ATSAL Specification

v0.6, 25-Sep-2013

by Randall K. Smith and Thomas H. Kent

ABSTRACT

ATSAL is an interactive application for performing astrophysical spectral analysis, implemented for Macintosh and Linux operating systems. It is layered on top of existing, proven tools, especially XSPEC along with many others. It enhances the productivity of those tools by improving concurrency and ease of use. ATSAL also supports LBA (Line-based Analysis), in conjunction with atomic databases. It employs a notebook metaphor to capture the steps of an analysis and repeat them later or apply them to new data. Notebooks also serve as exchange media, project data repositories, and archives. ATSAL simplifies creation of publication-ready graphs. It supports Python as an extension language.

Contents

Figures

Tables

1 Overview

ATSAL is an analytical tool for performing spectrographic analysis of luminous sources in the universe. More correctly, it is a pretty face for existing tools for this purpose, with a few new tricks of its own. It is layered on top of existing tools, principally XSPEC; also Python, TGCat and other archives, HEASARC, AtomDB, various libraries, and other optional analysis tools.

An ATSAL notebook stores analysis steps, displays results, exports publishable data, and allows you to share results with others. It minimizes programming but does not attempt to avoid it entirely. Users select analysis tasks from a list of templates and extend them as needed for their own purposes. ATSAL does most of the work via the user interface.

ATSAL is an exploratory tool. The graphing and analysis modules are designed to allow users to experiment quickly with different approaches until they settle on an approach. ATSAL also addresses publication. Users can create multiple graphs and tables and set each to illustrate a particular result, then export publication quality results.

ATSAL is, in principle at least, neutral about the portion of the electromagnetic spectrum in which users work, and the types of sensors supported for analysis. For example, it is designed to interface to multiple analysis packages. However, given substantial variations in analytical techniques and in sensor properties, early releases are focused on X-ray and γ-ray analysis, and for the sensors in use on ASTRO-H.

A common problem among analytic tools and open source software in general is version skew—incompatibilities resulting from mixing different versions of necessary tools. ASTAL confronts this issue directly, by bundling most external dependencies into a single package, by issuing updates infrequently, and by taking more explicit control over integration with dependencies. It also bundles user-written extension programs with notebooks to create notebook archives, so that colleagues can reproduce results consistently.

ATSAL also cures warts and reduces cholesterol.

1.1 Documentation Conventions

All user interface drawings are schematic.

[THK: This Remark style, together with the remarker's initials, mentions issues that reviewers may wish to comment on. These entries will be removed as they are resolved. Use this or Word's comments, whichever you prefer.]

Design Note

This brings attention to design considerations and typically remains in the document.

1.1.1 Priorities

Circled numbers are sometimes used to indicate the relative priority of a feature. ① means "first release goal." ② is proposed for second release, and ③ is a possible future.

1.1.2 Class Names

In order to cluster related classes together in alphabetical lists, class names start with their most abstract properties and work downward, the reverse of many naming schemes. For example, class Plot has subclasses PlotScatter and PlotSpectrum rather than ScatterPlot and SpectrumPlot.

To keep class name length a bit more manageable, the more abstract portions of class names are abbreviated. For example, class GraphLayer has subclasses GLTickMarks, GLSpectrum, and GLLabels.

Coding and naming conventions are described in a separate document, *ATSAL Coding Conventions.*

2 Design Goals

ATSAL overall goals:

- Support line-based analysis (LBA)
- Increase analysis speed
- Reduce learning time for new users
- Simplify sharing and reuse of analyses
- Simplify publication
- Use existing resources efficiently
- Support advanced users
- Design for longevity

Each of these goals is broken down further below.

Line-based Analysis (LBA)

- Support interactive tools to isolate features of interest
- Develop fitters that label emission or absorption lines or line groups
- Use data from continuum fitters to improve LBA

Increase Analysis Speed

- Allow interactive manipulation of graphs and overlays
- Take better advantage of parallel processing by running multiple copies of XSPEC on local or remote computers
- Use multiple copies of XSPEC for work on multiple models at the same time, or for generation of long-running results without interrupting work in other areas
- Avoid parameter numbering
- Work while analyses proceed in the background
- Display as much context (e.g. parameters, graphs, results, online help) as desired at the same time
- Use cut and paste to preserve previous lines of inquiry and develop new ones
- Reduce "modality," allowing user to work on multiple models at once, calculating minimum XSPEC commands automatically
- Resume a previously saved analysis without starting over

Reducing Learning Time

- ATSAL is a true graphical user interface, not a programming language.

- Supply annotated examples that illustrate various analyses, from downloading files to producing publication-ready results
- Examples supplied with ATSAL also serve as starting templates for new work
- Cut and paste from multiple examples or colleagues' work
- Apply an analysis to new input files from different instruments and sources, adjusting rather than starting from scratch
- Permit simple use of Python "glue code" as well as more advanced use of Python for custom analysis

Simplify Sharing and Reuse

- "Notebooks" include annotations, results, information needed to download files, custom Python code, etc.
- "Archives" are even more complete, and include other resources (e.g. libraries, PHA files, documentation files), providing a complete record that can be used to recreate or share an analysis
- Annotations, models, inputs and outputs, associated artwork or PDF files are all included in notebooks, providing a clear record of the analysis procedure

Simplify Publication

- Provide extensive control over graph generation, style details, and annotations to move more easily from the analysis phase to preparing publication-ready artwork
- Hide "cosmetic" controls until needed
- Export all publication artwork in a single operation
- Export complete sets of artwork for each of multiple input files
- Use Postscript quality art for both screen and publication
- Notebooks preserve all or part of the analysis, with annotations, making it easier to prepare papers

Use Existing Resources Efficiently

- Complement existing tools (primarily XSPEC) while imposing few constraints on their development process or time frame
- Maintain clear line of demarcation between analytical engines such as XSPEC and the user interface layer (ATSAL)
- Design to allow easier incorporation of other tools (e.g. Sherpa, ISIS, other atomic databases, other data archives)
- ATSAL will conform to the spirit and development standards employed by other HEASARC projects

Support Advanced Users

- Python supports simple glue code, advanced analysis operations, or (eventually) full-fledged user interface development
- Independent process model may be extended to incorporate other external resources such as DS9, XSTAR, or Sherpa
- User models added to XSPEC are available in ATSAL as well
- Matplotlib, numpy and probably astropy is included with ATSAL

Design for Long Life

- Use well-established, popular languages (C++ for performance-sensitive internals, Python for user extension)
- Use well-established tools (e.g. Qt for user interface, Python tools widely used in astrophysics community, HEASARC tools)
- Include regression tests, bug tracking
- Address version skew issues explicitly

3 Architecture

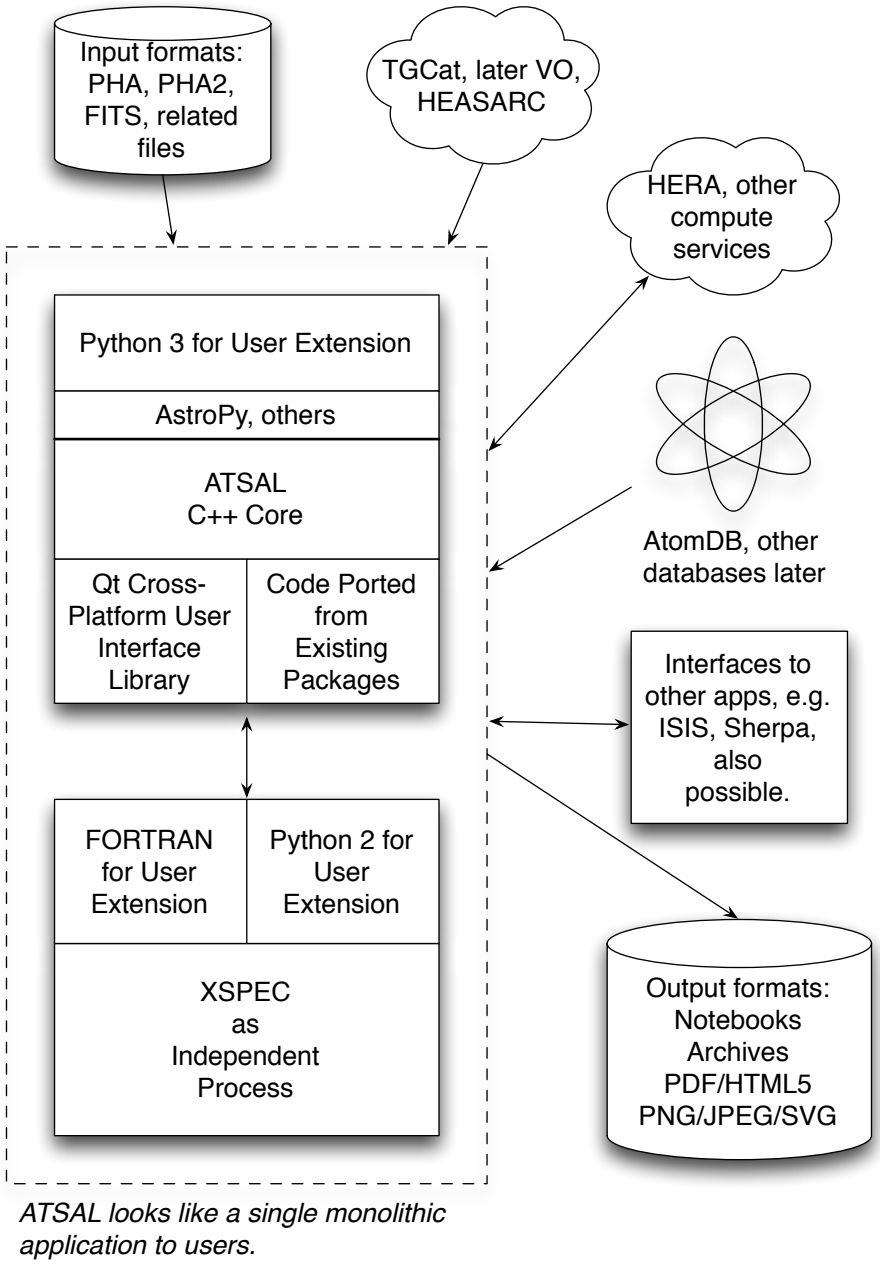


Figure 3-1. ATSAL architecture

ATSAL is heavily dependent upon XSPEC, which does most of the analytical heavy lifting, yet it is "loosely coupled" to XSPEC: the two programs run as independent processes and communicate via sockets. This is a central design issue. Linking XSPEC directly into ATSAL would improve performance by eliminating the interprocess communication bottleneck, and would make available math libraries that are otherwise lacking.

But we are reluctant to link them together, for the following reasons:

- ATSAL is, in principle at least, neutral about the energies for which it may be used and the analysis package(s) it employs, and will communicate with other analysis packages in the future. Linking multiple class libraries with partially overlapping capabilities into ATSAL would create formidable data conversion and maintenance problems. Loose coupling somewhat minimizes these problems.
- XSPEC is mostly single-threaded, so it cannot fully exploit processor power present in today's multicore computers. With loose coupling, though, it is possible to run more than one instance of XSPEC at the same time, taking somewhat better advantage of multiple processors.
- With loosely coupled processes, it is also possible to run the XSPEC processes on remote computer(s).
- The user can continue to make changes in the user interface while XSPEC is compute-bound.
- Communications between ATSAL and XSPEC are handled using a driver that make's XSPEC's C++ interface available remotely, over a socket connection. Because ATSAL developers maintain this driver, we anticipate fewer version skew issues. The driver also ensures that all necessary information is exchanged between the processes. (For example, floating point values are exchanged without any loss of precision due to rounding for display.) Hence XSPEC and ATSAL development schedules do not need to be synchronized, and XSPEC need not make any special allowances for ATSAL.
- ATSAL's code base is smaller and more maintainable when independent processes do most analysis functions.
- Since ATSAL includes a Python interpreter, a large code base of Python code, the extensive Qt library, plus the ATSAL application itself, overall reliability is a concern. ATSAL and XSPEC are complex enough to benefit from mutual isolation.

In addition to XSPEC, ATSAL can communicate with external web servers such as TGCat, compute servers such as HERA, other utility processes, and, in time, other analysis packages such as Sherpa and ISIS. ATSAL appears as a single monolithic process to users.

4 Notebooks

A notebook expresses an analysis procedure in ATSAL. It has these basic goals:

- To record a procedure in a way that others can understand and reproduce.
- To provide exploratory tools for developing a sound analysis.
- To reduce the need for, and complexity of, programming.
- To be able to repeat the analysis with the same or new inputs.
- To provide publication-quality results.

The notebook window consists of up to three panes. All the panes are shown below, but by default, only the notebook pane *(top right)* is displayed. The files pane is used in more complex notebooks, to manage other resources that may be part of a project. The helper pane displays information used to debug notebooks.

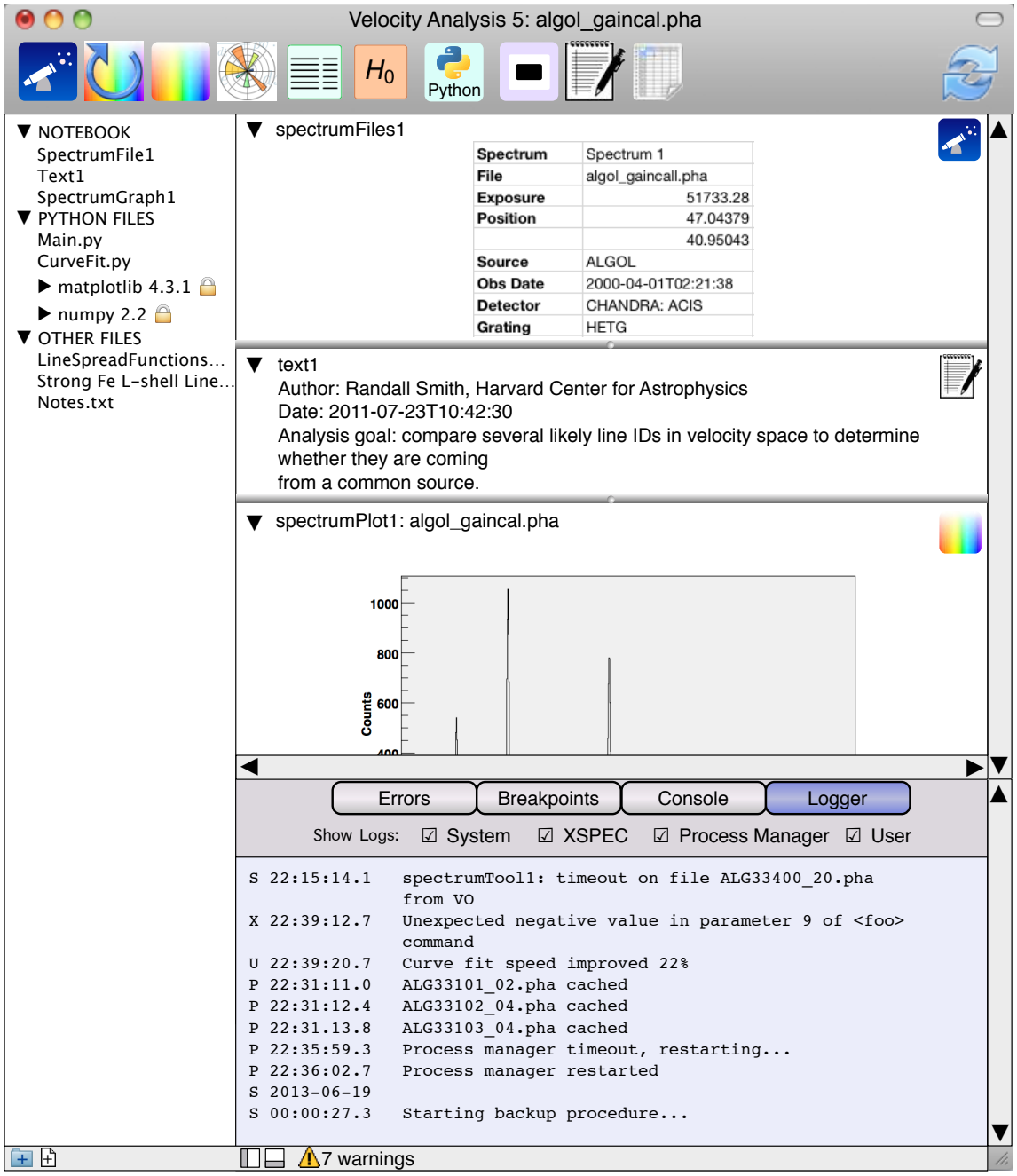


Figure 4-1. ATSAL notebook window with file and helper panes displayed

A notebook is a visual representation of an analysis procedure. Users add tools to the notebook to specify a sequence of file selection, analysis and display actions. Each tool can be collapsed to a name and comment, expanded to a pane in the notebook, or opened as a freestanding window for editing.

A tool represents an analysis step. Simply connecting tools together performs many tasks. Internally, though, each tool is an object, also accessible from a Python program. This provides an escape hatch for more advanced applications.

Some tools:

Tool	*Description*
Observatory	Downloads extractions from web archives and stores them locally.
Spectrum Files	Selects a subset of files from the local library.
Spectrum Plot	This tool plots raw spectra, theoretical spectra and continua, and various fits and other data. Users create selections used as input to analyses. Overlays exist in independently manipulated layers.
Matplotlib	Front end for matplotlib Python-generated plots
Model Parameters	Displays parameters for a model, for user entry.
Fit Results	Displays a table showing results for a fit.
Python Tool	Creates a Python program fragment.
Image Tool	Inserts an image as an annotation.
Text Block	A block of text that serves as annotation of an analysis procedure, presentation of results, or some other form of documentation. Supports variable substitution.
Table	Creates a table, a 2D array of strings. Users can enter data manually, or display arrays or other program results. Tables may also be embedded in text blocks.

