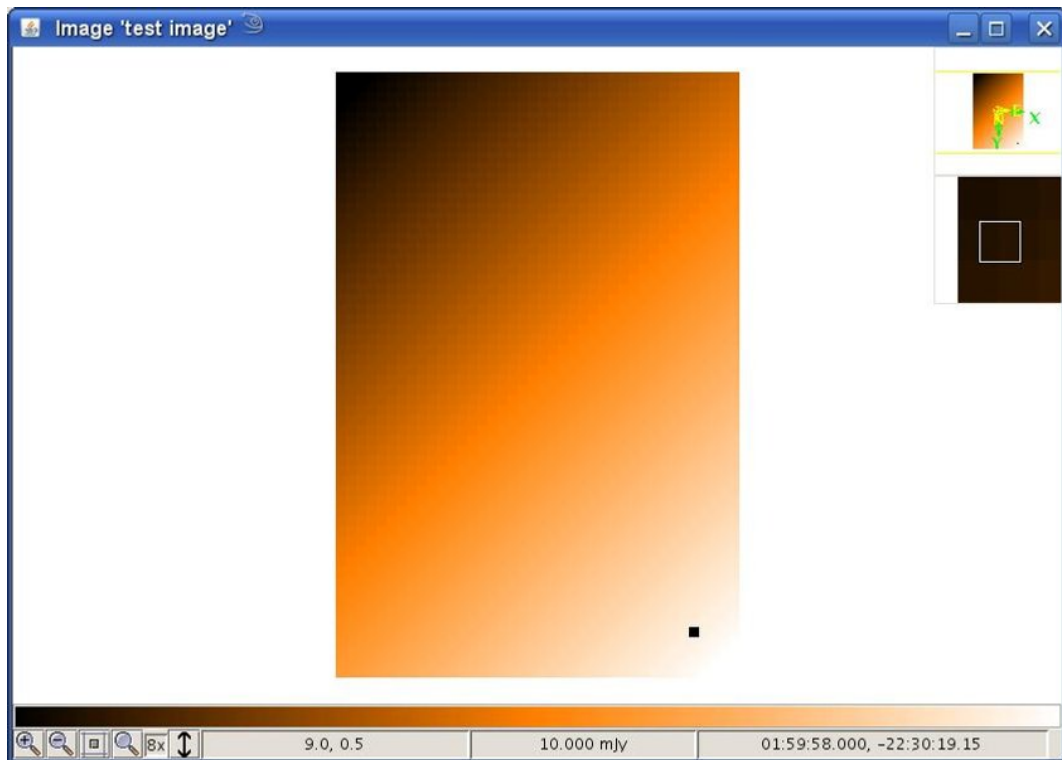


**Figure 2.33. Display of "myImage1".**

Flagged pixels are shown as black pixels. On the right hand side, you see two smaller frames. The upper one gives an overview of the image together with the x- and y-axes, and the north- and east-axes. The lower one zooms in on the image at the current mouse position. At the bottom of the window you see the colour bar (on which you can click and move the mouse to change its slope) and the status bar. The numbers in the status bar are the pixel coordinates (y,x), the intensity value and the sky coordinates (if available) at the current mouse position. The button allow you to zoom in, zoom out, zoom to fit, return to normal zoom, and to flip the image (along the y-axis). Also the current zoom factor is shown.

**Note**

The pixel position is currently displayed as (y, x).

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 ate 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)
```

## 2.3.3.1. Display in more Detail

In this section we will describe some, but not all, methods that are applicable on `Display` objects. For an exhaustive list of all methods, have a look in the `Display` javadoc. To apply a certain method, you must type **myDisplay2.<method>**.

**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 |

## 2.3.3.2. 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 :

```
myDisplay2.addLayer(myImage1)
```

This way we add myImage1 to myDisplay2, as shown on the screenshot. You also see that a slider appears in the status bar, which you can use to switch between the different layers.
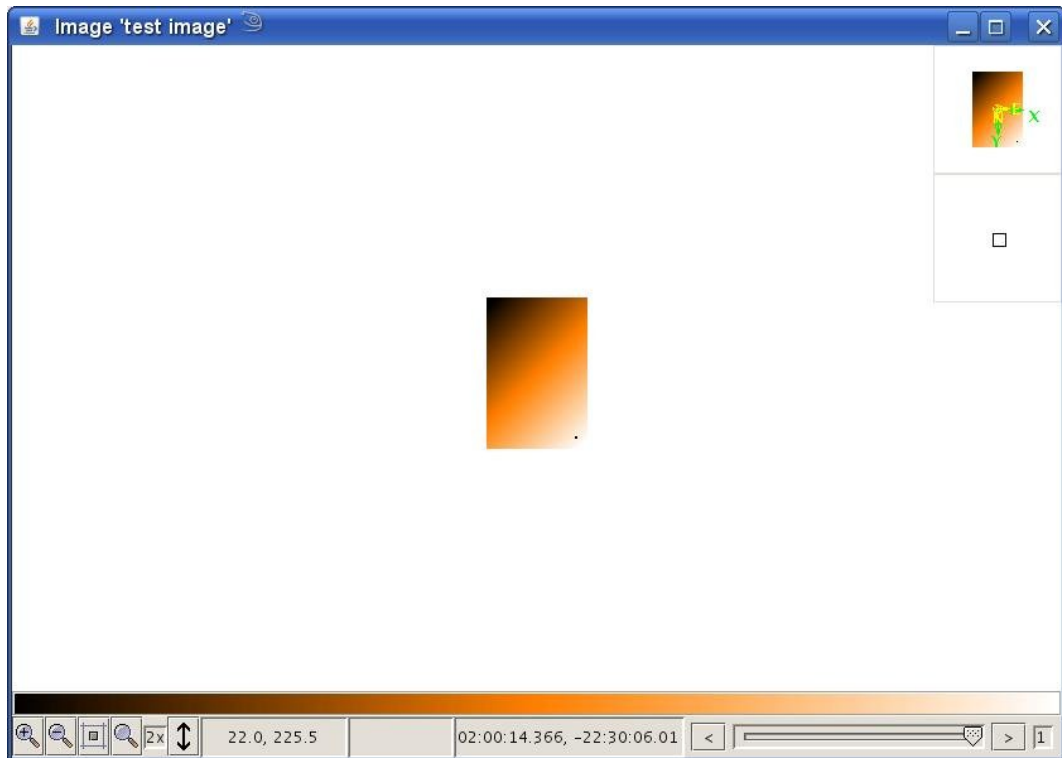
**Figure 2.34. Adding layers to a Display.**

> **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.

## 2.3.3.3. How to place Annotations on an `Image`?

It is possible to draw figures and put text, so called annotations, on an `Image`, using `Display`. This can be done in two different ways ;

1. Using the command line

2. Using the annotation toolbox

These are explained in the following two subsections.

### Annotations from the Command Line in your DP session

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)
```
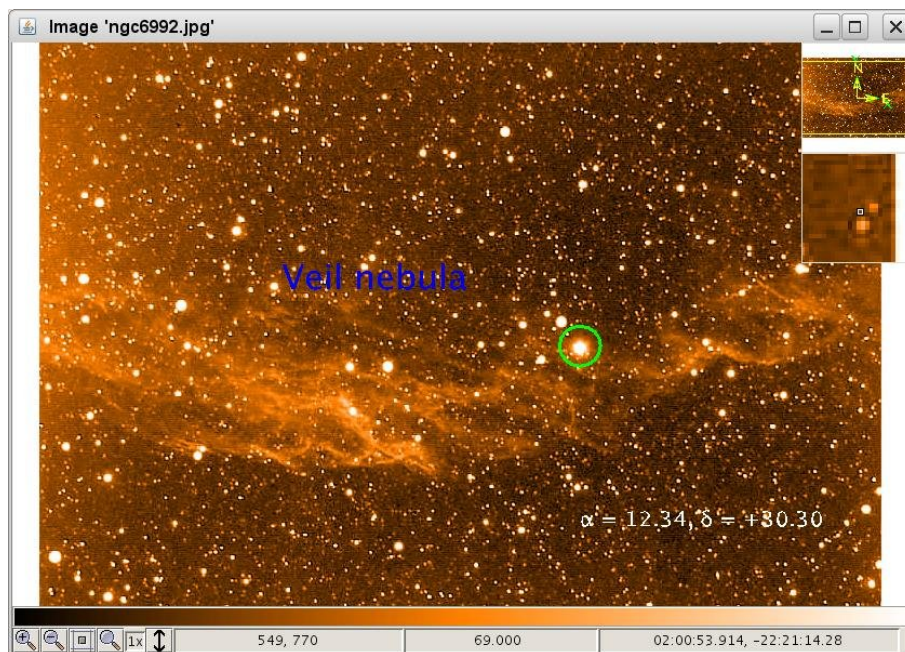


**Figure 2.35. Adding annotations to a Display.**

## The annotation toolbox

The annotation toolbox was covered in Section 2.2.5.3. You can open the annotation toolbox via the command line as follows:

```
myDisplay2.annotationToolbox()
```
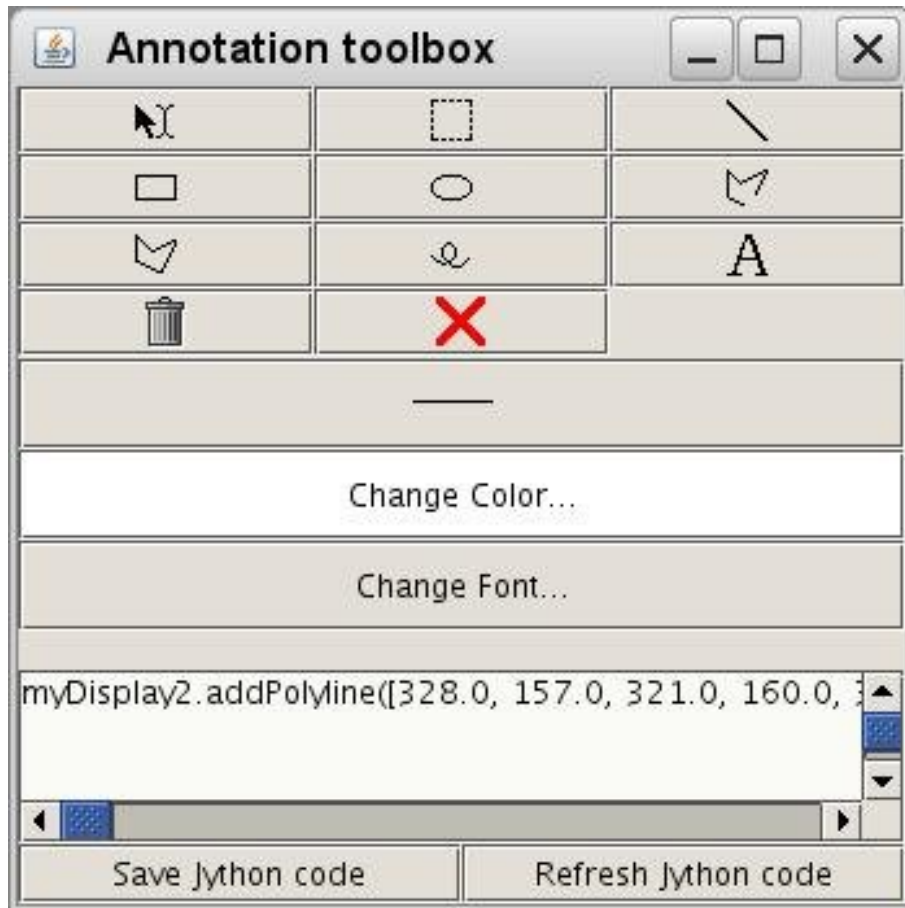
**Figure 2.36. The annotation toolbox.**

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.

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

## 2.3.3.4. Other Functionalities available via the Menu

### Editing the Colours

When you choose *Edit colors*, you will see a popup window appearing, where you can change

- the colour algorithm

- the colour map

- the intensity scale

The popup window is shown below.
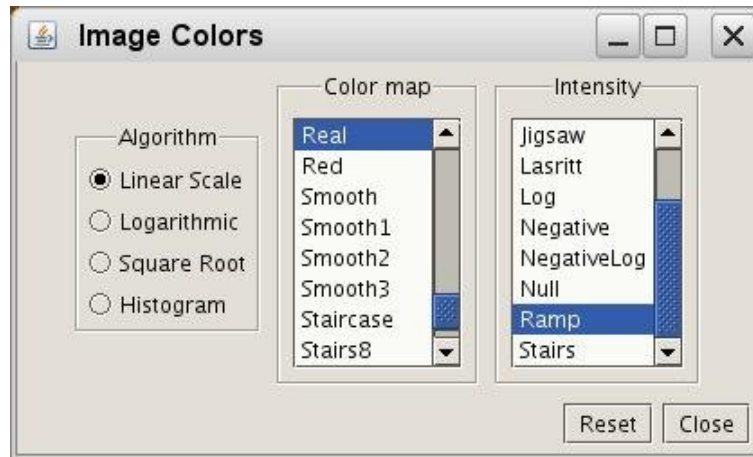
**Figure 2.37. Colour editting of image displays.**

The same popup window will appear when typing

```
myDisplay.editColors()
```

## Editing the Cut Levels

When you choose *Edit cut levels*, a window appears showing a histogram with the current cut levels and various possibilities to change the cut levels :

- you can play with the slider bar using the mouse

- you can cut away the outer parts of the histogram by specifying the percentage you want to keep

- you can apply a median filter

Either which possibility you choose, the histogram as well as the image will be adapted.

The popup window is shown below.
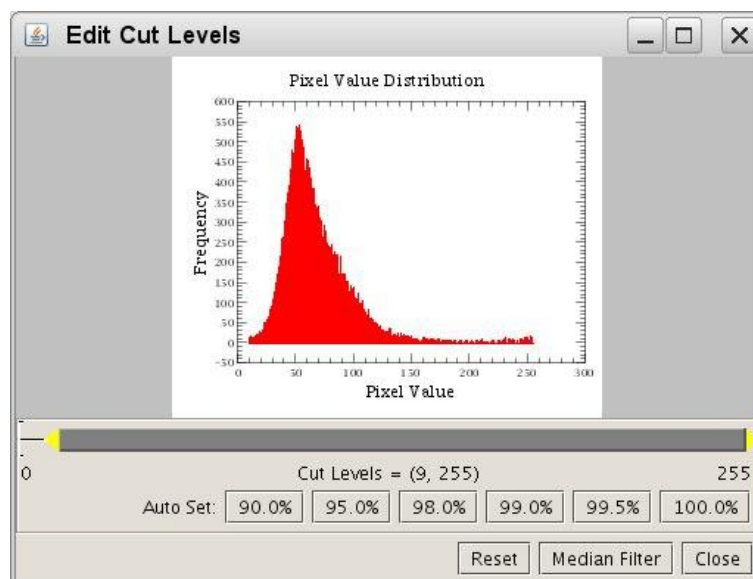


**Figure 2.38. Editting cut levels in displayed images.**

The same popup window can be opened, typing

```
myDisplay.editCutLevels()
```

## Printing and creating a Screenshot

When you choose *Create screenshot*, the current view or the image can be saved as \*.JPG, \*.PNG or \*.BMP. By *current view* we mean the image as you see it displayed, including the annotations. When saving the image you save the whole image, but without the annotations. When you choose *Print image*, you can print the image (without the annotations) on paper, or print it to a file.

## Flipping the y-axis

Instead of pressing the button with the arrow in the status bar, you can choose *Flip y-axis* from the menu that appears when right clicking on the image.

# 2.3.4. Viewing data cubes

A `SimpleCube` contains one or more 3d images and works in a very similar way to `SimpleImage`. A 3-dimensional datatypes should be given as inputs. For example:

```
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.
# See Figure 2.39.
```
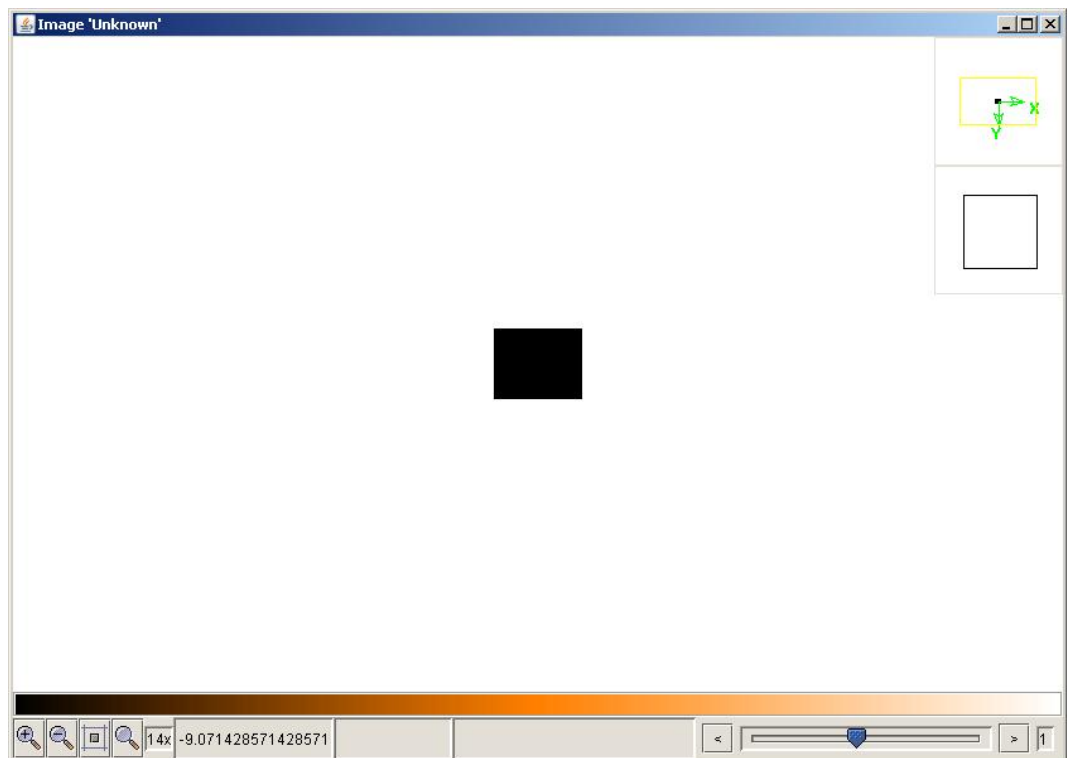


**Figure 2.39. Display of a cube. Note the slider to bottom right allowing the various images at different depths to be viewed.**

# 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 2.3.1.1 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.

You can find more demo scripts in Appendix A. The plot-related scripts are those beginning by `Test` (`TestPlotXY.py`, `TestAxis.py` and so on), at the bottom of the list.

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.title.text = -"Example plot"
plot.subtitle.text = -"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"
```

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 itt

```
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

```
layer.setAnnotation(0,Annotation(6.5,-10,"Test",color=java.awt.Color.GREEN))
```

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)
```

# 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).
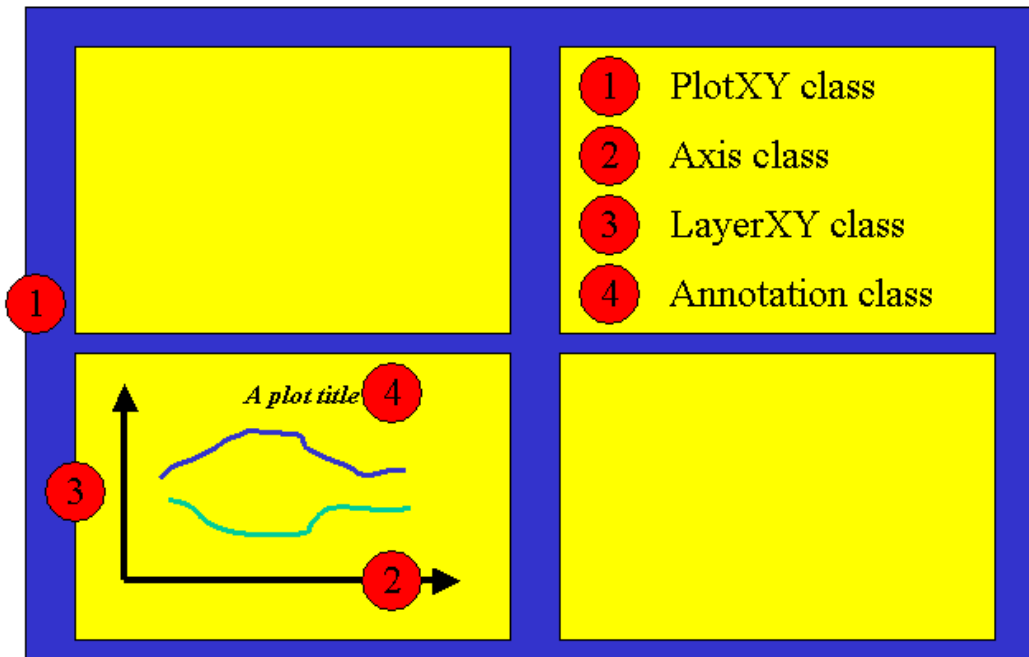
**Figure 3.1. Classes involved in plot operations.**

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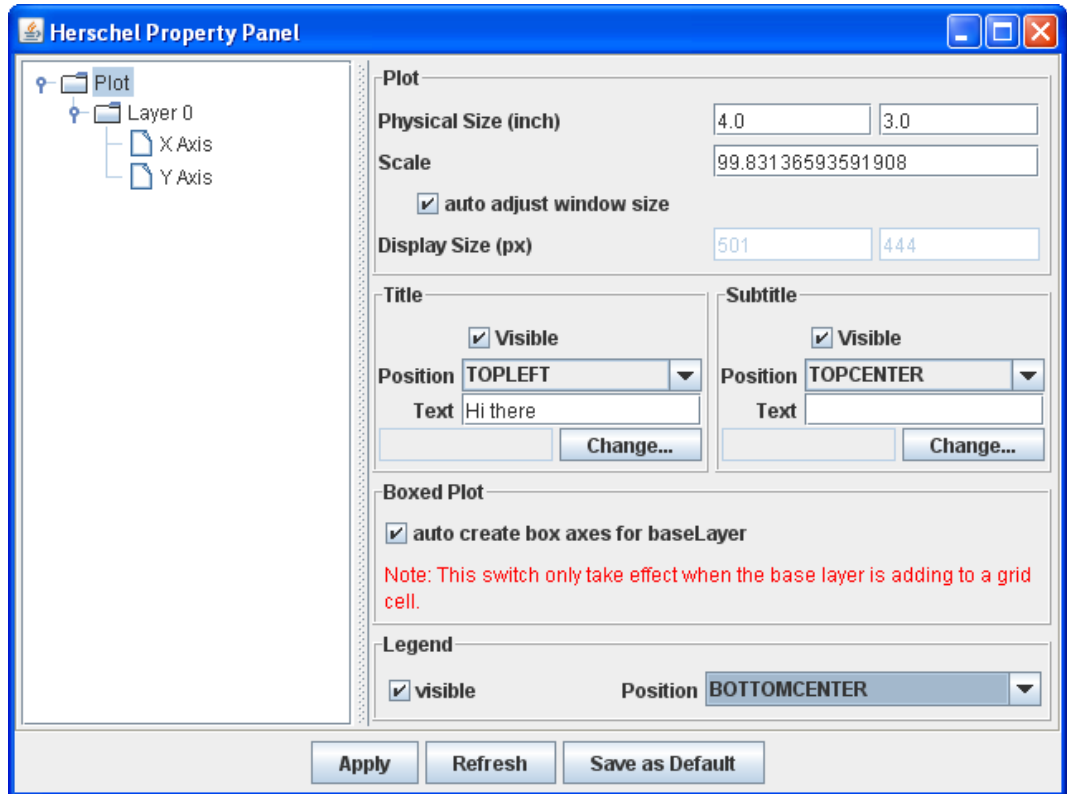
**Figure 3.2. The `Plot` section of the `PlotXY` properties dialog.**

## 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).
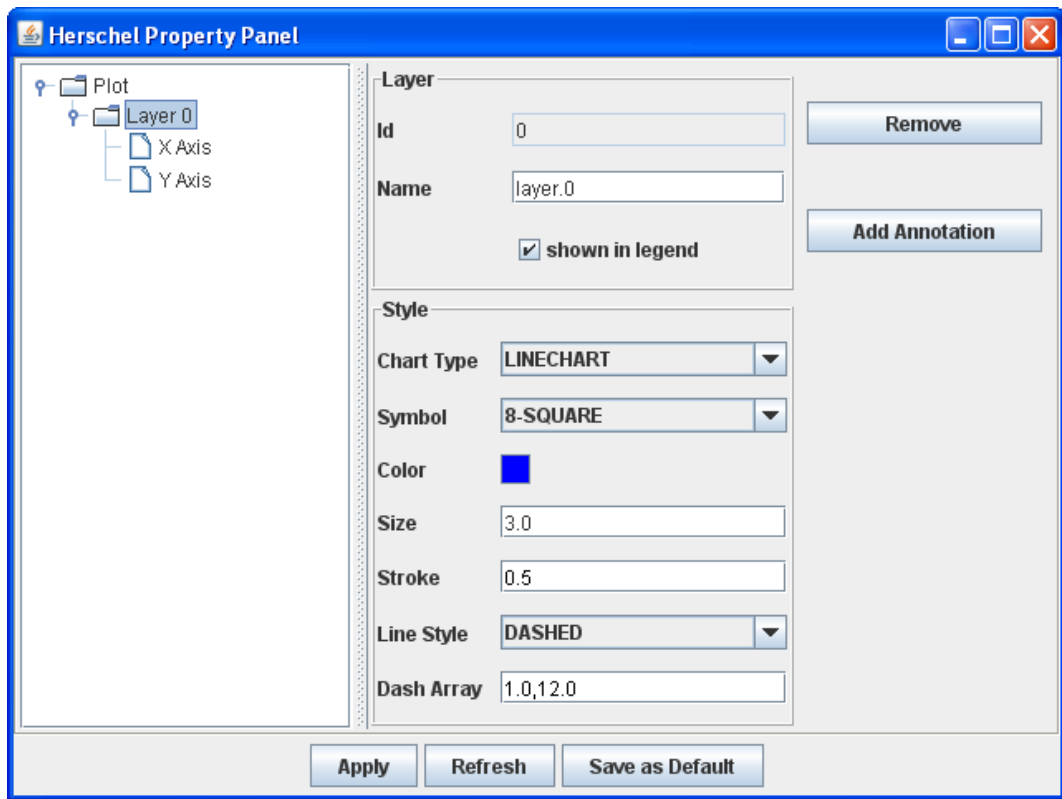


**Figure 3.3. The `Layer` section of the `PlotXY` properties dialog.**
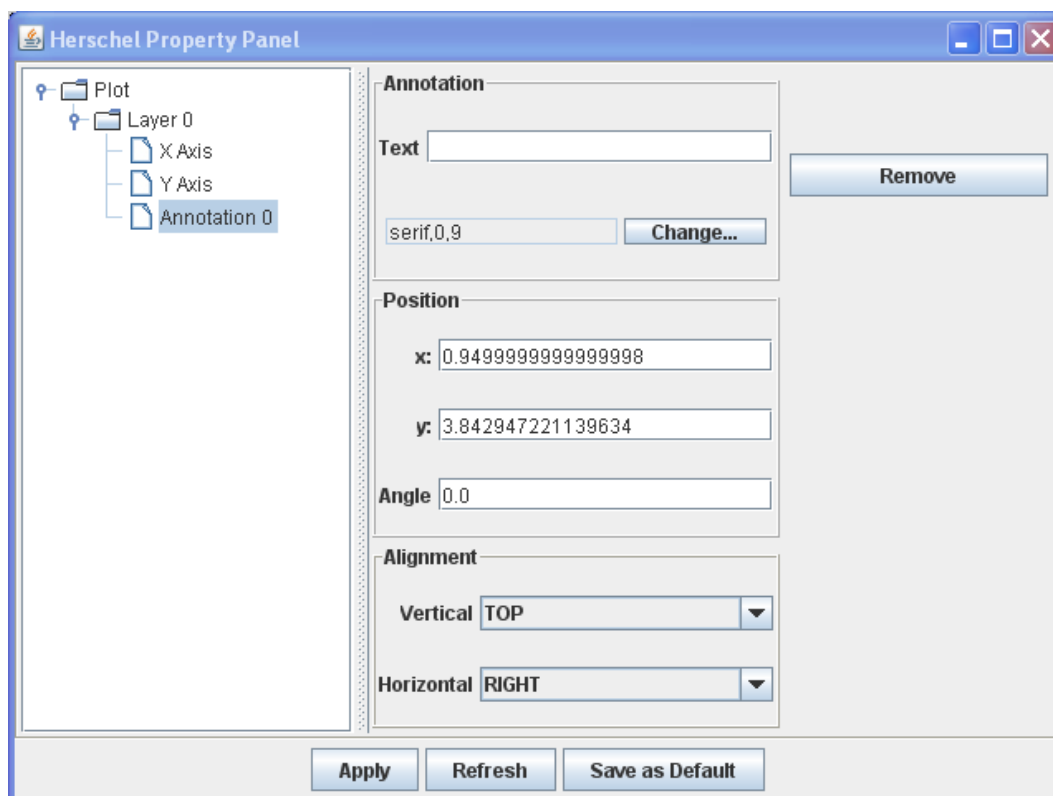
**Figure 3.4. Dialog for adding an annotation to a `Layer`.**

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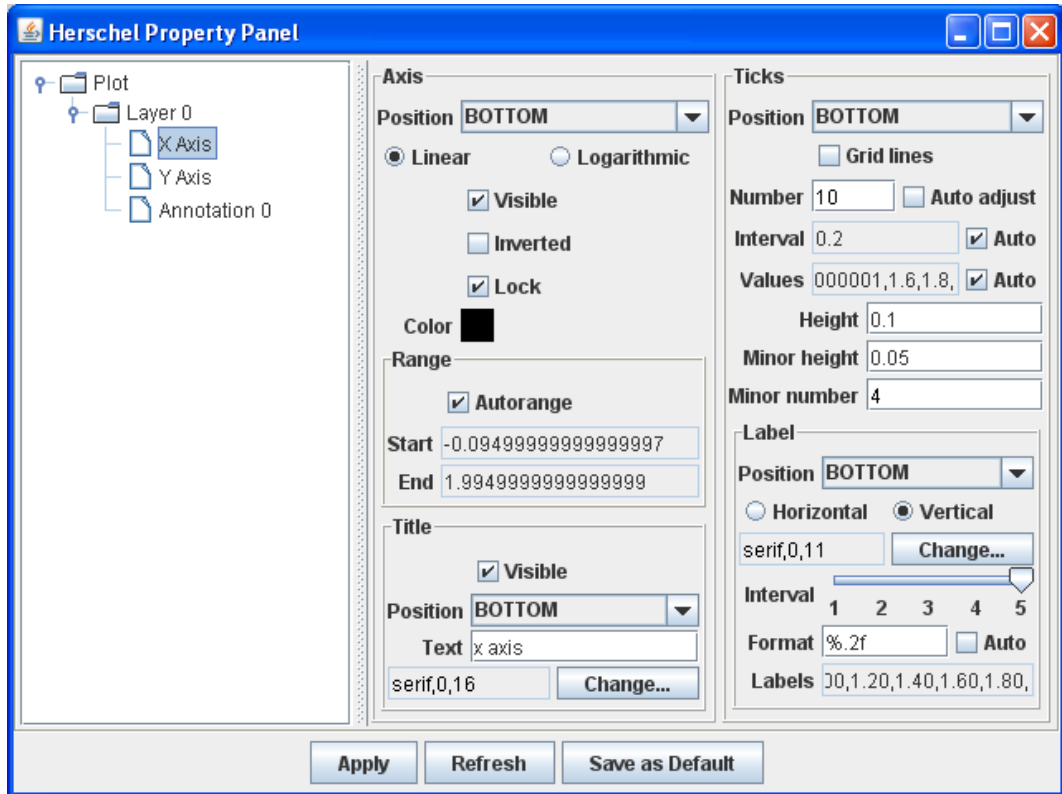
**Figure 3.5. The `Axis` section of the `PlotXY` properties dialog.**

## 3.3.1.4. How to use properties

The result of a property setup procedure (with a defined set of properties) is given in <u>Example 3.1</u>. 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.
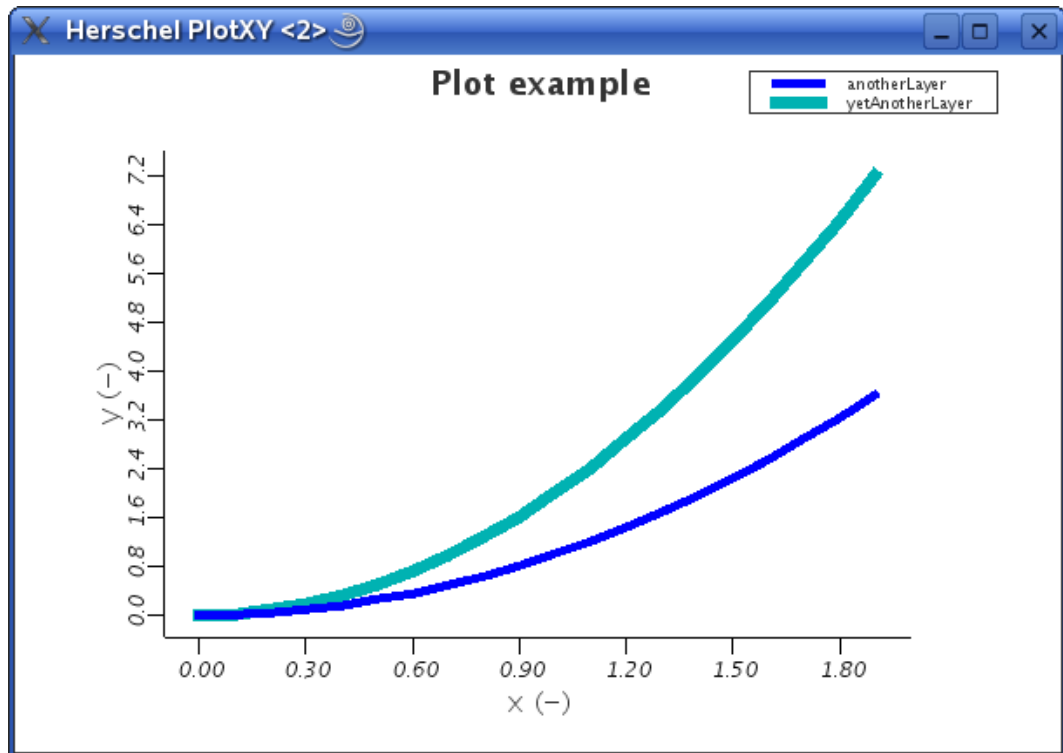
**Figure 3.6. This plot is the result of <u>Example 3.1</u>.**

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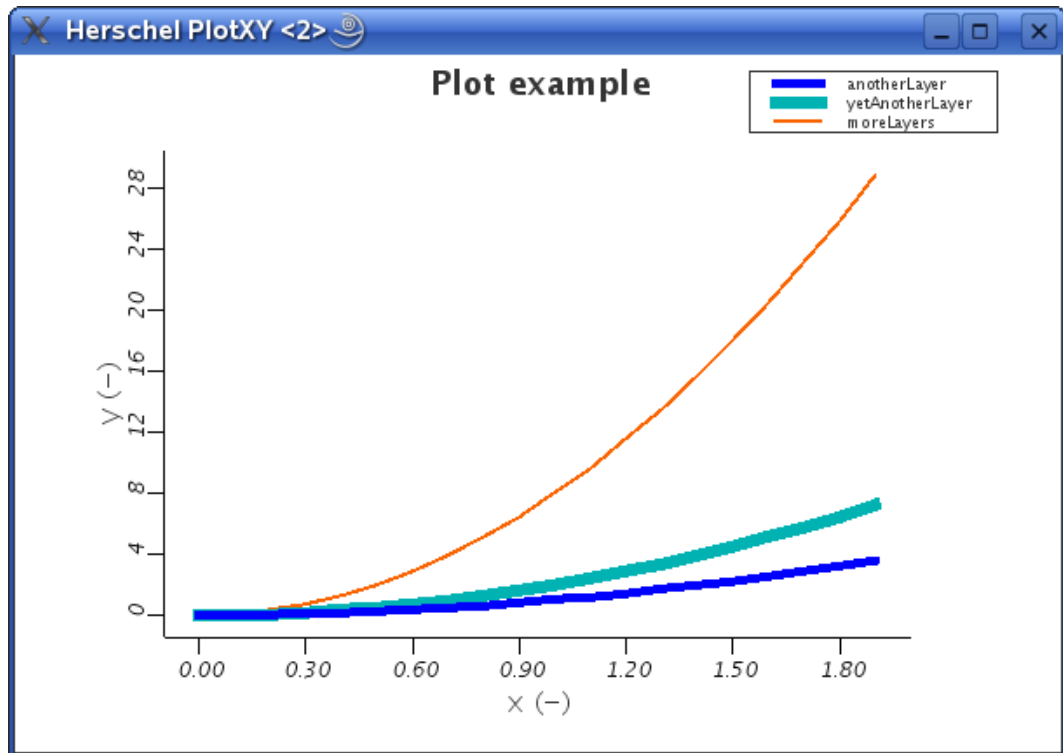
**Figure 3.7. Adding in another layer gives the orange curve (see text).**

# 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 in layers.**

| | |
|---|---|
| `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.<br><br>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. |
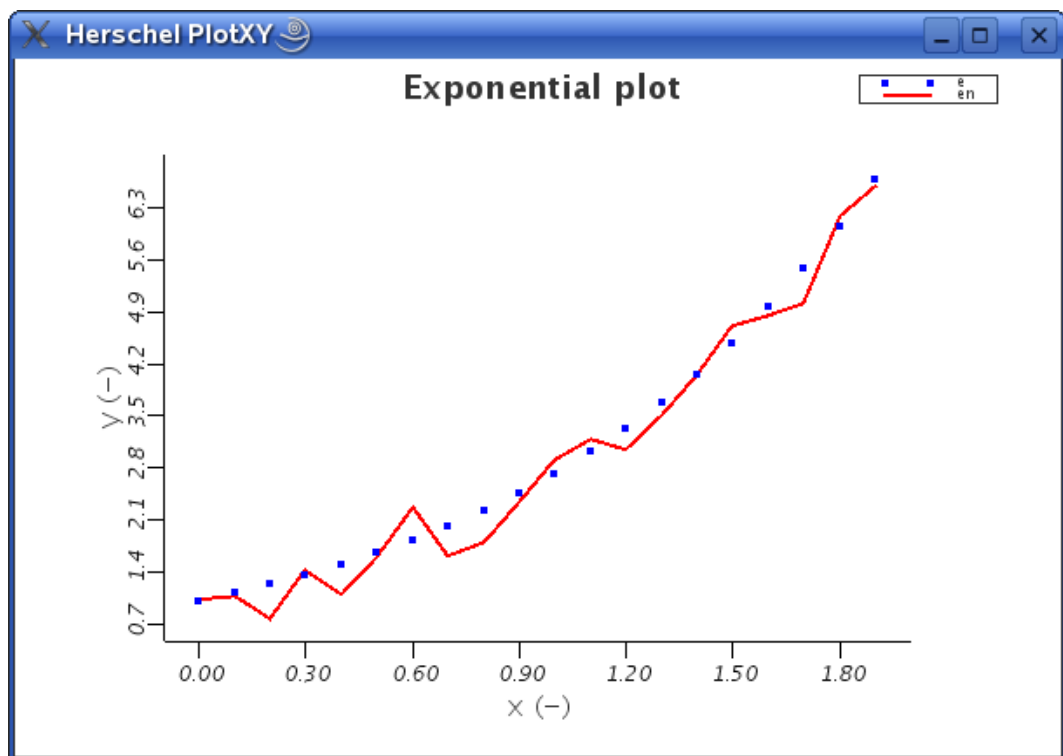


**Figure 3.8. Plot showing the result of manipulation of layers from the command line.**

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="$x_1^{2a}$"
```
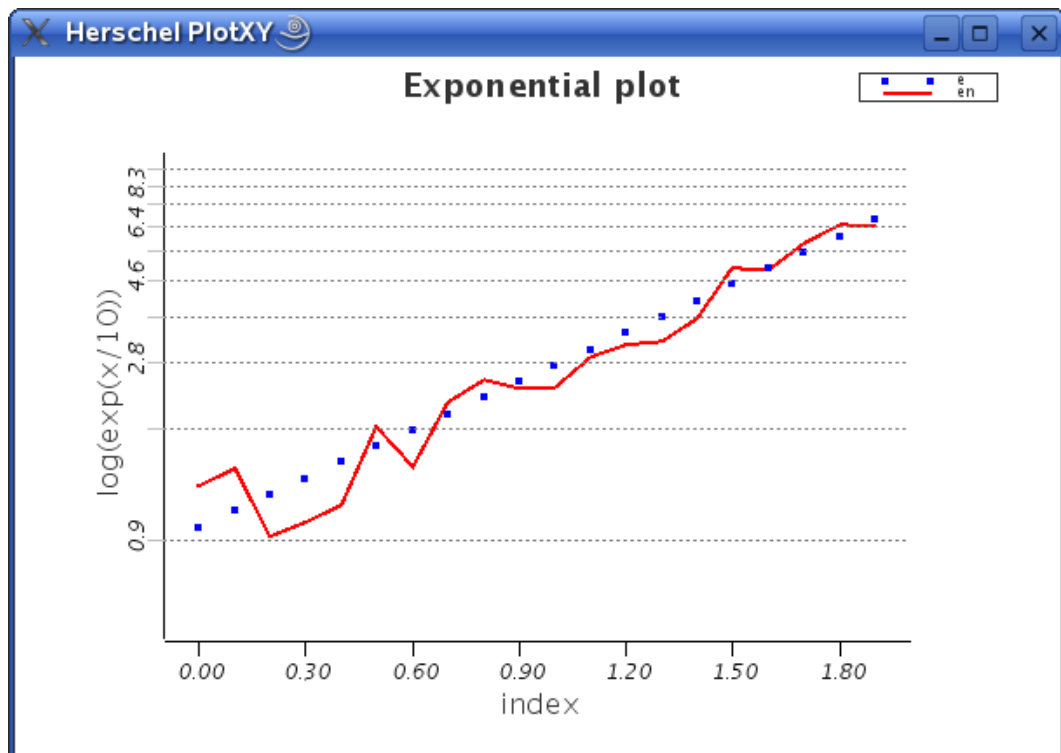