Table 4-1. ATSAL Tools

The basic sequence of an analysis procedure is as follows:

- Download extraction packages (files representing observations) that may be of interest.
- Search the downloaded packages and select specific extraction(s).
- Create a graph tool and display the file.
- Use the spectrum plot tool as a "model explorer," creating and adjusting various models and assessing their suitability.
- Add parameter blocks, results tables, generic tables, bits of Python, text blocks, etc., as needed to document or simplify the analysis.
- Create special-purpose plots using the matplotlib tool, or fine-tune the existing plots to prepare them for publication.
- For more advanced applications, add custom Python modules to the notebook.
- Decide which steps should save files during each analysis.
- Prior to running a completed analysis, create a new session, which simply places logs and exported files in a new, time-stamped directory.
- Run the entire analysis to produce all the exported results.
- When the analysis is complete, optionally save a notebook archive, which includes information necessary to reproduce the analysis, as well as optional file components.

- If desired, select new input files and apply the same analysis to them.

4.1 Notebook Pane

The notebook pane, which fills the entire window by default, normally displays the notebook. It may also be used to display the current Python source file, when Python programs are being edited.

In the example below, a simple notebook obtains an input spectrum and displays it.

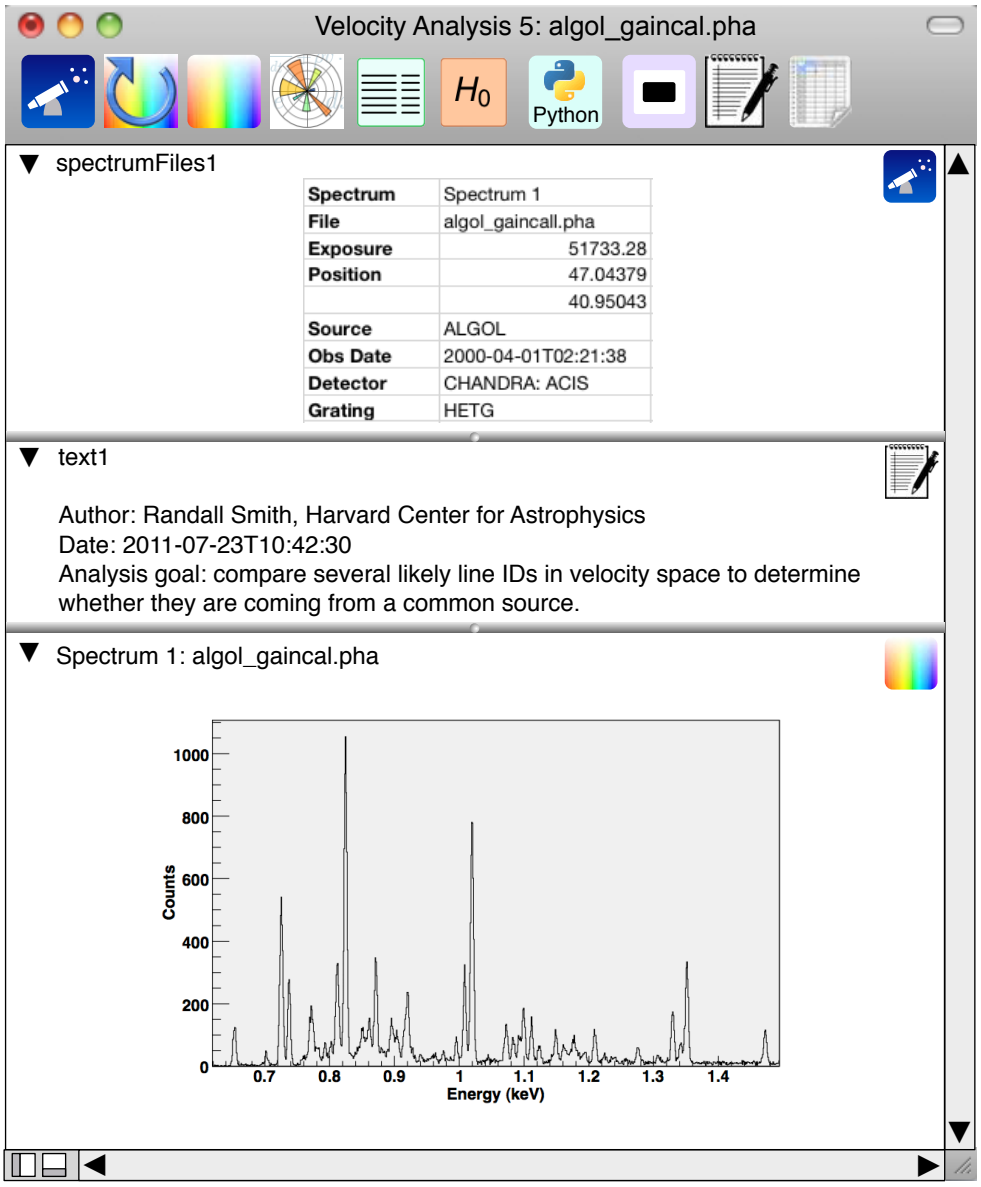


Figure 4-2. Notebook pane

Tools are shown along the top of the window. As more tools are added, these will probably become drop-downs with additional choices. Clicking a tool inserts a block into the notebook that displays a view of the tool. Some tools, like graphs,

expand and contract as pane height changes; others may behave differently. When printed, most tools are not broken across page breaks, but long text blocks or tables will be. The maximum height of a tool as displayed in the notebook is a single page, though at print time text and table blocks are extended to show their entire contents.

4.1.1 Notebook Palette

The notebook may be collapsed into a more compact form called the notebook palette, to reduce screen clutter. It may also be minimized.

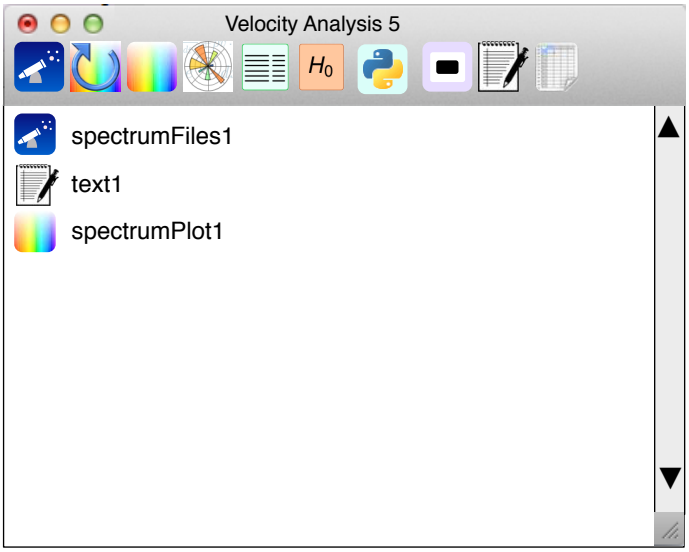
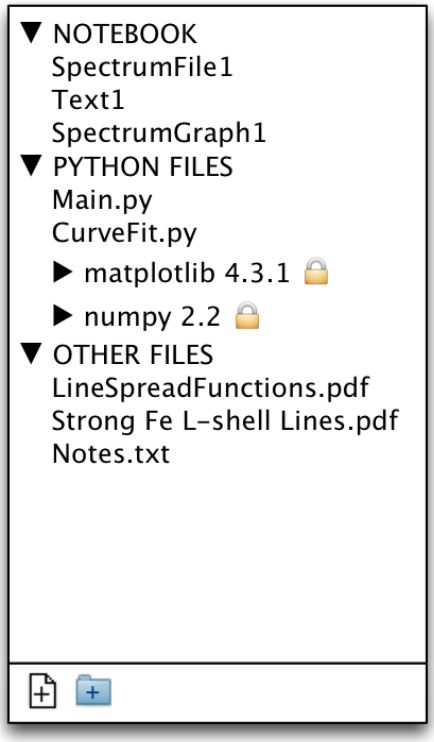


Figure 4-3. Notebook palette

4.2 Project Files Pane

An optional pane on the left side of the notebook lists files that are part of the notebook.



NOTEBOOK lists tools in the notebook. Selecting a tool scrolls to it. Double-clicking opens the tool editor.

PYTHON FILES lists files that are part of this analysis. There is always a Main.py, and others may be added.

OTHER FILES contains files that provide additional documentation or notes. These files do not play a role in the analyses.

Figure 4-4. Project files pane

New folders may be created, under each main heading, and files rearranged into hierarchies. For files stored within the notebooks, these "virtual folders" correspond to real folders. Only files within the project directory tree may be rearranged into subfolders—external file hierarchies are considered read-only.

New file creates a new file that is placed under Python Files if the file type is ".py", or under "Other Files" if the file type is not ".py". If there is no current selection in the files list, the new item is added to the end of the list. If there is a selection, it is added after the selected item.

New folder creates a folder after the selected item, which may be any type of item. If there is no selection, the new folder option is disabled.

To add files or folders to the notebook, the user drags the file(s) or folder between any two existing items in the files list. Items are rearranged in the list by dragging.

When a file or folder is added to the Python Files or Other Files section, the user may choose between copying the files into the notebook, in which case they are writable, and saved with the notebook. Or they may be left in their original locations so they are easily shared with other notebooks. In the latter case, they appear with lock icons. The user must unlock a file prior to editing it. An unlocked file is copied into the notebook, where it is modified locally. Users wishing to export the modified file back to the shared area must do so manually.

Any file that appears with a lock icon is stored outside of the notebook, and is not saved with the notebook. However, such files are saved by default with notebook archives.

Users delete files and folders by selecting them and pressing the delete key.

There is an icon to display or hide the files pane.

4.3 The Helper Window

The helper window may be displayed as a pane at the bottom of the notebook, or as a separate window. The pane has several tabs, described in the following sections.

Normally the helper pane is only displayed when the user requests it. However, if Python errors are detected, the window appears automatically, and is set to error mode.

[THK: Since errors are only detected when they are encountered, not ahead of time, perhaps the Errors and Console panes can be merged.]

4.3.1 Errors

The errors pane shows errors in Python syntax. ①

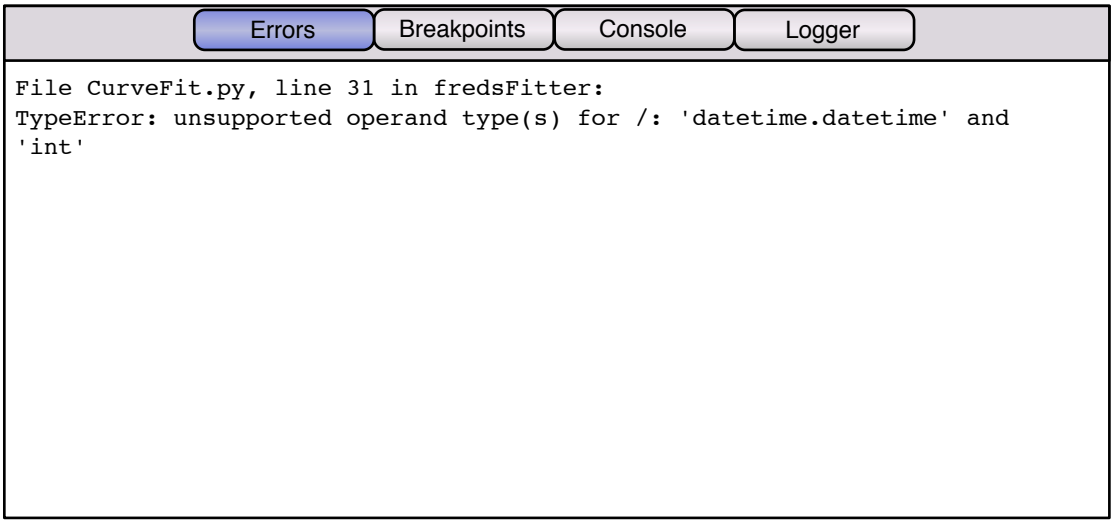


Figure 4-5. Errors tab in helper pane

Single clicking an error message displays the offending Python source file in the notebook window, and scrolls to it. Double-clicking displays it in a separate window. Errors are updated each time the notebook is run, or when Check for Errors is chosen from a menu.

4.3.2 Breakpoints Pane

The breakpoints pane lists all breakpoints. They may be enabled or disabled by clicking their checkboxes. Breakpoints are removed by selecting one or more and pressing delete. ②

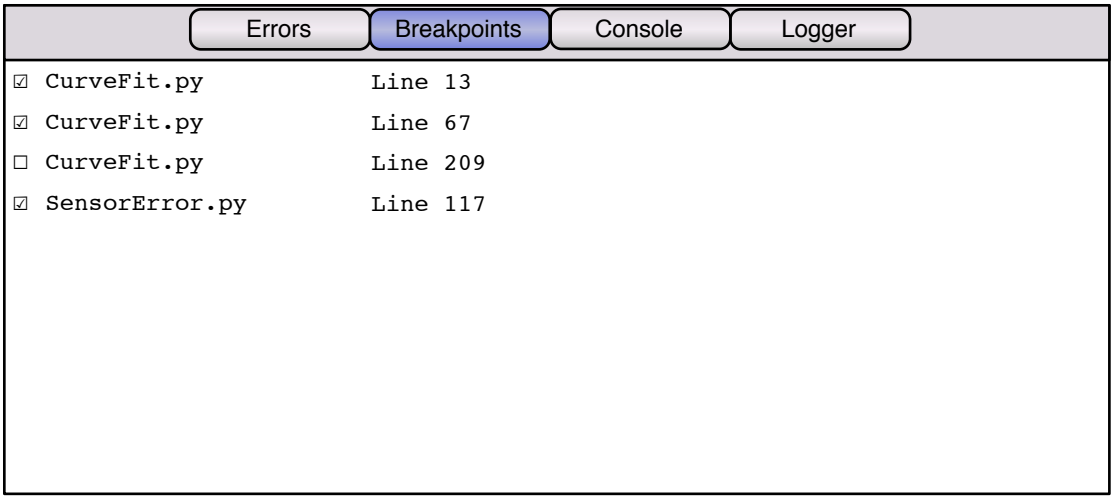


Figure 4-6. Breakpoints tab in helper pane

4.3.3 Console Pane

When a breakpoint is hit, the console is used to explore values in the running program, by typing print statements. Users are cautioned about calling functions, which could have side effects that change the execution path of the notebook. ②

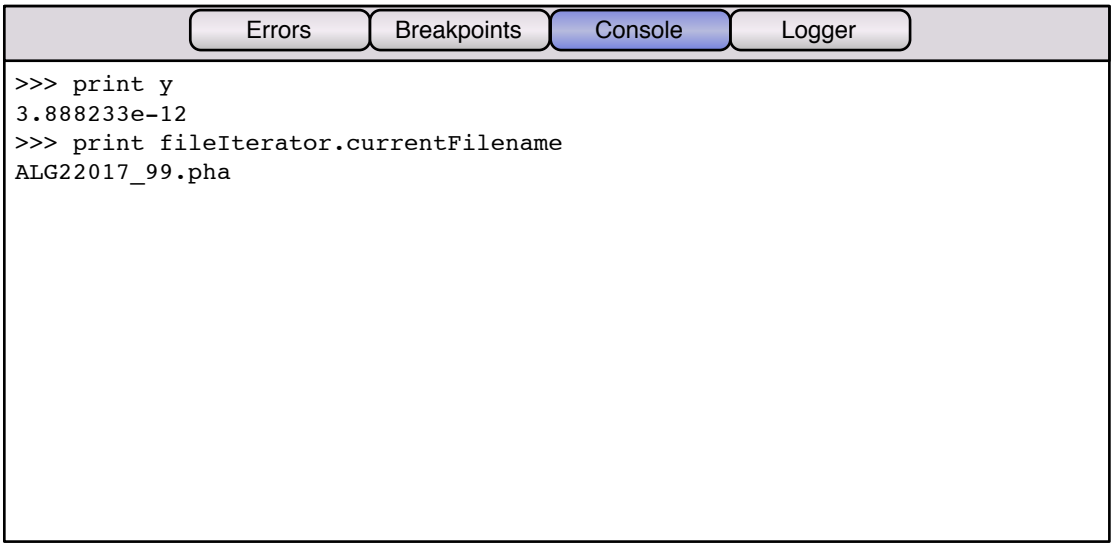


Figure 4-7. Console tab in helper pane

4.3.4 Log Pane

The log pane is mostly of value to ATSAL developers, but it can help users debugging Python programs too. Log entries are time stamped in GMT time. At day boundaries, a date stamp is inserted. Data may be shown from several sources, with each source marked by a single character at the beginning of each line. ①

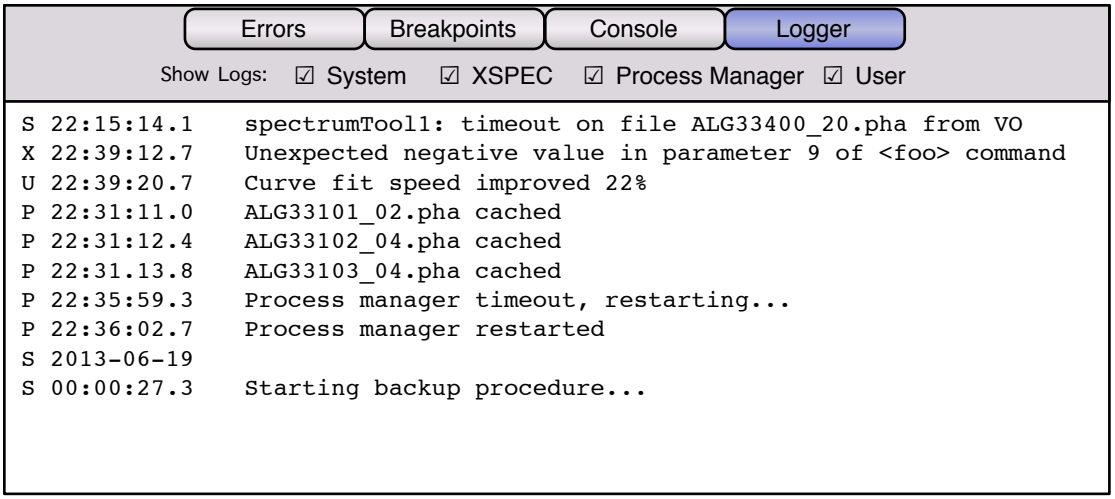


Figure 4-8. Log tab in helper pane

By default, only user information is logged. For example, the Python line:

```
print "Curve fit speed improved 22%"
```

produces the line starting with "U" above. If "System" is checked, system log information is shown. "XSPEC" shows commands to and results from XSPEC. "Process manager" shows status messages from the process manager. Logfile size is limited to a maximum. Individual logs are stored on disk, but the merged log format shown above is not stored.

4.3.5 The Refresh Command

The user interface does not fully update immediately in response to any user input, as is the case with many programs. Since a series of user interface changes sometimes cause redundant or superfluous compute-intensive calculations, ATSAL relies on the Refresh command to bring the program fully up to date. If changes in the notebook require corresponding changes in XSPEC, ATSAL issues the minimum necessary commands to XSPEC to complete the refresh.

XSPEC may encounter errors, warnings, or "convergence warnings" (the last a warning about fits that do not seem to be converging on a solution). Because a cascade of warnings may occur in response to a single user action, and because the same warnings may be repeated later as a side effect of other actions, ATSAL does not interrupt execution to present each warning, or even show them by default. However, the logger pane displays these warnings, and if the pane is hidden, an icon still appears to indicate the presence of warnings, along with a count, in the bar beneath the window. When a warning occurs, the icon highlights and the warning count goes non-zero.

The user can click the icon to display the helper pane in order to review the warnings, or click the warning icon to reset the warning count to zero.

Errors, on the other hand, abort the flow of XSPEC commands, and display a modal dialog reporting the error. The user resolves the problem and reissues the Refresh command when ready.

Design Issue

It will be easy—perhaps too easy—to fall into the habit of ignoring warnings. Randall suggests that we address this by permitting some warnings to be marked so they are always ignored, making the warning icon reflect only more serious warnings.

Convergence warnings will probably be handled as a special case in the user interface, with some visual indicator of convergence and perhaps also a graph of the fit itself as it progresses.

4.3.5.1 Python Errors

In addition to XSPEC errors, Python errors may occur. ATSAL will highlight the tool in which an error occurred, and direct user attention to the source of the error.

4.3.5.2 Refresh Single-step

Normally the Refresh commands issues all the XSPEC commands necessary to bring the user interface up-to-date. For debugging, the user can elect to issue these commands one at a time. Option-clicking (or alt-clicking) the Refresh icon issues a single XSPEC command, allowing for its results to be examined before proceeding.

Within a Python fragment, refresh single-stepping will (probably) single step through the code.

4.4 Notebook Global Operations

Multiple (perhaps limited to two) notebooks may be open at once. The operations listed below apply to the current notebook.

Action	*Result*
Tool icon	Add a tool after selected tool or at beginning if no selection
Delete tool	Remove a tool, by selecting it and pressing the delete key
Collapse or expand	Collapse/expand a tool pane by clicking the disclosure triangle.
Double click tool icon	Open a tool for editing.
Click boundary between two tools	Insert or remove a page break between tools.
Reorder tools	Drag one tool up or down, or do the same operation in the files list.
Print (menu)	Print a notebook.
Refresh	Requests any updated calculations necessary from external tools such as XSPEC; displays results.
Export Notebook (menu)	Export a notebook as a PDF or a Python file.
Export Notebook Archive (menu)	Select archive options and export a notebook archive.

Save Notebook (menu)	Save a notebook.
Open Notebook (menu)	Open a notebook.
Quit ATSAL (menu)	Quit ATSAL.

Table 4-2. Operations that apply to the current notebook

Debugging operations:

Run icon	Run a notebook.
Suspend icon	Suspend a run.
Abort icon	Abort a run.
Examine variables by name	
Examine variables by mousing over symbol names	
Set/clear breakpoints	
Restart	Reinstantiates all tools and reruns notebook. Equivalent to quitting and restarting ATSAL.

Table 4-3. Debugging operations

Note: there is no undo support for any of these operations.

4.5 Tool State

Objects representing tools in a notebook are instantiated as soon as they are inserted into a notebook (or when an existing notebook is opened). They remain in existence until the notebook is closed. Tool settings may be altered either interactively, or under control from a Python program. When a notebook is saved, current settings of all tools are saved, along with results calculated by XSPEC. Upon reopening the notebook, the saved results from the previous session are displayed. This makes it easy to open a notebook for review without waiting for extensive recomputation.

XSPEC's internal state is not saved, however, so it is necessary to reissue the commands that XSPEC previously executed to fully restore the saved state. This process commences as soon as the user uses the Refresh command, and may be time consuming. During this initial period of catching up, changes may be made in the user interface.

5 Models, Files, Selections, and Parameters

In an analysis package such as XSPEC, a *model* is an analysis function performed on one or more observed spectra. It accepts *observed spectrum files*, related files such as *response matrix files* and *ancillary response files,* and a series of *parameters* that guide its operation, and produces a fit, as well as summary data. Each parameter is a set of one or more values that may specify a scalar value, a range, or other related group of values. Multiple models are often combined to produce a composite fit that better approximates the physical processes that account for the observed spectra.

In ATSAL we formalize the concept of *selections,* which in their purest form represent a range of energies. In practice, selections are usually associated with one or more observed spectra, and represent ranges in bins. Models that operate on observed spectra usually operate as well on subsets of spectra expressed as selections. *Selection sets* are groups of selections that may be used as shorthand for listing each selection separately, again to restrict the operation of a model to a series of ranges within a spectrum file.

5.1 Models and Parameters

ATSAL does not compute models; it simply manages their inputs and outputs for the user. Hence an ATSAL model is composed of input files or selections, parameters, and results. A *basic model* is a model supplied by XSPEC or another analysis package. A *compound model* combines several basic or compound models, and is specified as an expression that shows how they are combined.

A model or compound model is like a class in a programming language: it acts as a template. Users create *model instances* (akin to objects) within ATSAL—a model instance is a model and a set of associated parameters. To distinguish between models and model instances, we use the same naming convention Python uses for classes and objects: models (classes) begin with an uppercase, and model instances (objects) begin with a lowercase. Model instances are sequence-numbered with an appended sequence number by default. (For this reason, it is strongly recommended that new models should *not* be sequence numbered. If some sort of sequencing is desired, use sequence letters instead.) Each model instance begins life with default parameters inherited from the model. Model instances are separate entities: changing a parameter in one does not affect another one.

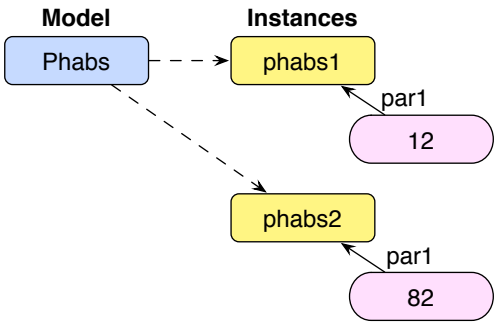


Figure 5-1. Model instances phabs1 and phabs2 have separate parameters

In this example, the user created two model instances, phabs1 and phabs2, from the basic XSPEC model Phabs. Each starts with the same set of default parameter values, but a change to a parameter for 1 has no effect on the other.

To share a single parameter intentionally between multiple model instances, the user supplies a variable name for it.

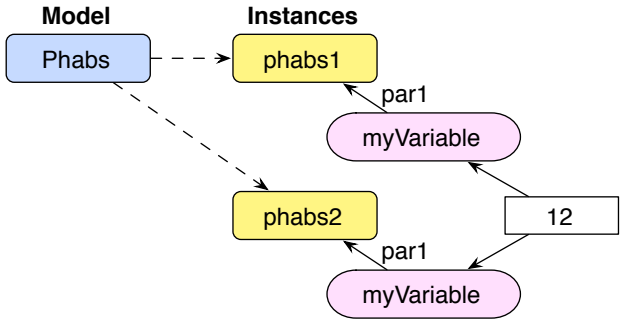


Figure 5-2. Using a variable to share a parameter

myVariable may be set in a Python program or another tool, or it may be set for either model, and the change will cause both models to be updated. Sharing, and the possible attendant confusion, requires an explicit user action.

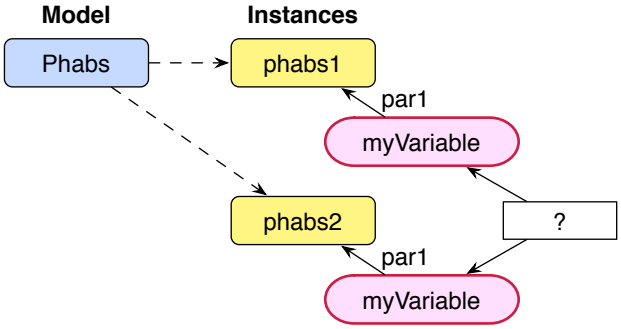


Figure 5-3. Undefined variables are marked by ATSAL

If a model instance is saved for later use, or copied and pasted into another notebook, myVariable may become undefined. ATSAL highlights any undefined references so users can correct them.

Users may define and name their own compound model instances:

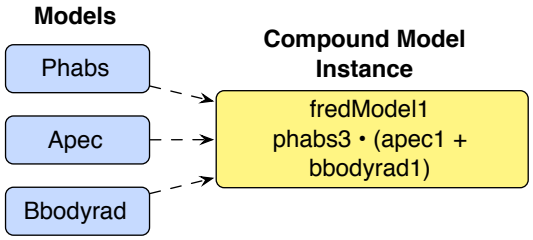


Figure 5-4. Creating a compound model instance

In this example, the user created a model instance called fredModel1, which combines new instances of the multiplicative model phabs with the sum of apec and bbodyrad. The compound model inherits all its initial parameter settings from the basic models. Users then modify the parameters as needed. Model instances are created in the context of an analysis, via the model editor.

The compound model instance shows its results on a graph. It is stored within the current notebook and does not appear in other notebooks' model lists unless a graph containing the model definition is copied to another notebook. But if the model is likely to have value for future analyses, the user can create a new model from the model instance:

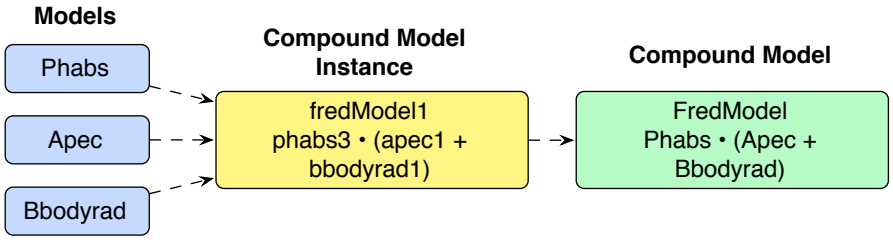


Figure 5-5. Creating a new model from an instance

The compound model acts as a template for future use. It is added to the list of basic models. The compound model's starting parameters are copied from the model instance. This is roughly equivalent in Python terms to subclassing Phabs, Apec, and Bbodyrad, with different initial parameters; hence the components are models, not model instances. Compound model parameters may include variable references. Compound model instances follow the same sharing rules as basic model instances: parameters are not shared by default.

Parameters for each fredModel instance include all those for the phabs, apec, and bbodyrad. Both instances start out with identical settings, but changes to either do not alter the other.

Model instances cannot be composed from other model instances, because this complicates the user interface design:

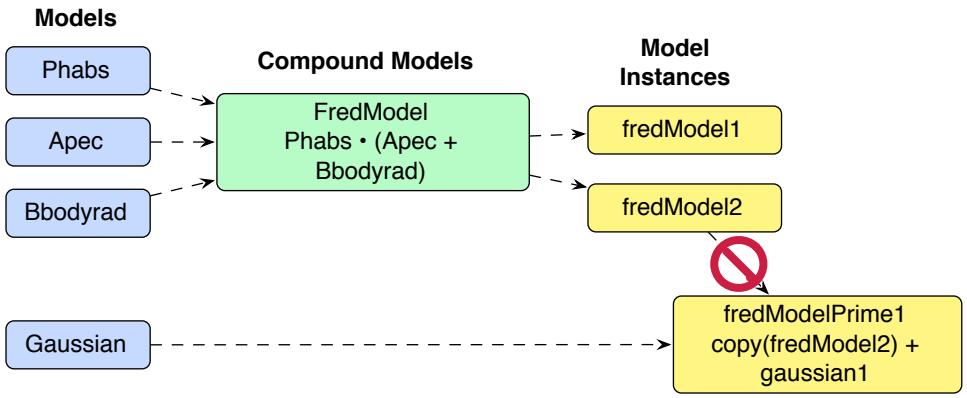


Figure 5-6. A model instance cannot be composed from other model instances

But it is legal to save a model instance as a new model. The new model may be included in another model:

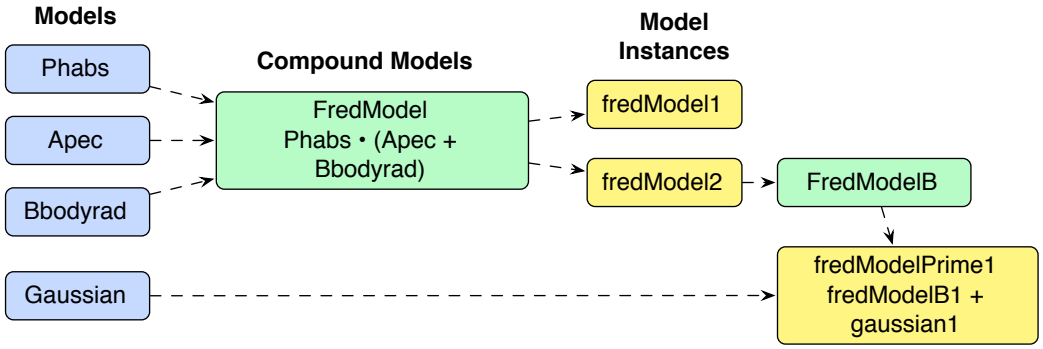


Figure 5-7. Composing compound model instances from other compound models

In this example, a compound model and a basic model have been combined into the fredModelPrime1 model instance. It is legal to save fredModelPrime1 as a new compound model.

It is also possible to share parameter blocks among models of the same type. This is equivalent to sharing model instances. In the parameter editor, the user can specify another model instance whose parameters are to be shared. Changes to the parameter blocks for either model instance apply to both.

5.1.1 Implicit Model Results Sharing

Although model instances are not shared, the results of model computations *are* shared, in a manner that is transparent to the user. As long as two or more model instances share *identical* inputs and parameters, XSPEC is asked only once to calculate results, which are shared among all the identical instances:

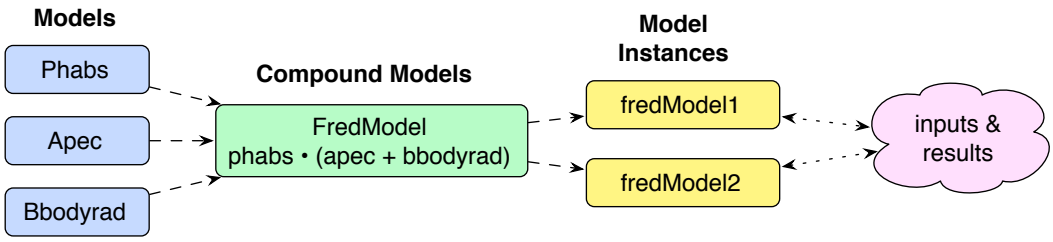


Figure 5-8. Results are shared if all inputs are identical

This is important because duplication of plot tools is a common technique for preserving results from one fit while exploring others. The duplication operation may include results from many fits, and they would have to be recomputed if they were not implicitly shared. As soon as the user alters any parameter or input file, though, the altered model must be recalculated, and two sets of results are subsequently tracked:

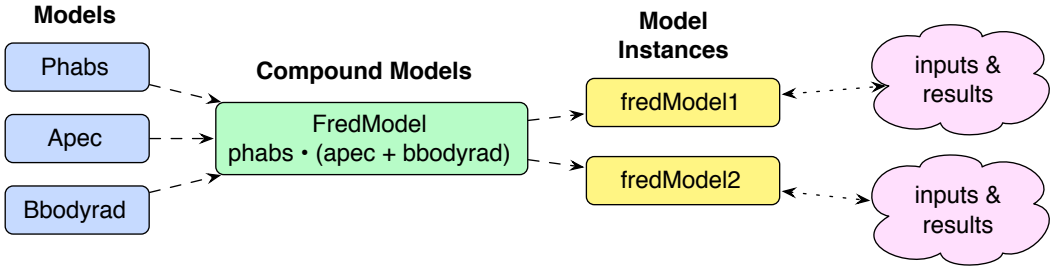


Figure 5-9. If any parameter is changed, the sharing is abandoned

This level of sharing improves speed but has no other consequences for the user.

5.1.2 Sharing User-defined Models with Others

When a user receives a copy of a notebook for another user, any user-defined model instances are present within the notebook, but any user-defined models are *not* added to the recipient's list of models. Users may add these models themselves, by selecting the instances and saving them as new models. If a compound model depends on other compound models, they are also added if necessary to the recipient's list.

In other words, notebooks from other sources never modify the recipient's list of models, and a recipient sees only model instances used in the notebook, not others defined for other projects.

5.1.2.1 Name Conflicts

Since a recipient's list of models is not automatically updated from incoming notebooks, the only time a model name conflict can arise is when the user saves an incoming notebook's model instance as a model, and chooses a conflicting name. Name conflicts of this sort are detected and prevented.

However, it is possible for a model instance in an incoming notebook to have a name that could create confusion. If the recipient has a model called Foo and the notebook has instances called foo*n*, this is confusing. It should probably be reported as a warning when the notebook is opened.

Model instance name conflicts can also occur when cutting and pasting between notebooks. These conflicts are resolved automatically by changing sequence numbers when necessary for the pasted object.

5.1.2.2 User Model Import/Export

User-defined models are stored in a human-readable text file format (probably XML) in the file ~/Library/ATSAL/User Models (Mac) or ?/ATSAL/User Models (Linux). This file may be edited or copied to transfer models to other users or other computers.

Import/export controls may be added in a later release.

5.1.3 Dependency Tree Analysis

Modifications to models and their parameters typically require that XSPEC recompute all or part of them to update their results. Since many modifications may be performed (for example, altering or setting many parameters) before the user wants an update to be performed, updating is done only when the user requests it via the Refresh command.