**Figure 3.9. Changing Axes, labels and added grid lines.**

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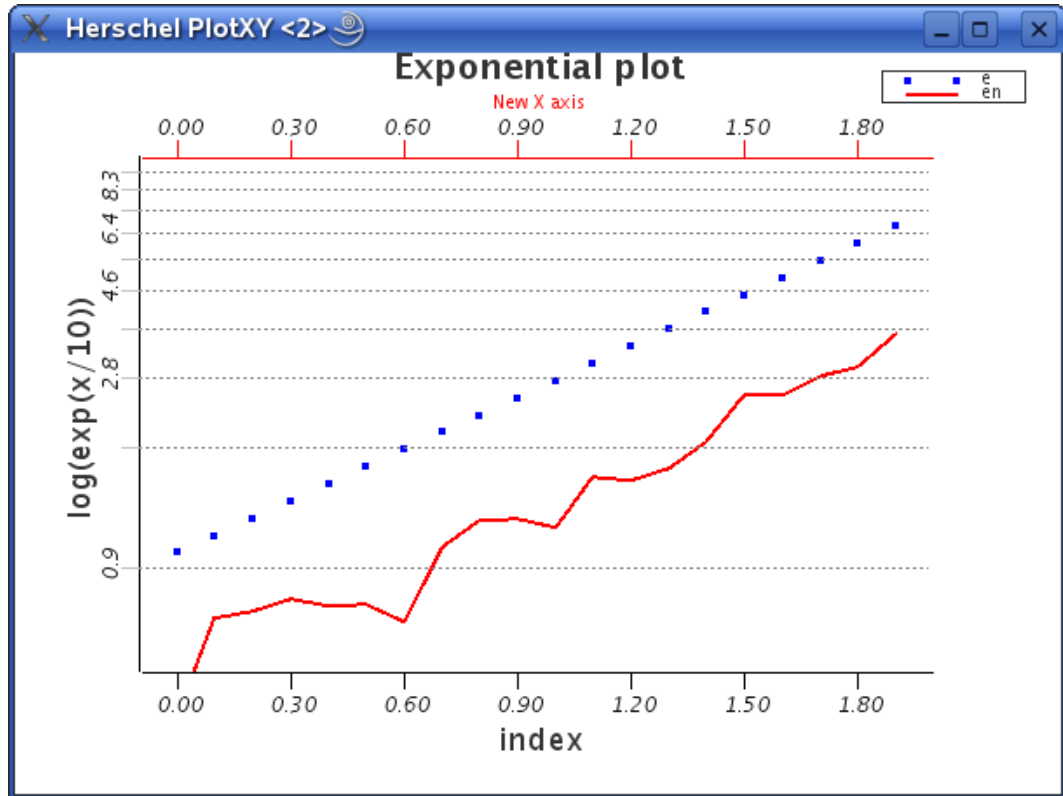
**Figure 3.10. Example of a second X-axis label relevant to the red line plot.**

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.**

| | |
|---|---|
| `getTick().getLabel().setColor (java.awt.Color colour)` | Sets the colour of labels. Get method available. |
| `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().`<br>`setOrientation(int n)` | Sets the orientation of the labels (0 for horizontal, 1 for vertical). Get method available. |
| `getTick().getLabel().setStrings`<br>`(String[] labels)` | Replaces the current labels with the values in an array of `String` objects. Get method available. |
| `getTick().getLabel().setPosition`<br>`(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`<br>`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`<br>`interval)` | Sets the interval (in axis units) between ticks. Get method available. |
| `getTick().setPosition`<br>`(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`<br>`minors)` | Sets the number of minor ticks displayed between two major ticks. Get method available. |
| `getTick().setValues(Double1d`<br>`values)` | Sets the values where ticks are to be placed. Get method available. |
| `getTick().setAutoAdjustNumber`<br>`(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`<br>`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.
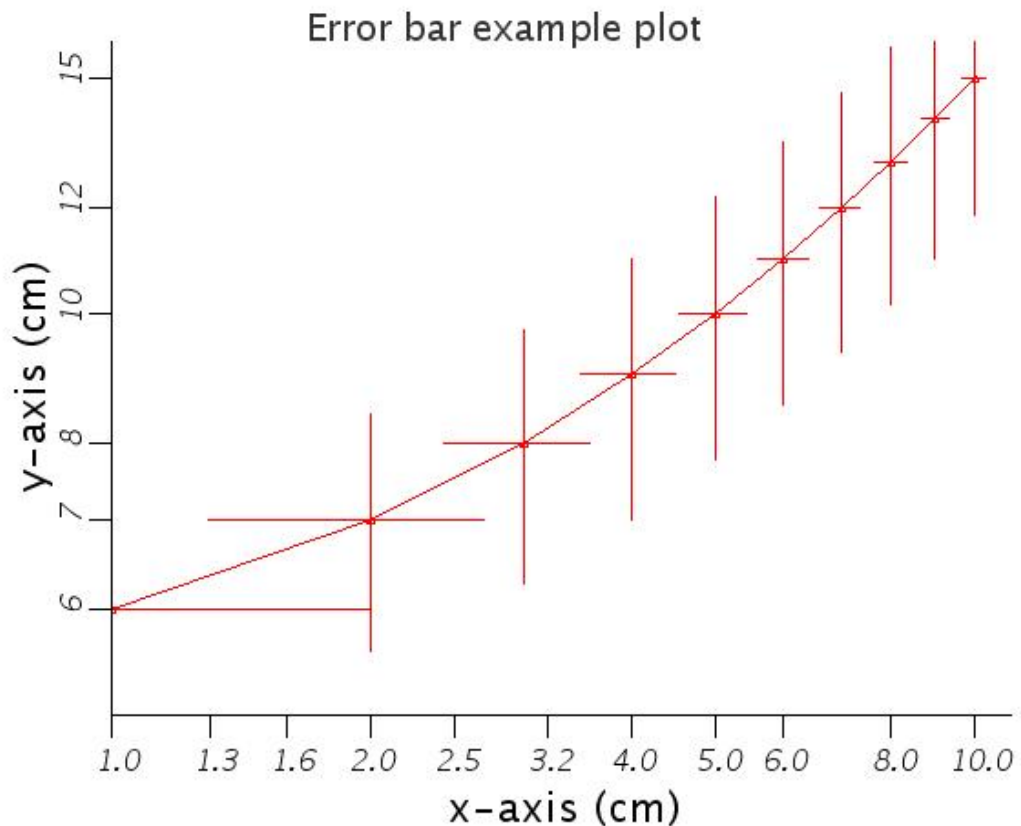


**Figure 3.11. Setting errors in a plot**
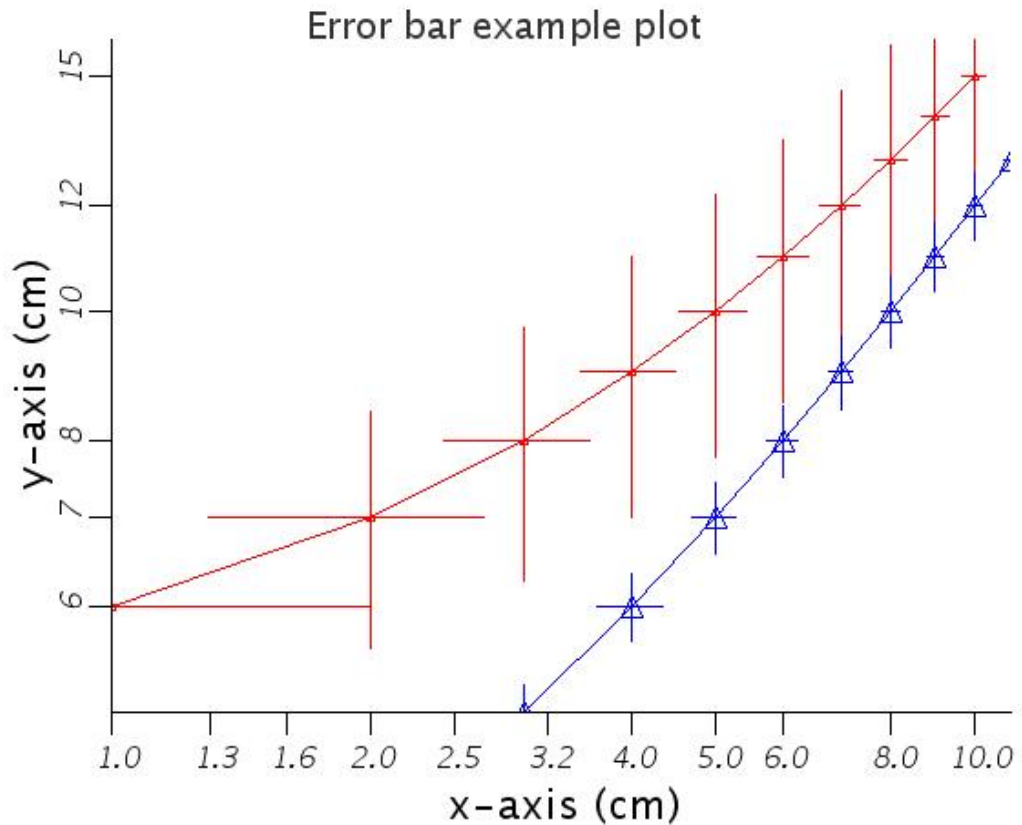
**Figure 3.12. Applying errors to a specific layer of a plot**

# 3.3.5. Annotating, 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(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)
# Change the name (and the legend) for this layer to say what we want
layer.setName("exp+noise")
# Place some annotation at position 1, 2
layer[0] = Annotation(3, 6, -"Noise on top of exp()", color=java.awt.Color.blue)
# Get the first layer of the plot...
layer = p.getLayerXY(0)
# -...and change its name
layer.setName("exp")
# Set a new style
layer.setStyle(Style(line = Style.MARKED, symbol = Style.FTRIANGLE, \
  color = java.awt.Color.green, symbolSize=7))
# Give the plot a title
p.title.text = -"Example of a layered plot"
# Save it as a PNG file for importing as a picture into documents etc.
p.saveAsPNG("myPlot.png")
# Alternatively, save it as a JPEG file...
p.saveAsJPG("myPlot.jpg")
# -...or an EPS file
p.saveAsEPS("myPlot.eps")
```

**Example 3.6. Decorating a plot.**

Note that we changed the name of both layers in the second and fifth line of the script. Changing the name also changes the legend displayed on the plot.

For the `exp+noise` layer we put an annotation at a specific point (layer coordinates) in the plot. Please check the detailed package documentation of the `Annotation` class for methods to change the font, the size and other properties of an annotation.

For the `exp` layer we have changed the appearance of the datapoints to a line with triangles on top of it. Please refer to Section 3.3.2 for information on basic manipulation methods for layers.
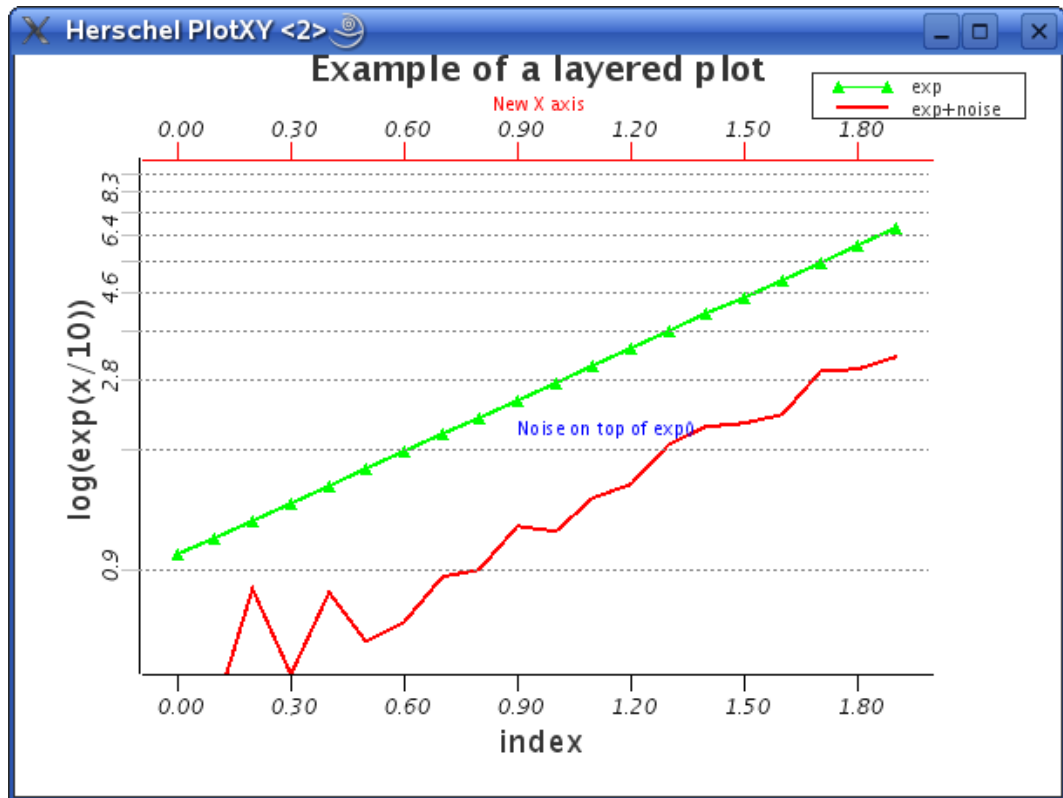
**Figure 3.13. The plot has been annotated and decorated.**

# 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)
```
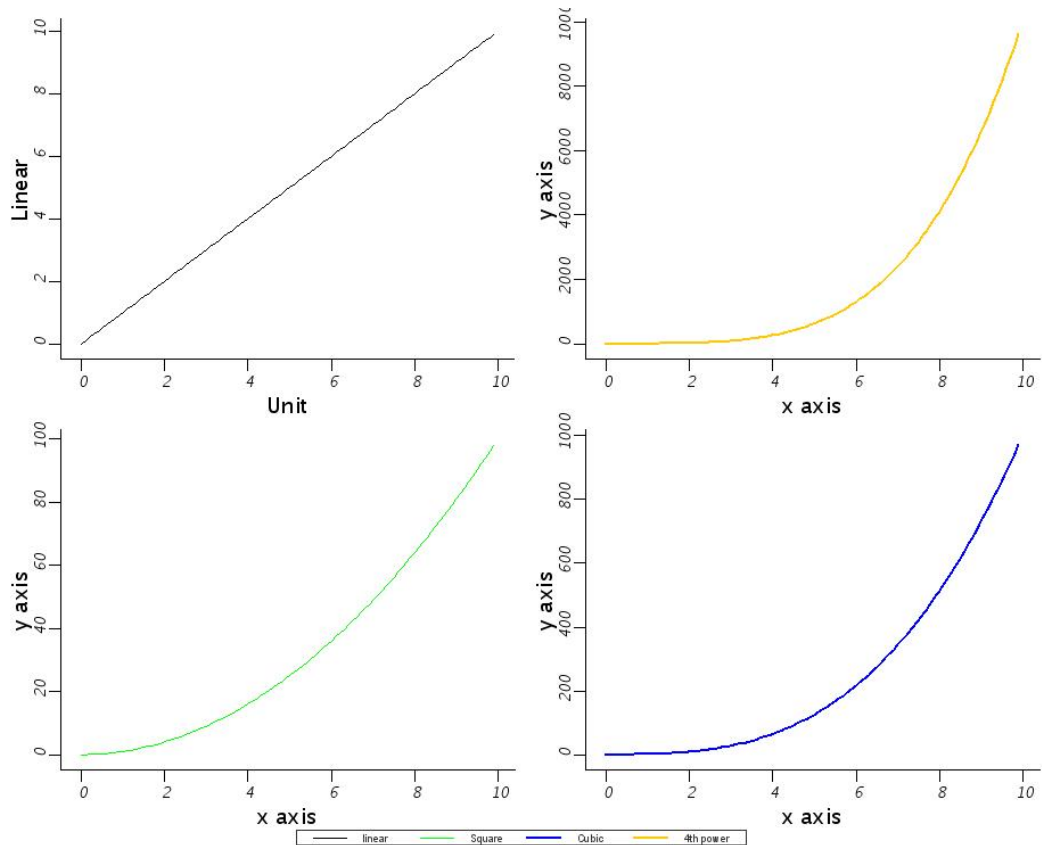
**Figure 3.14. Example of multiple plots in a window from Example 3.7.**

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 2.2.2.

## 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.
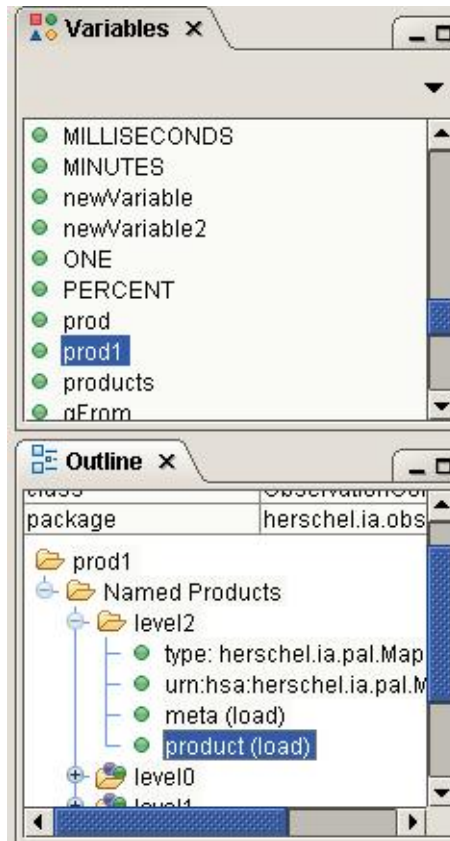
**Figure 4.1. An `ObservationContext` called "prod1" has been obtained from the HSA. Clicking on the folders it contains in the "Outline" window allows us to get at the Level 2 product -- the final pipeline output for this observation.Contents of an `ObservationContext`**
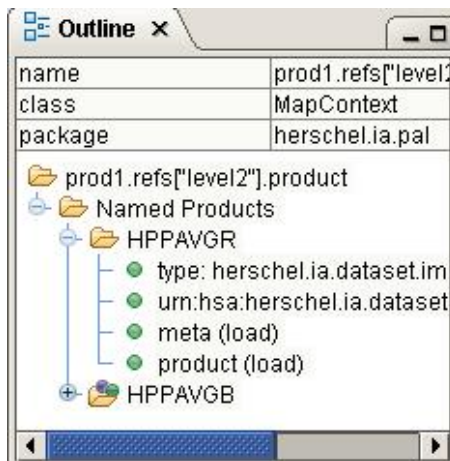


**Figure 4.2. A double-click on the product highlighted in blue in Figure 4.1 provides this Outline view. A double-click on the product highlighted displays the image from the green channel of this PACS observation.PACS green channel image access**
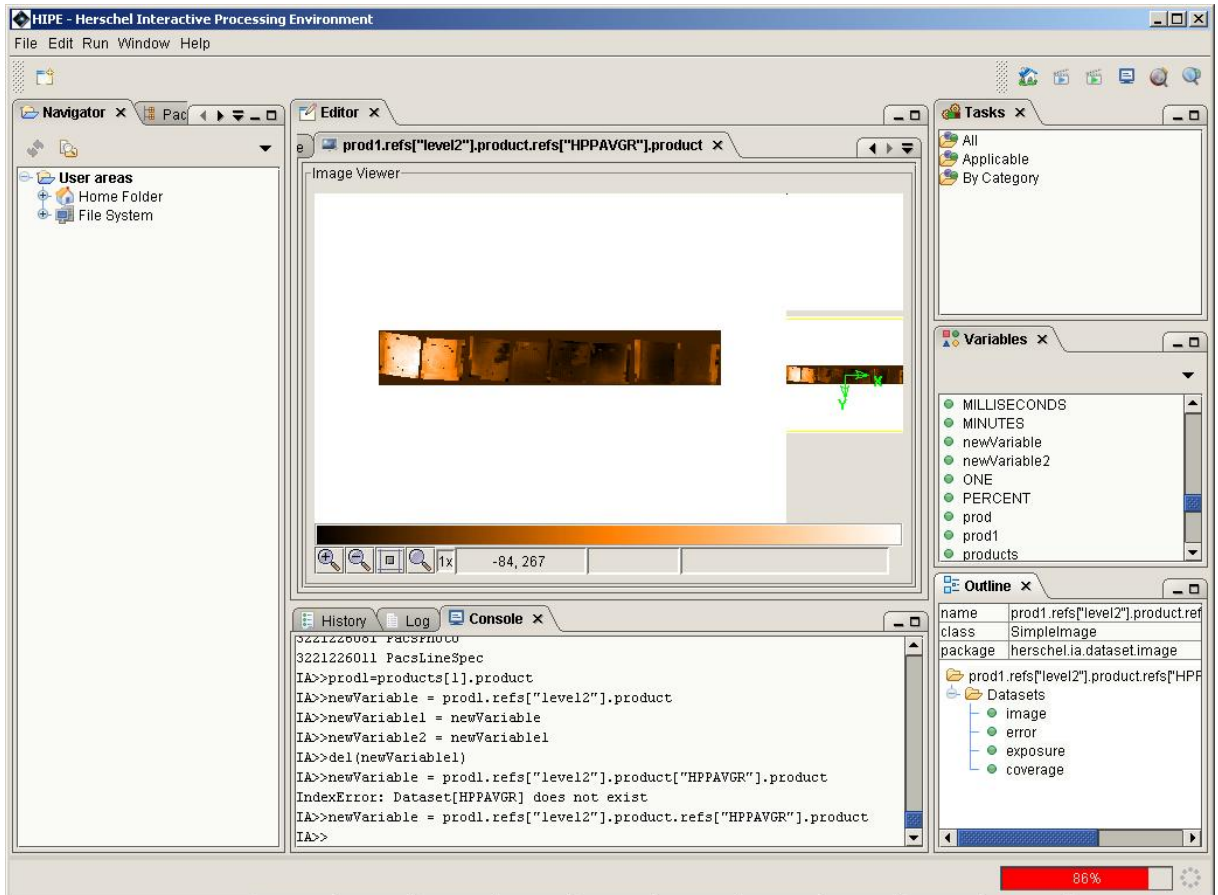
**Figure 4.3. The PACS green channel image displayed in the full work bench of HIPE.**

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. Applying 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. This will bring up a dialog for the task.

Dialogs work in a similar fashion for all image transformations. You can set parameters via pull-down menus or text boxes. Clicking on Accept runs the task.
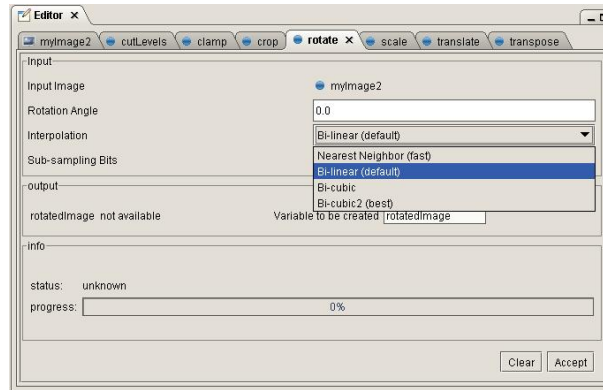
**Figure 4.4. Example image transformation dialog. Rotating an image using the "rotate" task. Several interpolation options are available.**

- **Clamping:** this allows the floor and ceiling of an image to be set. Values above the max or below the min input by the user are set to the max and min values assigned by the user respectively.

- **Cropping:** a section of the image to be extracted to another SimpleImage. The range of X and Y pixel coordinate values are input by the user.

- **Rotating:** when rotating the image, several types of interpolation are possible. By default, bi-linear interpolation is used. There are four types of interpolation possible.

  - **Bi-linear [default]** -- *the default* - interpolates one pixel to the right and one below.

  - **Nearest neighbour [fast]** -- direct pixel copying, the fastest.

  - **Bi-cubic** -- uses interpolation via a piecewise bi-cubic polynomial.

  - **Bi-cubic2 [slow]** -- variant of bicubic interpolation that can give sharper results than bicubic.

- **Scaling:** allows for different magnification in X and Y pixel directions. Possible 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") can be input by the user.

- **Transposing:** allows for different simple transpositions of images. The following transpositions can be done with this task:

  - 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/Divide/Multiply/Subtract (imageAdd, imageDivide, imageMultiply, imageSubtract). This allows either a scalar or a second image as the amount to be added/divided/multiplied/subtracted. The second image can be input into the dialog by click-and-dragging of it from the "Variables" view to the orange dot position in the dialog for the second image (see Figure 4.5) For images, the combination is by pixels or WCS reference.

- 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 (imageRound).

- Square and square root of the image (imageSquare, imageSqrt).

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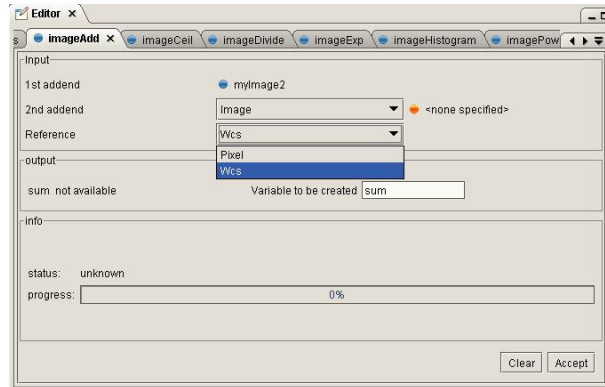


**Figure 4.5. Example image arithmetic dialog.**

# 4.2.4. Working with the World Coordinates System (WCS)

The WCS information for an image is stored in its metadata which can be viewed using the Dataset viewer.

The Wcs class enables the user to define a transformation between the pixel coordinates and world coordinates. The following illustrates how we can type in (at a command-line) a WCS. We create our Wcs() object which we then add to a fake `SimpleImage` we set up to start with.

```
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 section *A note on array ordering* in the *User's Manual*.

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.

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.

# 4.2.5. 1D Profile Plotting

The 1D profile plotting capability allows the user to draw a straight line on an image and plot the intensity along that straight line.

After double-clicking on the Variable "myImage2" in the previous section the image was displayed in an "Editor" window. The Applicable Tasks are also available including the task `profile` (see Figure 4.6). A double-click on this item in the tasks list brings up another display of the image and allows interaction with the mouse
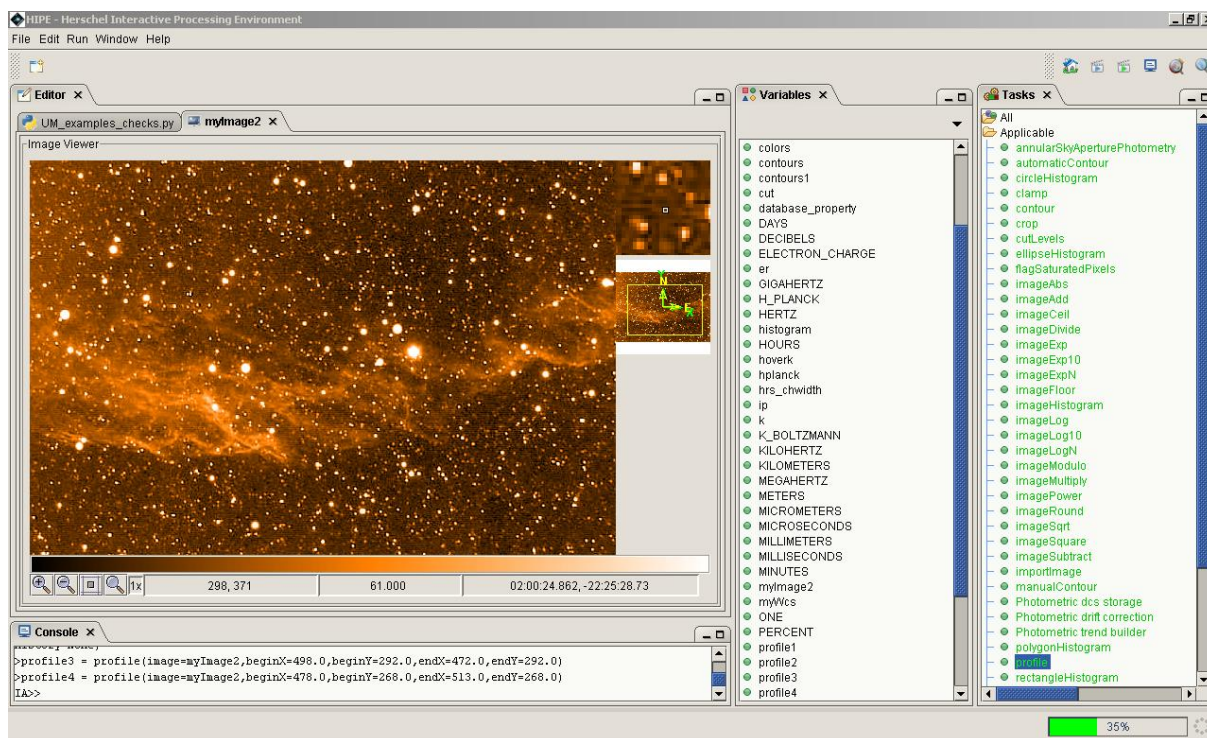
**Figure 4.6. The available tasks show `profile` is available for "myImage2". Double-clicking on profile after first highlighting myImage2 in the Variables window creates a new display of the image together with the `profile` tool capabilities.Accessing the `profile` task**

Now we can start drawing the straight line on the active image. The beginning of the line can be fixed by clicking once on the image. While moving the mouse over the active iamge, the straight line will be updated, until the end of the straight line is fixed by clicking a second time on the image. Simultaneously, the intensity plot along the straight line in an extension of the window below the displayed image (see Figure 4.7). The window is scrollable so the whole profile display can be seen by scrolling down (see Figure 4.8).
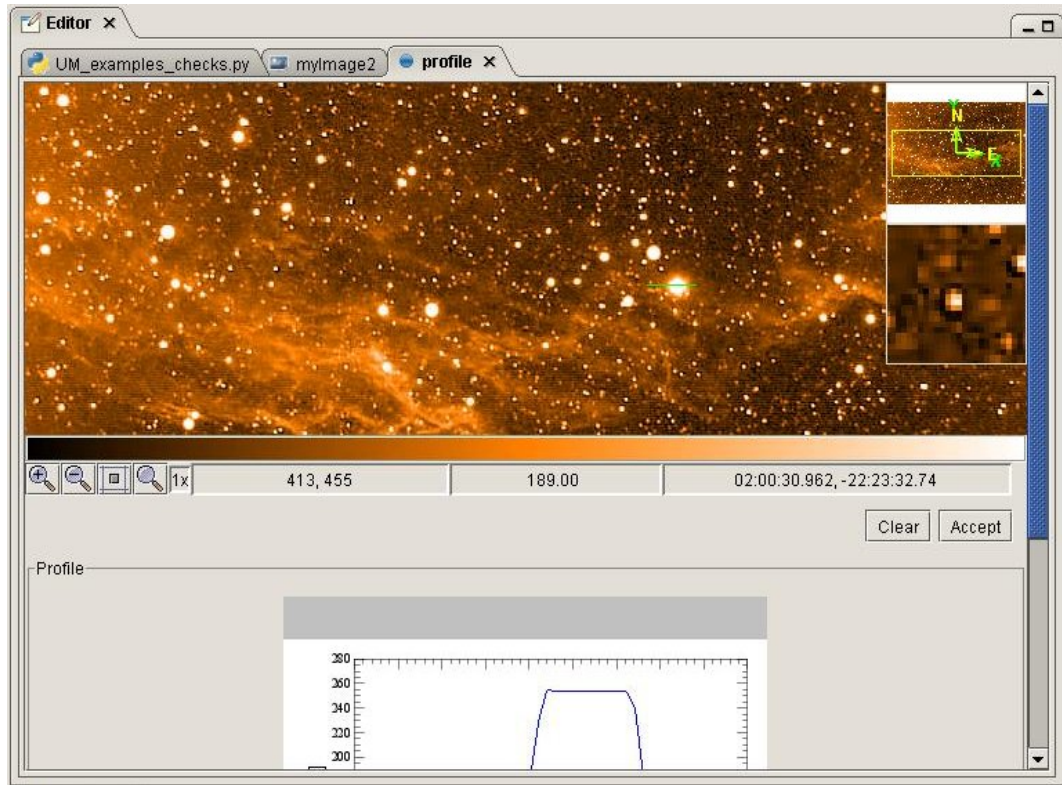
**Figure 4.7. A 1D profile plot interaction**
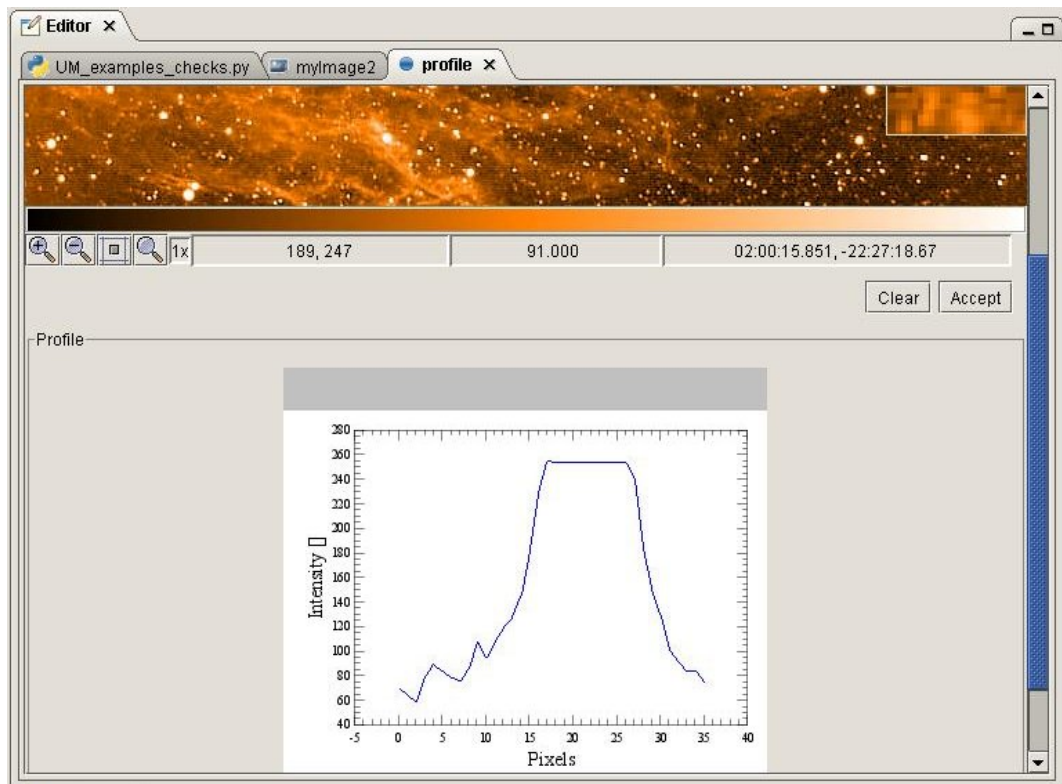


**Figure 4.8. Same as for Figure 4.7 but scrolling down to show the profile display.**

# 4.2.6. Area Histogram

One can make a histogram of an image as a whole, or of a certain region of interest which is specified by the user. This region can be bounded by a CIRCLE, an ELLIPSE, a RECTANGLE or a POLYGON, which has to be drawn on the image, or can be for the whole image.

We start the procedure for making an area histogram by choosing one of the `imageHistogram`, `polygonHistogram`, `circleHistogram`, `ellipseHistogram` or `rectangleHistogram` tasks.

First click on a `SimpleImage` in the list of Variables, e.g. "myImage2". Then double-click one of the histogram tasks in the Tasks view. This brings up a new image and activates the mouse so that an image area can be selected. To cover an area with a circle, ellipse or rectangle do a click-and-drag. On release, the area selected is shown overlaid on the image.

The histogram is constructed from the intensity values of the selected pixels and the input of the min and max cut levels in the boxes provoided, plus the number of bins for the histogram. Hitting the "Accept" button does several things.

- A histogram is formulated in the Editor window (scroll down).

- The equivalent command line is shown in the Console view which includes a named output object.

- The histogram values are placed in a dataset that appears in the Variables list. In Figure 4.9 this is called "histogram2".

- The output (e.g., "histogram2") appears highlighted in the Variables view and appears in the Outline view. Double-clicking this output value in the Variables view provides the histogram together with key information (see Figure 4.10).

For the `polygonHistogram` the only difference is that each corner is indicated by a single mouse click. The polygon area completion is indicated by a mouse double-click. Otherwise, this works in the same way as the other histogram tasks.
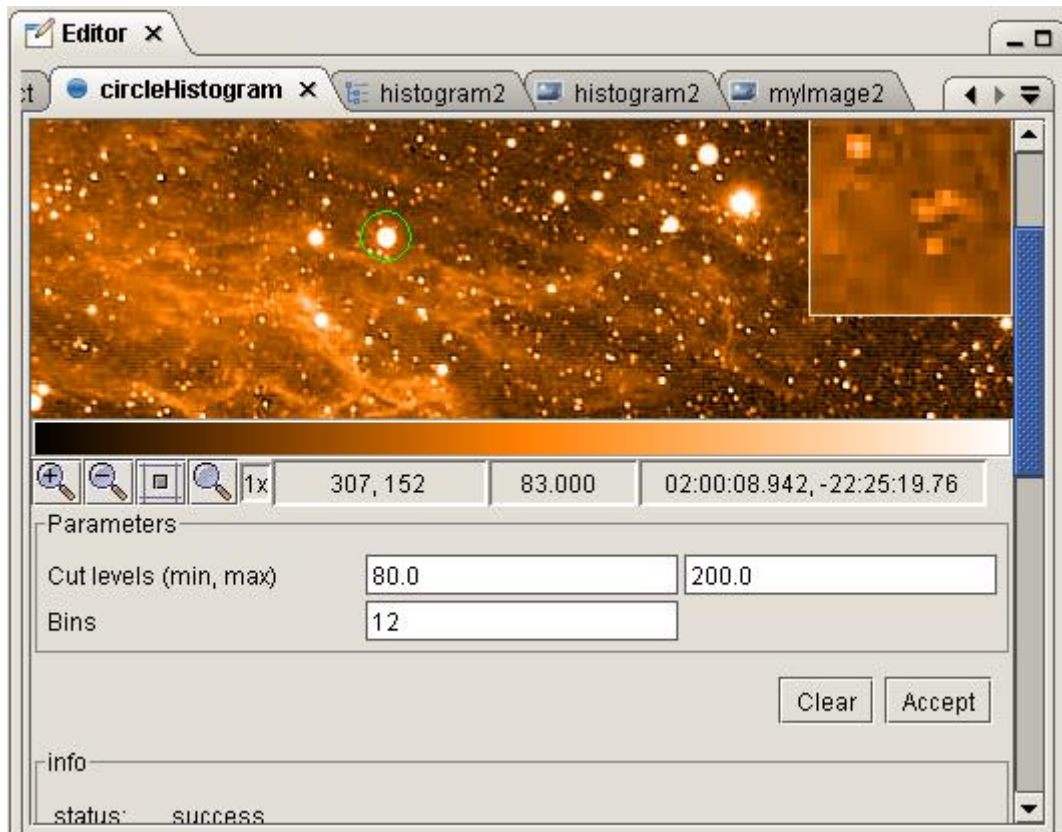
**Figure 4.9. Circle histogram area selection and parameter selection. These appear in the HIPE "Editor" view.Circle histogram**
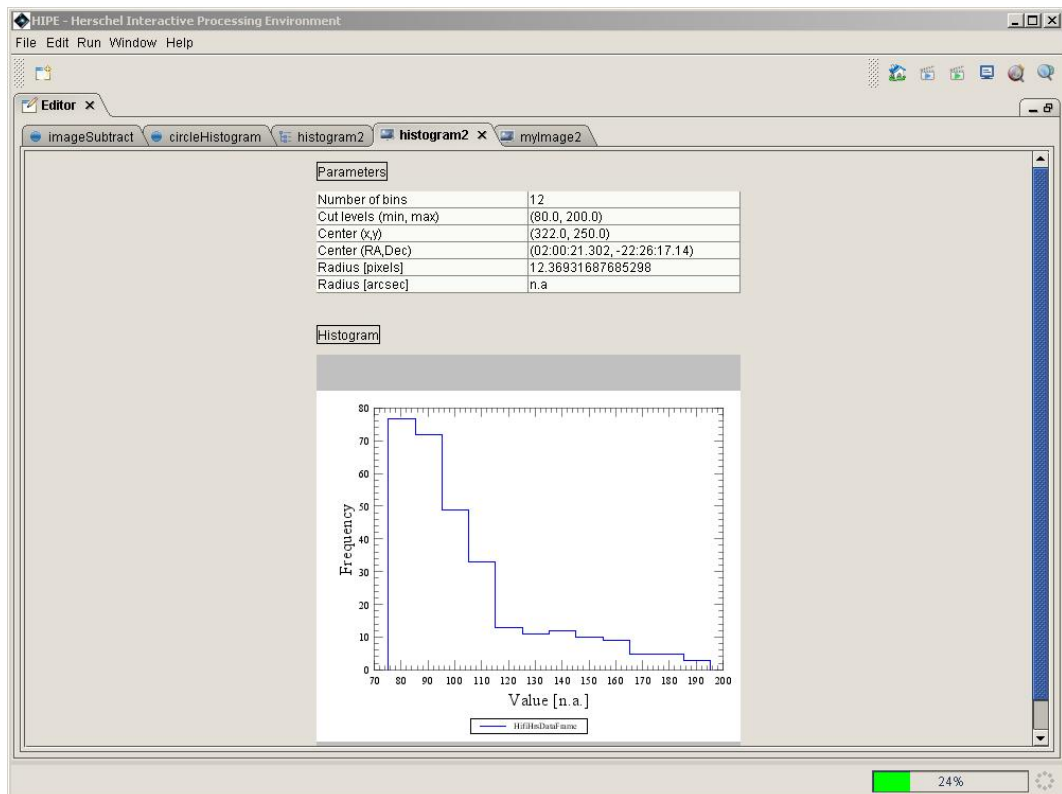


**Figure 4.10. Display of the histogram results held in the histogram output in an expanded Editor view.Histogram display**

# 4.2.7. Aperture Photometry

One can also perform aperture photometry on an image, using a circular target aperture and an annular or a rectangular sky aperture. There are five algorithms that can be used to estimate the sky : average, median, mean-median, the synthetic mode and daophot. In the mean-median method all values further away from the median than a specified number of times the standard deviation (i.c. 1.5) are discarded and the remaining values are averaged. The `daophot` method is a translation of the algorithm used in the aophot package from IDL to Java.

A start to doing annular aperture photometry can be made by choosing the `annularSkyAperturePhotometry` item in the Tasks menu following selection of the appropriate image in the "Variables" menu. This provides the image in the "Editor" view below which is the dialog for the aperture photometry task options. This is easiest seen by expanding the "Editor" view window and scrolling down below the image.

There are three mechanisms by which the photometry area can be identified.

• By click-and-drag mouse interactions.

• By pixel region selection.

• By sky coordinate selection.

The default is by mouse interaction. A single click on the image allows places a circle on the image at the mouse point. The user then inputs a value for the object aperture radius and inner and outer radii for sky subtraction. A selection should be made for the appropriate fitter (e.g. daophot) and pressing accept leads to an output in the variable "result". The circular radii are shown on the image (see Figure 4.11). To redo -- press the "Clear" button.

A sky position or pixel position can also be selected. Selection of either of these possibilities enables an update to the input screen allowing the sky/pixel values to be input. For the sky position (at present) the format of input is "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. This is likely to be changed to allow various input types in the future.

The results can be displayed by double-clicking on the "result" variable shown in the "Variables" menu (see Figure 4.12)
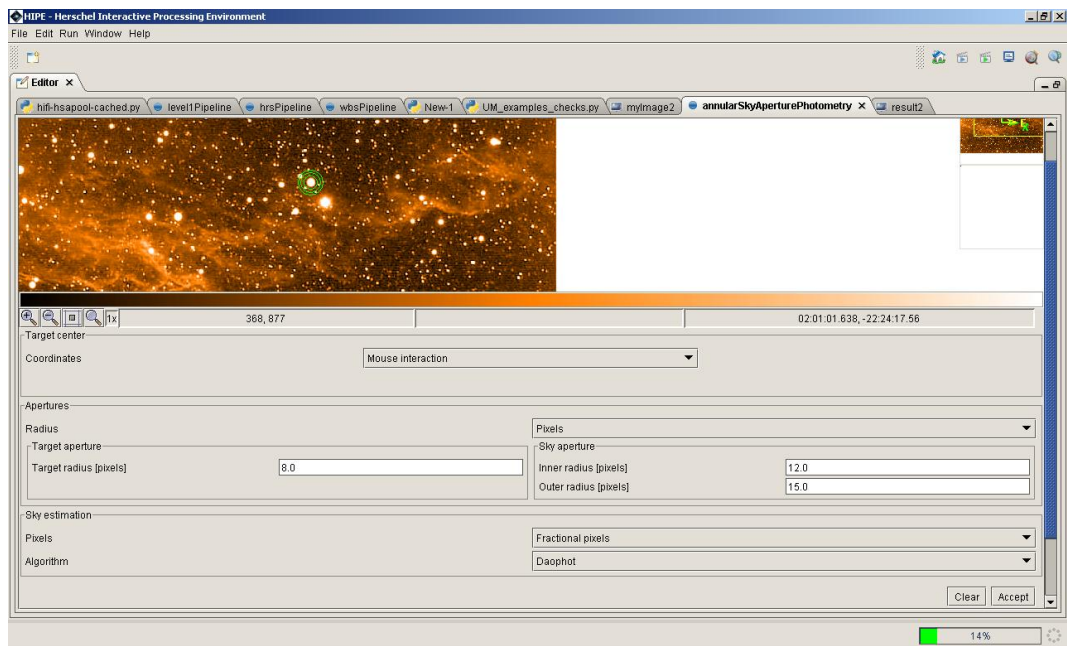


**Figure 4.11. Aperture photometry with an annular sky aperture as displayed in HIPE.**
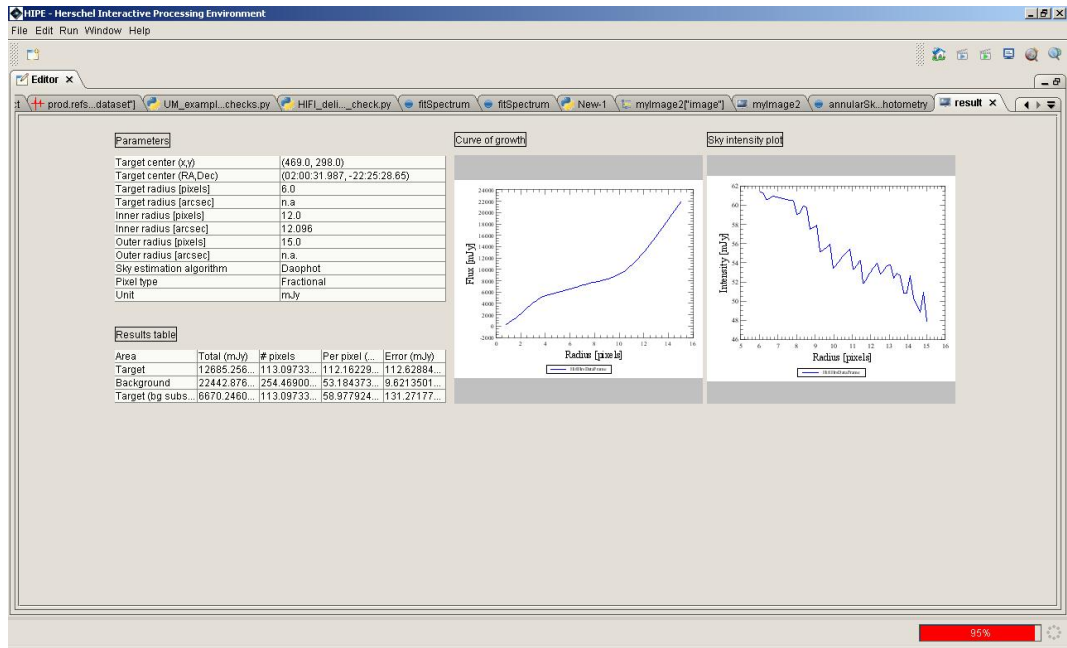
**Figure 4.12. Aperture photometry results plot and tables. Note that n.a. relates to "not applicable" and typically will occur when units are not assigned to the image.Results of sky aperture measurement.**

A similar capability is available for using a rectangular sky aperture. Rectangular aperture photometry can be done by choosing the `rectangularSkyAperturePhotometry` item in the Tasks menu following selection of the appropriate image in the "Variables" menu. Similar to the above, a single mouse click can be used to identify the target or a sky or pixel position can be indicated by the user. A rectangular sky aperture can then be selected by a click-and-drag across a region of the image (see Figure 4.13). Following the calculation for the first position, the same rectangular box can be used for the sky and a further single click on the image picks out a new object. Hitting the "Accept" button allows another result for this new position.
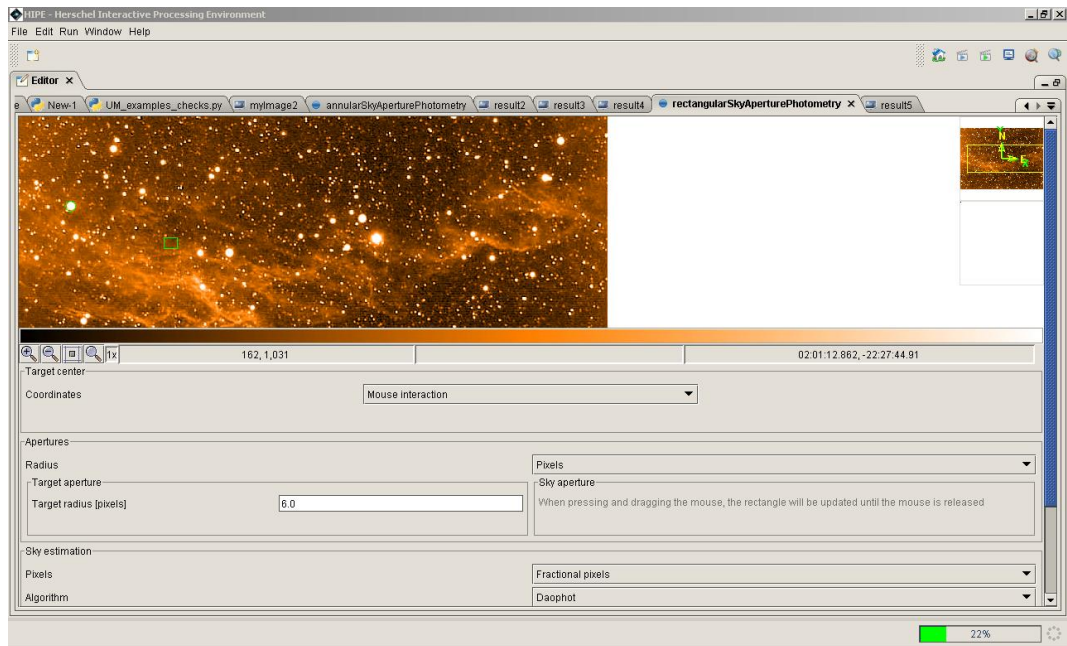


**Figure 4.13. Aperture photometry with an annular sky aperture as displayed in HIPE.**

The results for both aperture photometry tasks provide the curve of growth. This is a plot of the target flux as a function of the target radius. Such a plot can be used to see whether a valid target radius has

been given. When an annular aperture is used to estimate the sky, a sky intensity plot is also shown. This plot shows the intensity per sky pixel as a function of a varying inner radius (the outer radius is fixed).

# 4.2.8. How to run the source extractor in HIPE

## 4.2.8.1. Introduction to source extraction in HIPE

HIPE includes a source extractor ("sourceExtractor"), designed primarily for use on PACS and SPIRE maps. It currently features implementations of the DAOPHOT (classic) and SUSSEXtractor algorithms for extraction of point sources with a known profile. This chapter explains how to use the source extractor; advanced usage is described in the User's Reference Manual entry for the SourceExtractorTask.

## 4.2.8.2. Extracting sources from a SimpleImage

The example image used in this section is a simulation of a SPIRE 250 micron image of a grid of sources. This is imported into HIPE, where it appears as a Simple Image. In the Work Bench perspective, in the Tasks view, under "Applicable", "sourceExtractor" appears as one of the applicable tasks for a SimpleImage. This is shown in Figure 16.13.
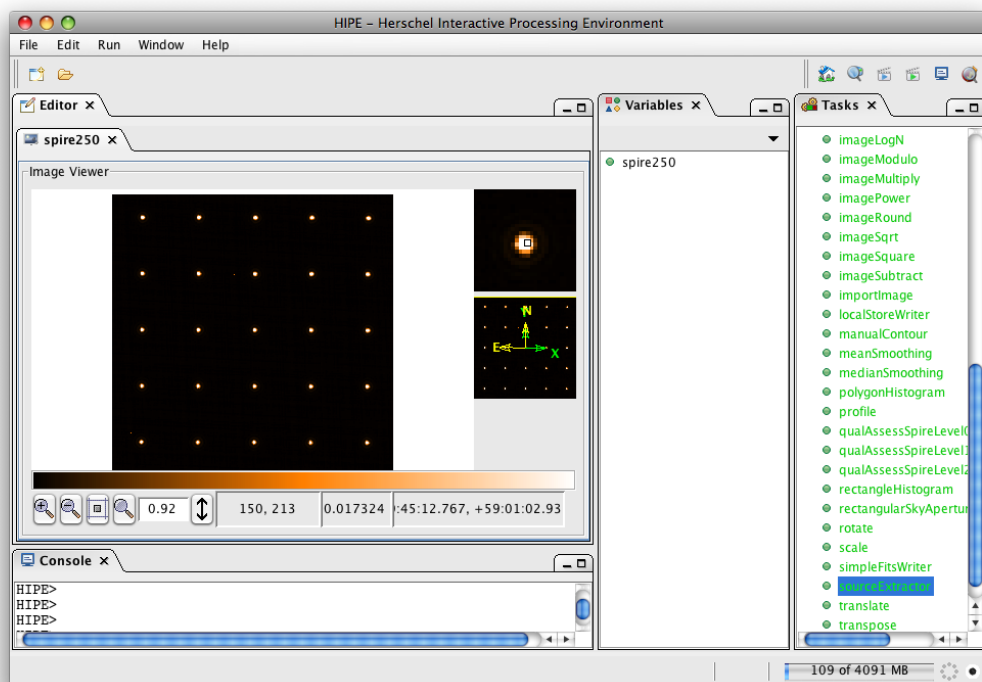


**Figure 4.14.    The Work Bench perspective, showing the example image, with "sourceExtractor" highlighted in the list of applicable tasks.**

Experienced users are likely to use the sourceExtractor task as part of a Jython script. However, the easiest way to learn how to use the task is through the graphical user interface (GUI). Double-clicking on the "sourceExtractor" in the applicable tasks list will open the GUI for the source extractor. This is shown in Figure 16.14.
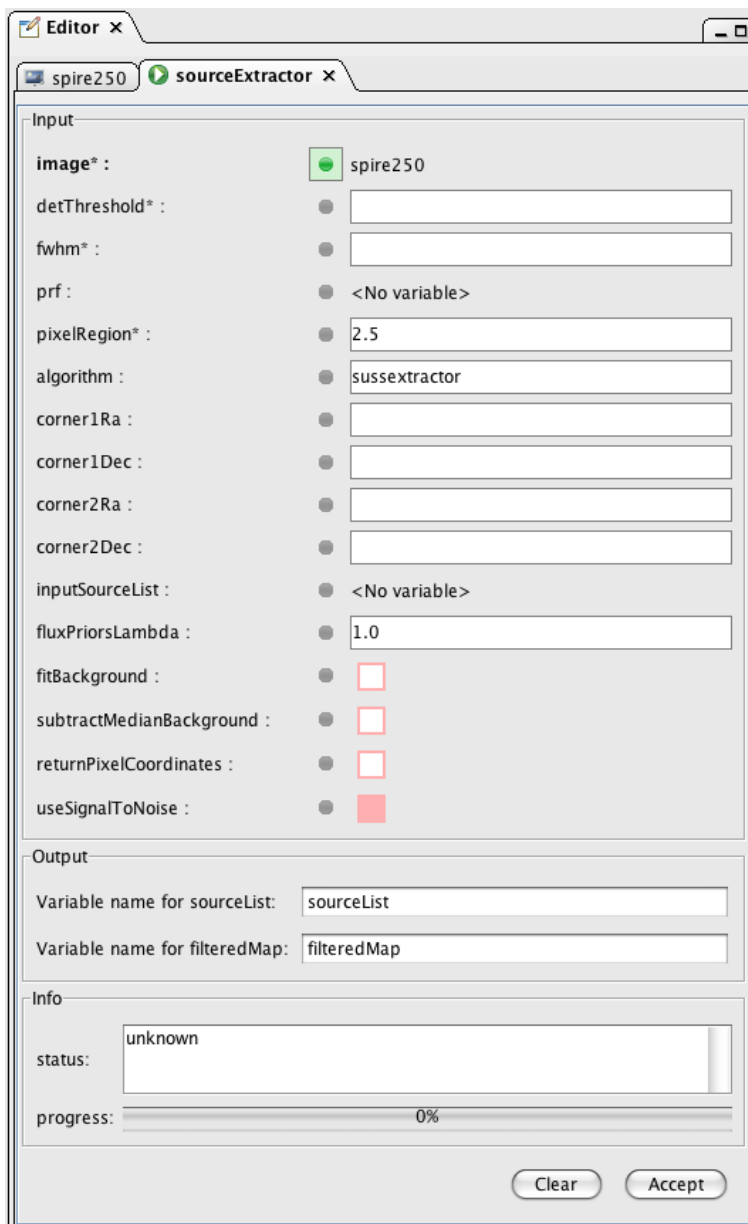
**Figure 4.15. The GUI for the sourceExtractor task, displaying the default values of the parameters. If the task is launched from the list of applicable tasks for a SimpleImage, that SimpleImage will appear as the "Image" parameter of the task, as in the Figure. Otherwise, a SimpleImage may be dragged and dropped from the Variables view.**

In order to launch the task, the mandatory parameters must be filled in. These are marked with an asterisk (*) in the GUI. If the mouse pointer is placed over the name of a task parameter, a box will appear giving information about that parameter. In this example, the parameters still to be filled in are the detection threshold ("detThreshold") and the full-width-half-maximum (FWHM) of the (Gaussian) point response function ("fwhm"). In this example, these are set to be 5.0 for the detection threshold (corresponding to a 5-sigma detection) and to 18.0 arcsec for the FWHM, corresponding to the approximate size of the SPIRE beam for the PSW array. Clicking "Accept" launches the task. The status and progress may be followed in the task GUI. Long-running tasks may be interrupted by selecting "Stop" from the "Run" menu. Figure 16.15 shows the task GUI once the task has been completed.
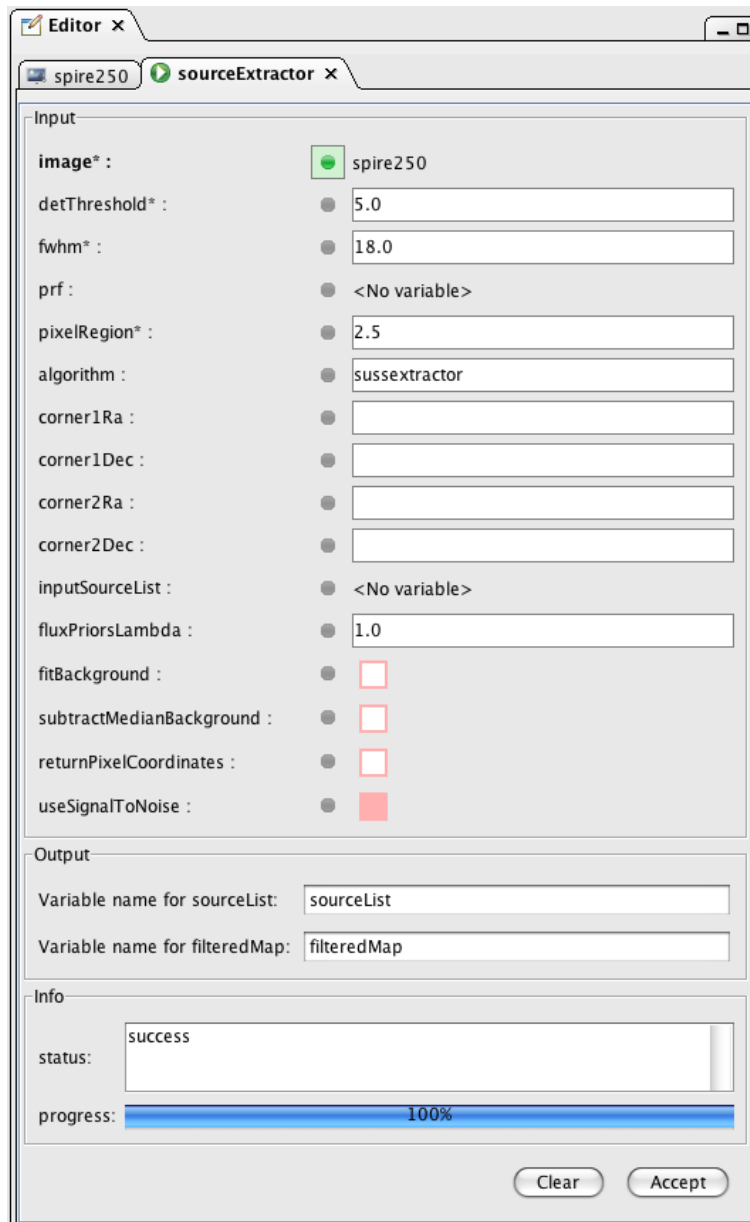
**Figure 4.16.  As Figure 16.14, but for the completed source extraction task.**

The task GUI is essentially a convenient method of created a Jython command to execute the task. Once the task has been completed, the Console view displays the Jython commands executed to perform the sourceExtractor task. This shown in Figure 16.16.
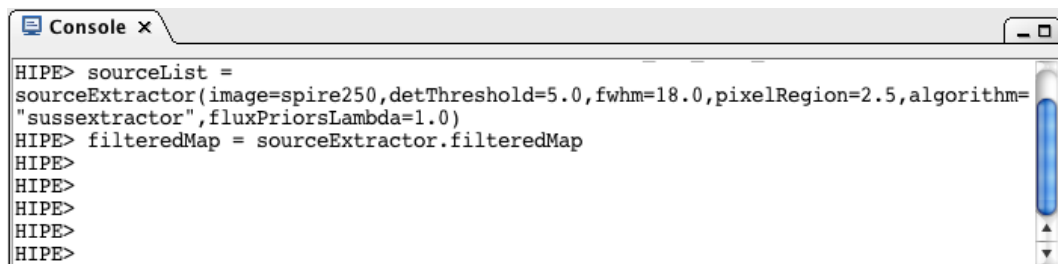


**Figure 4.17. The Console view after the sourceExtractor task has been launched from the task GUI.**

In order to repeat the source extraction as part of a Jython script, the user need simply copy these commands into the script, rather than entering parameters manually into the GUI. Two commands

were executed by the task. (1) The first created the primary output, a SourceListProduct, given the name "sourceList". This will appear in the Variables view. Double- clicking on "sourceList" opens the SourceListProduct in the Product Viewer. This is shown in Figure 16.17, with the Dataset highlighted. The Dataset in a SourceListProduct is a "SourceListDataset", which is a kind of TableDataset.



**Figure 4.18. The SourceListProduct shown in the Product Viewer, with the dataset highlighted.**

(2) The second command obtains an optional output from the sourceExtractor task, the "filteredMap", which is another SimpleImage. For the SUSSEXtractor algorithm, this 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 filteredMap gives the input map convolved with the DAOPHOT kernel. This may be viewed by double-clicking on "filteredMap" in the Variables view.

## 4.2.8.3. Displaying extracted sources on an image

Having extracted the sources in an image, the user is likely to want to see where the extracted sources lie in the image. This can easily be done by displaying the original SimpleImage in the Editor view, and then dragging and dropping the SourceListProduct from the Variables view onto the displayed image. A circle is now overlaid at the location of each source. The result of this is shown in Figure 16.18.
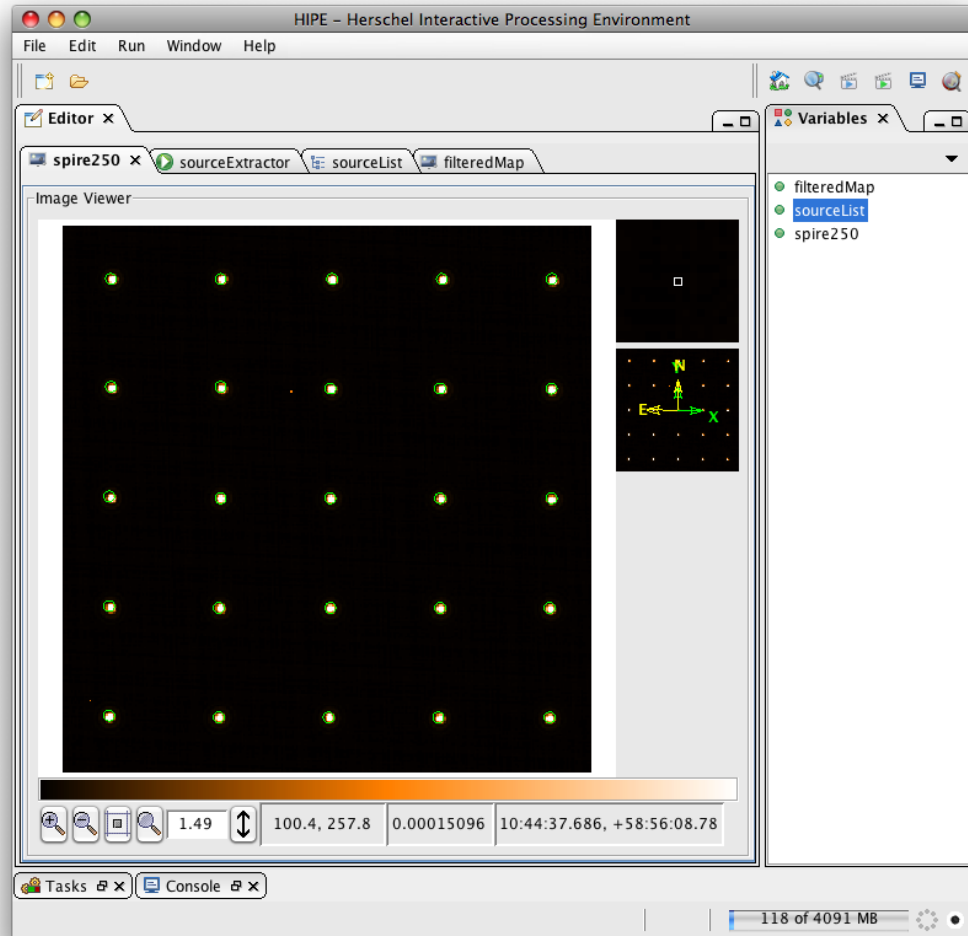
**Figure 4.19. The original image with the locations of the extracted sources overlaid as circles.**

## 4.2.8.4. Specifying positions of known sources

Often the user will be interested in the fluxes of sources at known positions, e.g., from previous observations made at mid-infrared wavelengths. If these sources are contained in a SourceListProduct, this can be used as an input to the sourceExtractor task to specify the positions of sources. The task will then give the fluxes of sources at those positions. This is shown in Figure 16.19, where the known sources are provided by another SourceListProduct, "knownSourceList". This is specified as the "inputSourceList" parameter for the task.
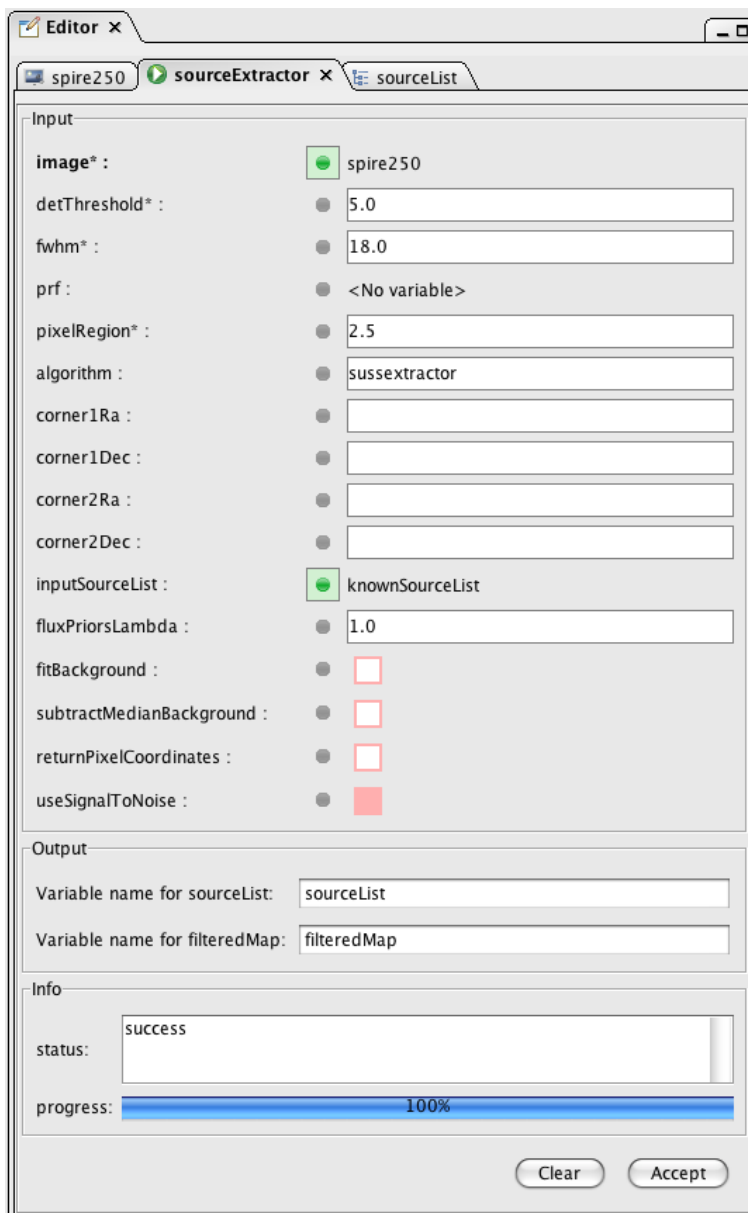
**Figure 4.20. The sourceExtractor task GUI, with "knownSourceList" specified as the "inputSourceList" parameter.**

## 4.2.8.5. Specifying a custom point response function

By default, the sourceExtractor task assumes a Gaussian point response function (PRF), with full-width-half-maximum (in arcsec) provided by the "fwhm" parameter. Alternatively, a custom PRF may be specified by the "prf" parameter. This should be a SimpleImage of odd dimension, with the peak at the centre, normalised such that the PRF image gives the (central pixels) of a point source of flux 1 Jy, in the units of the input map.

## 4.2.8.6. Working with source lists in ASCII files

One may wish to work with external ASCII files, either to export Herschel source lists, or to import other source lists, either to provide known positions for the HIPE source extractor, or for use in other analyses. This may easily be done using the asciiTableWriter and asciiTableReader tasks. To save a Herschel source list as an ASCII file, first the SourceListDataset must be retrieved from the SourceListProduct. A SourceListDataset is a kind of TableDataset, and consequently the asciiTableWriter will appear in the applicable tasks list. This is shown in Figure 16.20.
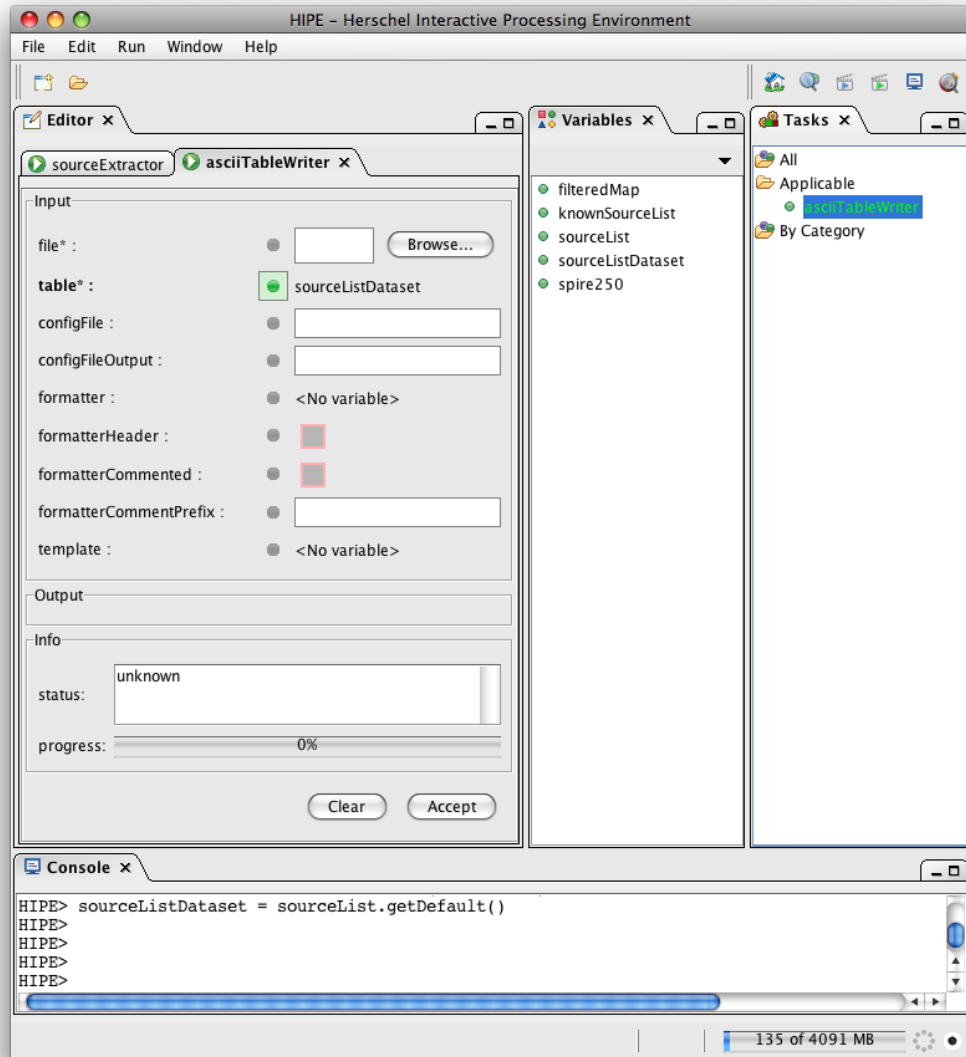
**Figure 4.21. Exporting a SourceListProduct as an ASCII file. First, the SourceListDataset is retrieved using the command "sourceListDataset = sourceList.getDefault". Then "sourceListDataset" is highlighted in the Variables view. In the Tasks view, "asciiTableWriter" now appears as an applicable task. Double-clicking this opens the GUI for the asciiTableWriter, as shown in the Figure.**

To import an ASCII file as a SourceListProduct, first the asciiTableReader should be launched. This is shown in Figure 16.21, for a source list contained in "sourceList.txt".

This creates a TableDataset, "table". This may be converted into a SourceListProduct called "importedSourceList" using the command "importedSourceList = SourceListProduct(table)". It is important to note that the column names in the imported source list must match the default column names in a SourceListDataset ("ra", "dec", "flux", etc.), although whether these names are upper, lower or mixed case does not matter.
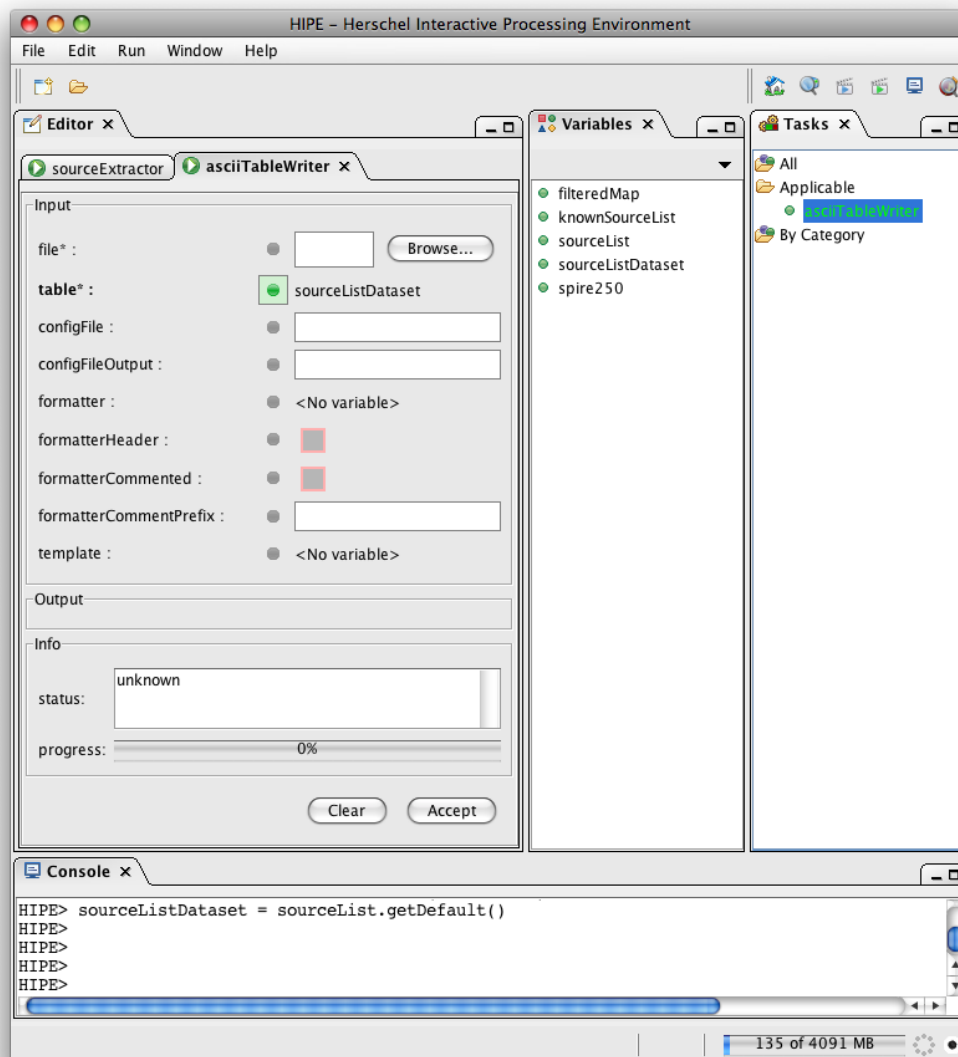
Figure 4.22. The asciiTableReader task for importing sourceList.txt.

# 4.2.8.7. Working with source lists in FITS files A SourceListProduct may be exported or imported as a FITS file.

This may be useful for working with applications such at TOPCAT. To export a SourceListProduct as a FITS file, simply select the name of the SourceListProduct in the Variables view, and the "simpleFitsWriter" task should appear in the list of applicable tasks. To import a FITS file as a SourceListProduct, simple select "Open" from the "File" menu, and select the FITS file. This launches the simpleFitsReader, and the FITS file will be imported as a Herschel Product. If the FITS file contained an exported Herschel SourceListProduct, then the imported Product will be of this type. Otherwise, the imported Product will be a generic Product, with the source list contained in a Dataset. For example, if the Dataset has the name "HDU_1", then a SourceListProduct may be created using the command

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

This is shown in Figure 16.22.
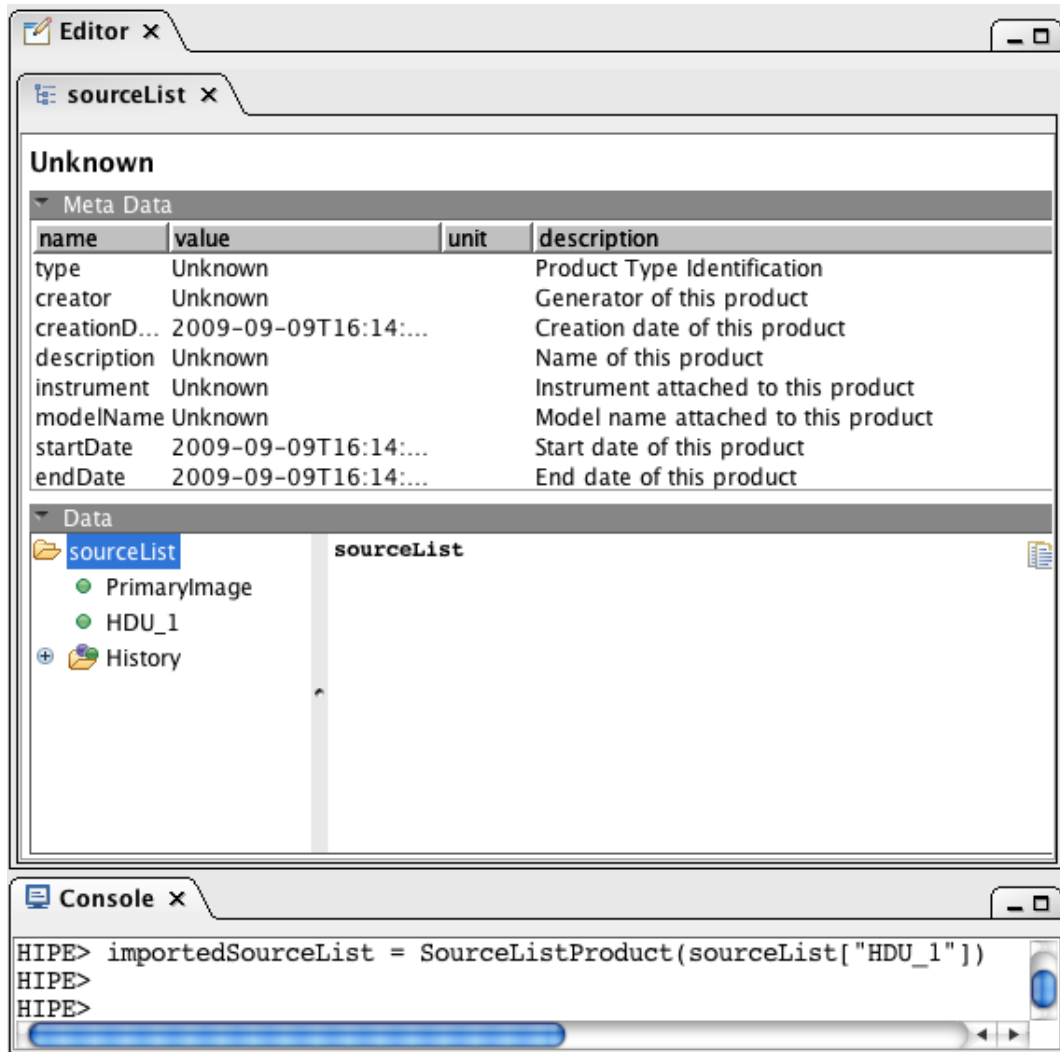
**Figure 4.23. The asciiTableReader task for importing sourceList.txt.**

# 4.2.9. Contour Plotting

Another functionality of the toolbox is contour plotting. A contour plot connects all points in the image with the same intensity, like isobars on a weather map.

First we create the contours. We then, later, overlay these contours on any image we wish.

There are two methods for providing a set of contours for display. The first is an `automaticContour` where the number of contour levels and a min and max contour level are selected and the intermediate levels are generated automatically with linear, ln or log intervals of intensity. The second is a `manualContour` where the values of each contour level are individually put in by the user.

In either case we, as usual, start by clicking the name of the image we want to be countoured from the "Variables" list. Then we choose either `automaticContour` or `manualContour` by double-clicking these items in the "Tasks" list (see Figure 4.24for example).
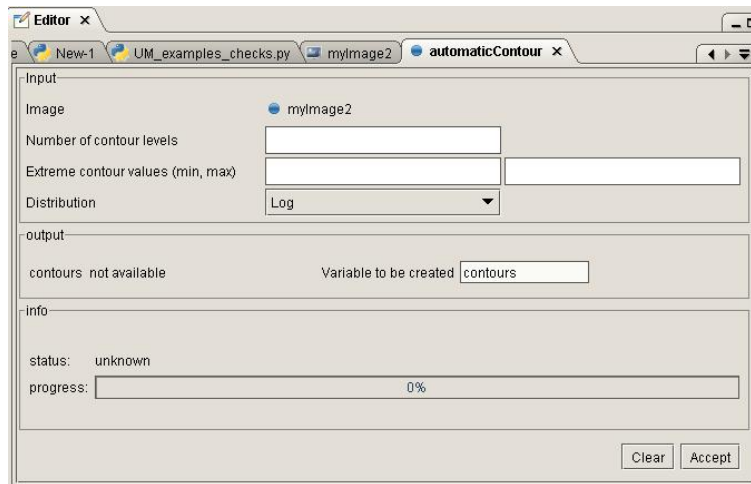
**Figure 4.24. Dialog for `automaticContour`.**

In either case, we create an output (editable value for user), e.g. "contours". When hitting the "Accept" button this is the variable that will store the contour results.

# 4.3. In depth

A large set of functionalities is available on the command line in HIPE and most of them are also available in a GUI-driven environment. In the past, this environment used to be the `ImageAnalysisToolbox`, but now we're making efforts to integrate everything into HIPE. Thus, in this section, you will find how you can do everything, as integrated into HIPE (if available). Here is a list with all available functionalities :

1. Profile plotting

2. Aperture photometry

3. Histograms

4. Contour plotting

5. Mosaicking

6. Smoothing

7. Clamping/clipping

8. Cropping

9. Rotating

10. Scaling

11. Translating

12. Transposing

13. Flagging saturated pixels

14. Getting the cut levels

15. Image arithmetics

In the following subsections, we will elaborate on all of these functionalities.

# 4.3.1. Profile Plotting

`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. You can do this on the command line in HIPE, or via a HIPE-integrated GUI.

### *On the command line*

The only input parameters are

- the *image* (`Image image`)

- the *begin 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, simply type