Graphs, model instances, their parameters, and their computed results are tied together into an internal dependency tree. ATSAL tracks modifications that are performed interactively, marking nodes in the tree for update. When parameters are specified by variable name, ATSAL keeps a copy of each parameter as it existed during the last dependency analysis, and compares the copy to the present value of the variable to detect modification.

After dependency analysis, ATSAL requests XSPEC to recalculate all changed nodes, combining the new results with cached results from prior computations.

5.1.4 Projections

Projections evaluate fit quality more comprehensively. They may require large amounts of computation. So ATSAL initiates projections by instantiating another XSPEC process. The second process runs at lowered priority, leaving the user free to work interactively with the main XSPEC process.

Details TDB.②

5.2 Model Macros

A spectrum plot consists of one or more panes or channels, called **zones** (to avoid confusing them with the other terms, which have alternate meanings). Layers are overlays that are added to a particular zone in a plot. A layer may be a spectrum file, a model, or an annotation. Since a model is added to a zone, by default, it operates on the spectrum (or spectra) displayed in that zone. Models are known by their XSPEC names, and are combined using a macro language that is similar to the command language employed by XSPEC. The macro

```
gaussian
```

means "apply a Gaussian fit with default parameters to the spectrum file(s) in the current zone." The current zone is the zone in which the model's spectrum is

displayed. The zone can also be expressed explicitly. If zone 0 is current, the above expression is equivalent to

```
gaussian(z0)
```

Also, z is shorthand for "current zone," so

```
gaussian(z)
```

means "gaussian of the file(s) in the current zone."

We can also specify the files rather than the zones:

```
gaussian(f2)
```

Here, f2 selects the third (zero-based) foreground file defined by the spectrum files tool (more about this in a moment). This is useful, for example, for specifying a background file for a model.

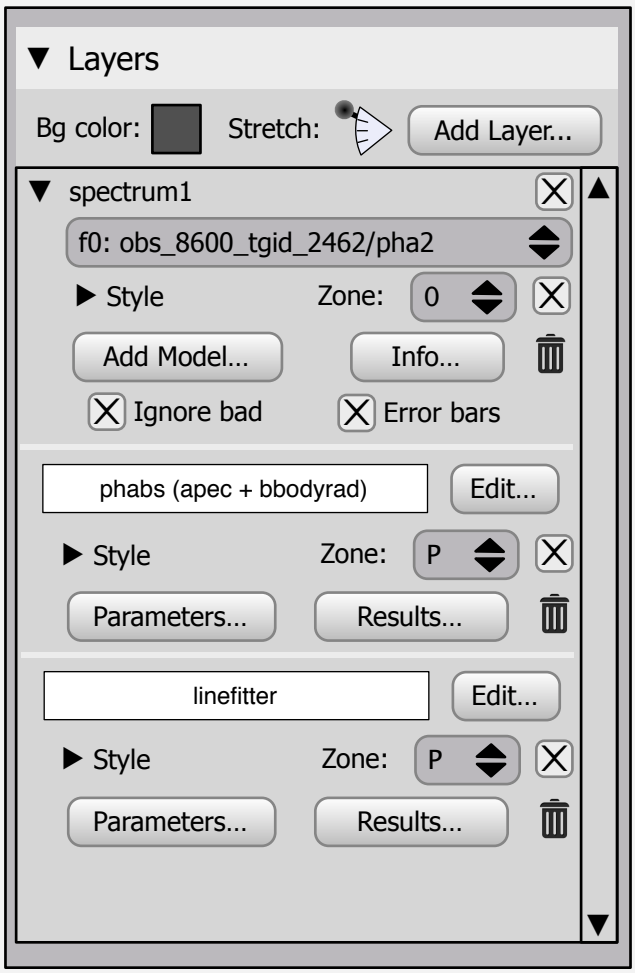
Rarely, a model needs to reference files defined in a different layer than the model itself is defined in. For example, if a gaussian is defined in z1, but operates on file(s) from z0,

```
gaussian(z0)
```

must be specified.

Design Note

It is relatively common for a fit to be displayed in a different zone than the one in which it is defined, but the way the layer manager works, the defining zone is the current zone, not the displayed zone, so "z" is still sufficient. Here is an example:



You can't add a model until you have added at least one spectrum file, so a model always operates on a spectrum file by default. In this case, although the first model's fit is *displayed* in zone 1, it is *defined* in z0, so the model's z = z0, not z1. The user might add a second spectrum, and choose to display it in z0 too. Now z = f0, f1. (One more thing about the example above. The line fitter appears in zone "P," meaning the parent zone. By default, all new models are displayed in their parent zone.)

The distinction between the displayed zone and the defining zone is undeniably confusing, but I can't think of a more natural way to obtain the desired default behavior that a model operates on the spectra in its own zone.

Models may be combined via expressions like this:

```
phabs * (apec + bbodyrad)
```

This is equivalent to

```
phabs(z) * (apec(z) + bbodyrad(z))
```

It means apply the multiplicative model phabs to the current zone's files, and multiply this by the sum of additive models apec and bbodyrad, also applied to these files.

A given file, f0, is likely to be, say, obs_8600_tgid_2462/pha3, which is somewhat lacking in syntactic convenience. Hence in the spectrum files tool, we create a mapping between files of use to a particular analysis and shorthand names that are used when defining models or macros. The spectrum files tool makes the following assignments:

a*n*	Ancillary response files (.arf) are mapped to a*n*. They are not normally referred to explicitly in later analysis, because they are already associated with appropriate spectrum files and background files.
r*n*	Response matrix files (.rmf). They are not normally referred to in later analysis, because they are already associated with appropriate spectrum files and background files.
b*n*	Background files (.pha). These are files that will be used as background files. (It is legal for the same file to be considered either a background or a foreground in different contexts, by associating both an f*n* and a b*n* with it.
f*n*	Files that may be used as foreground spectra (.pha). Each of these have been assigned default RMF and ARF files.

Since the RMF and ARF associated with each f*n* and b*n* are pre-defined, the model macro language does not need to make reference to these. If a given file may need to be evaluated against a different RMF or ARF, multiple files may be defined. For example, f0 could be defined as obs_8600_tgid_2462/pha3 with r1 and a2, while f1 could be defined as obs_8600_tgid_2462/pha3 with r1 and a3.

Since most models operate on the current zone's files (z) or background files (b*n*), most model templates are defined in terms of these elements.

A model may refer to multiple spectra:

```
phabs(f0, f1) * (apec(f0, f1) + bbodyrad(f0, f1))
```

Or to different spectra:

```
phabs(f3) * (apec(f3) + bbodyrad(f2))
```

It is often necessary to refer to a subset of the range of values in a spectrum, and this is done using **selections**. In pure form, a selection is simply a pair of wavelengths expressing a range. For a selection to be useful in a model calculation, it must be specified in conjunction with one or more associated spectrum files. The wavelength range is used to select a corresponding range of bins. Selections are created interactively as overlays on a graph, and are referred to as s*n* in the macro language. Thus e.g. s0 is defined to consist of the wavelength range $\lambda 1$–$\lambda 2$. It has an associated file(s), say f1, for which it also specifies a range of bins that most closely correspond to the wavelength range. Hence

```
gaussian(s0)
```

means "calculate a Gaussian fit for the f1 bin range specified by selection s0."

s0 may have more than one associated file. If s0 has two such files, f1 and f2, for example, then

```
gaussian(s1)
```

means "calculate Gaussian fits for the f1 and f2 bins specified by selection s0."

If multiple files are associated with a selection, they may come from different detectors and therefore be binned differently. Hence s0(f1) might evaluate to a bin range of 3-94, while s0(f2) might evaluate to 8-165. Since the associated files may change after a selection has been defined, we always apply the rule of finding the closest equivalent range of bins from the wavelength, rather than allowing the range to be defined in terms of a binning scheme specific to a particular detector.

For selections with more than one file, the user may override the default of using all the files by supplying some subset of the files instead:

```
gaussian(s0(f2))
```

means "calculate Gaussian fits for only f2's bins, as specified by selection s0."

Selections are more often defined in terms of zones.

```
s3
```

means

```
s3(z)
```

and

```
s4(z3)
```

indicates the specified bin ranges defined in s4 from the files shown in zone 3.

Multiple selections may be specified, for example:

```
gaussian(s1, s2, s4)
```

Selection sets are a shortcut for expressing multiple selections. They are composed via the graph selection bar, by double clicking any selection name. For example, ss1 may be defined to include s1, s2 and s4. Then

```
gaussian(ss1)
```

is equivalent to the preceding. A selection set may only contain selections in the same zone.

Selections and selection sets may be inverted by prepending with "!". !ss1 means "everything that is not within the selection set ss1."

Finally, combinations are legal:

```
gaussian(s2, z3, f0)
```

A warning is issued if a combination like this specifies the same file more than once.

5.3 Macro Language Summary

A *compound model expression* relates a series of *model expressions* with operators that determine how their results are to be combined:

cme	me me op me cme op cme

me is a model expression. op is an operator, and may be "*" (multiply), "+" (add), or "c" (convolution). Parentheses may be used to alter operator precedence. Operator precedence follows the same rules as those for C++ and Python, with the extension that the convolution operator applies only to its immediate operand.

[THK: Is this the right definition for convolution?]

A model expression specifies a *model* and an optional list of *selection expressions*:

me	m[([se_1, [, se_2 [, se_3, ...]]])]

A model is an ATSAL model name. (ATSAL model names are similar to, but not identical to, their XSPEC counterparts.)

A selection expression consists of a *selection,* a *selection set* (group of selections), a *zone*, a file, or a background file:

se	s*n* (selection *n*) ss*n* (selection set *n*) z*n* (zone *n*) f*n* (file *n*) b*n* (background file *n*) !se (everything outside of se)

A zone, z*n*, is a region in a spectrum plot, in which one or more spectra are displayed. A file, f*n*, is a specific spectrum file. b*n* refers to a specific background file. When a selection expression refers to a zone or file explicitly, rather than a selection, the selection range is the entire file.

A selection, s*n*, expresses a contiguous range of wavelengths within a zone or list of files:

s	no argument: applies to all files within the zone in which sn is defined (z*n*) (applies to files within zone z*n*) [(f0 [, f1 [, f2 [, ...]]])] (applies to files in this list)

A selection set, ssn, is a list of selections:

ss	no argument: a list of selections composed elsewhere (z*n*) (applies to files within zone z*n*) [(f0 [, f1 [, f2 [, ...]]])] (applies to files in this list)

6 Line-based Analysis

Most models calculate fits to spectral data. Line fitters, though, calculate fits to fits, finding the ions and transitions that are associated with specific lines in a spectrum, working in conjunction with ATOMDB. This is called *Line-Based Analysis* (LBA). Although the operation is different in this respect, its specification and usage is essentially the same as that for other models.

Design Note

Determining how to perform the line fits and judge their quality is a GSRP—Graduate Student Research Project. This section describes the overall goals of line-based analysis, and proposes the user interface, but only touches on design.

The linefitter model performs this function. To use this model, the user first creates one or more selections encompassing interesting areas of the spectrum. To label a specific strong line, the selection should roughly encompass 3-5 times its FWHM. If a selection encompasses multiple peaks, the linefitter model labels all major peaks. The linefitter may also be applied to the entire spectrum as a quick means of identifying the largest peaks.

The linefitter operates on the selections in three phases:

- A peak detection algorithm selects suitable peaks and/or troughs as candidates for identification.
- A centroid-detecting model is applied to each of the peaks above.
- A database of emission and absorption lines is consulted to associate labels with the centroids.

All of these phases may feed back on each other, especially when decomposing overlapping peaks.

The selections and selection sets on which the linefitter operates may include hints as to the type of data contained within the selection. The fitter uses these hints to achieve improved labeling. These types are shown below.

Absorption line, single	Assumes selection encompasses a single absorption line
Absorption line, doublet	A pair of nearby absorption lines
Absorption lines, multiple	Fits all prominent troughs within the selection (a fit parameter determines the threshold of "prominence")
Continuum	An error—this is used by the continuum fitter
Emission line, single	A single emission line
Emission line, doublet	A pair of nearby emission lines
Emission line, multiple	All the prominent peaks within the selection (a fit parameter determines the threshold of "prominence")
Ignore	Ignore this region
Manual	Assigns a label of "Unknown" over the centroid. The user then assigns a label manually from a list of nearby emission lines
Unknown	Try to fit prominent peaks and troughs in this range. This is the default

Table 6-1. Selection types

The peak detection phase uses these hints to decide whether to look for peaks, troughs, or both. The algorithm will probably employ Continuous Wavelet Transforms (CWT), as described by Pan Du, Warren A. Kibbe, and Simon Lin, in a 2007 paper, *Using the MassSpecWavelet Package.*[1] Multiple such peak detection algorithms may be available, selectable as a linefitter model parameter. The algorithms accept a threshold parameter used to select more or fewer dominant peaks for labeling.

Next, a user-specified model (gaussian by default) is applied to each of the candidate peaks in order to find its centroid. The selection type also influences the algorithm. For example, if a selection's type is emission line doublet, the linefitter will try to assign two labels even if one is substantially lower in emissivity than the more prominent one. If the selection is marked as a single absorption line, it will try to find a trough even if a peak is the dominant feature.

Finally, the line fitter consults the ATOMDB database (and possibly other resources, such as XSTAR) to determine the lines that best fit, and displays label(s). These matches may be overridden if the user has some other means of selecting a better choice.

Each linefitter has a "clutter control" that shows some subset of the calculated results, showing the highest quality lines first. The clutter control and the zoom level together determine the number of labels that are actually shown, but all the labels found are available to software tools following the plot.

Usually a single linefitter is sufficient for each zone. It operates on a selection set, so it handles multiple selections. However, multiple linefitters may be applied to

[1] http://bioconductor.wustl.edu/bioc/vignettes/MassSpecWavelet/inst/doc/MassSpecWavelet.pdf

handle special cases, for example, to apply different parameters to different regions.

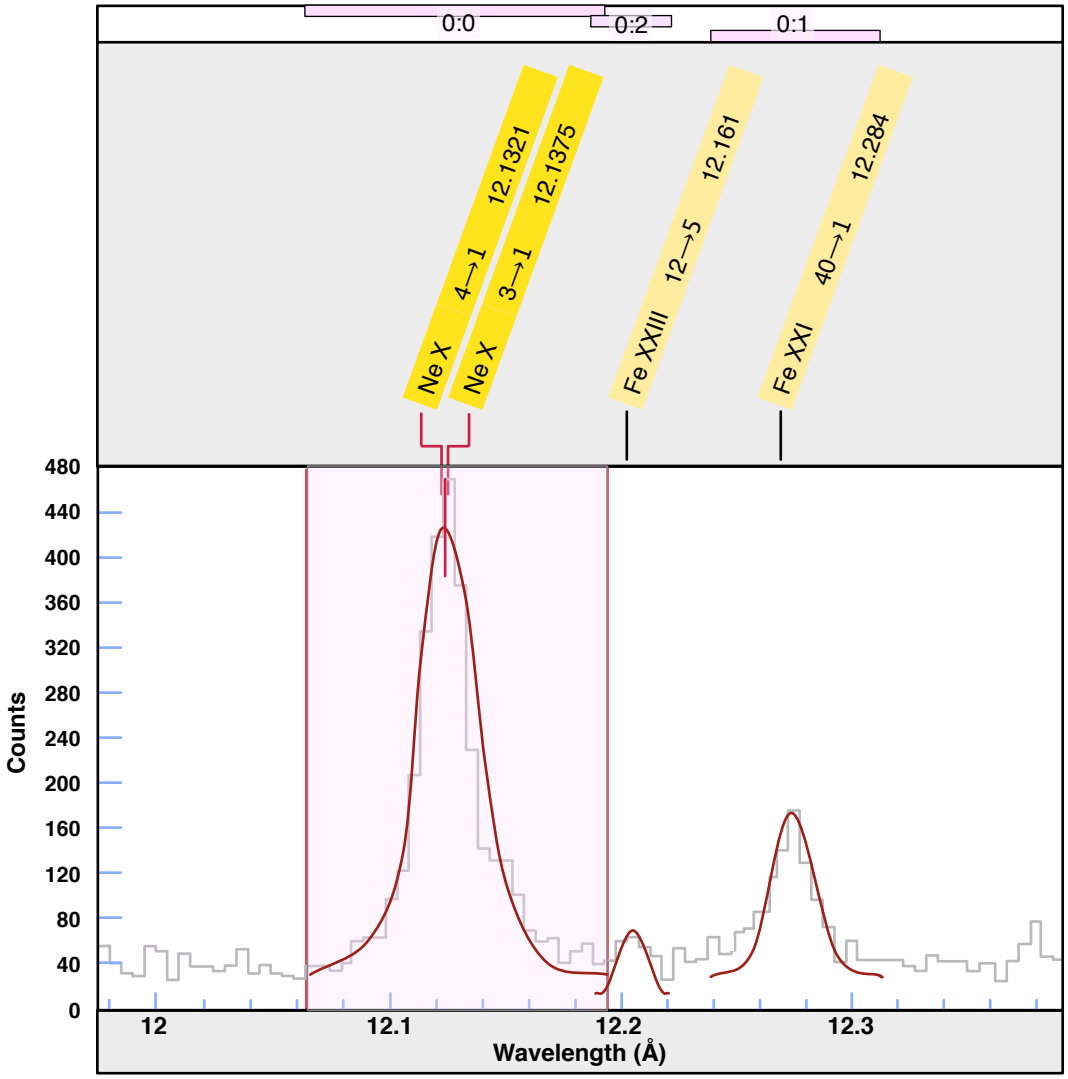


Figure 6-1. Line fitter applied to a selection set of 3 selections, with single line matching requested. (Simulated data)

Selections are shown in the selection bar at the top. There are three in this example, all colored the same because they are members of the same selection set, set 0. The numbering format is *set:selection*. The current selection's edges are shown in red. Its centroid is shown at the line's peak, also in red. Although single line matching was requested, the selection matched a doublet, so labels for both lines are shown, along with their positions relative to the centroid. The fits are also shown for two more selections. (Selections are shown in the selection bar at the top.)