```
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`) will appear in the *Variables* view in HIPE.

### *Via the HIPE-integrated GUI*

If you select `myImage2` from the *Variables* view and then click on *profile* in the *Tasks* view, a task dialog opens in the *Editor* in HIPE. In the upper part, the `Image` (`myImage2`) is shown. Here you must click a first time to fix the begin of the straight line. As you move the mouse over the image, you will see a straight line being updated, as well as the corresponding profile plot in the lower part of the task dialog. If you click a second time on the image, the straight line is fixed, and a new variable `profile` appears in the *Variables* view and the corresponding command is echoed to the *Console*.

When you press the *clear* button, the straight line will be removed from the image and the intensity plot in the lower part of the task dialog will be cleared.

You can also drag the straight line across the image and manipulate it from the moment you have fixed it (by clicking a second time). You will see that the profile plot in the lower part of the task dialog will be updated. To get the adapted output product in the *Variables* view however, you must press the *Accept* button again. If you hit this button accidentaly, before finishing drawing the straight line, a popup window will appear to warn you about this. If you click *OK*, you can continue drawing the straight line.
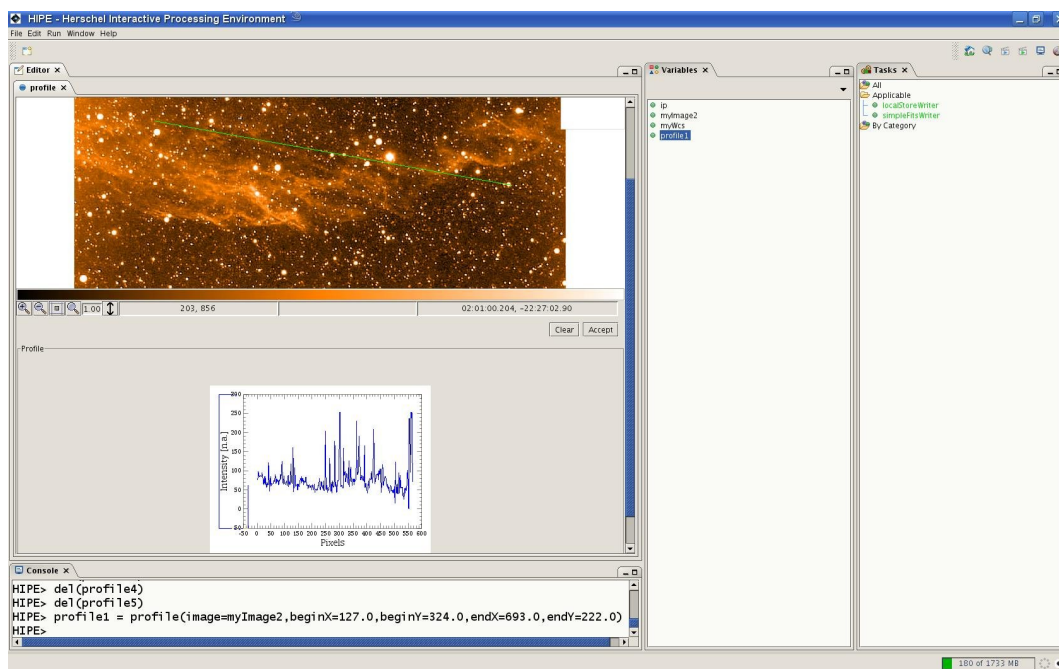
**Figure 4.25. Using the profile task on an image.**

### Inspecting the result

The resulting product, `profile`, can be inspected by double-clicking on it in the *Variables* view. An explorer will be opened in the *Editor*, showing a table with the parameters (begin and end of the straight line in pixel and - if available - sky coordinates) and an intensity plot of the pixels along the corresponding straight line. The data in this product can also be inspected on the command line:

```
# Returns a Double1d with the pixel coordinates of begin and
# end of the straight line
profile.getBeginPixelCoordinates()
profile.getEndSPixelCoordinates()

# 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()
```

This is what the profile explorer looks like :
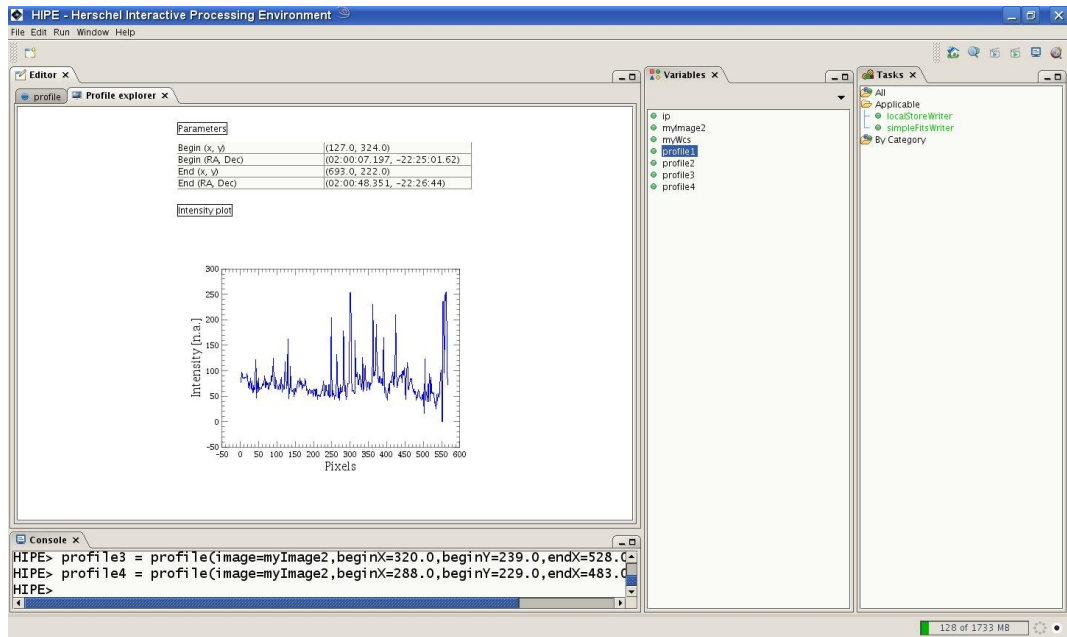
**Figure 4.26. Profile explorer appearance.**

# 4.3.2. Aperture Photometry

Another kind of analysis you might want to do on an `Image` is aperture photometry. This can be done in several 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

In all cases, you can perform the calculations on the command line in HIPE or via a HIPE-integrated GUI.

## 4.3.2.1. Aperture Photometry with a circular Target Aperture and an annular Sky Aperture

If you want to do aperture photometry for a circular target, you can choose to estimate the sky using an annular sky aperture, centered around the target. This can be done using the `AnnularSkyAperturePhotometryTask`.

**On the command line**

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, just type

```
# 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`.

> **Note**
>
> All distances must be specified in the same unit, so either pixels or arcsec.

You have the possibility to choose between five sky estimation algorithms : average, median, mean-median, synthetic mode and the algorithm used by Daophot. Here is how you can choose between these options on the command line :

```
# 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.

**Via the HIPE-integrated GUI**

If you select `myImage2` from the *Variables* view and then click on *annularSkyAperturePhotometry* in the *Tasks* view, a task dialog opens in the *Editor* in HIPE. In the upper part you see the image on which you can mark the target center, if the *Coordinates* combo box is set to *Mouse interaction*. If you click on the `Image`, a circle (with a radius of five pixels) will indicate where you have marked the

target. You can drag this circle over the image if you want it to be elsewhere. If you set this combo box to *Pixel coordinates* or *Sky coordinates*, you must specify the appropriate coordinates in the foreseen fields that will appear.

The next thing you should do, is specifying the target radius and the inner and outer radius of the annular sky aperture in the *Apertures* part of the task dialog. But first you must indicate - using the *Radius* combo box - which unit you're going to use (pixels or arcsec).

In the *Sky estimation* part of the task dialog you can choose either to use fractional or entire pixels to do the calculations, and indicate which sky estimation algorithm you want to use.

If you are happy with all filled out parameters, you must press the *Accept* button in order to start the calculations. If something goes wrong, you are warned with an appropriate error message. For example, if you choose to mark the target center via mouse interaction but forget to click on the image, a popup window will appear.

At this point the apertures will be drawn on the image, the result will appear in the *Variables* view and the corresponding command will be echoed to the *Console*.

Clicking the *Clear* button will make the apertures disappear from the `Image`.



**Figure 4.27. Sky aperture photometry task.**

**Inspecting the result**

The resulting product, `phot`, can be inspected by double-clicking on it in the *Variables* view. An explorer will be opened in the *Editor*, showing a table with all parameters, and a results table with all calculated values (total flux, number of pixels, intensity per pixel and error on the flux for the sky and the target, with and without the sky). This is the normal output, all other software packages yield for aperture photometry.

Besides that, you also have two plots as output :

- a curve of growth

- a sky intensity plot

The curve of growth shows the target flux (without the sky) for a varying target radius, while the sky intensity remains the same. The sky intensity plot shows how the calculated sky intensity changes if the outer radius is fixed and the inner radius varies. Both plots can be used as diagnostics to see whether you have chosen decent values for the radii.

You can also inspect this output product on the command line :

```
# 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()
```

This is how the explorer looks like :
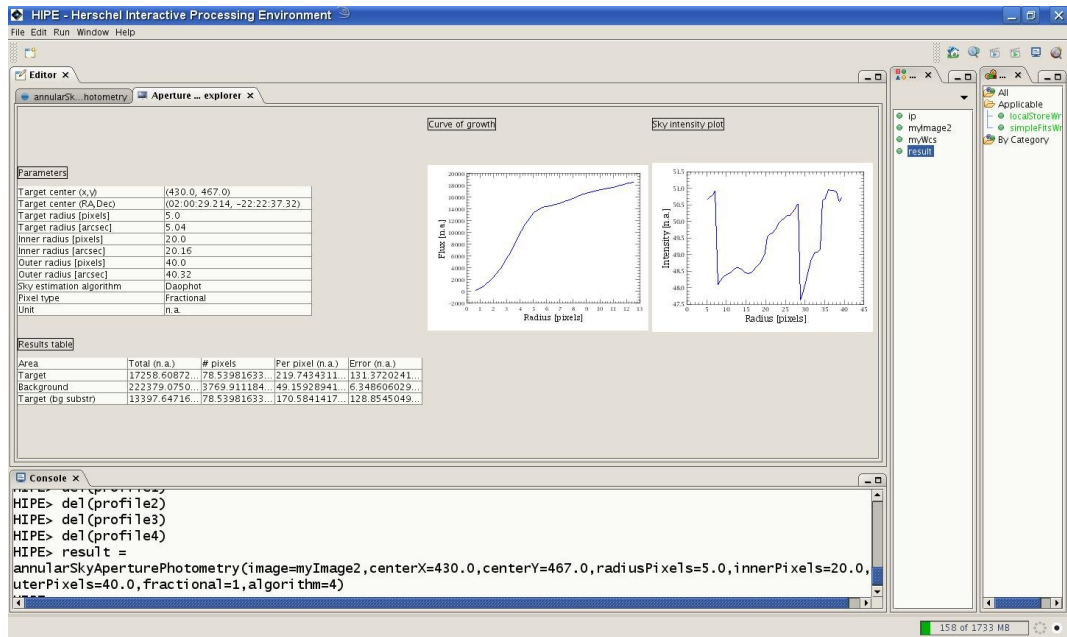
**Figure 4.28. Annular sky aperture measurement.**

# 4.3.2.2. Aperture Photometry with a circular Target Aperture and a 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`.

**On the command line**

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`.

> **Note**
>
> The target center and the corner of the rectangle with minimal row and column must be specified in the same coordinates (pixel/sky).

> **Note**
>
> 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.

### Via the HIPE-integrated GUI

You can do the calculations via a HIPE-integrated GUI, just like for the `AnnularSkyAperturePhotometryTask` (select *rectangularSkyAperture* instead of *annularSkyAperture* in the *Tasks* view). Specifying the target center, the target radius, the kind of pixels and the sky estimation algorithm is done exactly the same way. To draw the rectangle on the `Image`, just press the mouse, drag it across the `Image` and release it when it satisfies you. Afterwards you can move or manipulate this rectangle.

Starting the calculations and clearing the image is done as for the `AnnularSkyAperturePhotometryTask`.
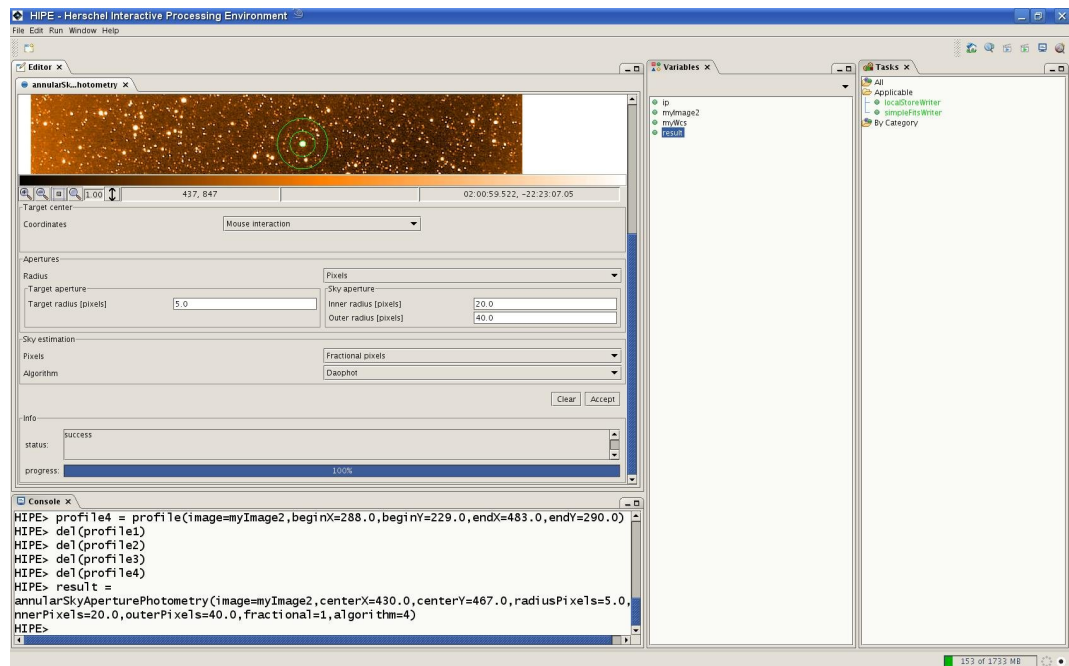


**Figure 4.29. Rectangular aperture photometry task.**

### Inspecting the result

The resulting product, `phot`, can be inspected by double-clicking on it in the *Variables* view. An explorer will be opened with great resemblance to the explorer for the result of the `AnnularSkyAperturePhotometryTask`. Only the sky intensity plot is missing, of course.

To inspect this 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, you can 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()
```

This is what the explorer looks like :
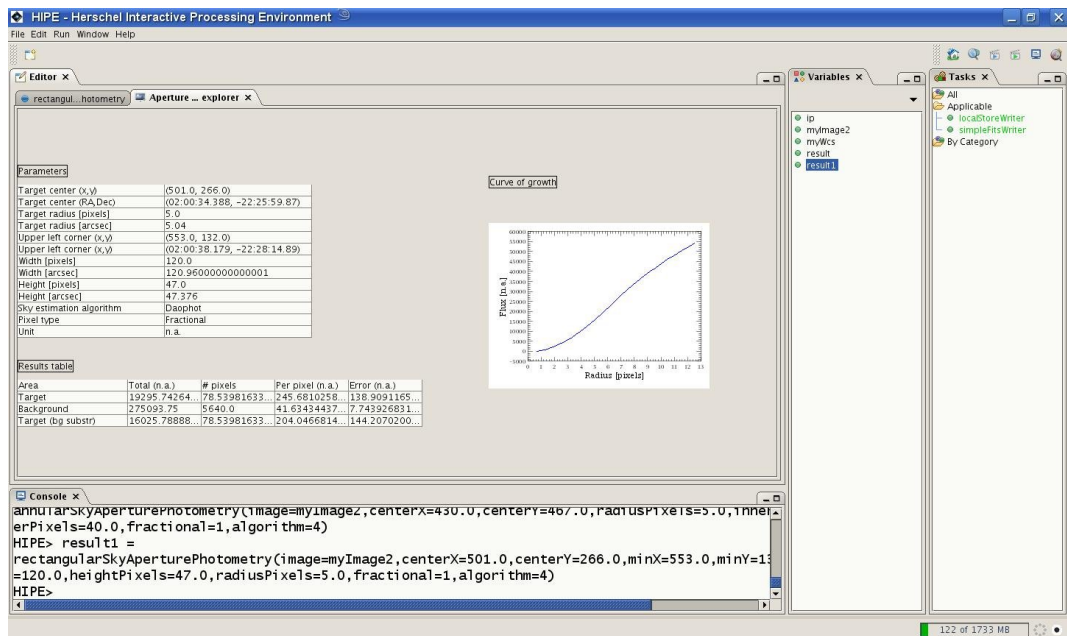


**Figure 4.30. Results of using the rectangular sky aperture photometry task.**

# 4.3.2.3. Aperture Photometry with a 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.

**In the HIPE-integrated GUI**

Also for this task, a HIPE-integrated GUI can be used to do the calculations. Simply select *fixedSkyAperturePhotometry* in the *Tasks* view. In the task dialog that will open in the *Editor*, you can specify the target center and target radius just like for the other aperture photometry tasks. Finally, you also need to enter the value you wish to use for the sky intensity. This is to be specified in the same unit as the one for the `Image`.

When pressing the *Accept* button, the calculations will be done : the result will appear in the *Variables* view and the corresponding command will be echoed to the *Console*.
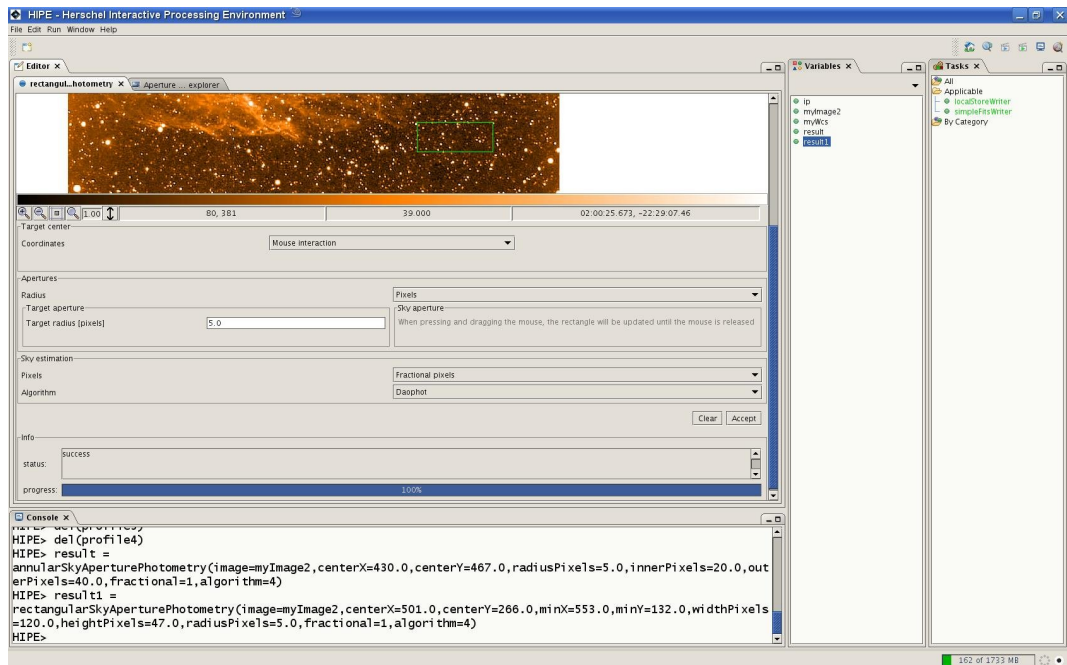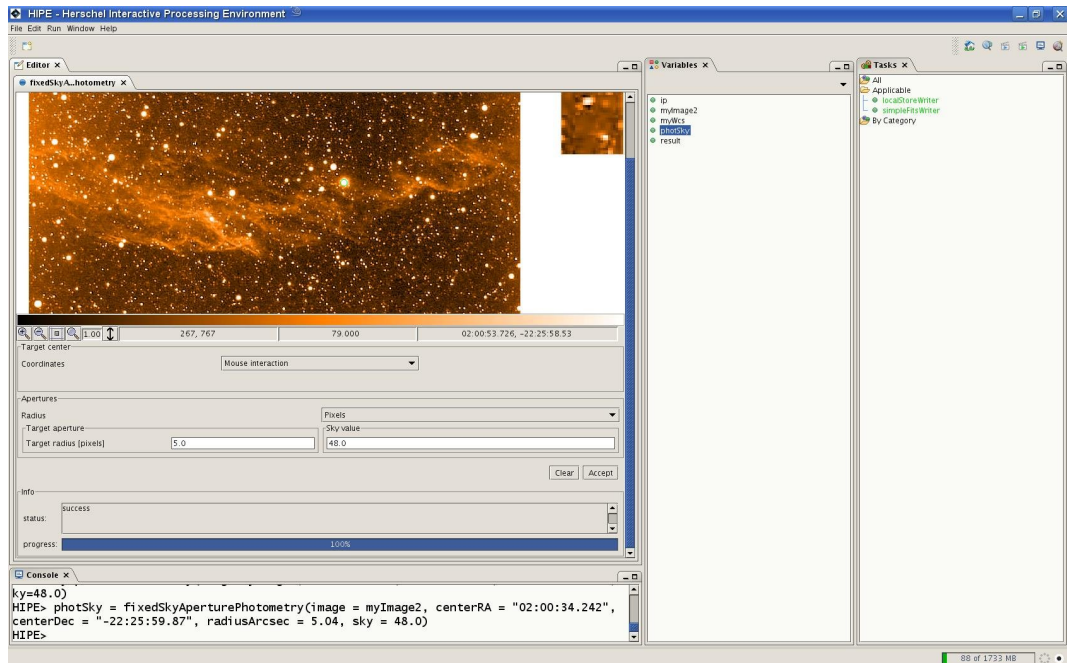


**Figure 4.31. Fixed sky aperture photometry task.**

**Inspecting the result**

The resulting product, `phot`, can be explored by double clicking on it in the *Variables* view. The explorer that will be opened in the *Editor* looks again very similar to the ones for the other aperture photometry products, as you can see here :
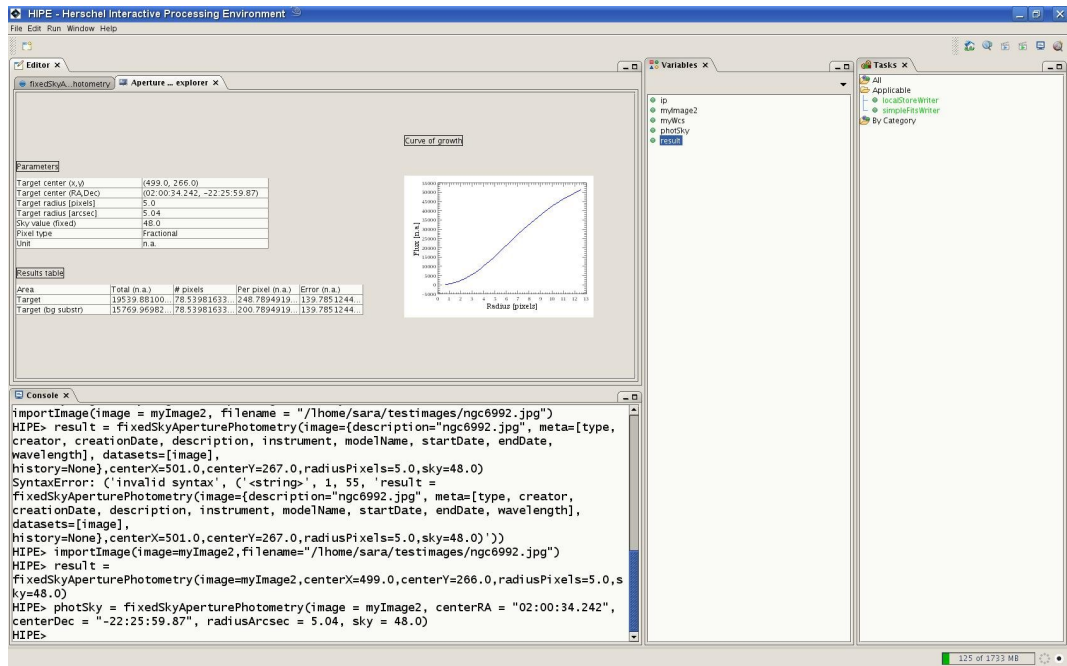
**Figure 4.32. Results of using the fixed sky aperture photometry task.**

You can also inspect `phot` via the command line. You can use the same commands as for the other aperture photometry tasks, except for those referring to the sky aperture. To obtain the sky intensity value, type

```
phot.getSkyValue()
phot.getIntensityPerSkyPixel()
```

# 4.3.3. Histograms

Several tasks have been written to make a histogram of an `Image`, or of a region you are interested in. Such a region of interest can be bounded by a circle, an ellipse, a rectangle or a polygon.

**On the command line**

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, just type

```
# Making a histogram of an image
imageHistogram = 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.

**Note**

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.

**In the HIPE-integrated GUI**

To run a histogram task from a HIPE-integrated GUI, you must open its task dialog (in the *Editor*) by selecting the appropriate task in the *Tasks* view. In all task dialogs, filling out the parameters it quite straightforward. Only the drawing of the figures needs more explanation.

To draw a circle, an ellipse or a rectangle you must press the mouse, drag it, and release it when you're satisfied with the result. Drawing the polygon can be done by clicking on the image. Double clicking means fixing the last vertex of the polygon. Afterwards you can always move/manipulate this figure.

You also notice that the cut levels are already filled out for you. These values are the cut levels of your `Image` and only serve as initial guess, in order to avoid empty histograms.

When you press the *Accept* button, the histogram will be made. You will see it at the bottom of the task dialog, a new variable will appear in the *Variables* view and the corresponding command will be echoed to the *Console*. If you now adapt your figure, only the histogram at the bottom of the task dialog will be updated. To transfer this to the *Variables* view, you need to press the *Accept* button again.

This is how the task dialog for the `ImageHistogramTask` looks like :
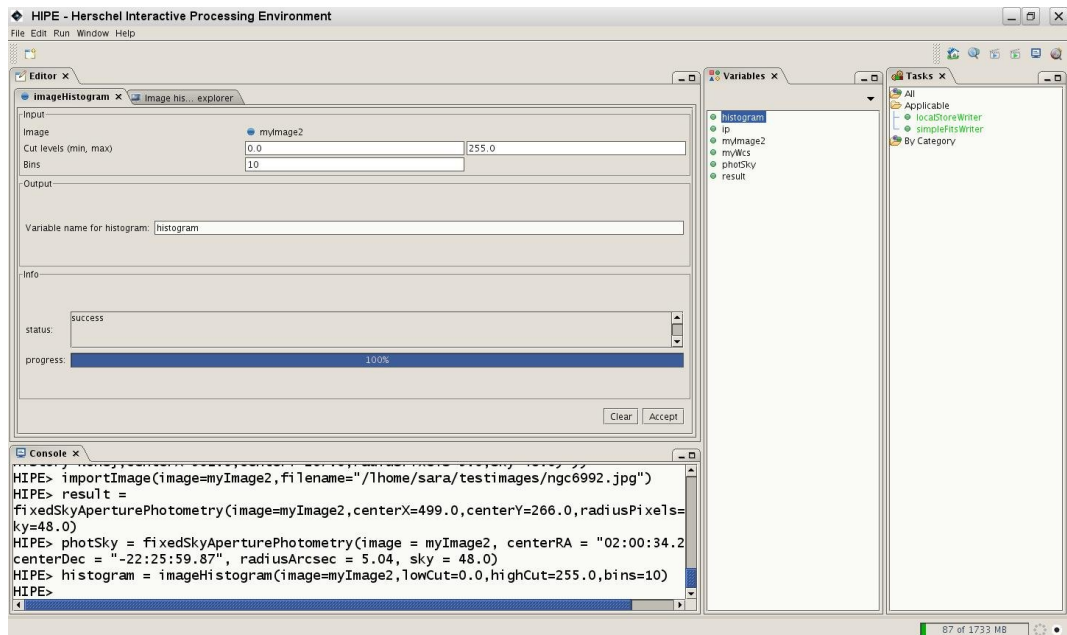


**Figure 4.33. Image histogram task.**

The task dialogs for the other histogram tasks look very similar :
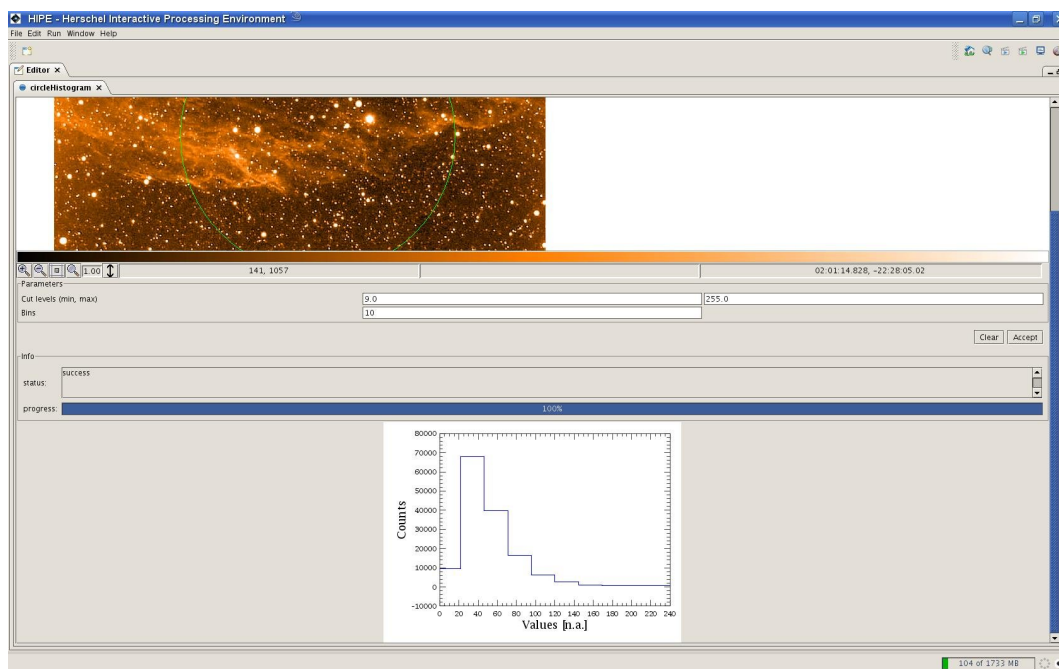
**Figure 4.34. Circle histogram task.**

### Inspecting the result

The resulting product, `histogram`, can be explored by double-clicking on it in the *Variables* view. You will see a table with the number of bins and the cut levels, and - if "drawing" a figure was required - the parameters specifying this figure. Also the histogram itself it shown.

This is how a histogram explorer will look like :
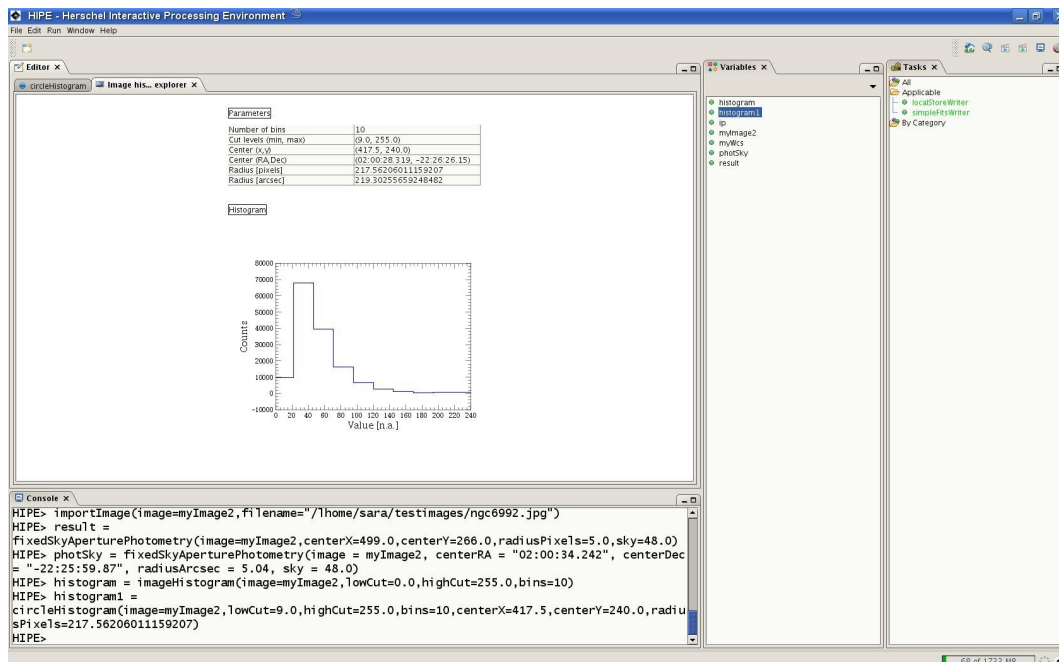


**Figure 4.35. Circle histogram task output.**

You can also explore `histogram` via the command line, using the following commands :

```
# 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

```
# 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()
```

# 4.3.4. Contour Plotting

Yet another way to inspect `Images`, is to make contour plots. This can be done by specifying one (`ContourTask`), or several (`ManualContourTask`) contour values, or to let them be calculated automatically (`AutomaticContourTask`).

**On the command line**

If you know in advance, you want to plot only one contour value, you can use the `ContourTask`. The only input parameters are :

- the *image* (`Image image`)

- the *contour value* (`Double value`)

To run this task, simply type

```
contours = contour(image = myImage2, value = 100.0)
```

If you want to specify multiple contour values yourself, use the `ManualContourTask`. This task takes the following parameters as input :

- the *image* (`Image image`)

- a l*ist of contour values* (`Double1d values`)

The commands you need, are

```
# 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, logarithmic or ln), using the `AutomaticContourTask`. The task will then determine the corresponding contour values and calculate the contours. The input parameters are :

- 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`)

The command is

```
# 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 these results will appear in the *Variables* view.

**In the HIPE-integrated GUI**

Plotting contours for one or several value is fully integrated into HIPE, as well as the the
`AutomaticContourTask`.

If you want to plot only one contour value, you can open the task dialog for the `ContourTask` by
double clicking on *contour* in the *Tasks* view. In this task dialog you must only fill out the contour
value you are interested in. Here is a screenshot of the task dialog :
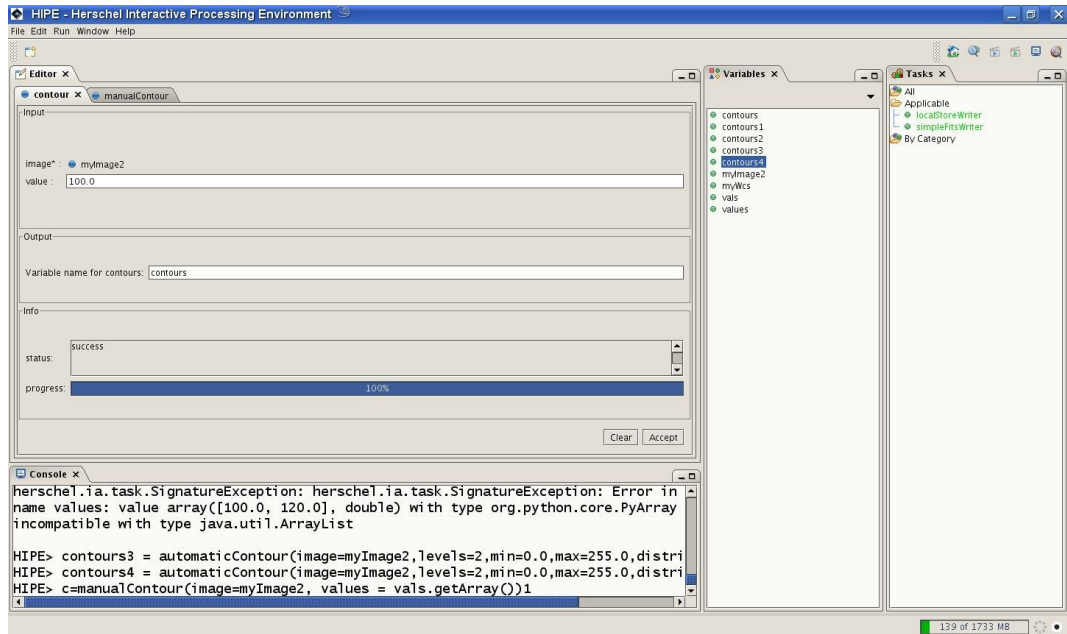


**Figure 4.36. Image contour task.**

Another option is to specify multiple contour values, by double-clicking on *manualContour* in the
*Tasks* view. In the task dialog that opens in the *Editor*, you can add the contour values you are interested
in, to a list. It is also possible to remove values from the list (by selecting the value in the less and
press the *Remove* button) or to clear the whole list (by pressing the *Clear* button). This is how the
task dialog looks like :
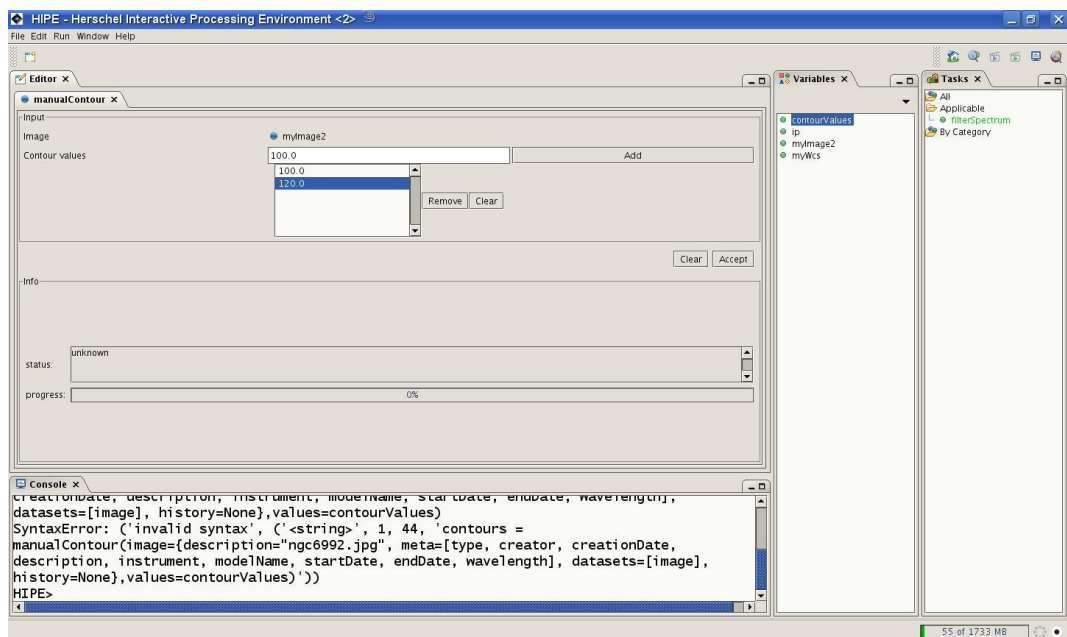


**Figure 4.37. Manual setting of contour levels with the image contour task.**

To let the contour values be automatically calculated, just double-click on *automaticContour* in the *Tasks* view. In the task dialog that will open by doing this, you must specify all input parameters. The cut levels of the Image are used as initial guess for the extreme contour values. A combo box allows you to use the distribution of the contour values. The task dialog is shown here :
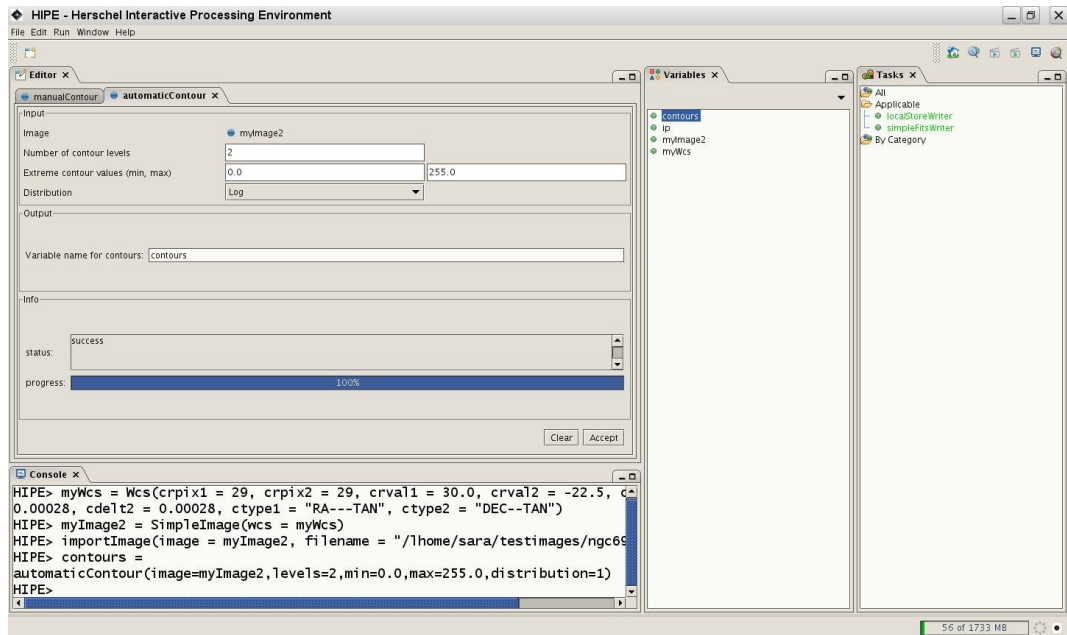


**Figure 4.38. Automatic contour level setting with the image contour task.**

If you press the *Accept* button, the contours will be calculated and appear in *Variables* view and the corresponding command will be echoed to the *Console*.

**Inspecting the result**

The resulting product, `contours`, can be explored - as all `Products` - with the product viewer, but this is not of much use. You want to plot your contours on your image. This can be done by opening your `Image` (by double-clicking on it in the *Variables* view) and then dragging contours from the *Variables* view to the image explorer. The contours will be plotted instantaneously on the `Image`. This will all be done in green for the moment. The idea is to use different colours for different contour levels in the future.

If the the contours are calculated for an `Image` with a valid `Wcs` and you drag it on an `Image` with a valid `Wcs`, the plotting will be done based on the sky coordinates. In all other cases, the pixel coordinates will be used.

You are not obliged to drag your contours over the `Image` for which they were calculated. This can be convenient if you want to compare `Images` at different wavelengths.
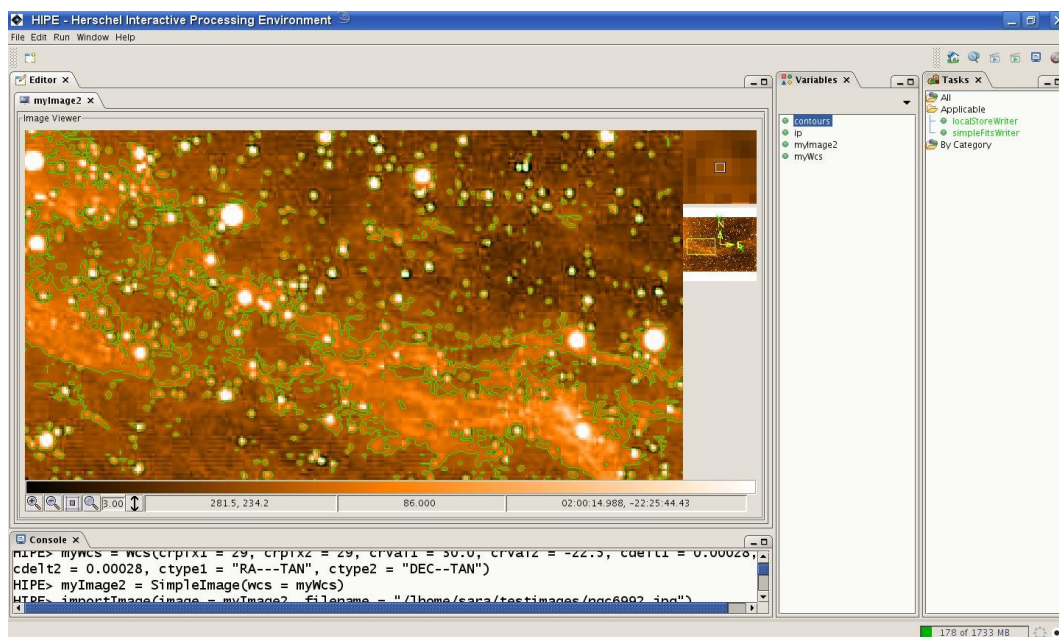
**Figure 4.39. Output of the contour task.**

# 4.3.5. Mosaicking

This task is not yet integrated in HIPE, so it is only available from the command line. The only input parameters you need, are :

- a *list with images you want to combine* (`ArrayList<Image> images`)

- *oversampling* (`Boolean oversampling`) - optional (per default : `True`)

**On the command line**

To combine n `Images`, say image_1,..., image_n, to a mosaic, you must type

```
# 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)
```

**Inspecting the result**

The result, `mosaic`, is a `SimpleImage` and can be treated like any other `Image`.

# 4.3.6. Smoothing

Four different smoothing algorithms are available :

- average smoothing

- median smoothing

- boxcar smoothing

- gaussian smoothing

These can be executed on the command line or via a HIPE-integrated GUI. They all take the following parameters as input :

- the *image* (`Image image`)

- the *width of the filtering window/boxcar/gaussian* (`width`)

The parameters `width` must be an odd positive `Integer` for mean and median smoothing, a positive `Integer` for boxcar smoothing and a positive `Double` for gaussian smoothing.

### On the command line

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, width = 2.5)
```

### Via the HIPE-integrated GUI

In all tasks dialogs (to be opened by double clicking on *meanSmoothing*, *medianSmoothing*, *boxcarSmoothing* or *gaussianSmoothing* in the *Variables* view) have an empty field where you should enter the value for the parameter `width`.
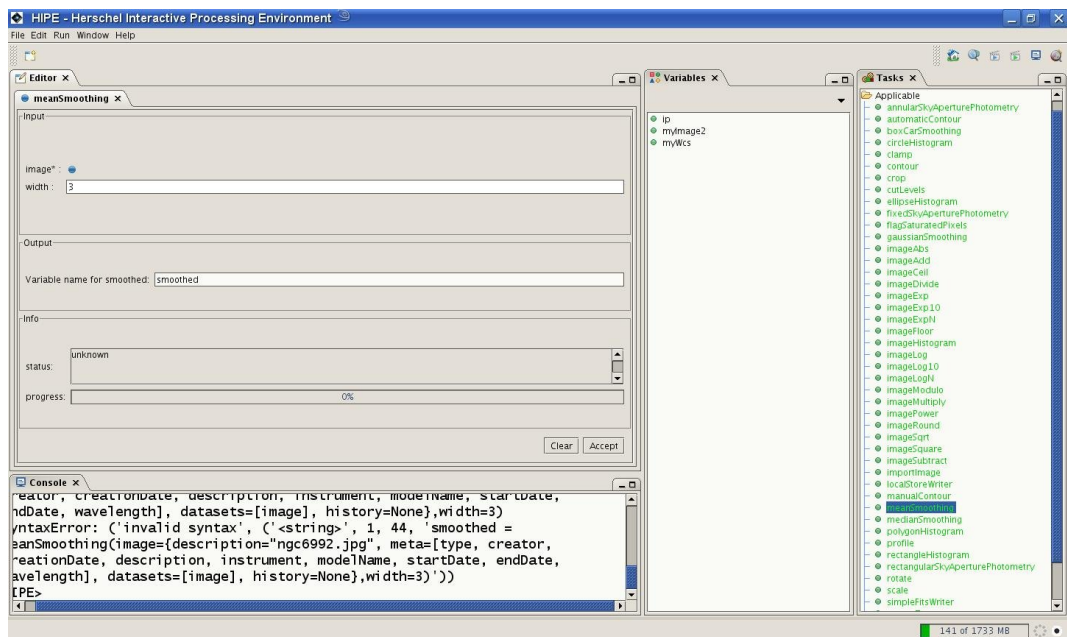


**Figure 4.40. Application of the image smoothing task.**

**Inspecting the result**

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.