The linefitter's output is the labels themselves, which appear in the label region of the graph zone, along with the fits. (Other calculated results are described shortly.) The boundary between graph and label regions is adjustable, and label

information is truncated if necessary to fit. Clicking a label scrolls the surrounding lines into view in the control panel:

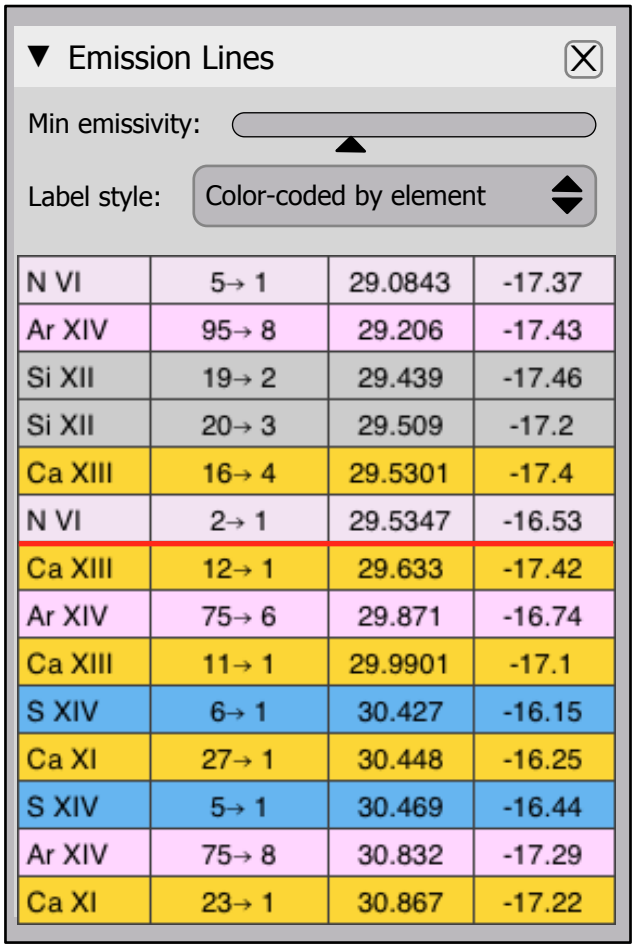


Figure 6-2. The emission lines displays scrolls to the selected label's wavelength

This provides a means of viewing surrounding lines in crowded areas of the spectrum. The red line indicates the position of the selection's centroid. The Min Emissivity sets a threshold below which lines are omitted, restricting the list to more dominant lines. Double-clicking these lines, or a label, displays their detailed information:

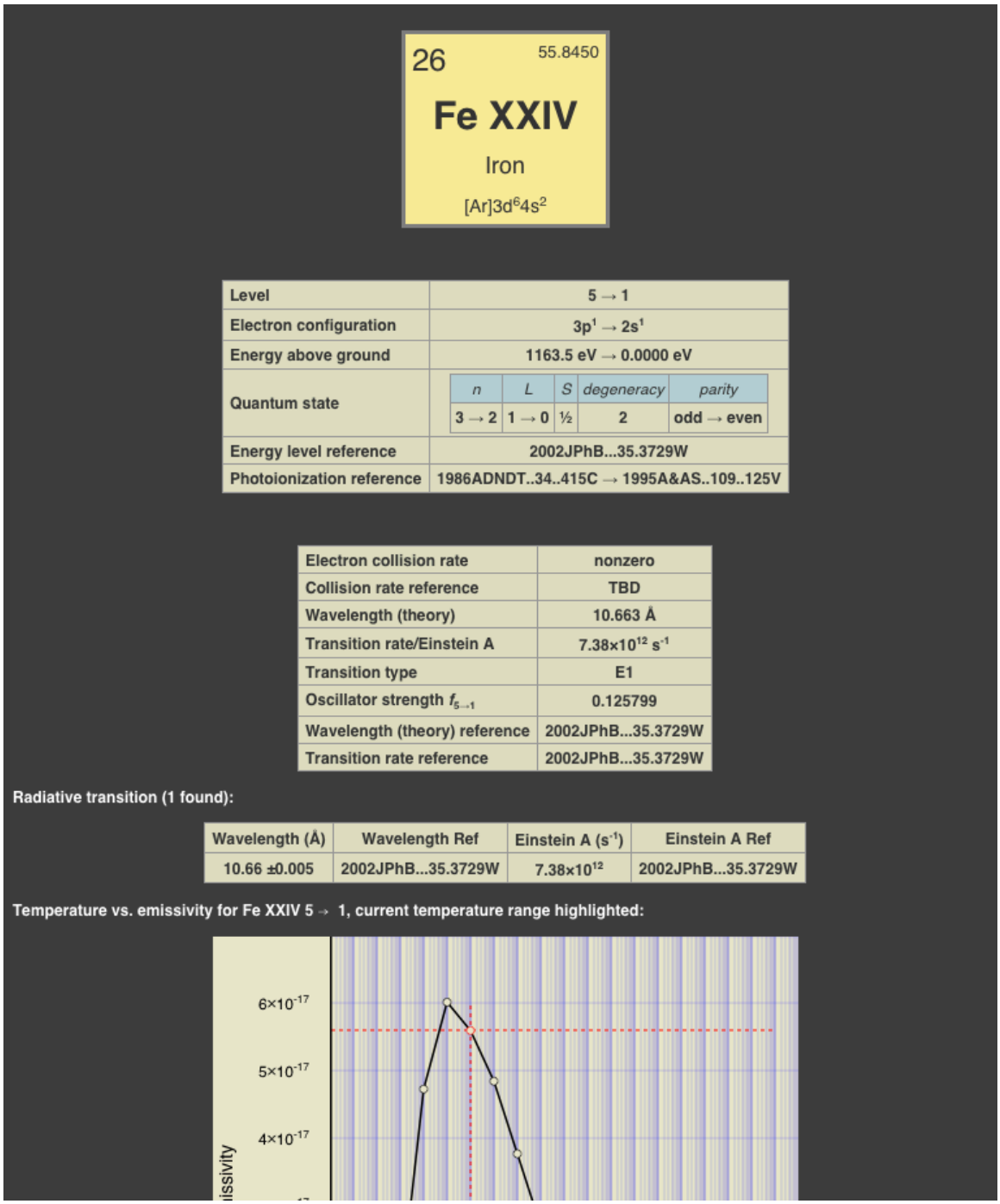


Figure 6-3. Emission line info (mockup from ATOMDB for iPad app – format will vary)

This display will also allow a user to override the binding between a fit and an emission line in ATOMDB, when there is reason to do so. The Reassign Line Fit... command *(not shown)* displays a list of nearby lines, from which the user can select an alternate, better match. Reassigned labels are marked differently.

In the example below, we pretend that the asymmetry in the predominant peak is due to an unobtainium absorption line instead of a doublet:

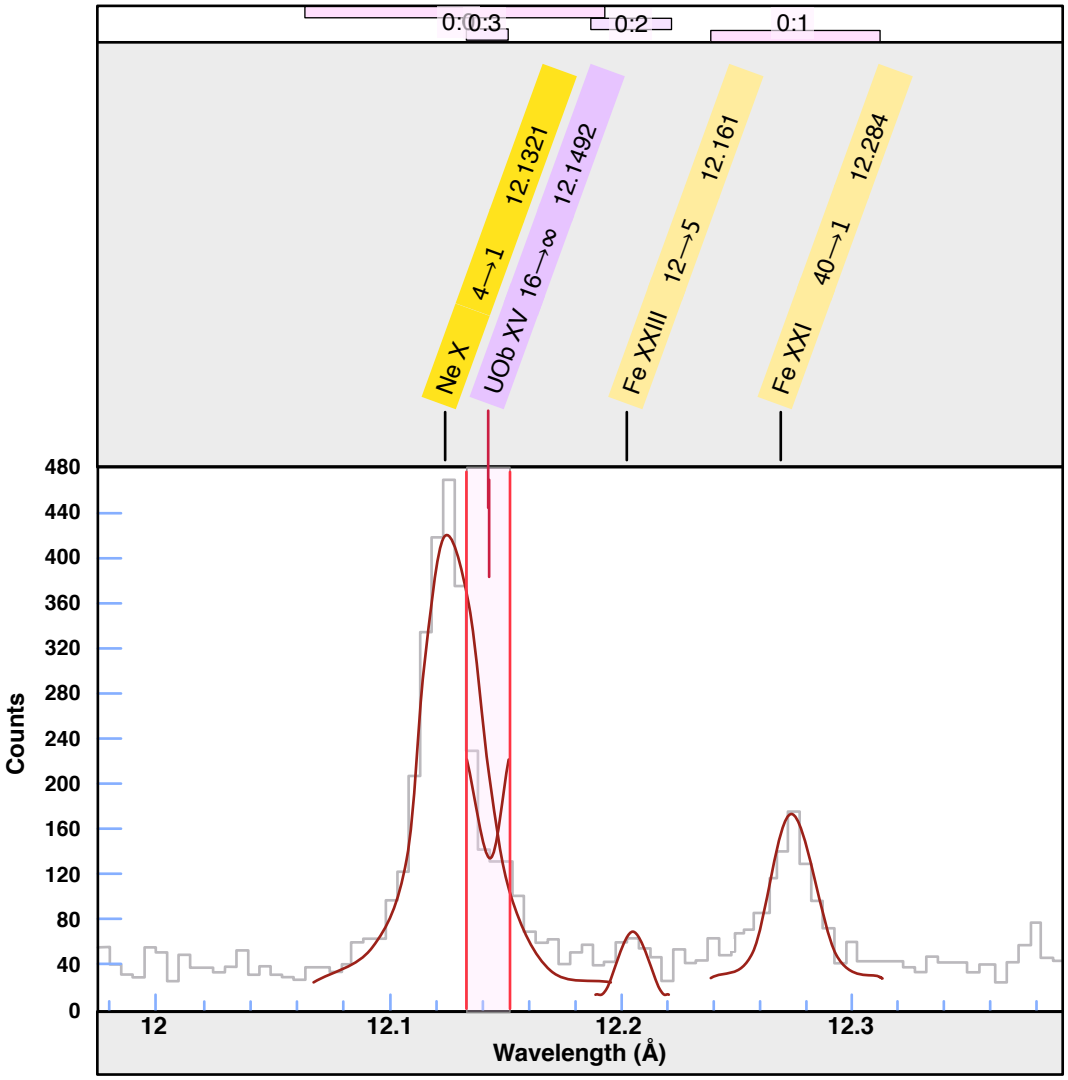


Table 6-2. Overlapping emission and/or absorption lines improve fit quality

Because the Ne X and UOb XV lines overlap, the fitter can improve the quality of the fits for both lines by subtracting the effect of each from the other. Similarly, users can subtract some or all the identified lines in a selection set temporarily from the plot, in order to better view the remaining data. This makes it easier to isolate and identify additional peaks and troughs.

6.1.1 Absorption

[RKS: Absorption is a bit trickier than emission, because there are more places where you can get 'continuum absorption' than there are 'continuum emission'—and any continuum process is more complex than any line process. There's another issue as regards absorption, which I think I've mentioned before—emission can be of any strength, while absorption can't absorb more than the total available emission. This has a mathematical consequence when writing down models—you can do three emission lines in a model as "gaussian + gaussian + gaussian" but absorption has to be "continuum * absline * absline * absline". At least, that's how XSPEC wants to handle it. One could imagine

using gaussians with negative normalizations (and this works, to an extent) but then you can get negative fluxes, which is a no-no. The downside of all of this is that fitting gaussians is relatively quick, but fitting multiplicative absorption lines can get really, really bogged down. Our selections approach might be a big help here, don't know yet.]

6.1.2 Linefitter Parameters

The linefitter supports the following parameters.

Peak detection model	Selects the algorithm used to isolate peaks and troughs. Default is continuous wavelet transforms.
Minimum match quality	A value of 0-1 that determines the minimum match quality required to generate a label. Lower values produce more labels. The number of labels found is usually higher than the number displayed, since the latter is also determined by the clutter control. Default is 0.5.
Peak fitting model	Determines the model used to find peaks within the selection(s). Default is gaussian. Lorentzians and Voigt profiles are other options.
(Parameters for the selected model)	TBD
...	

6.1.3 Linefitter Results

A linefitter produces an array of LineInfos, where each LineInfo consists of:

- the model instance that was used to find the peak, which contains the fit results, including the centroid, the width of the line, the normalization, and other results.
- an EmissionLine or AbsorptionLine object that describes the line (essentially a pointer into AtomDB or a similar database). This is usually calculated by the line fitter but may be assigned by the user.

The results also include a kT and normalization for the group of lines as a whole.

This data is displayed when the label is double-clicked. It may also be displayed in tabular form by a model results tool, or accessed from a Python program.

6.2 Continuum Fitter

The continuum fitter tool also operates on a selection set, but in this case the selection set encompasses portions of the spectrum that contain only (or mostly) continuum, and the goal is to match the continuum.

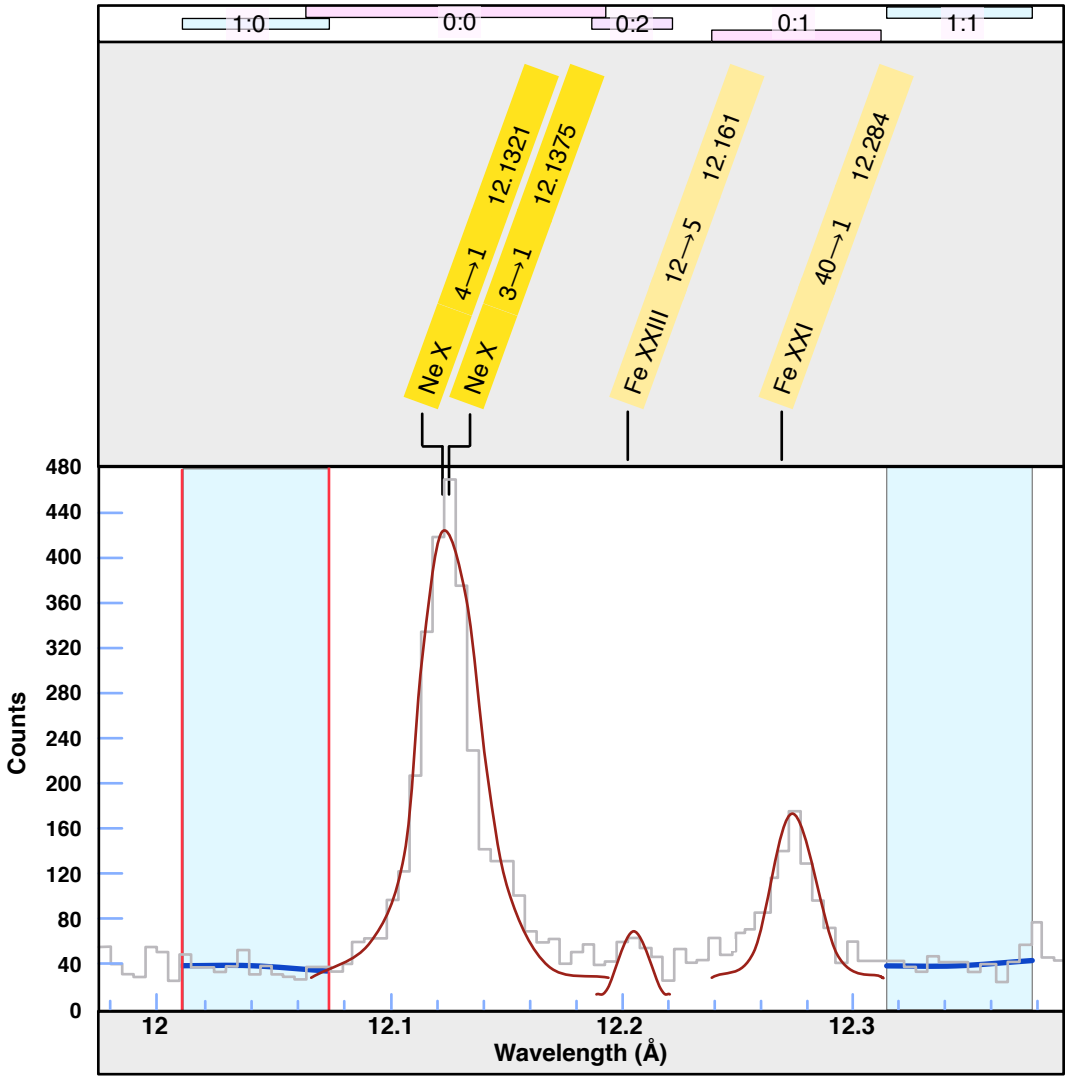


Figure 6-4. Continuum fitter example. The selection set shown in blue is used for the continuum fit

6.2.1 Continuum Fitter Parameters

The linefitter supports the following parameters.

Model type	Apec models, others?
Temperature estimate?	
...	

6.2.2 Continuum Fitter Results

Fit results are the same as for the underlying models, and presented the same way.

7 Tools

A tool is a view or control element that plays a role in a Python analysis. Each tool has a collapsed view or an expanded view within the notebook, as well as a tool editor that appears in a separate window.

7.1 Tool Views

7.1.1 Collapsed View

Each tool has a collapsed view, which consists of a name, a one-line comment, a progress indicator that appears when needed, and a tool icon. The name must be a valid Python symbol name. By convention, such names begin with a lowercase. Both the collapsed and expanded forms of the tool fill the width of the page.

Figure 7-1. Collapsed view of a tool

In a Python program, the tool is accessed via this name. For example,

```
spectrumFiles1.comment() # returns “Beta Lyrae sources from Chandra”
```

7.1.2 Expanded View

Clicking the disclosure triangle shows the tool’s expanded form. In this example, the expanded form shows the search parameters used to fetch and download a set of files for analysis. The expanded form of the tool may be adjusted in height as needed, up to a maximum of one page. An expanded tool may not be edited though.