# 4.3.7. 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`)

**On the command line**

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.

**In the HIPE-integrated GUI**

Opening the task dialog in the *Editor* by double clicking on clamp in the *Tasks* view, will show you that you must fill out the lower and upper intensity between which the `Image` must be clamped. Pressing the *Accept* button, will start the clamping, after which the resulting clamped `Image` will appear in the *Variables* view and the corresponding command will be echoed to the *Console*.

In the future the cut levels of the `Image` will be used as initial guess for the lower and upper intensities.
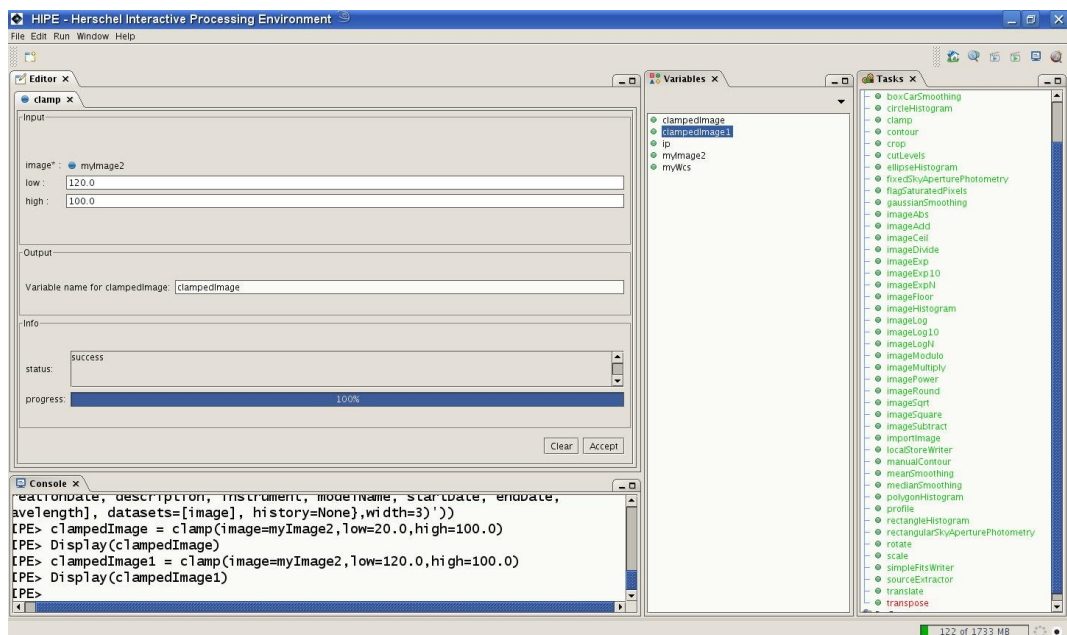


**Figure 4.41. Application of the image clamping task.**

**Inspecting the result**

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

# 4.3.8. 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

**On the command line**

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)
```

In the *Variables* view `cropped` will appear as new variable.

**In the HIPE-integrated GUI**

For the moment you can use the cropping task with the default task dialog. This can be opened by double-clicking on *crop* in the *Tasks* view. You have to specify the bounding rows and columns in the foreseen fields. In the future, this task dialog will be reworked, such that you can draw the bounding rectangle on the `Image`.

When you press the *Accept* button, the cropped `Image` will appear in the *Variables* view and the corresponding command will be echoed to the *Console*.



**Figure 4.42. Application of the image cropping task.**

**Inspecting the result**

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`.

# 4.3.9. 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)).

**On the command line**

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 will appear as a variable in the *Variables* view.

**In the HIPE-integrated GUI**

If you double-click on *rotate* in the *Tasks* view, a task dialog will open in the *Editor*. There you must fill out the rotation angle (in degrees) and via a combo box, you can choose the type of interpolation to be used for rotating. If you select one of the bicubic interpolation types (*Bi-cubic* or *Bi-cubic2*), the *Sub-sampling Bits* field becomes editable, so you can enter the number of subsampling bits.

When you press the *Accept* button, the result appears in the Variables view and the corresponding command is echoed to the *Console*.

**Inspecting the result**

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 :
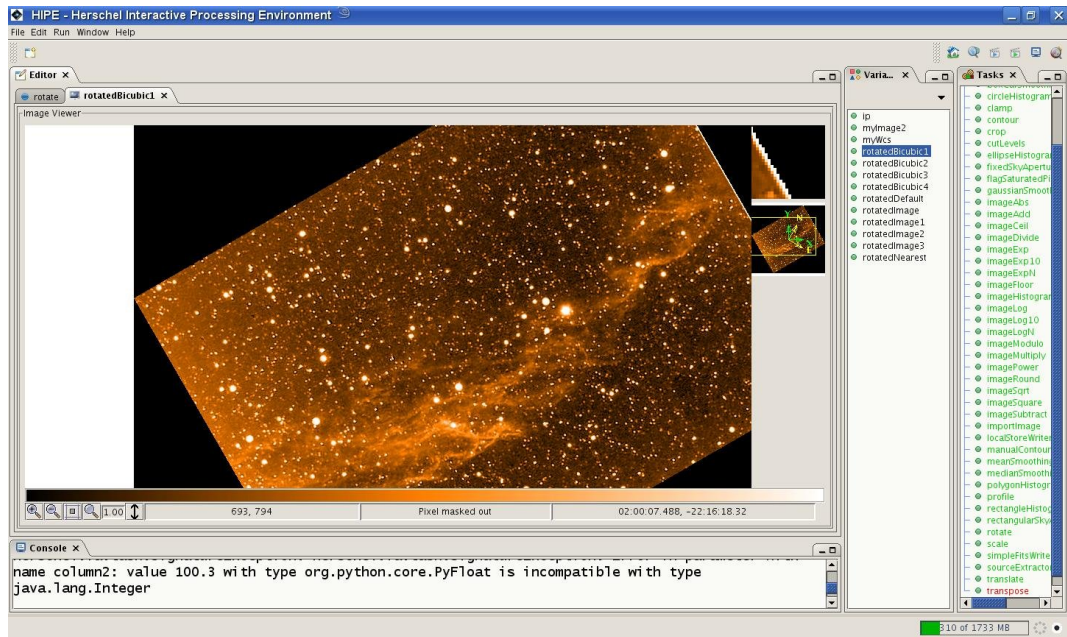
**Figure 4.43. Image rotation task.**

# 4.3.10. 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.

**On the command line**

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.

**In the HIPE-integrated GUI**

If you double-click on *scale* in the *Tasks* view, a task dialog opens in the *Editor*, where you have to specify the magnification factors in the x- and y-direction. Parameters for the interpolation can be specified as the same parameters for rotating.

If the *Accept* button is pressed, the scaled Image will appear in the *Variables* view and the corresponding command is echoed to the *Console*.
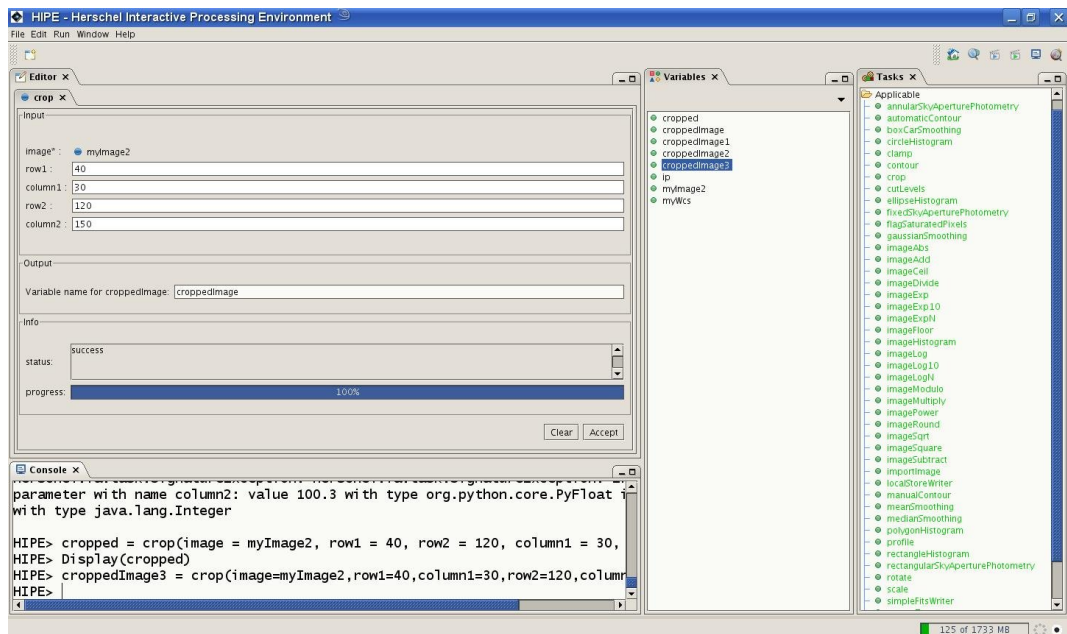
**Inspecting the result**

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`. An example is shown here :
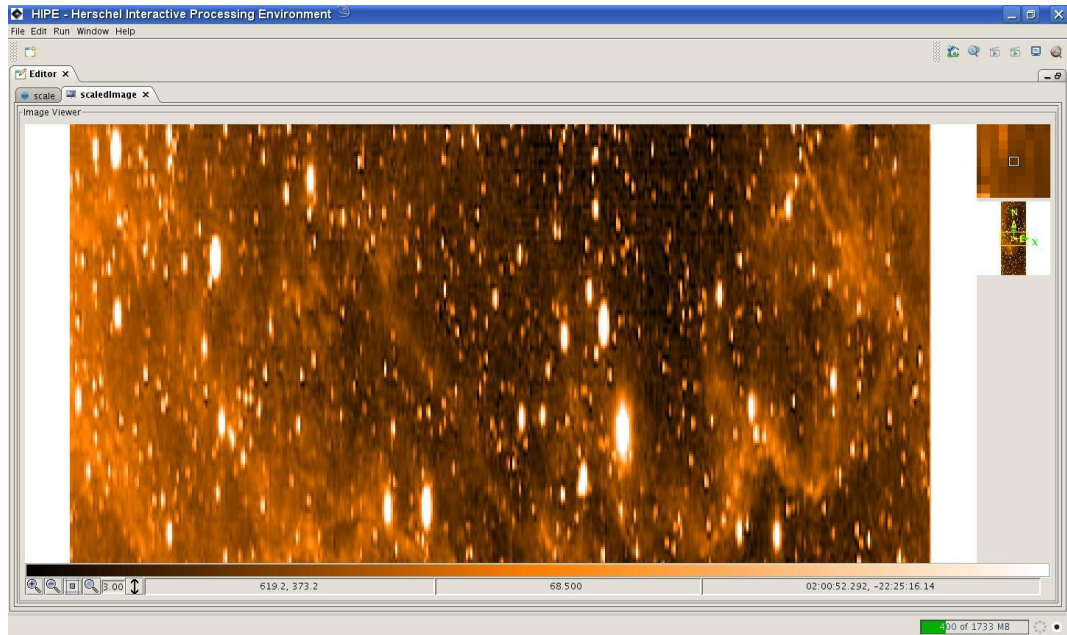


**Figure 4.44. Application of the image scaling task.**

# 4.3.11. 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`)

**On the command line**

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")
```

The result will appear in the *Variables* view.

> **Note**
>
> For the moment you can specify the pixel and sky coordinates at the same time. This should be prohibited in the future.

**In the HIPE-integrated GUI**

For the moment, the default task dialog is opened when double clicking on *translate* in the *Tasks* view.

When the *Accept* button is pressed, that `Image` is translated and appears in the *Variables* view and the corresponding command is echoed to the *Console*.

**Inspecting the result**

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.

# 4.3.12. 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

**On the command line**

To transpose an Image, type

```
# Flip vertically
flippedVertically1 = transpose(image = myImage2, type = TransposeTask.FLIP_VERTICAL)
flippedVertically2 = transpose(image = myImage2, type = 0)
```

The transposed `Image` appears in the *Variables* view.

**In the HIPE-integrated GUI**

If you double-click on *transpose* in the *Tasks* view, a task dialog is opened in the *Editor*. Here you can choose via a combo box which tranposition type you want to use.
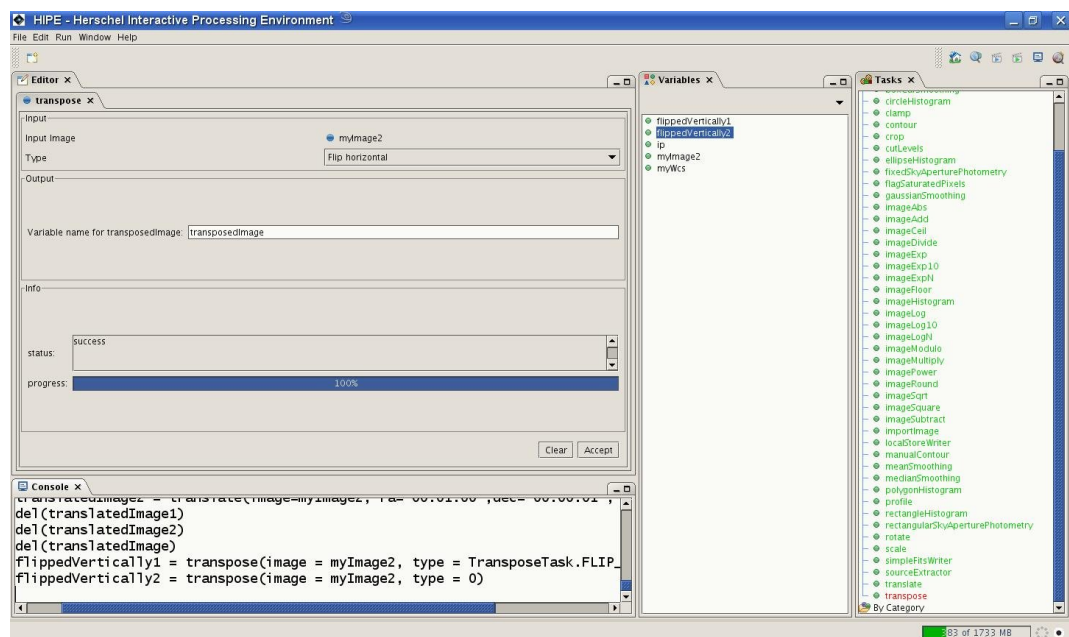


**Figure 4.45. Application of the image transposing task.**

**Inspecting the result**

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`.

# 4.3.13. 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 `FlagSaturatedPixelsTask`, by specifying these parameters :

- the *image* (`Image image`)

- the *cut off value* (`Double value`)

**On the command line**

To flag the saturated pixels, type

```
flagged = flagSaturatedPixels(image = myImage2, value = 100.0)
```

The resulting `Image` will appear in the *Variables* view.

**In the HIPE-integrated GUI**

By double-clicking on *flagSaturatedPixels* in the *Tasks* view, a task dialog is opened in the *Editor*. There you need to enter the cut off value, above which pixels are said to be saturated.

After having pressed the *Accept* button, the flagged `Image` appears in the *Variables* view and the corresponding command is echoed to the *Console*.

**Inspecting the result**

The result, `flagged`, looks like a copy of the input `Image`, except that pixels whose value lies above the given cut off value, are flagged out with the `SATURATED` flag type, as shown here :
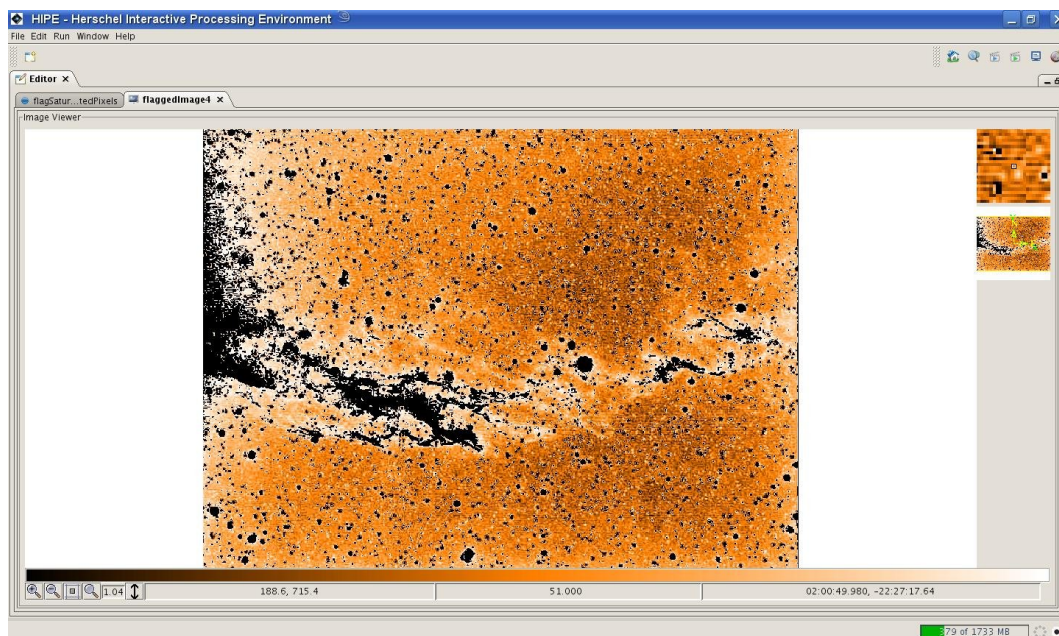


**Figure 4.46. Application of the image flagging task.**

# 4.3.14. Getting the Cut Levels

Using the `CutLevelsTask`, you can determine the cut levels of an Image, either using the percentage method or applying a median filter. You have to specify the following parameters

- the *image* (`Image image`)

- the *method* used for determining the cut levels (`Integer method`)

The method can be

- `CutLevelsTask.PERCENT` = 0 : percentage method

- `CutLevelsTask.MEDIAN_FILTER` = 1 : median filter

If you choose the percentage method, an extra parameter must be given a value :

- the *percentage* (`Double percent`) - per default : 99.5

**On the command line**

To calculate the cut levels of an `Image`, you must type

```
# 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 appears in the *Variables* view.

**In the HIPE-integrated GUI**

When double-clicking on *cutLevels* in the *Tasks* view, a task dialog will be opened in the *Editor*. There you can indicate via the *Method* combo box which method you want to use to determine the cut levels. When selecting *Percent*, the *Percent* field will become editable, so you can change the default value for the percentage.

After pressing the *Accept* button, the cut levels will be calculated and they will appear as one variable in the *Variables* view. The corresponding command will be echoed to the *Console*.
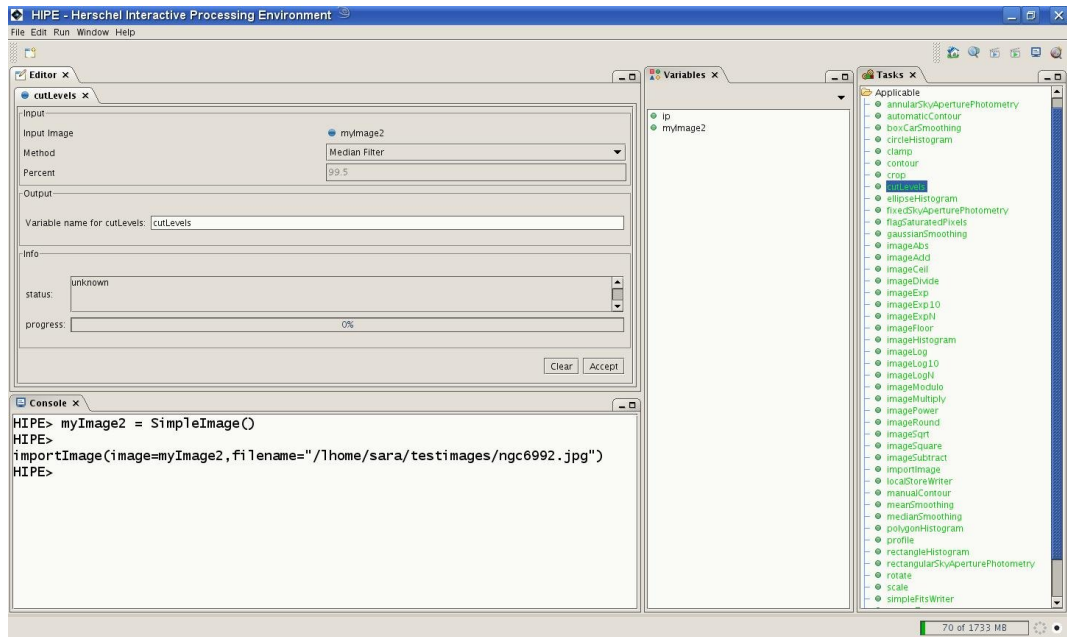
**Figure 4.47. Application of the image cut levels task.**

**Inspecting the result**

The result, `cutLevels`, is a double array. To gain access to the low and high cut, type

```
# The low cut
low = cutLevels[0]

# The high cut
high = cutLevels[1]
```

# 4.3.15. Image Arithmetics

The last functionality on Images we describe in this chapter, is image arithmetics. These options 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.

## 4.3.15.1. 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`)

**On the command line**

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.

**Note**

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`.

**In the HIPE-integrated GUI**

If you double click on *imageAdd*, *imageSubtract*, *imageMultiply*, *imageDivide* or *imageModulo* in the *Tasks* view, the corresponding task dialog opens in the *Editor*. In the modifier for *1st addend*,

*minuend*, *multiplier* or *dividend*, parameter `image1` is given as value. The combo box for *2nd addend*, *subtrahend*, *multiplicand* or *divisor* allows you to choose whether to use a second `Image` or a scalar as input.

If you choose for a second `Image`, a second combo box (*Reference*) appears, allowing you to choose to do the calculation either pixel-to-pixel or based on the `Wcs`. Also the second `Image` must be dragged from the *Variables* view to the task dialog.

If you choose to use a scalar, the second combo box disappears and a field appears where you can give the value for the scalar.

When you press the *Accept* button, the resulting `Image` is constructed and will appear in the *Variables* view. The corresponding command is echoed to the *Console*.

## 4.3.15.2. Absolute values

The only parameter that needs to be specified is

- the *image* (`Image image`)

**On the command line**

To take the absolute value of all intensity values in an `Image`, simply type

```
abs = imageAbs(image = myImage)
```

**In the HIPE-integrated GUI**

Double-clicking on *imageAbs* in the *Tasks* view, will open the default task dialog. You only need to drag the `Image` you want to use as input to the foreseen modifier. If you press the *Accept* button, the resulting `Image` will appear in the *Variables* view and the corresponding command will be echoed to the *Console*.

## 4.3.15.3. Rounding/Flooring/Ceiling

If you want to round, floor of ceil all intensity values, you only have to specify

- the *image* (`Image image`)

**On the command line**

The commands are

```
# Rounding
rounded = imageRound(image = myImage2)

# Flooring
floored = imageFloor(image = myImage2)

# Ceiling
ceiled = imageCeil(image = myImage2)
```

**In the HIPE-integrated GUI**

The corresponding default task dialogs can be opened in the *Editor* by double-clicking on *imageRound*, *imageFloor* or *imageCeil* in the *Tasks* view. It looks exactly the same as the task dialog for the `ImageAbsTask`.

## 4.3.15.4. 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`)

**On the command line**

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)
```

**In the HIPE-integrated GUI**

You can also run these tasks in a GUI in HIPE. Then you must double-click on *imagePower*, *imageSquare* or *imageSqrt* in the *Tasks* view. This will open the default task dialog in the *Editor*, to which you can drag the `Image` you want to use as input. For the `ImagePowerTask`, a field is foreseen where you can give the power.

# 4.3.15.5. 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`)

**On the command line**

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)
```

**In the HIPE-integrated GUI**

To open the task dialog in the *Editor*, you must double-click on *imageLog*, *imageLog10*, *imageLogN*, *imageExp*, *imageExp10* or *imageExpN* in the *Tasks* view. An `Image` can be dragged from the *Variables* view to be used as input. In the task dialogs for the `ImageLogNTask` and `ImageExpNTask` a field is foreseen to enter the value for parameter `n`.

# 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.

- 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 1.2). 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."
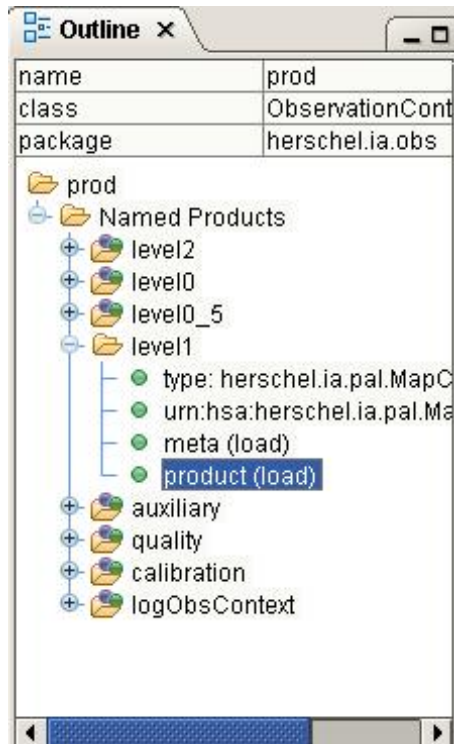
**Figure 5.1. Selecting Level 1 data from a downloaded archive observation done by HIFI.**
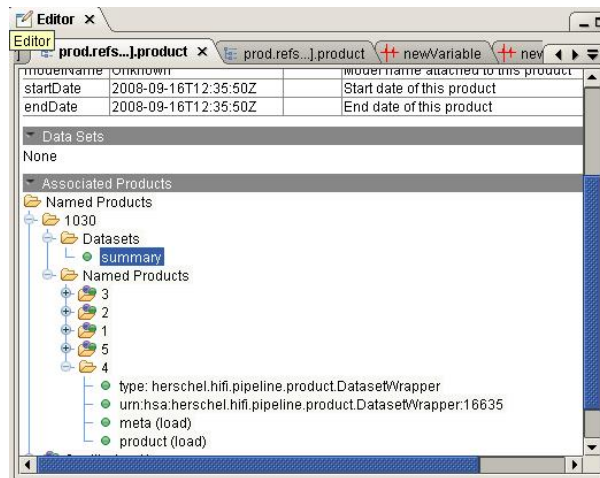


**Figure 5.2. Display of product set.**
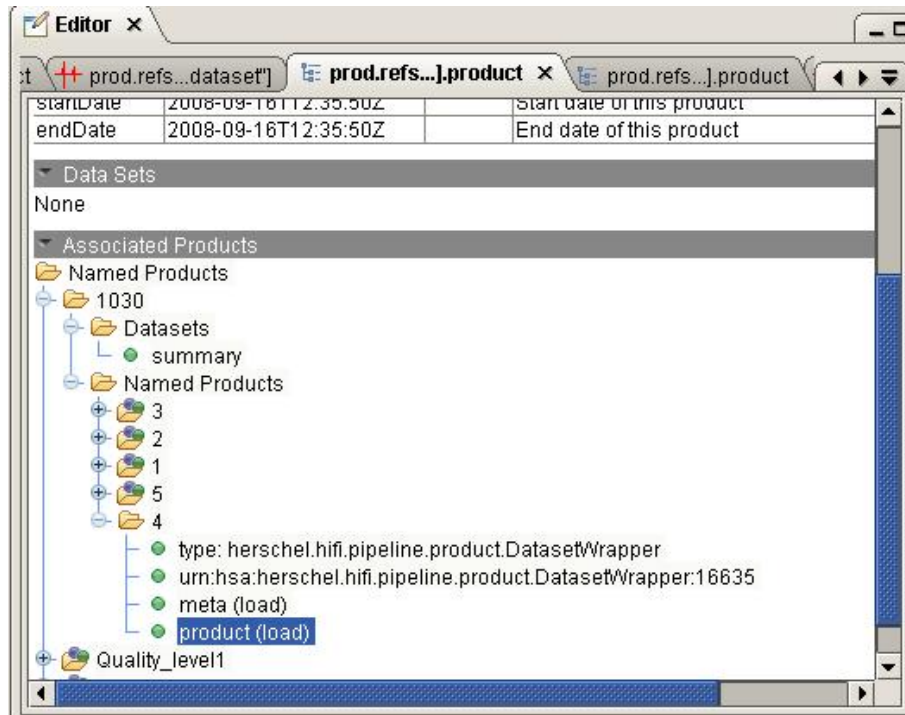
**Figure 5.3. Choosing the product with the dataset we want.**

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.
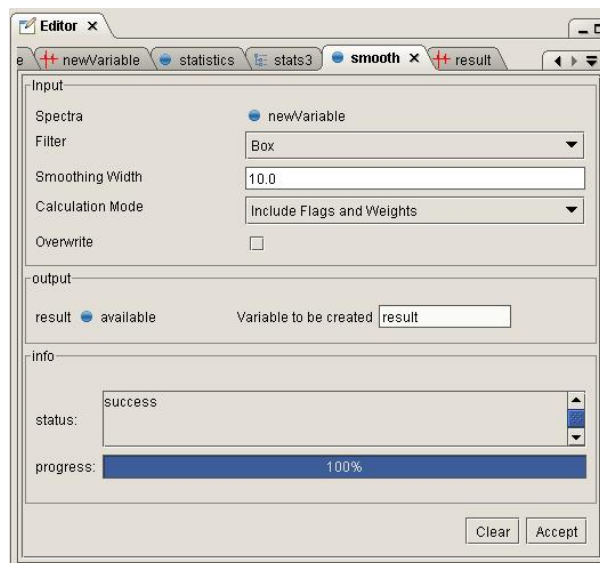


**Figure 5.6. Using the smooth task**
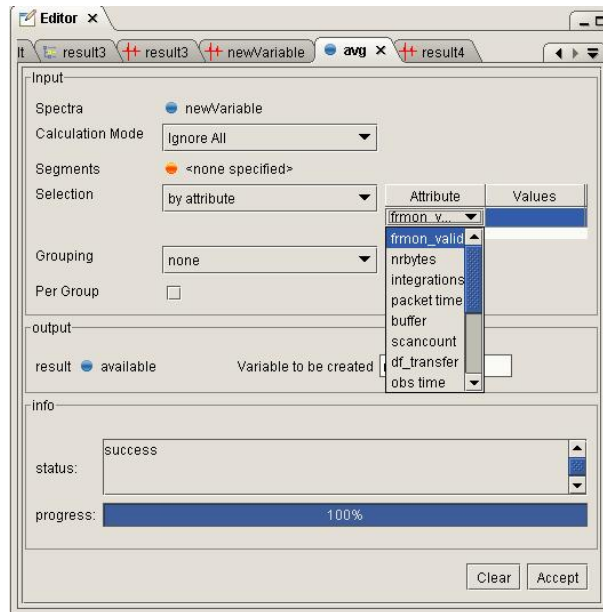
**Figure 5.7. Using the `avg` task**
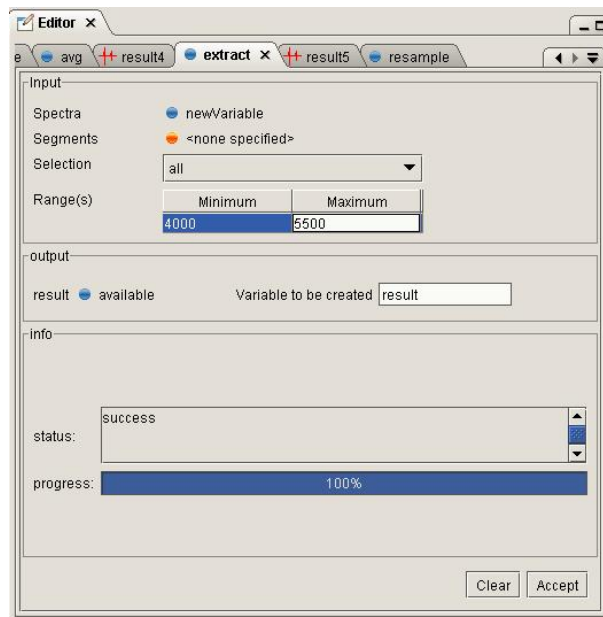


**Figure 5.8. Using the `extract` task**
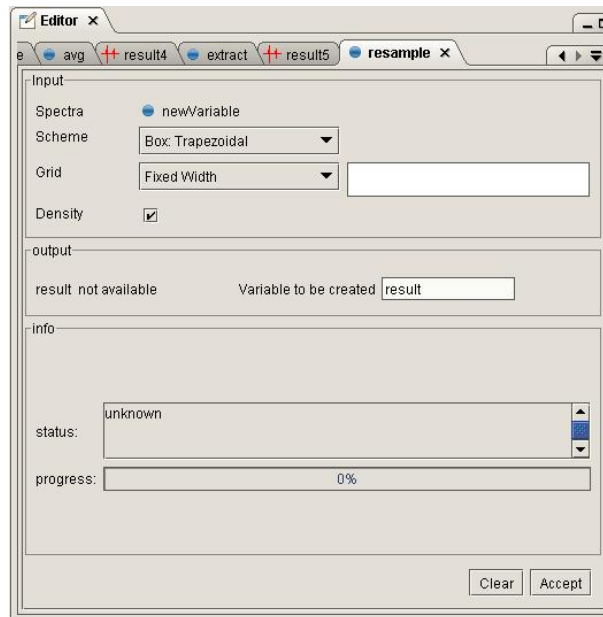
**Figure 5.9. Using the `resample` task**
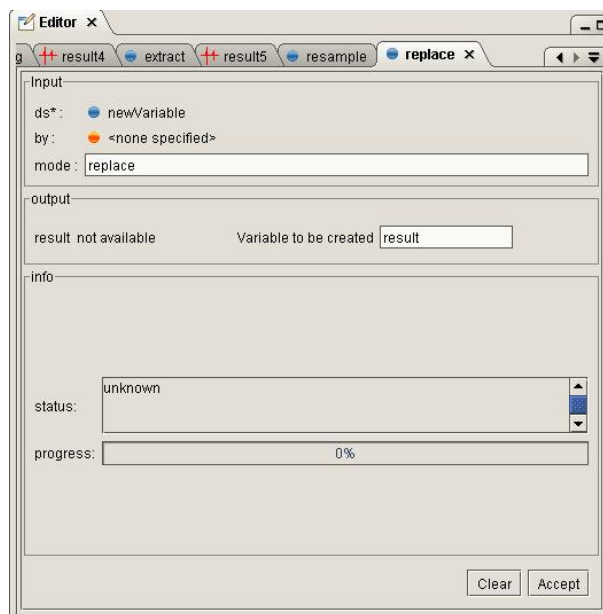


**Figure 5.10. Using the `replace` task**

- *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](#)).
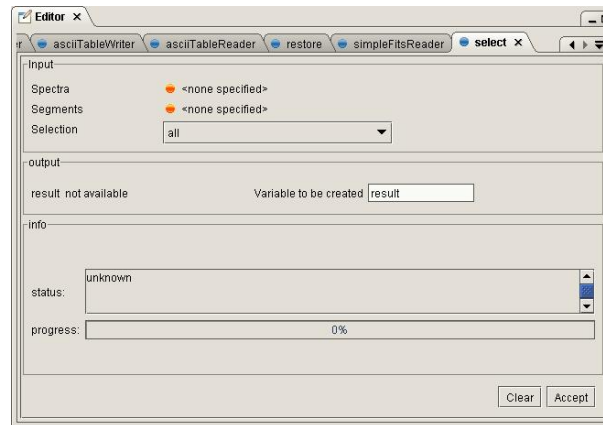
**Figure 5.4. Using the `select` task**

- *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)
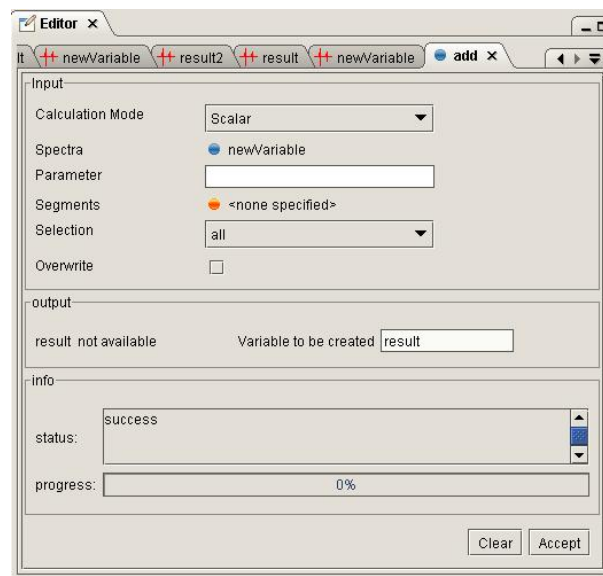


**Figure 5.5. Using the `add` task**

- *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:
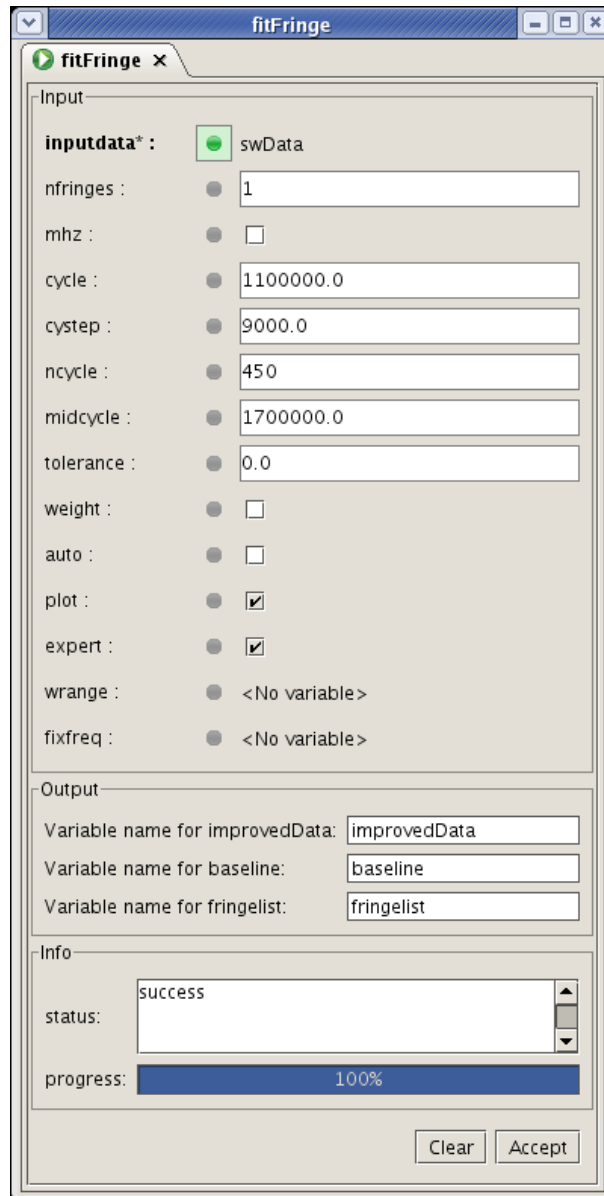
**Figure 5.11. The FitFringe task interface**

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.
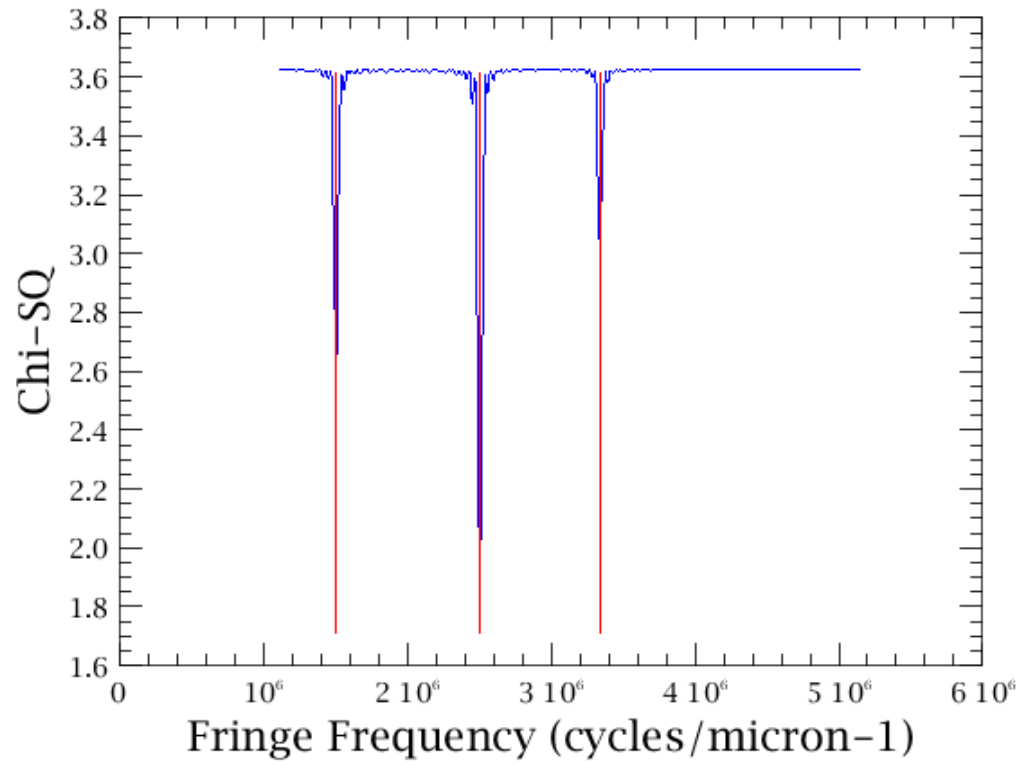
**Figure 5.12. Sine wave period plot**

The second plot shows the original data, the baseline, the sine-wave subtracted data, and the mask:
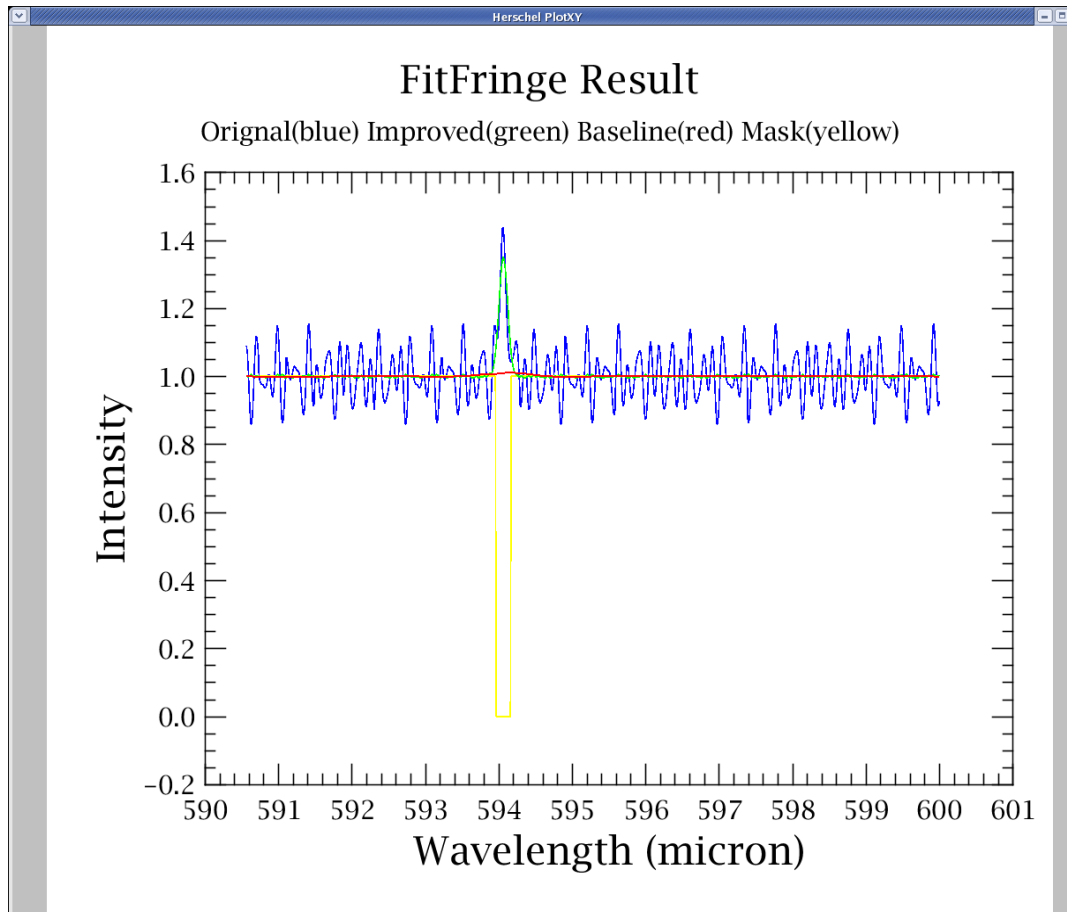
**Figure 5.13. FitFringe result plot**

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:
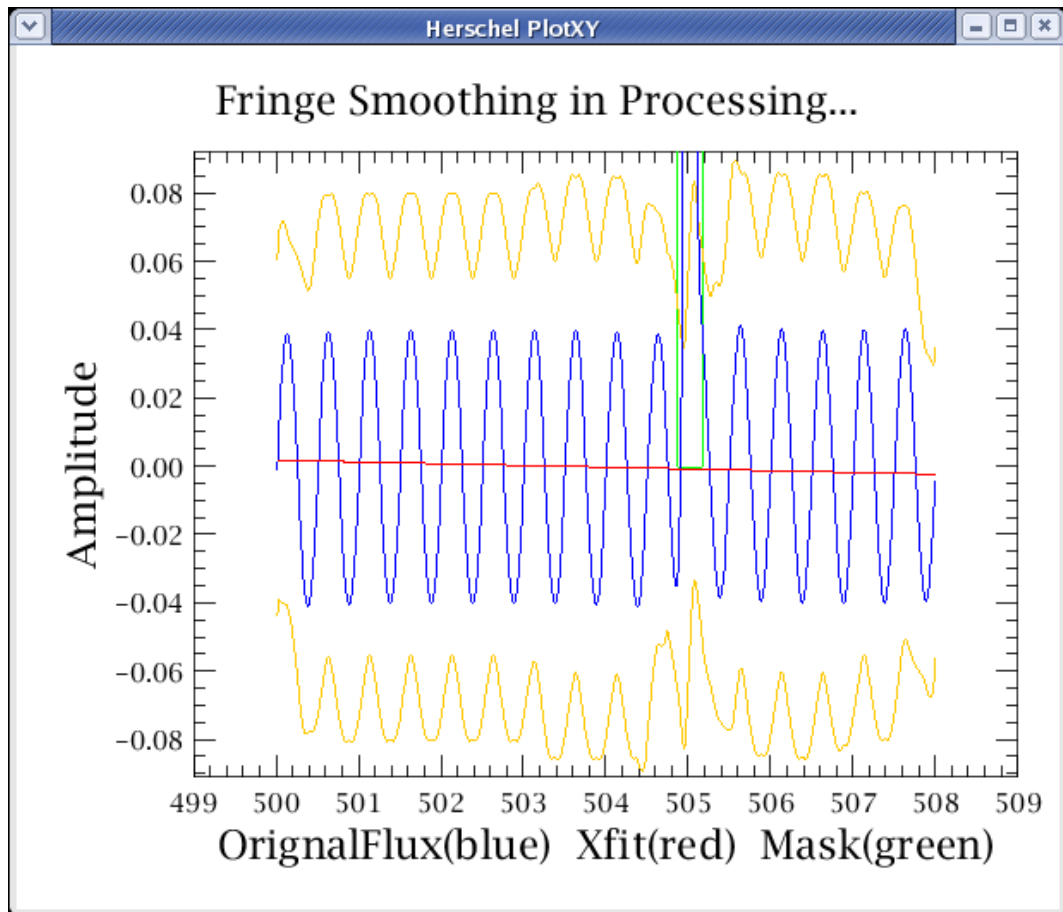


**Figure 5.14. The first plot generated by SmoothBaseline shows the initial baseline (red) and the limits above and below which signal will be masked (yellow). Clearly the emission line is masked, as indicated by the green line.**
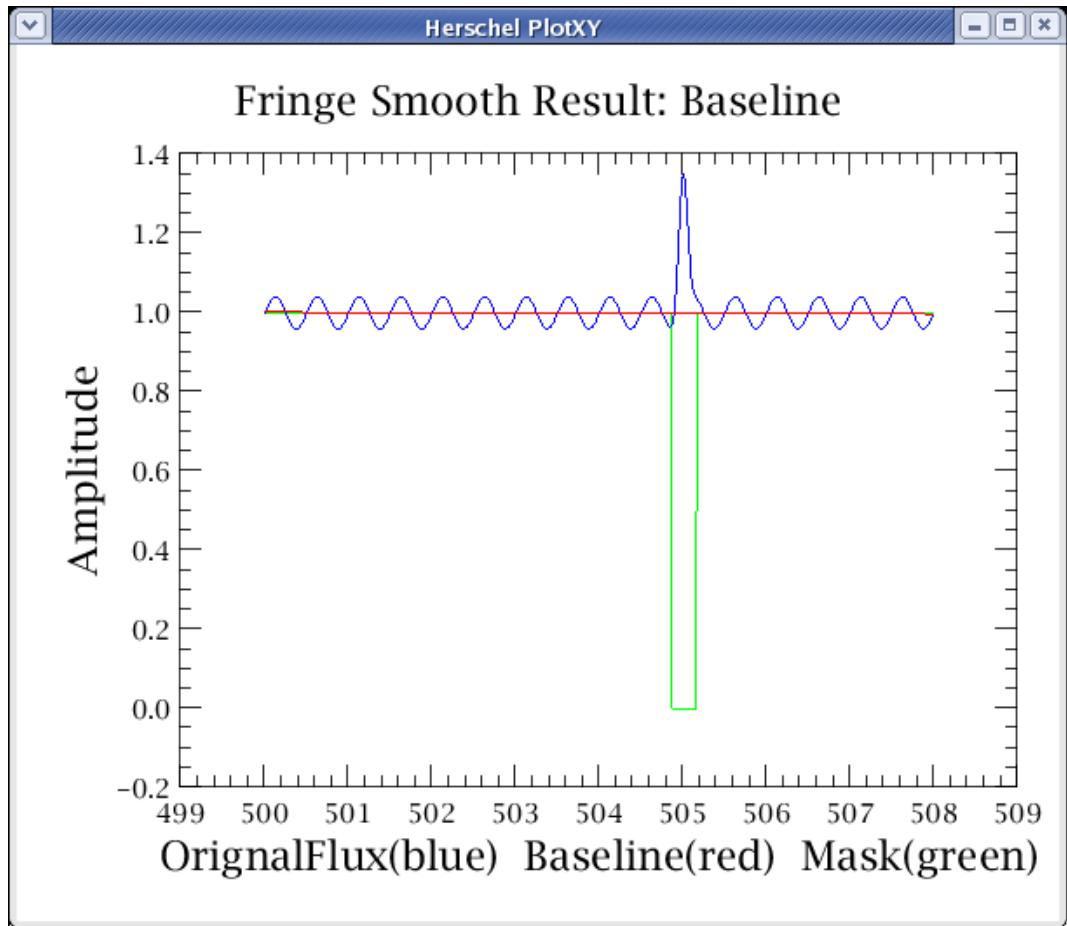
**Figure 5.15. The second plot generated by SmoothBaseline shows the baseline in red and the masked regions indicated in green.**

# 5.2.6. 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.

## 5.2.7. 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.

## 5.2.8. Using the GUI

When you launch the CSAT the following window will appear:
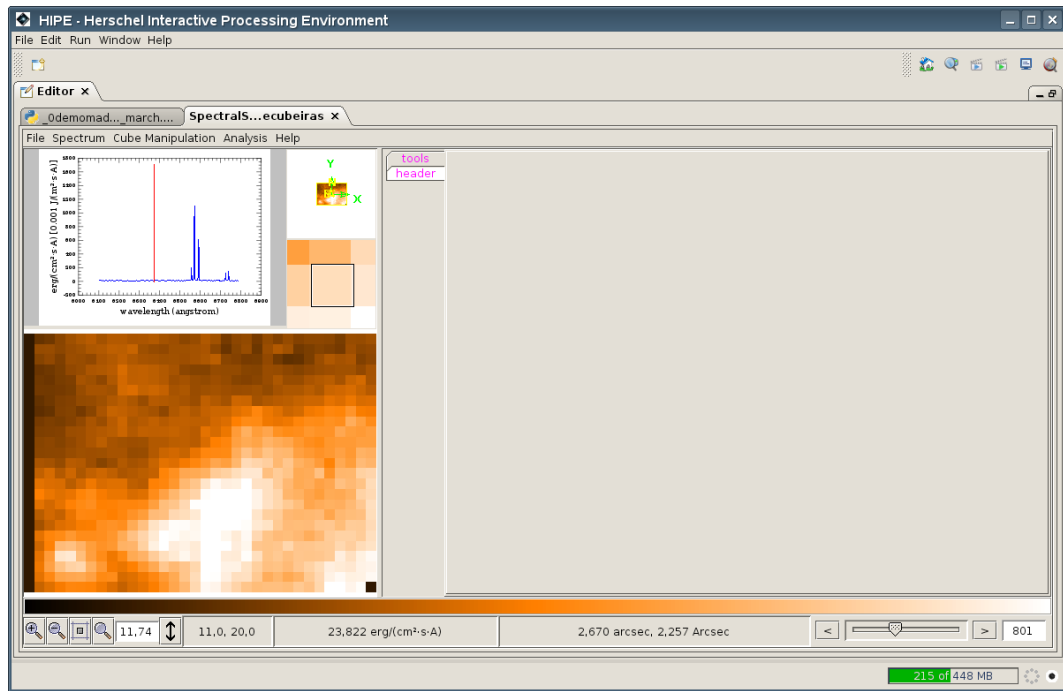
**Figure 5.16. The first view you will have of the cube tool GUI**

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.

## 5.2.8.1. 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.
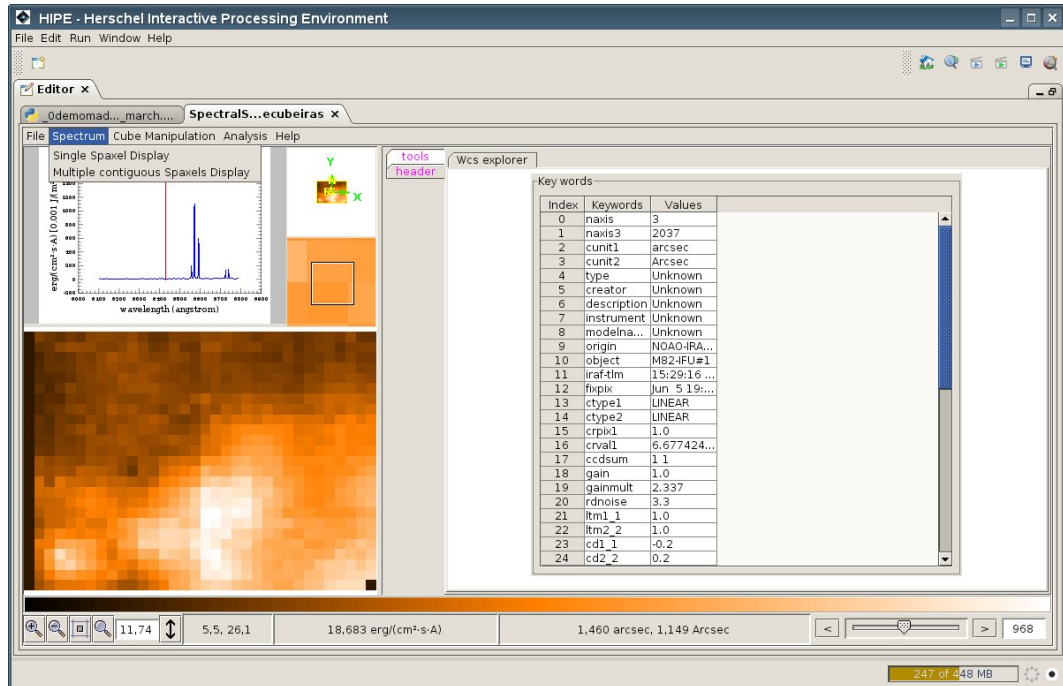


**Figure 5.17. The header explorer**

## 5.2.8.2. 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.
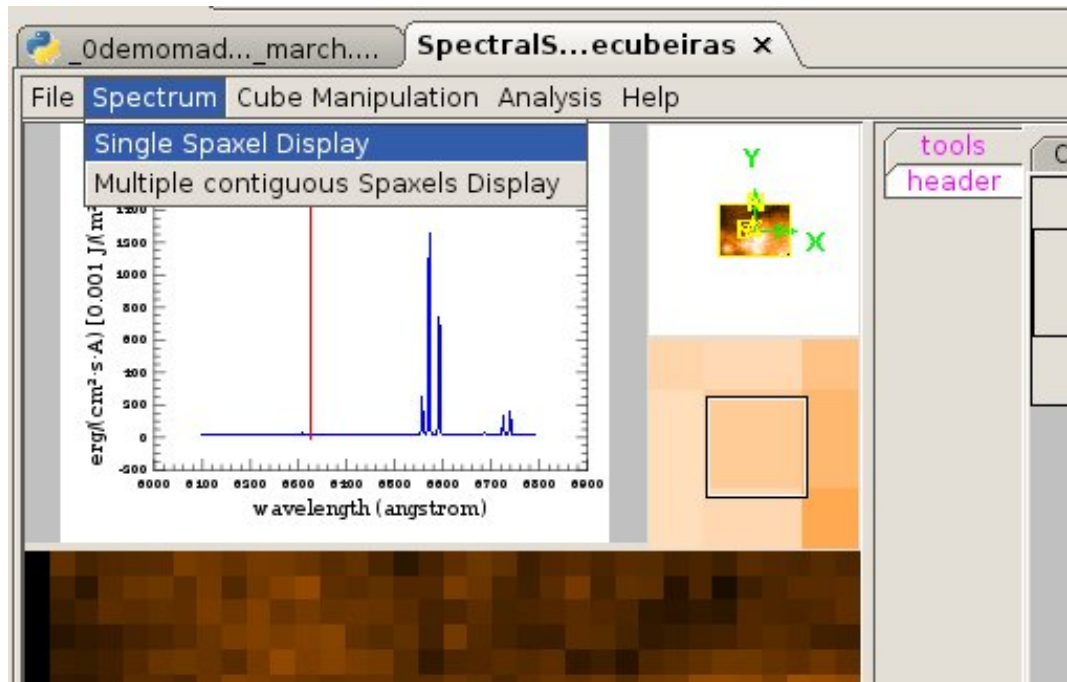
**Figure 5.18. The Spectrum menu**

## 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
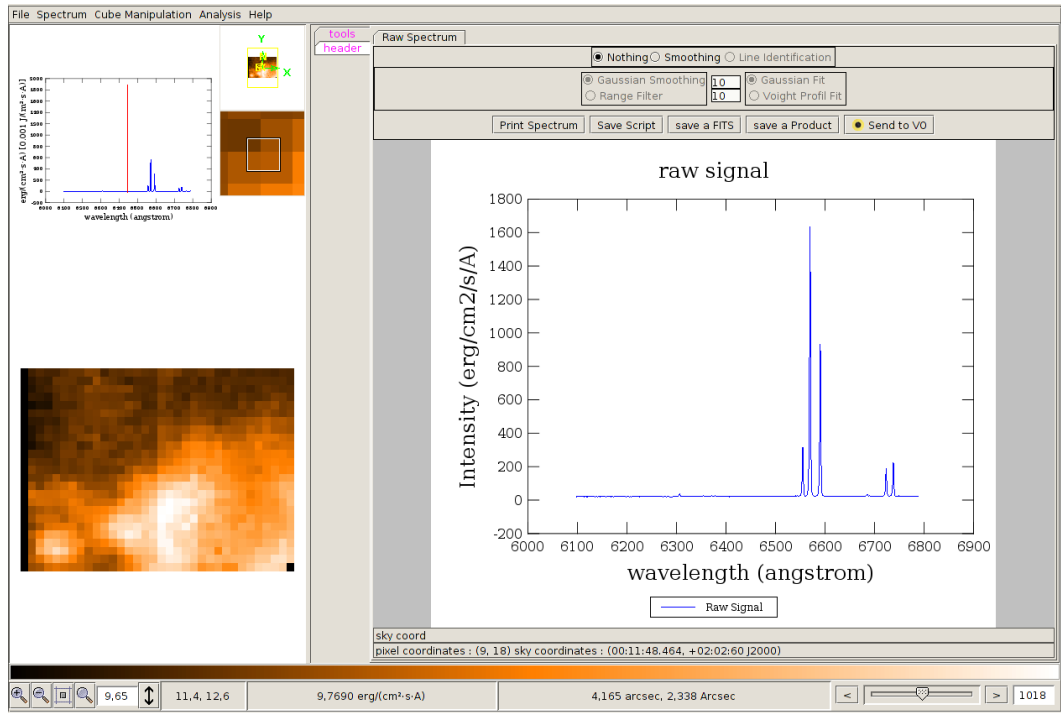
**Figure 5.19. The single spaxel spectrum extraction window**

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
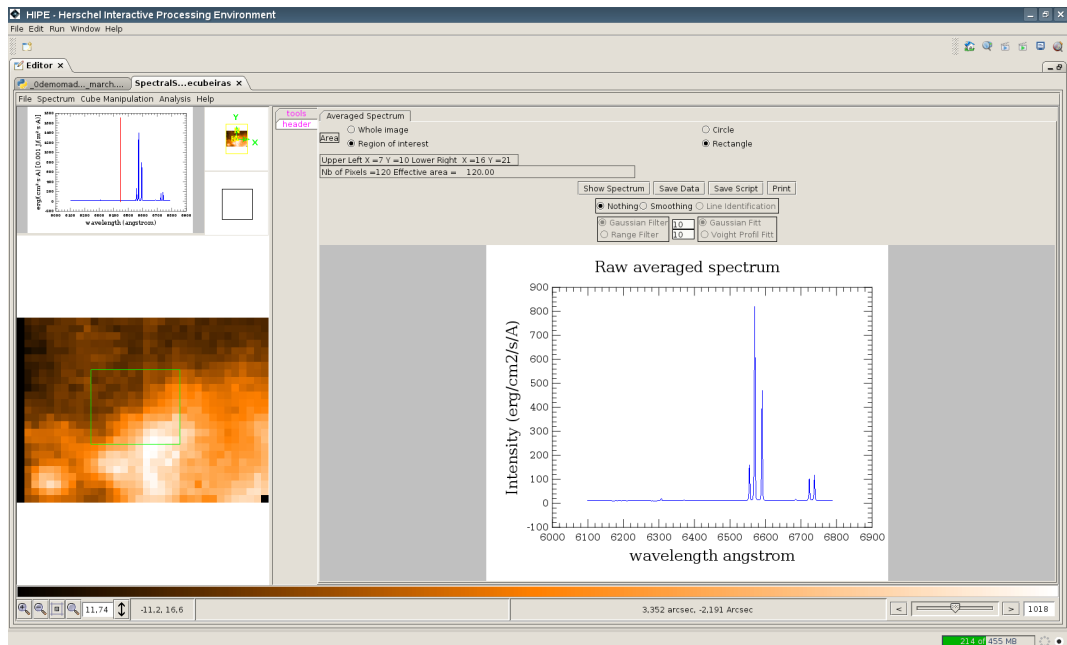


The spectrum shown is the average from the region drawn by the user on the cube image (which is in the left part of the window).

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.

## 5.2.8.3. 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.
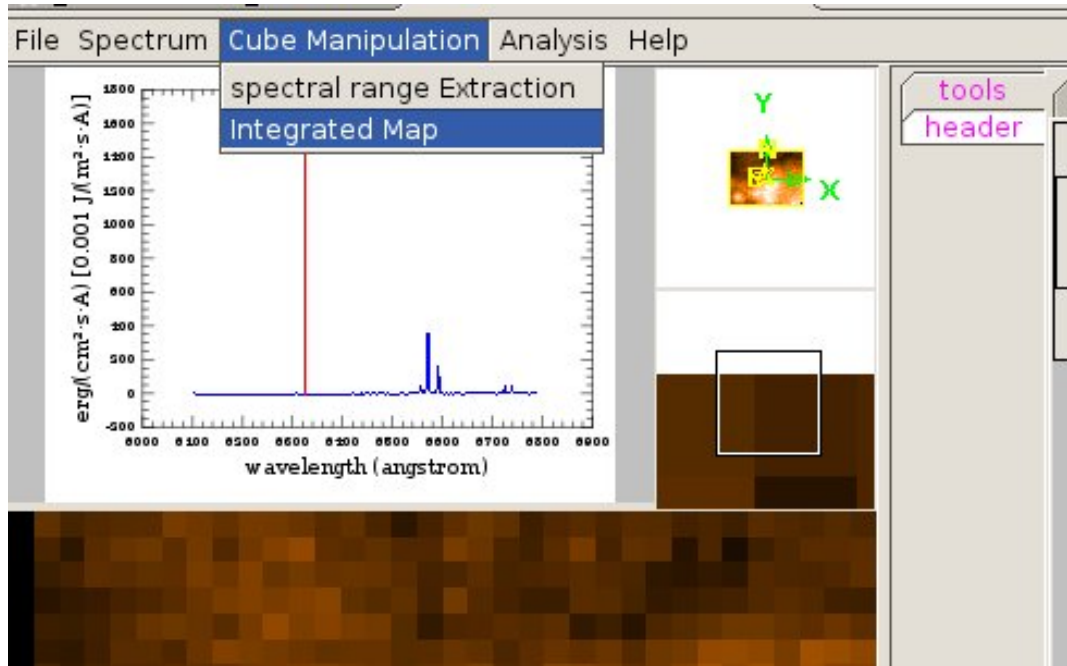
**Figure 5.20. The Cube Manipulation menu**

## 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.
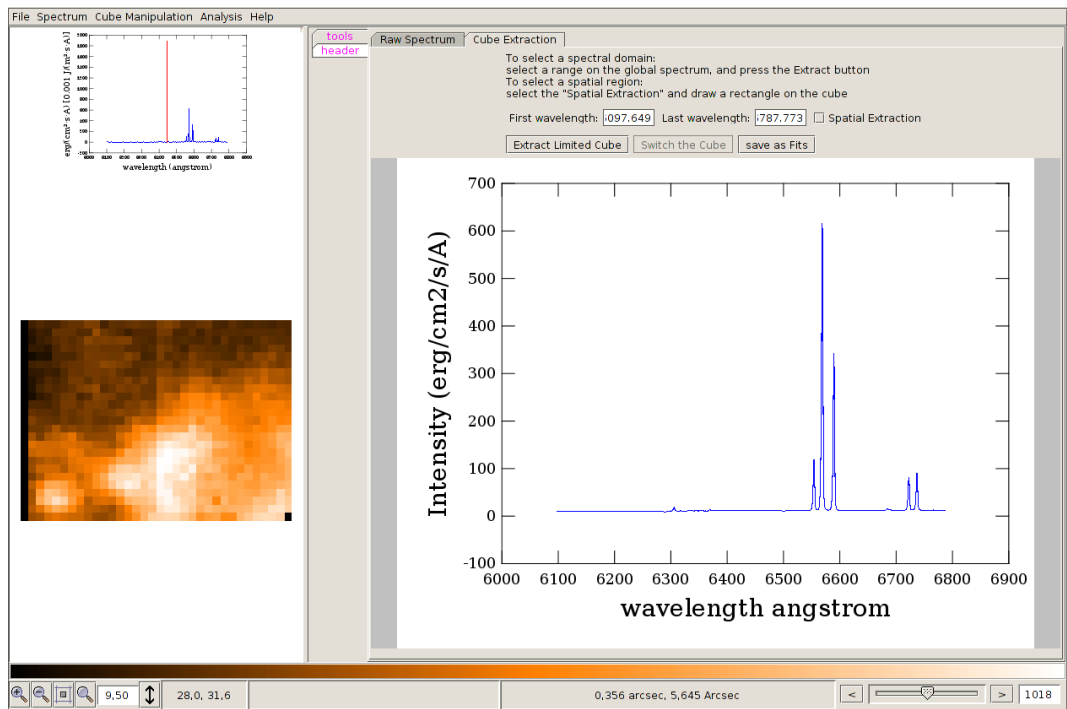


**Figure 5.21. The Range extraction window**

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).
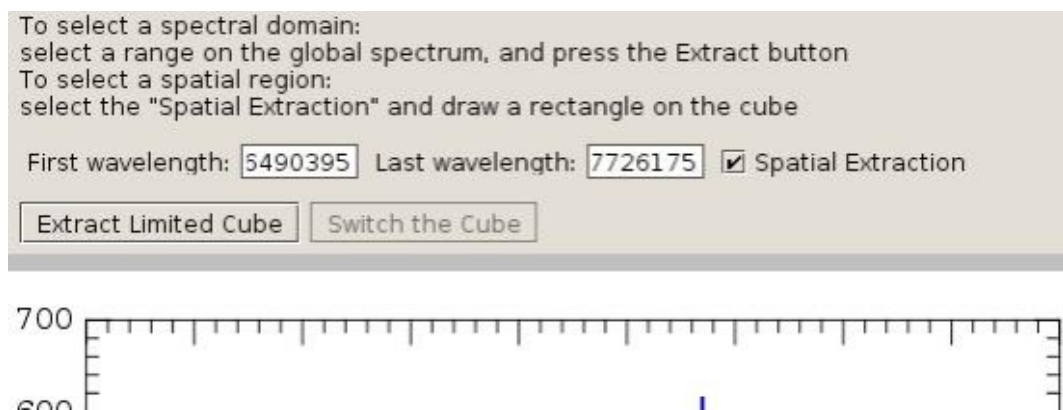


**Figure 5.22. The range of extraction and the radio button for the spatial extraction**

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.
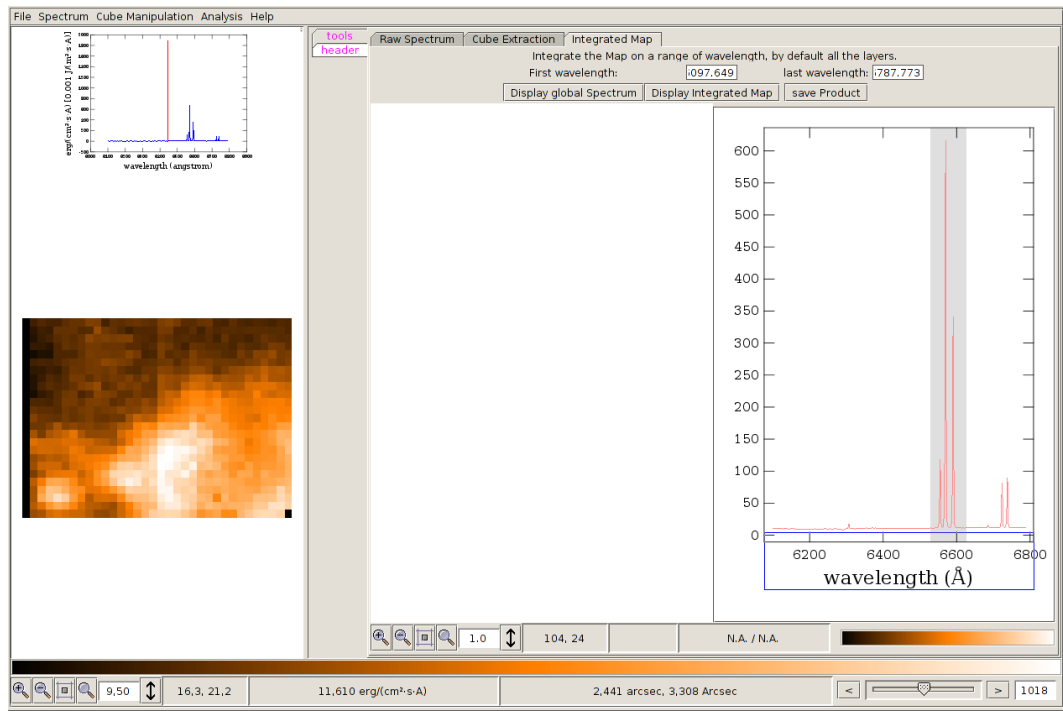
**Figure 5.23. The integrated map interface**

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
```