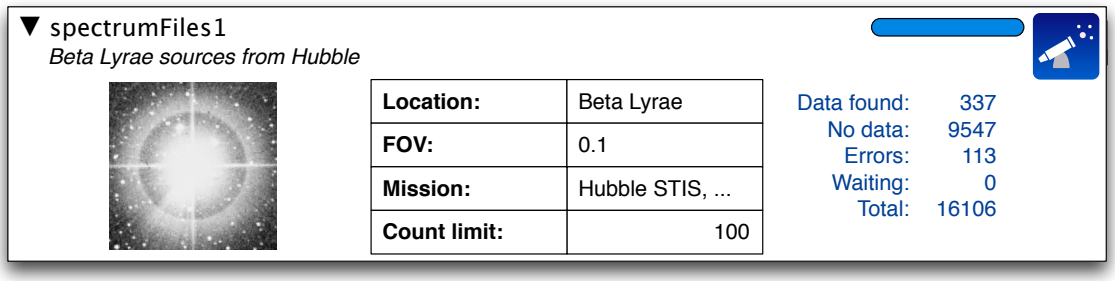


Figure 7-2. Expanded view of a tool

7.1.3 Tool Editor

Double clicking anywhere in a collapsed or expanded tool opens the editor for that tool in a separate window. The editor varies greatly depending upon the type of tool. It consists of an editor window and a control panel to the right.

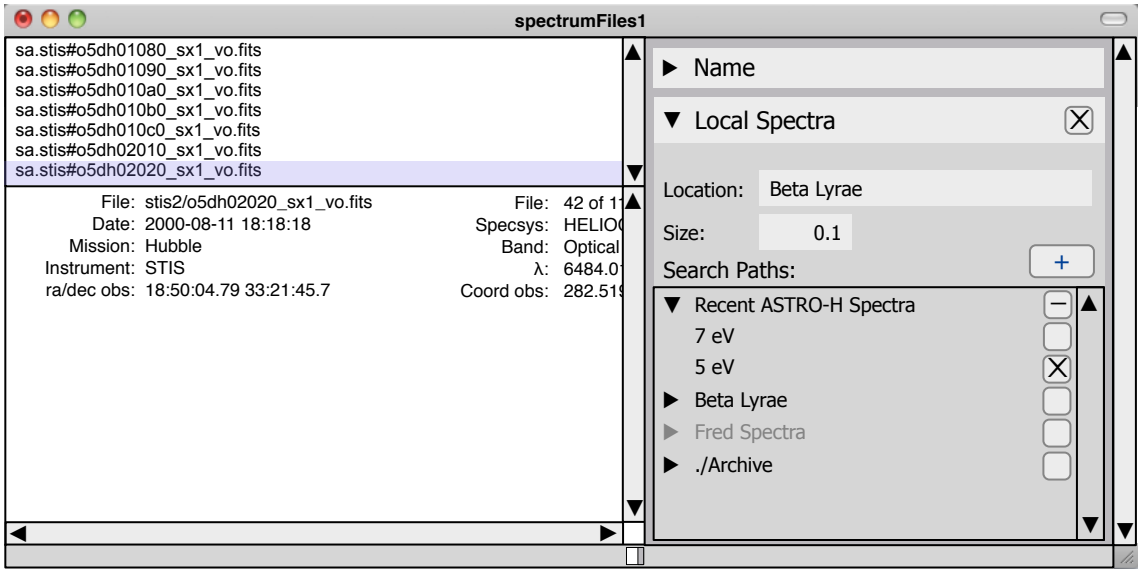


Figure 7-3. A tool editor with control panel displayed

The control panel is scrollable, and composed of collapsible panes, allowing the user to display those panes that are most commonly used. In the example above, the Name pane is collapsed, while the Local Spectra panel is expanded. By hiding the control panel after configuration, users have a bit more room to arrange windows to watch progress during notebook runs. The control panel is fixed in width, so resizing the window resizes the tool editor portion, not the control panel.

8 Observatory Tool

The observatory tool presents a search interface to retrieve files from web sites for analysis. Its purpose is to download the files to a local library, and enter their metadata into a local database. The downloading may be initiated at any time.

Design Note

Downloaded files are stored in a library that is global to the user, shared among all notebooks. A single download daemon manages all pending download requests, so that even if two separate notebooks issue simultaneous download requests, they will be handled correctly. Since the download cache is not specific to a particular notebook, a second tool—the spectrum files tool—is used to search against the local cache. That is:

- The **observatory tool** gets files from one or more archives, using archive-specific search tools, into a local library.
- The **spectrum files tool** searches the *local* library for the subset of those files that are needed for an analysis.

[THK: This decoupling of downloading from searching the local files makes the observatory tool optional, since it replicates capabilities already in place in the archive web sites. For example, users can download extractions from TGCat using the web interface, place them in a directory that is in the spectrum file tool's search path, update the local database (using the Update Database button), and work with that. One benefit of this decoupling is that if our downloader breaks because of unanticipated changes on the web site, users can still download by hand.]

8.1 Tracking Downloaded Data

Notebooks preserve information about the files that are analyzed, but because the files are generally large, notebooks do not include copies of the files themselves. A problem can arise because downloading with identical search criteria may not produce identical results as a result of changes in the archive. Hence ATSAL must track additional information to ensure that a reproduced analysis matches the original analysis.

Phase	*Remarks*
Search for a collection of files	Search may yield different files later. ATSAL saves search criteria.
Download file archives	ATSAL keeps track of the original contents of each extraction, and can highlight missing and new files.
Select subset for analysis	ATSAL keeps a list of the files used for an analysis.
Individual files	The file may have been regenerated for some reason, for example, a data reduction bug fix. ATSAL tracks information (checksum?) that detects any files that have changed since the last analysis.

Taken together, this stored information is sufficient to download the observed spectra and related files and to verify that the same subset of these files is used to repeat an analysis. The user is warned of any changes. Users may also elect to export a notebook archive, which optionally includes copies of the files for analysis. This provides a way to ensure that the original data are preserved, for example, prior to publication.

8.2 Downloaded File Storage

Downloaded files are stored in ~/ATSAL Spectra/*Archive/Source*, where *Archive* indicates the source of the data, e.g. TGCat, and *Source* indicates a name for the approximate source area. Extractions are stored in the format in which they are supplied by the archive (by observation number), except that the outer compression layer is removed.[2] It is legal to rearrange these files manually if desired, but it will confuse ATSAL's cached database. If ATSAL detects a mismatch, it offers to update the database to match the current file tree within the search path. However, ATSAL won't detect a mismatch unless a requested file is not present. The "Update Database" button, described later, forces a database update.

Although ATSAL downloads new files using the rule above, it is neutral about where files are placed once they are downloaded (except that all files that are part of an extraction must remain in the same directory). ATSAL only sees and catalogs files that are part of the current notebook's search path. (The database actually spans the search paths of all notebooks, but when the user updates the database, only records associated with the current notebook's search path are updated.)

The search path usually includes the ATSAL notebook directory itself. If identical files are found in more than one part of the search path, ATSAL discovers them in order by search path and takes only the first one found. Usually the notebook is first on the list, so anything included in a notebook archive takes precedence.

[THK: Need to add a way to optionally move extractions included in a notebook into the user's library.]

8.3 Observatory Tool in Notebook

The observatory tool appears in the notebook with a summary of search criteria and some progress information if a download is active. The contents of this pane may vary depending on the source of the data.

[2] TGCat files are supplied with compression to both the individual files and the extraction group as a whole. The outer layer of compression probably adds little or no additional compression and is done simply to create a single file for downloading. The inner level of compression is removed only for files that are actually used in an analysis, and the decompressed files are later discarded if the cache fills.

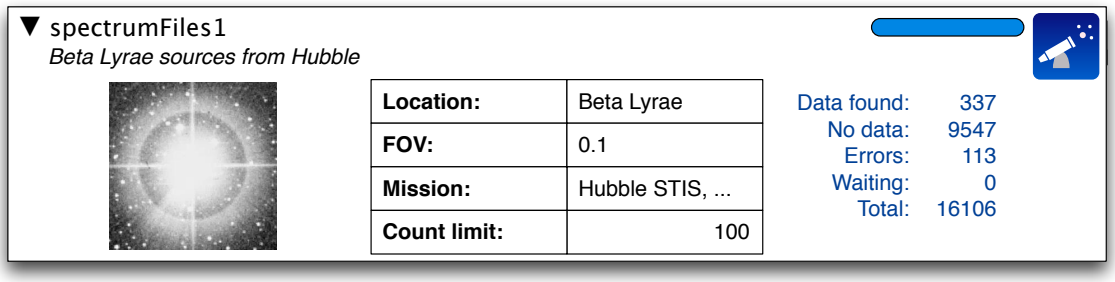


Figure 8-1. Expanded view of spectrum file tool

8.4 Observatory Tool Editor

The observatory tool editor displays all files that presently match the search criteria. If a file is selected, it is downloaded and its key metadata displayed. In the sample below, the window is in view by source mode. A single source is selected, and the bottom half of the window displays information about its extractions. In this case, there is only one extraction. In view by extractions mode, each extraction is listed separately.

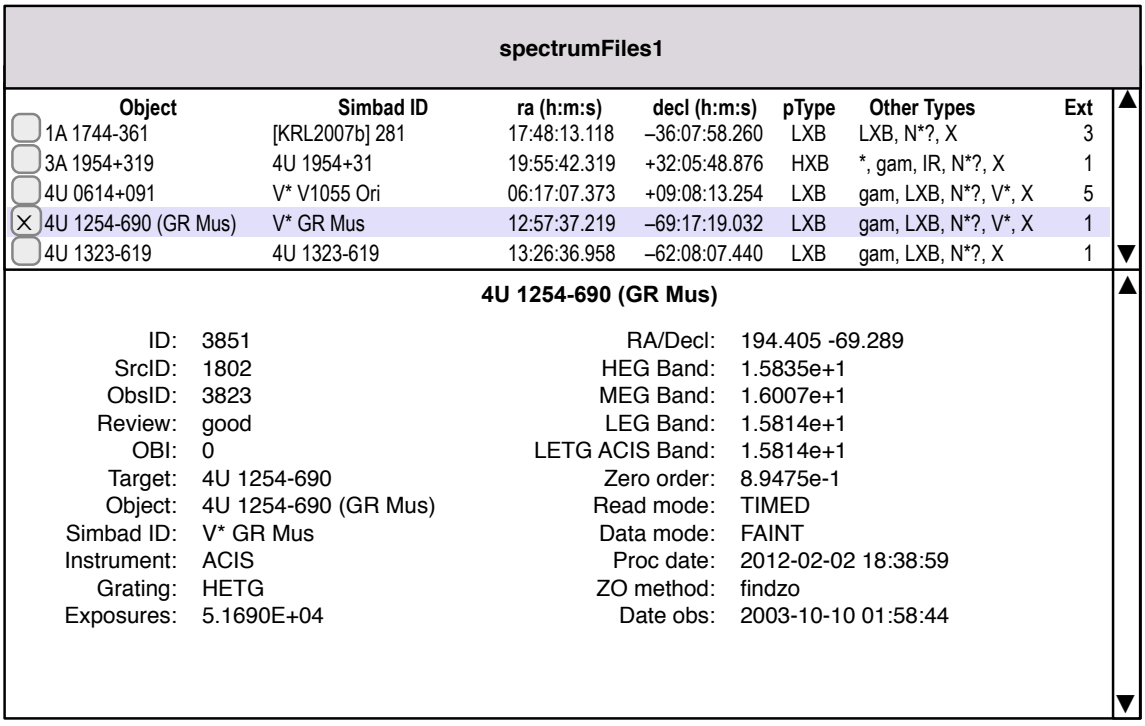


Figure 8-2. Spectrum file tool editor, TGCat example

This interface browses the available data and permits selection of the desired data for download.

The control panel allows searches from the following resources. Searches for data that is not already pre-converted for use with XSPEC, marked as release 2 below, probably won't happen any time soon, as this is just one part of the problem.

Resource	*Description*
TGCat	Archive of pre-processed data (http://tgcat.mit.edu) ①
VO	NVO's Virtual Astronomical Observatory (http://www.us-vo.org) ③
HEASARC	NASA's High Energy Astrophysics Science Archive Research Center files (http://heasarc.gsfc.nasa.gov) ③
DARTS	Data ARchives and Transmission System files (http://www.darts.isas.jaxa.jp) ③

Table 8-1. Possible archive sources

Each searchable archive has its own CollapsiblePane subclass, since search options may differ by archive. A single spectrum downloader can acquire spectra from multiple sources at once.

More complex downloads, for example, files with multiple search criteria, are created by inserting multiple downloader tools.

8.5 Observatory Tool Control Panels

8.5.1 CPObsToolName

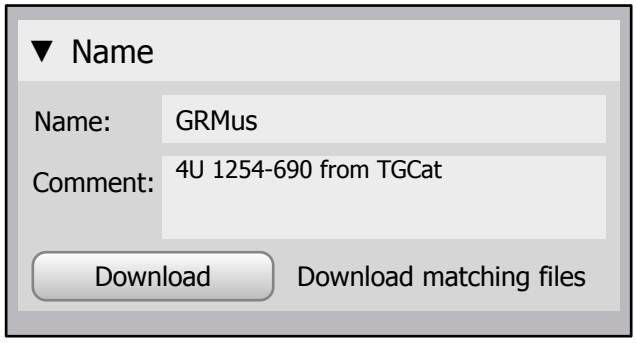


Figure 8-3. CPObsToolName

This provides a name for the tool and a brief description of its purpose. The Download button causes files matching the current search criteria to be downloaded. Downloading is performed by a separate process and continues even when the notebook is running (although of course the notebook won't be able to see any files that are not yet downloaded). Any change to search criteria first offers to abort any pending download. The download must be restarted after changes are completed.

If the user runs the notebook, it will pause until downloading has completed.

8.5.2 CPObsTGCat

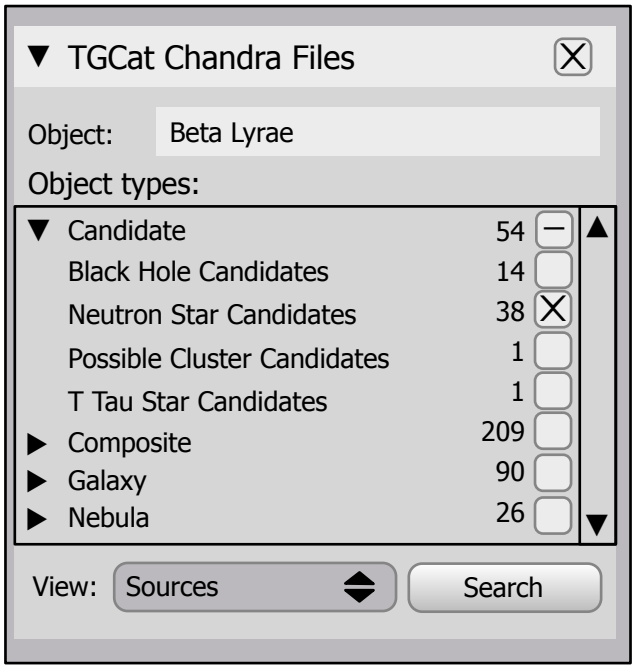


Figure 8-4. CPDownloadTGCat

The Object box allows quick searches by source. The Object types display supports searches by type of object. These two are mutually exclusive. When a search is completed, results are displayed in the search window. The Search button initiates a new search. Search matches may be listed by source or extractions.

spectrumFiles1

Object	Obs ID	Inst/Grating	ra (h:m:s)	decl (h:m:s)	Exposure	Date Obs
4U 0614+091	10858	ACIS/HETG	17:48:13.118	–36:07:58.260	34436.8	2009-01-19 17:45:01
4U 0614+091	10857	ACIS/HETG	19:55:42.319	+32:05:48.876	58053.2	2009-01-21 13:49:21
4U 1254-690 (GR Mus)	3823	ACIS/HETG	06:17:07.373	+09:08:13.254	51689.6	2003-10-10 01:58:44
4U 1323-619	3826	ACIS/HETG	12:57:37.219	–69:17:19.032	39245.5	2003-09-25 06:13:11
4U 1323-619	14377	ACIS/HETG	13:26:36.958	–62:08:07.440	59940.8	2011-12-19 01:02:43

4U 1254-690 (GR Mus)

ID:	3851	RA/Decl:	194.405 -69.289
SrcID:	1802	HEG Band:	1.5835e+1
ObsID:	3823	MEG Band:	1.6007e+1
Review:	good	LEG Band:	1.5814e+1
OBI:	0	LETG ACIS Band:	1.5814e+1
Target:	4U 1254-690	Zero order:	8.9475e-1
Object:	4U 1254-690 (GR Mus)	Read mode:	TIMED
Simbad ID:	V* GR Mus	Data mode:	FAINT
Instrument:	ACIS	Proc date:	2012-02-02 18:38:59
Grating:	HETG	ZO method:	findzo
Exposures:	5.1690E+04	Date obs:	2003-10-10 01:58:44

Figure 8-5. ObsTGCat search window, viewed by source (rather than by extractions)

Sources are selected for downloading, and when they are needed, they are downloaded. The Precache command downloads all source files.

9 Spectrum Files Tool

While the observatory tool downloads files into a local library for later use, the spectrum file tool searches this cache to select file(s) for a current analysis. The observatory tool has interfaces tailored to each online archive, while the spectrum file tool uses a consistent interface to search local files regardless of their source.

When used to search the local library, the expanded tool pane shows the search criteria in use, and looks very similar to the downloader tool:

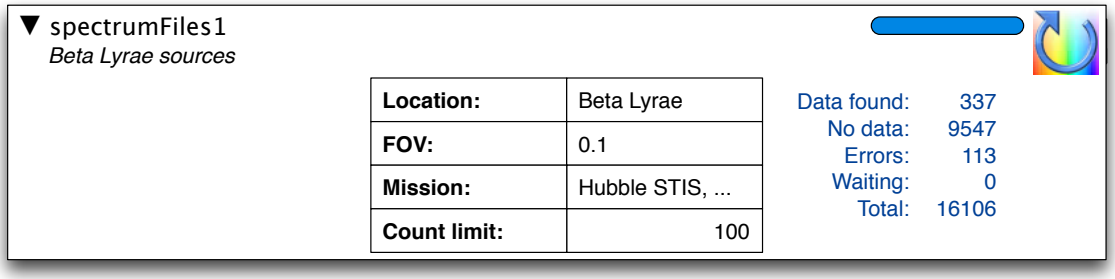


Figure 9-1. Spectrum files pane when searching

While a notebook is executing, this pane shows information pertaining to the current state of the execution, for example, a spectrum file currently being analyzed:

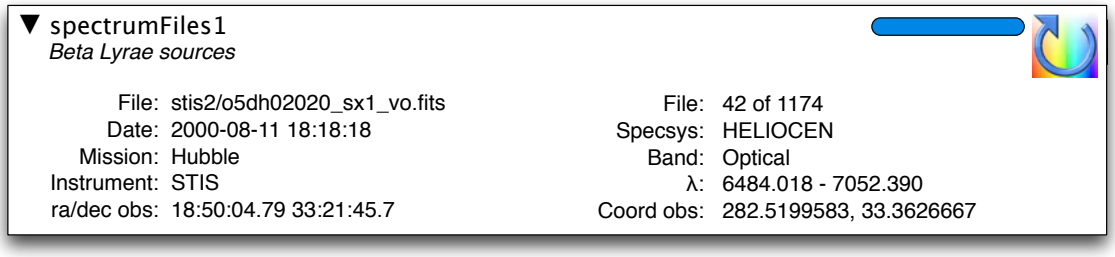


Figure 9-2. Spectrum files pane during execution

9.1 Spectrum File Tool Editor

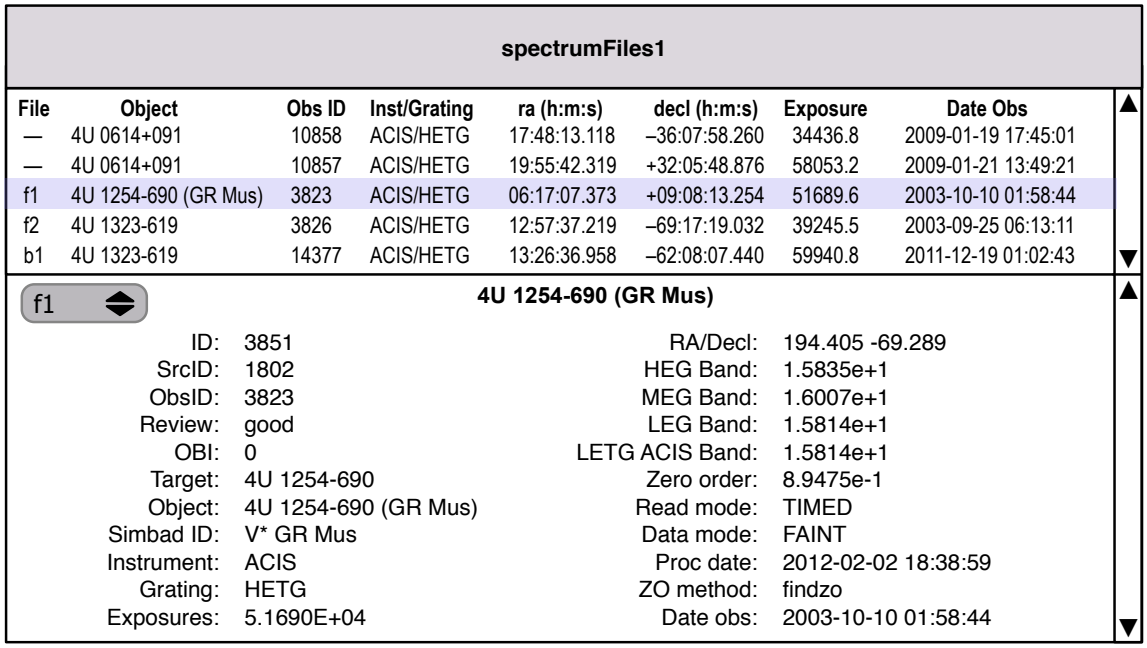


Figure 9-3. Spectrum file tool editor window

Like the observatory tool, this tool lists all the extractions that match the search criteria. But searches made with this tool are restricted to the local filesystem or any mounted remote volumes.

The user first performs a search to obtain the desired extractions, and marks those of interest. Files of interest are assigned temporary names of f*n* and b*n*, designating foreground and background files that will be used by various models during the analysis. Assigned files are shown in the Files column.

To select a different file(s) to be analyzed by existing models in the notebook, the user reassigns these labels.

[THK: By giving files of interest temporary names, we avoid the need to change the files associated with each model. We are naming the file(s) to fit the model(s) instead of the other way around.]

[THK: This isn't quite right, because the list above is extractions, not individual files. There will have to be two levels, opening up the extraction to list its PHA files, and assigning those temporary names.]

9.1.1 CPSearchParams

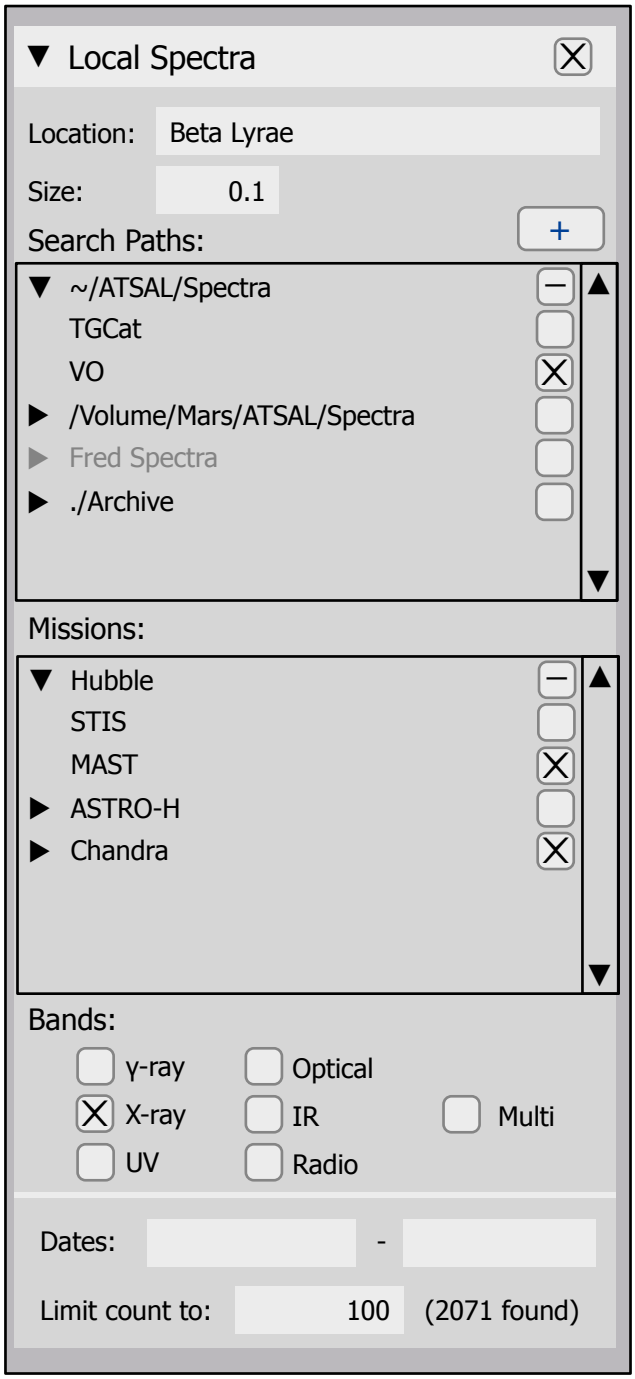


Figure 9-4. CPSearchParams

This panel allows searches through extraction files that are stored on the local system or on mounted filesystems. The search path is composed of directories that are part of the user's global search path, set via Preferences, and any additional directories added for the current project. The checkboxes indicate: (blank) no search; "-" search some subdirectories; "x" search all subdirectories. If a

search path is dimmed, it is not present on the system. In the first release, only data from ASTRO-H will be formally supported.

The checkbox in the upper right corner enables or disables a search of this resource.

9.1.1.1 Search Paths and Other Systems

When a notebook is given to another user, the global search path is replaced by that in use on the recipient's system. Notebook-specific search paths are shown dimmed if they do not exist on the recipient's system. The user can remove unused search paths by selecting and deleting them. Since notebooks do not include spectrum files, a recipient can apply the notebook's analysis procedure to other files, but not to the original files.

If a notebook archive is created, the creator has the option of including any local files as part of the archive. Such files appear in ./Archive.

This names the tool for Python access.

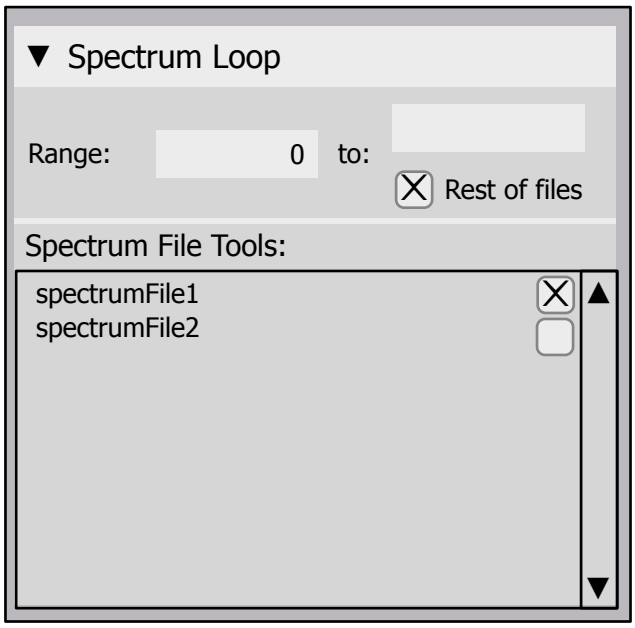


Figure 9-5. Spectrum loop settings

This option allows loop parameters to be set, and specifies which spectrum files tools will be used as input.

10 Spectrum Plot Tool

This tool displays a spectral *x/y* plot, and a series of optional overlays that display curve fits, theoretical data, etc. It allows users to locate specific emission or absorption lines, or to designate wavelength ranges for subsequent analysis. Plots are typically corrected for red shift prior to analysis, but they can also be left uncorrected.

These classes are involved in creating any kind of plot:

- A Plot, such as a PlotSpectrum, is a QWidget that embodies a complete plot and related areas, such as labels, axes, and annotations. Plot subclasses begin with "Plot."
- A GraphLayer is a transparent overlay for the graph area (graph + labels) in a plot. GraphLayers include data plots, tick marks, labels, etc. Any number of such labels may be stacked atop each other to produce a plot. GraphLayer subclasses have a "GL" prefix.
- An Axis subclass displays the numbering for an axis. Subclasses begin with "Axis."
- An AxisUnits subclass displays the currently selected units for an axis, and may support clicking to select other units. Subclasses begin with "AxisUnits."

10.1 Spectrum Plot User Interface Classes

10.1.1 PlotSettings

This is a data class. It contains access to the plot data itself, as well as settings that control its display. The class keeps information that is shared with various user interface elements, such as the units, pan and zoom settings, plot maxima and minima, etc.

10.1.2 PlotXY/PlotSpectrum

PlotXY is the base class for X/Y plots, while PlotSpectrum is a subclass designed to display spectral data. PlotSpectrum is the outer QWidget that contains all the elements in a plot, including titles, axes, annotations, etc.

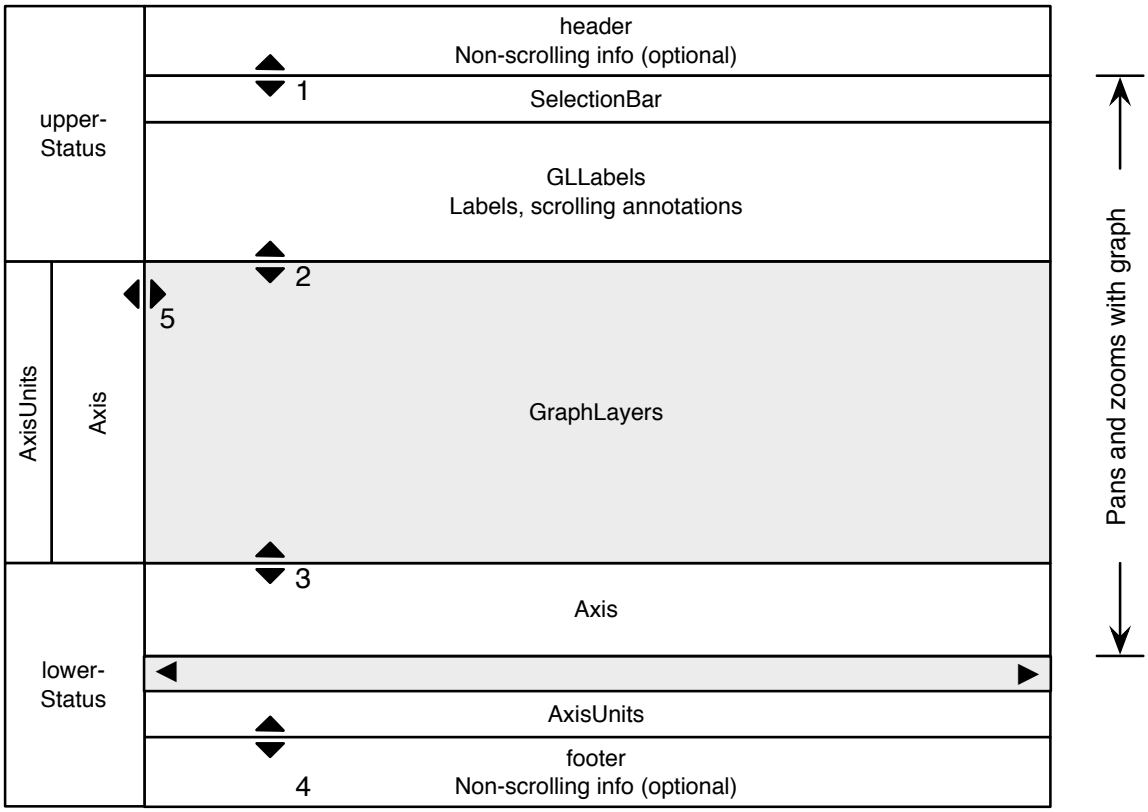


Figure 10-1. PlotSpectrum (or PlotXY) geometry and resizing rules

All PlotXYs contain the regions described above, though some may be hidden.

The regions marked upperStatus, lowerStatus, header, and footer, are text blocks, program-accessible by these names. By default, these fields are empty.

10.1.2.1 Graph Geometry

If a user adjusts the size of the graph window, the GraphLayers and GLLabel regions change to track the window size, while other graph elements remain unchanged (except to remain aligned with these elements). The boundaries between graph areas, as numbered in the preceding diagram, may be adjusted:

1. Adjusts the height of the top non-scrolling info area, shrinking or expanding the graph area (graph + labels) as needed.
2. Adjusts the boundary between the label area and graph area. If the label area is enlarged, the labels show more information, while the height of the graph decreases.
3. Adjusts the height of the AxisNumbers against graph area height. The numbering angle changes in relation to this adjustment. It can later be adjusted to taste.
4. Adjusts the height of the bottom non-scrolling info area, against the graph area height.
5. Adjusts the width of the AxisNumbers against the graph area width. This adjustment does not effect the graph zoom level, but alters the span of

wavelengths shown. The angle at which axis labels are displayed changes in relation to this adjustment, and can be overridden afterwards.

A graph's representation in the notebook changes to match the last shown settings in the graph editor window. If necessary, the graph is reduced in height or width to remain within the margins of the notebook window. Its aspect ratio is not changed.

10.1.2.2 Overlays in Plots

A graph is displayed in a graph pane. The pane consists is a rectangle subdivided into a label subpane on top, and a plot subpane on the bottom. The height of the label subpane may be zero.

The following overlays may be displayed in the plot subpane:

X/Y overlay	A scatterplot of *x* vs. *y*, where *y* is vertical and *x* is horizontal. Each point may be shown with an optional small icon. Points may be connected with line segments. Line segment color, style and thickness are settable. Opacity setting applies to both icons and line segments. If raw data includes error information, error bars may be displayed. Style, color, and line thickness are controllable.
Counts overlay	Shows each count as a vertical bar. Bars are connected together. At high enough zoom, counts optionally appear on top of each bar.
Matplotlib overlay	Displays a graph whose contents are determined by a Python program, using matplotlib.
Tick mark overlay	Displays a graticule (grid) or axis tick marks at the boundaries. Line thickness, color, and opacity are controllable, along with tick mark style.
Annotation overlay	Displays a text block anchored to a specific *x/y*, or to an *x* and the peak *y* of some data set for the *x*.

The label subpane displays the following overlay:

Ions overlay	Displays one or more labels showing information about a supplied list of ions.

10.1.2.3 Multizone Plots

A multizone plot[3] is a horizontal or vertical stack of graphs with an aligned axis. Each graph region is called a zone. For example, three zones may be stacked vertically such that each plot has a common wavelength axis. All graphs must be of the same units.

[3] I considered "strip charts" and "stacked plots" as names for awhile, but both of these have alternate meanings, so I selected multizone plots.