# 5.2.8.4. 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.
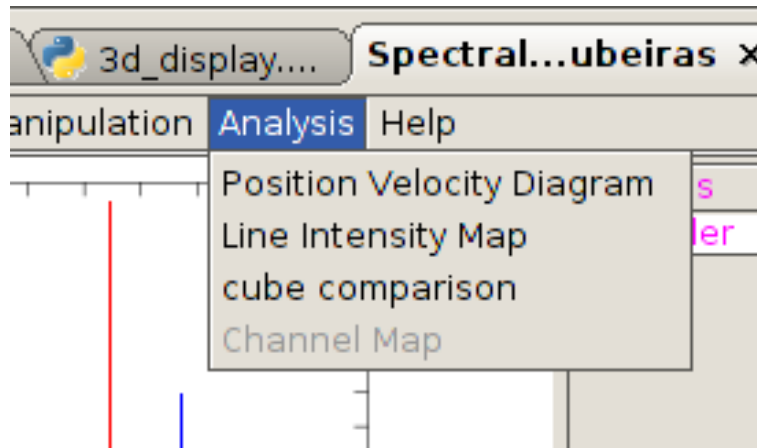


**Figure 5.24. The analysis menu**

## 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.
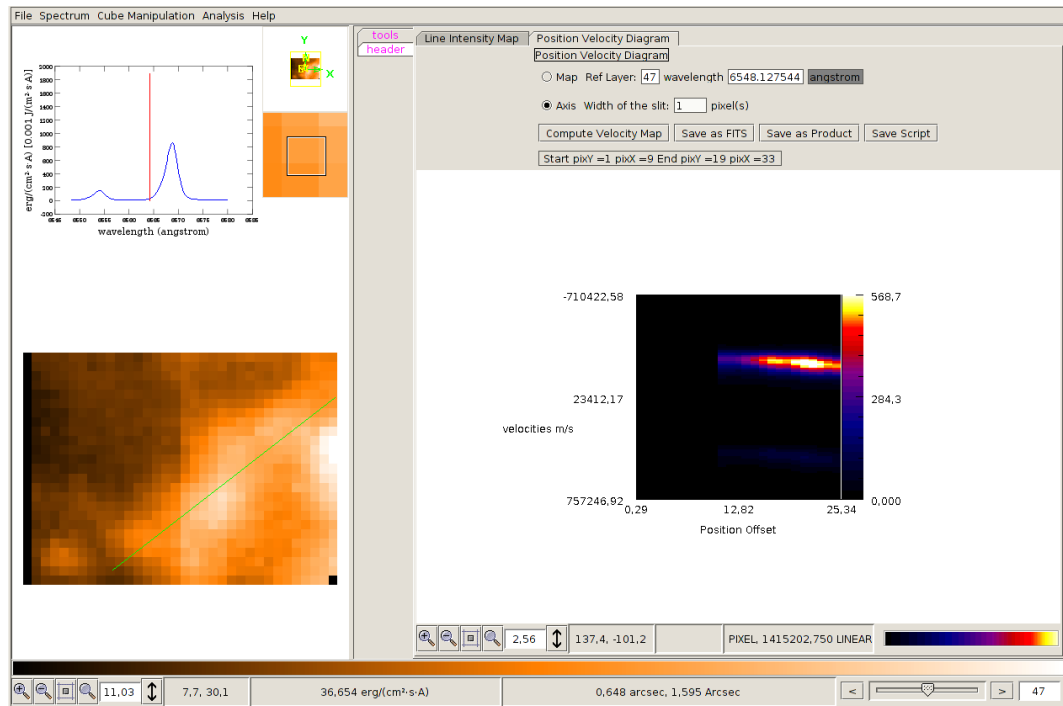
**Figure 5.25. Position-velocity diagram**

**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.

**Remark**s:

• 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.
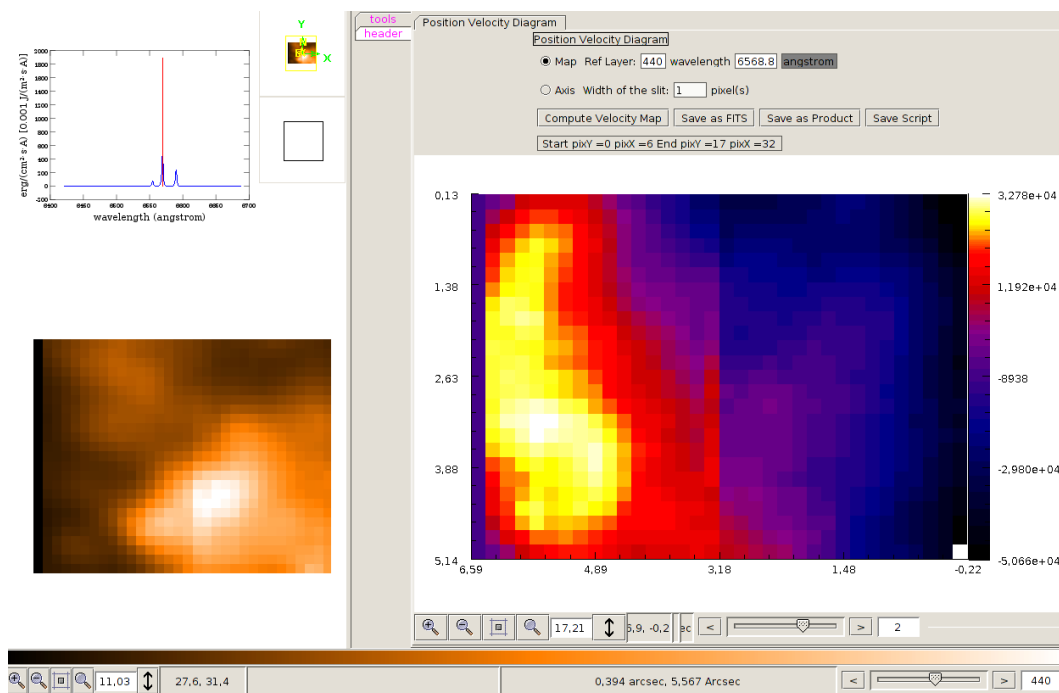


**Figure 5.26. Velocity map**

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
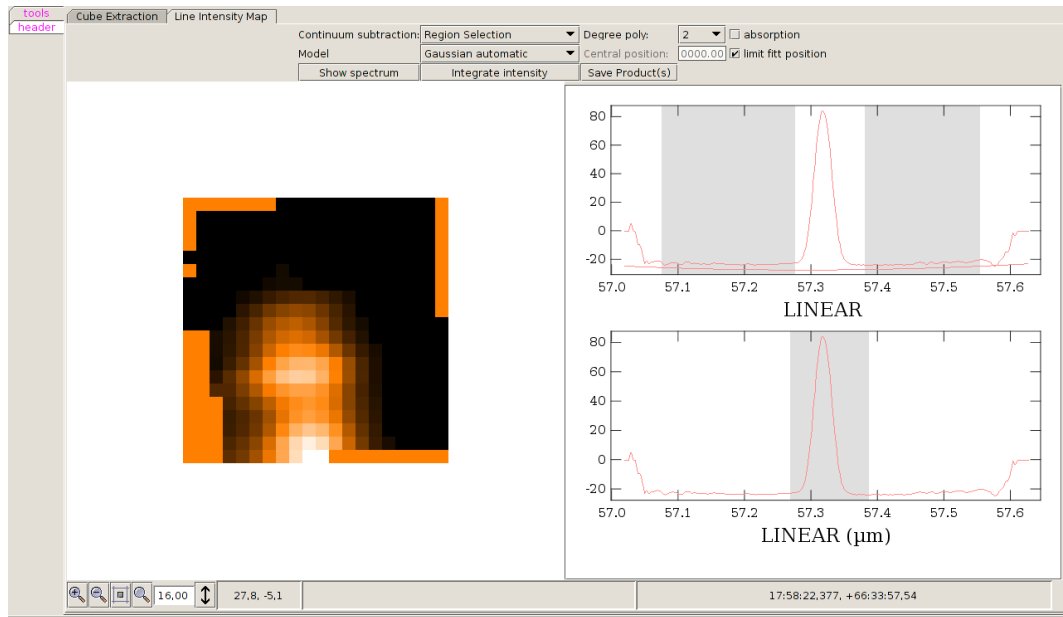
The computation is done by clicking on the "Integrate Intensity" button, and the integrated image is displayed in the left part of the GUI.

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.
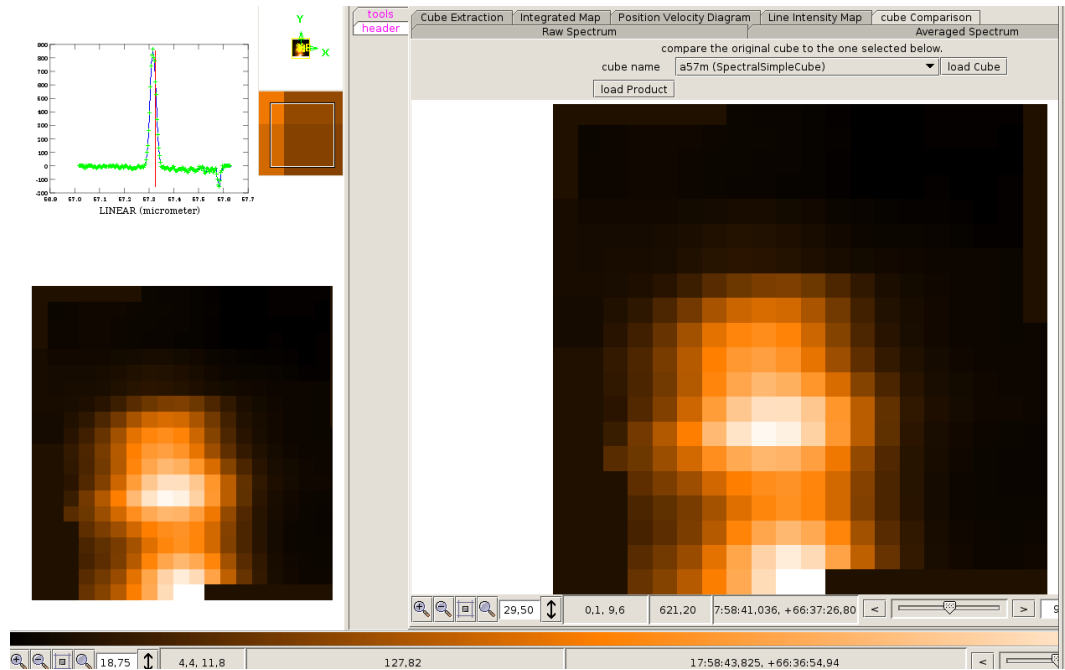
**Figure 5.27. The cube comparison GUI**

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.

# 5.2.9. 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.

## 5.2.9.1. 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 |

# 5.2.9.2. 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.

# 5.2.10. SpectralSimpleCube panel for the spectrum explorer

## 5.2.10.1. 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:
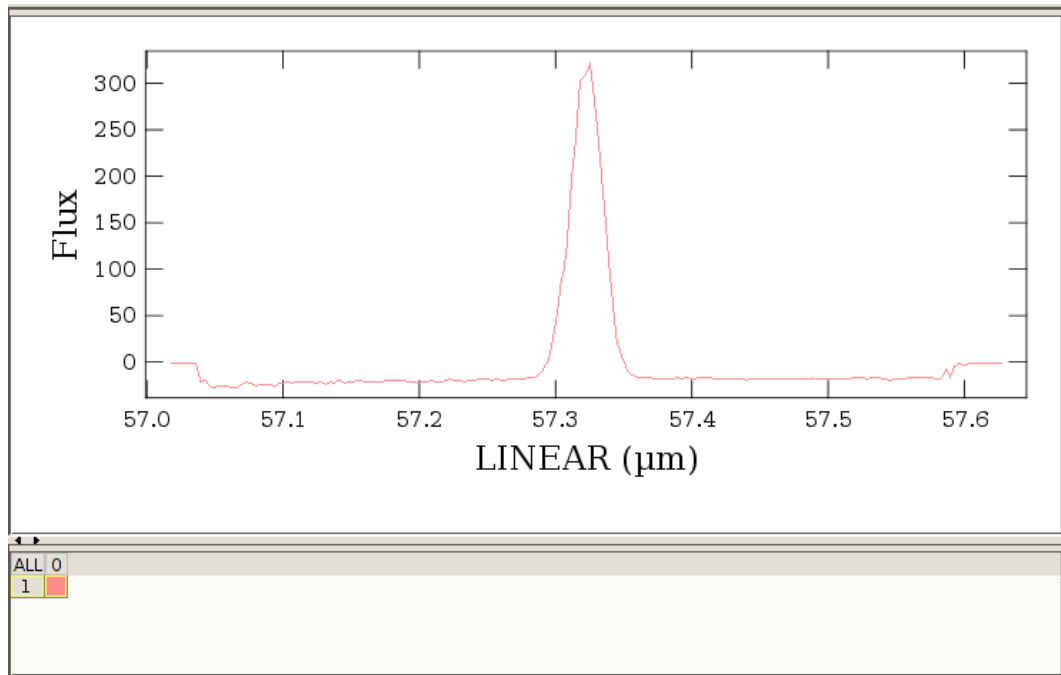


**Figure 5.28. The graphical interface of the spectrum explorer**

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.

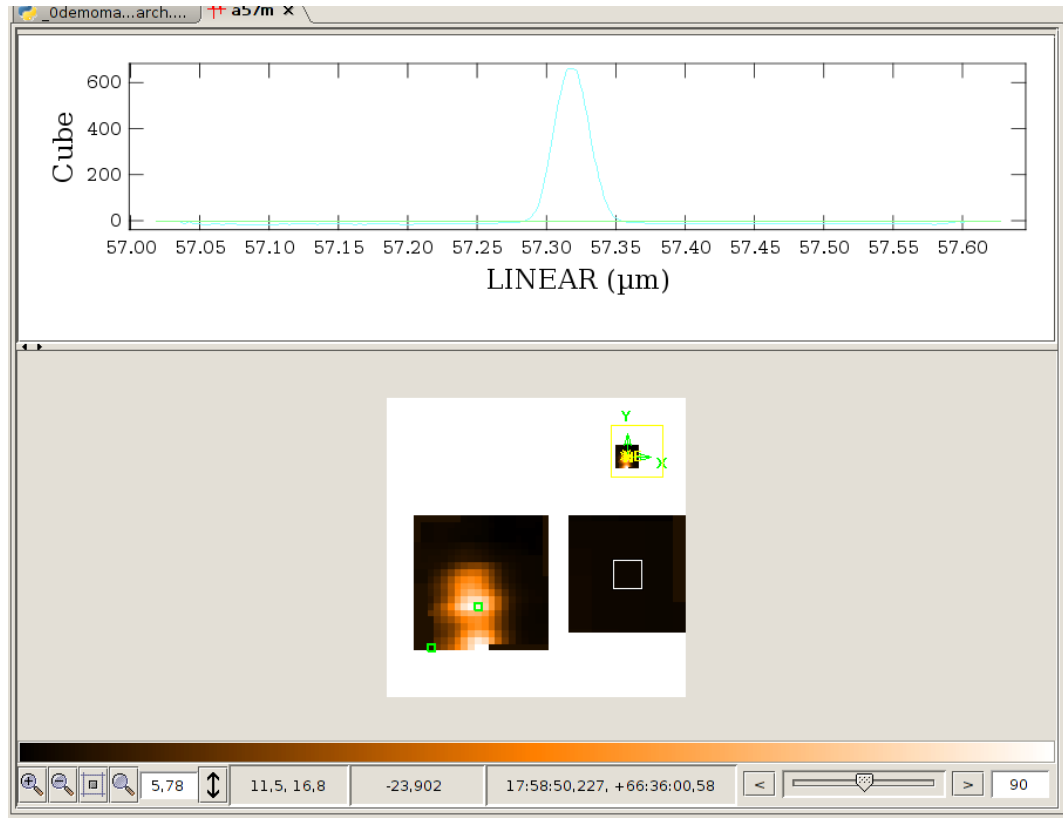The following figure shows the panel for SpectralSimpleCube.



**Figure 5.29. 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.

With these two modes you can select a rectangle and then remove pixels from inside it, thus creating a selection with *holes*.

# 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)
```
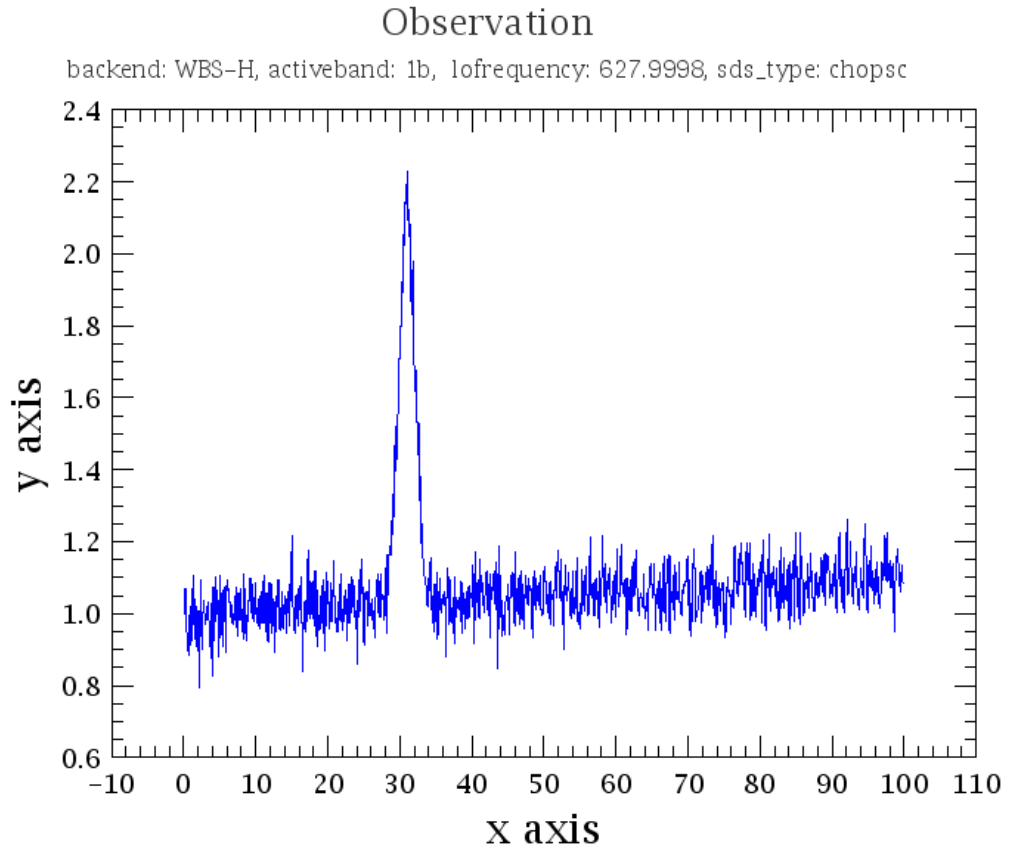
Assume that the data looks as shown in Figure 5.30.

**Figure 5.30. 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 | $f(x) = a_0 \, exp\left\{\dfrac{-(x-x_0)^2}{2s_0^2}\right\}$ | $a_0$ = amplitude of line | `sf.addModel` |
| | | $x_0$ = location of line peak | `('gauss',` |
| | | $s_0$ = width of line (sigma) | `[a0,x0,s0])` |
| Lorentzian | $f(x) = p_0 \left[\dfrac{p_2}{(x-p_1)^2 + p_2^2}\right]$ | $p_0$ = amplitude of line | `sf.addModel` |
| | | $p_1$ = location of line peak | `('lorentz',` |
| | | $p_2$ = half width at half maximum of line | `[p0,p1,p2])` |
| Polynomial | $f(x) = c_0 + c_1 x + ... + c_n x^n$ | n = order of polynomial | `sf.addModel` |
| | | $c_0 .. c_n$ = polynomial coefficients | `('poly', [n],` `[c0,c1, ...,` `cn])` |

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.
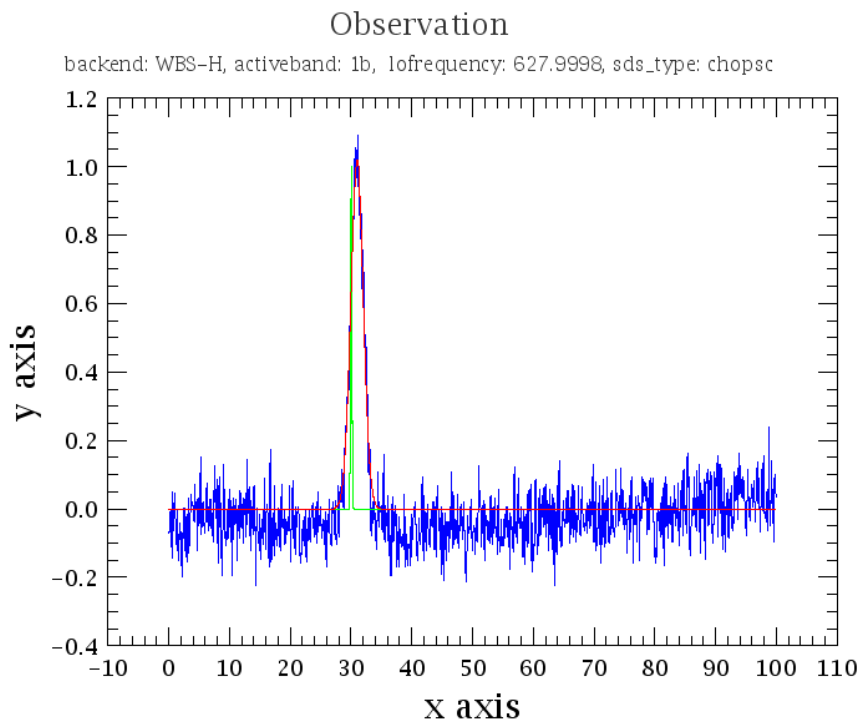


**Figure 5.31. Fit result. Fit results for spectrum**

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()**

Observation

backend: WBS-H, activeband: 1b,  lofrequency: 627.9998, sds_type: chopsc
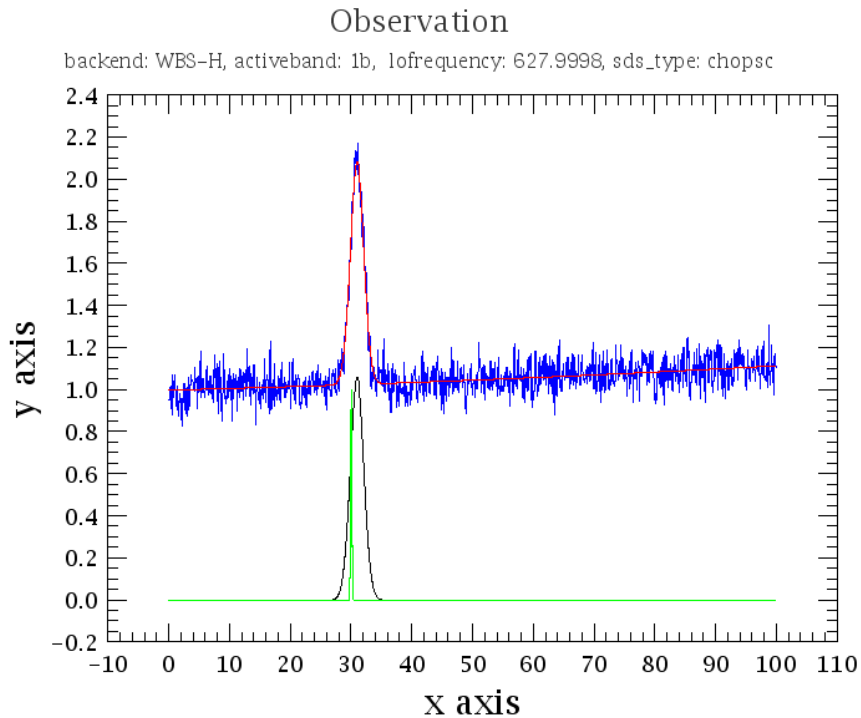
**Figure 5.32. Global fit. Use the models together in a global fit**

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

## 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.

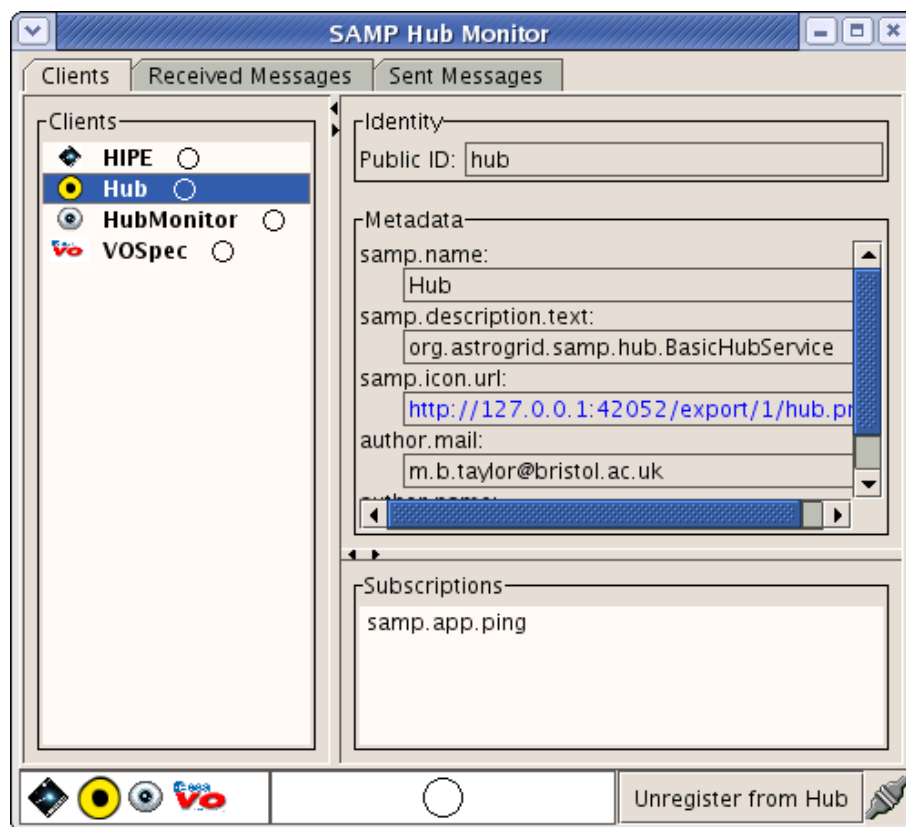When you select File → Interop → SAMP Hub Status the following windows appears:



**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.

# Appendix A. Demo scripts

This is a collection of many (but not all) of the available scripts all over the system. The collection is organized by package.

You can find these scripts in the `doc/ia/demo/scripts/` directory of your HIPE installation.

Note that the hyperlinks below do *not* work in the PDF version of this manual, but only in the browser-based version.

**Demo files**

| | |
|---|---|
| simple.py | Overview of Jython capabilities |
| help_demo.py | Demonstration of help facility |
| session_inspector.py | Demonstration of session inspector facility. |
| logging_demo.py | Demonstration of jconsole's message logging facility. |
| save_restore_demo.py | Demonstration of the save and restore of data feature in Jconsole. |
| numeric_whatisnew.py | Demonstration of how to use the new functionalities of the numeric library from Jython. |
| numeric_demo.py | Demonstration of how to use the 1D functionality of the numeric library from Jython |
| numeric_2D_demo.py | Demonstration of how to use the 2D functionality of the numeric library from Jython |
| numeric_reshaping.py | Demonstration of how to use the reshaping functionality of the numeric library from Jython |
| numeric_shifting.py | Demonstration of how to use the shifting functionality of the numeric library from Jython |
| numeric_slicing.py | Demonstration of how to use the slicing functionality of the numeric library from Jython |
| convolution_demo.py | Demonstration of how to use the convolution functions in the numeric library |
| fit_demo1.py | 1) Demonstration of how to perform fitting from the numeric library |
| fit_demo2.py | 2) Demonstration of how to perform fitting from the numeric library |
| fit_demo3.py | 3) Demonstration of how to perform fitting from the numeric library |
| fit_demo4.py | 4) Demonstration of how to perform fitting from the numeric library |
| fft_demo.py | Demo of FFT functionality |
| boxcar_demo.py | Boxcar filtering demo |
| gaussian_filter_demo.py | Gaussian filtering demo |

| | |
|---|---|
| [interpolate_demo.py](#) | Demonstrates iterpolation |
| [matrix_demo.py](#) | Demo of matrix functions |
| [dataset_demo.py](#) | Demonstration of how to use datasets and create products |
| [ascii_demo.py](#) | Demonstration of Import/Exporting of ASCII tables. The data file `ascii_demo_data.txt` is also required to run this demo. You can find it in the `data/ia/demo/data` subfolder of your HIPE installation. |
| [fits_demo.py](#) | Demonstration of Import/Exporting of FITS data |
| [imageExample1.py](#) | Demonstration of general image functionality. The image file `ngc6992.jpg` is required to run this demo. You can find it in the `data/ia/demo/data` subfolder of your HIPE installation. |
| [imageExample2.py](#) | Shows how to create an image from a simple numeric 2d array. |
| [task_example.py](#) | Demonstration of how to write a task |
| [task_array.py](#) | Demonstration on how to pass an array to a task. |
| [task_stop.py](#) | Demonstration on how to stop a task. |
| [TestPlotXY.py](#) | Demonstration of the new PlotXY capabilities |
| [TestAxis.py](#) | Demonstration of how to use PlotXY Axis |
| [TestAnnotation.py](#) | Demonstration of how to use PlotXY Annotations |
| [TestStyle.py](#) | Demonstration of how to use PlotXY Styles |
| [TestCompositePlot.py](#) | Demonstration of how to compose plots |