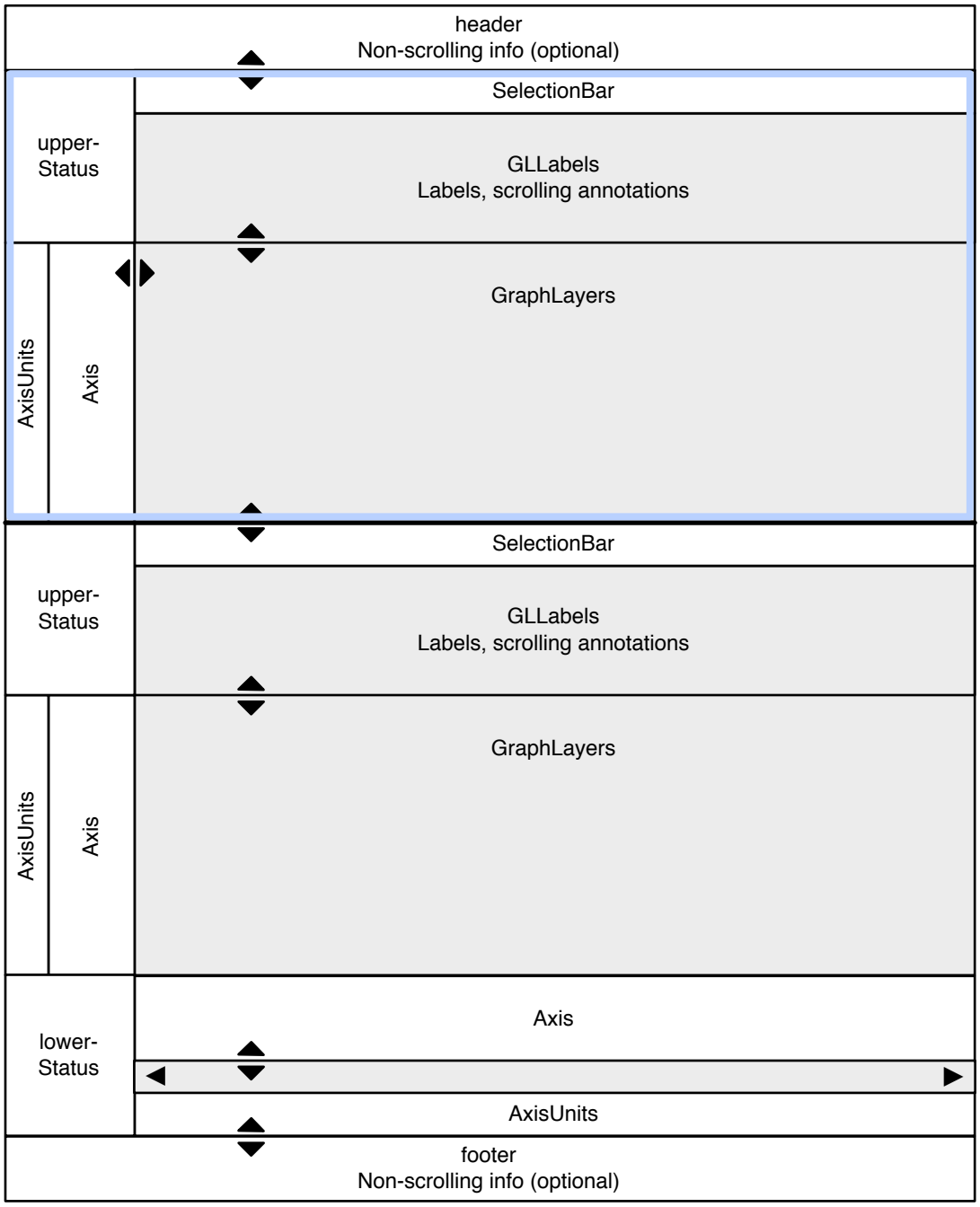


Figure 10-2. Vertical multizone plot geometry

The heavy dividing line shows how elements are associated with corresponding `PlotSpectrums`, which are stored in an array numbered from 0 (top or left graph) upward. So, for example, `zonePlot.plotSpectra[0]` has an `upperStatus` but no visible `lowerStatus` (it is actually present, but hidden), while `zonePlot.plotSpectra[1]` has both an `upperStatus` and `lowerStatus`. Since missing elements are present, but hidden, it is not an error to try to access them. Triangle pairs indicate boundaries where size is user-adjustable. The currently

selected zone is enclosed in a selection rectangle. Clicking anywhere in an inactive zone selects that zone. Zone 0 is selected above.

These diagrams look a bit cluttered, but many regions may be hidden.

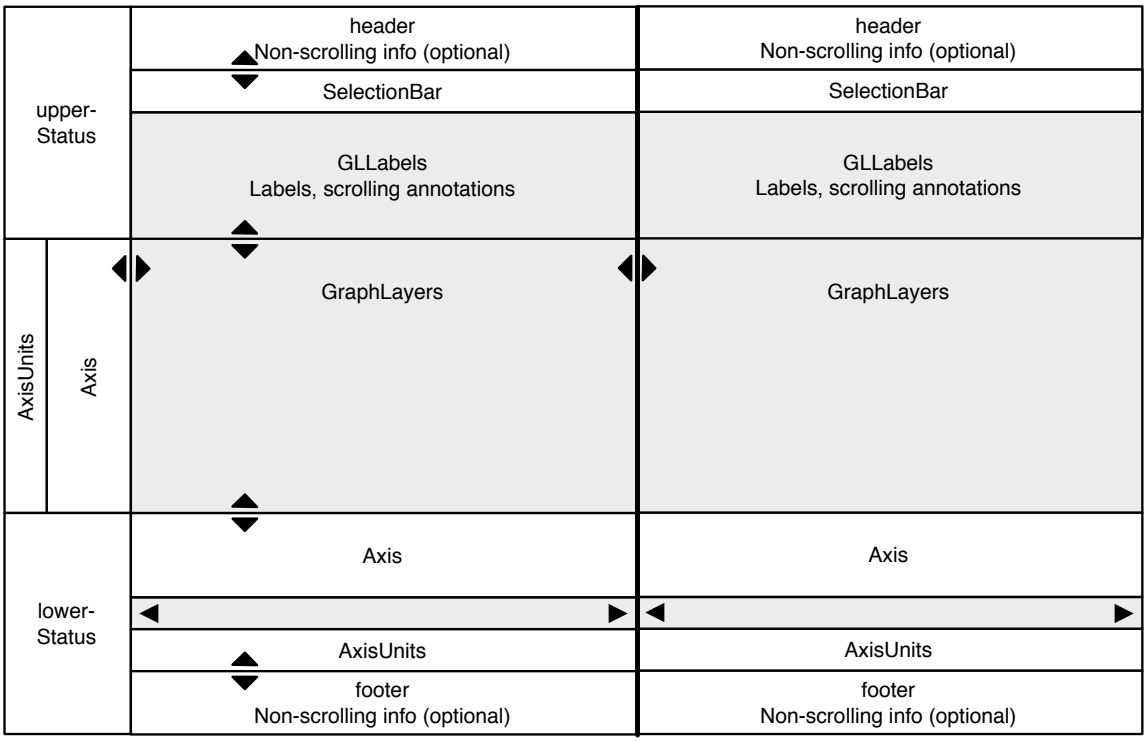


Figure 10-3. Horizontal multizone plot geometry

10.1.2.4 Wrapped Plots

Wrapped plots are like multizone plots, but instead of a synchronized axis, each pane consists of a new contiguous range of the *x* axis. For example, if the first pane of a 3-pane wrapped plot displays 1-2.5 keV, the second would display 2.5-4, and the third 4-5.5. Wrapped plots pan and zoom together.

10.1.2.5 Clicking and Dragging in Plots

Clicking in a plot has a fairly large number of possible meanings. To reduce confusion, we rely on a common technique here, using a zoom icon to toggle between normal and zoom modes. However, modifier key shortcuts permit advanced users to bypass zoom mode entirely. (Note: this table does not describe the behavior for clicking in the selections bar.)

Action	*Normal Mode*	*Zoom Mode*
Click in inactive zone [1]	Make zone active	Make zone active
Click on trace [2]	Select trace? (Probably not)	Select trace? (Probably not)
Click	Places graph cursor at this location [3]	Zooms in around location [4]
Option-click	No effect [3]	Zooms out around location [5]
Drag	Places and drags graph cursor [3]	Pans graph left/right [6]
Shift-click	Positions other endpoint; defines a new selection between graph cursor and new endpoint [3]	
Drag cursor or selection endpoint [10]	Repositions cursor or endpoint	Pans like a normal drag
Cmd-click [7]	Zooms in around this location	Zooms in around location
Cmd-option-click [8]	Zooms out around this location	Zooms out around location
Cmd-drag [9]	Pans left/right	Pans left/right
Click on line dividing plot sections	Repositions boundary between sections	Repositions boundary between sections
Click on units	Typically cycles through other units options	Typically cycles through other units options
Click on axis numbers	No effect	No effect
Click in status or header areas	Allows entry/edit of text	Allows entry/edit of text

Table 10-1. Clicking and dragging in plots

Notes:

1. Active zone denotes by a thick rectangle around zone in selection color. In single-zone plots, the zone is always active.
2. There is no defined need for selecting traces yet. If this is added, we will have to determine how to highlight a selected trace.
3. Cursor remains a pointer in this mode.
4. Cursor is a zoom in icon in this mode.
5. Cursor is a zoom out icon in this mode.
6. Cursor changes to dragging hand when a dragging begins.
7. Cursor changes to a zoom in icon when cmd is pressed. On Linux systems, "Cmd" is "Control."
8. Cursor changes to a zoom out icon when cmd-option is pressed.

9. Cursor changes to dragging hand when dragging begins.
10. When dragging a cursor or endpoint, autoscrolling is not performed. It is legal for one endpoint to cross the other one during a drag.

10.1.3 PlotSpectrum

This subclass of PlotXY is specialized to display λ vs. emissivity, and related overlays.

10.1.4 GraphLayers and CPLayerGroup

A GraphLayer is a translucent layer that displays some element of a graph, and multiple such layers are superimposed to produce a plot. Layers display tick marks, spectra, fits, labels, etc. The opacity and ordering of each overlay is adjustable. A GLSpectrum, for example, is a GraphLayer that displays a spectrum. A GLModel displays a model. A GLAnnotation displays a single text annotation. A CPLayerGroup pane controls all of these layer types.

This pane manages settings for one or more GLLayer overlays, displaying the layers in top-down order. Most layers are spectra, each of which is composed of sub-layers, including the spectrum itself and any models.

10.1.4.1 Spectra and Models

Users perform analyses by selecting models that are most likely to describe, either alone or in combination, the physical processes that account for observed spectra. The end result of applying a model is calculation of a model spectrum that may be compared to the observed data. To obtain these fits, the user must create and specify a series of model instances.

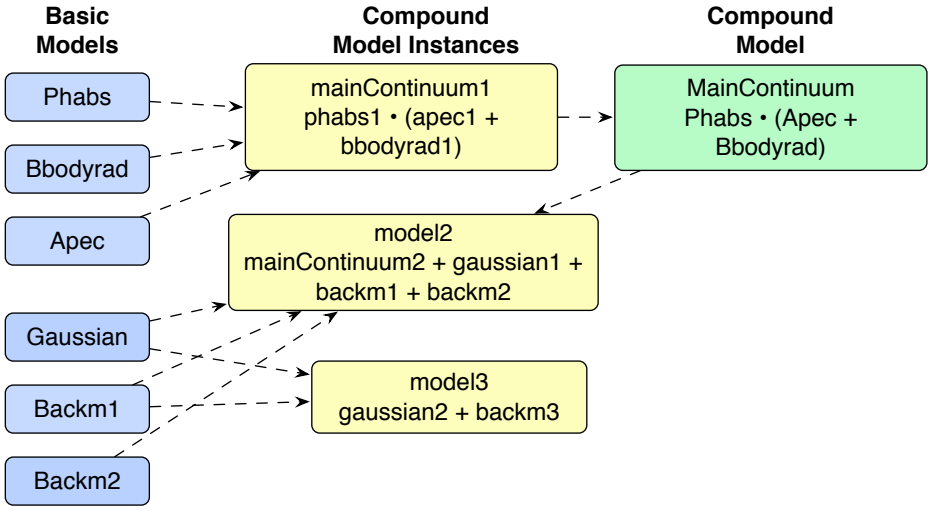


Figure 10-4. Creating model instances

Model instances are created directly from models—for example, the user could create instance gaussian1 to apply a Gaussian fit. But models are also frequently combined into compound model instances. (In this diagram, the dotted arrows means "instantiated from.") Here, the user creates a compound model instance

called mainContinuum1 to model some observed spectra. Next, she creates a second model instance that combines the mainContinuum spectrum with a gaussian, and two background models; this will be called model2. It is not possible to create a model instance directly from another model instance, so she first saves mainContinuum1 as a new model, MainContinuum, which is added to their list of models for future use. Then she includes this in model2. She next creates model3 as shown.

In the diagram following, the user creates two more model instances, backm1 and backm2. She associates model spectra (spectrum1-3) or background spectra (back1-2) with various model instances, and specifies response matrix files (and possibly ancillary response files, not shown) for the spectra. Assuming for the moment that each model's parameters are fully specified, everything necessary has been specified to generate each of the graphs and other fit results.

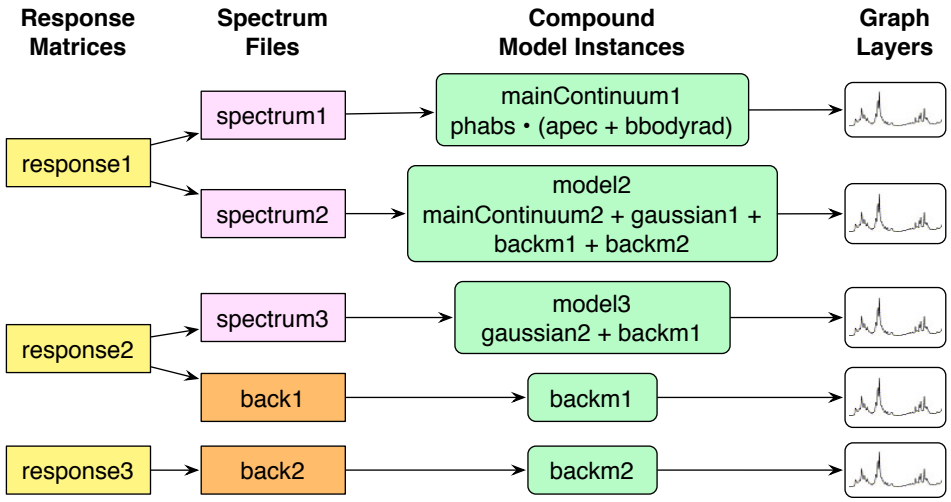
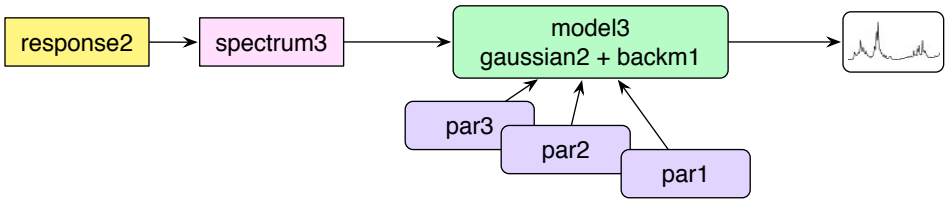


Figure 10-5. Dependency trees for several graph layers

Take spectrum3 as an example. We wish to calculate a fit that is a sum of a Gaussian model and a hypothetical fit called "backm1" that represents background radiation. The fit requires access to response matrix response2 as well as spectrum3. Each model's parameters



ATSAL must provide a way to construct these dependency trees, display their results, and minimize the amount of computation necessary to reflect a change in parameters that drive a model. Later sections address the construction of these dependency trees. Once such a tree is constructed, ATSAL uses the tree to optimize recomputation. For example, if the user selects a different response matrix file for a given spectrum, ATSAL recomputes that spectrum's entire model. If the user modifies a parameter for the gaussian model in model2, ATSAL recalculates

the Gaussian and combines it with the previously calculated phabs1, backm1, and backm2 results, producing updated fits for spectrum1 and spectrum2.[4] In the diagram below, these spectrum files are highlighted in red to indicate that their corresponding fits are updated.

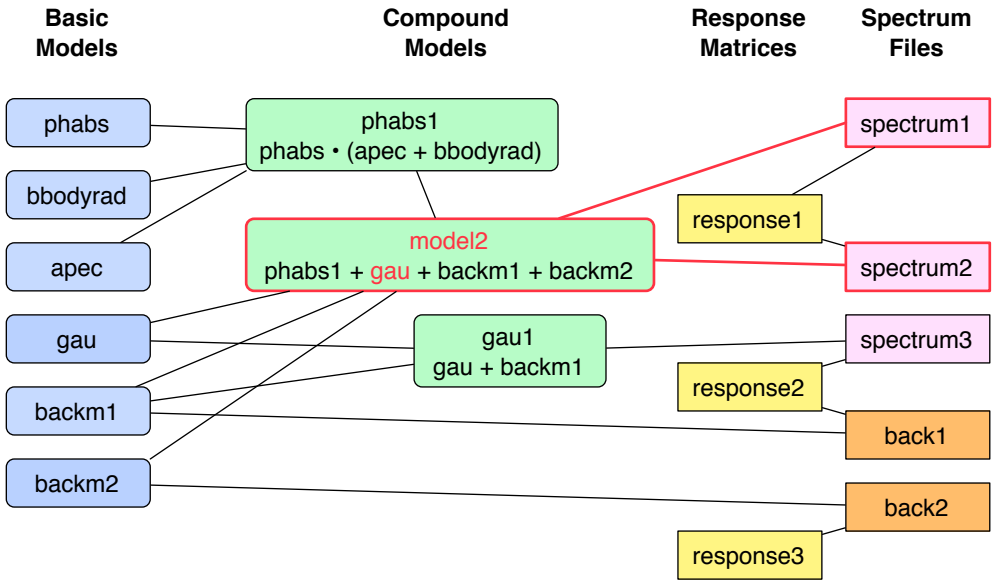


Figure 10-6. Modifying a parameter to the gaussian model repeats the calculations shown in red

Note that the modification of the gaussian parameter is shown inside the model2 box, not the blue box showing the basic model. This reflects the fact that the parameter change is specific to this analysis' use of model2. Changing its parameters does not affect compound model gau1.

Because dependency tree construction and parameter specification may take some time, ATSAL does not draw the results every time a change is made in the user interface. Instead, a Refresh command brings the graph up to date.

10.1.4.2 CPLayerGroup

The CPLayerGroup panel manages all the overlays that make up a single graph. It can overlay one or more spectra and annotations. Within each spectrum, it can overlay multiple fits. This is also the user interface that constructs dependency trees.

The example below shows two spectra, the first overlaid with two models:

[4] The actual amount of recalculation necessary may be larger, depending upon possible XSPEC constraints.

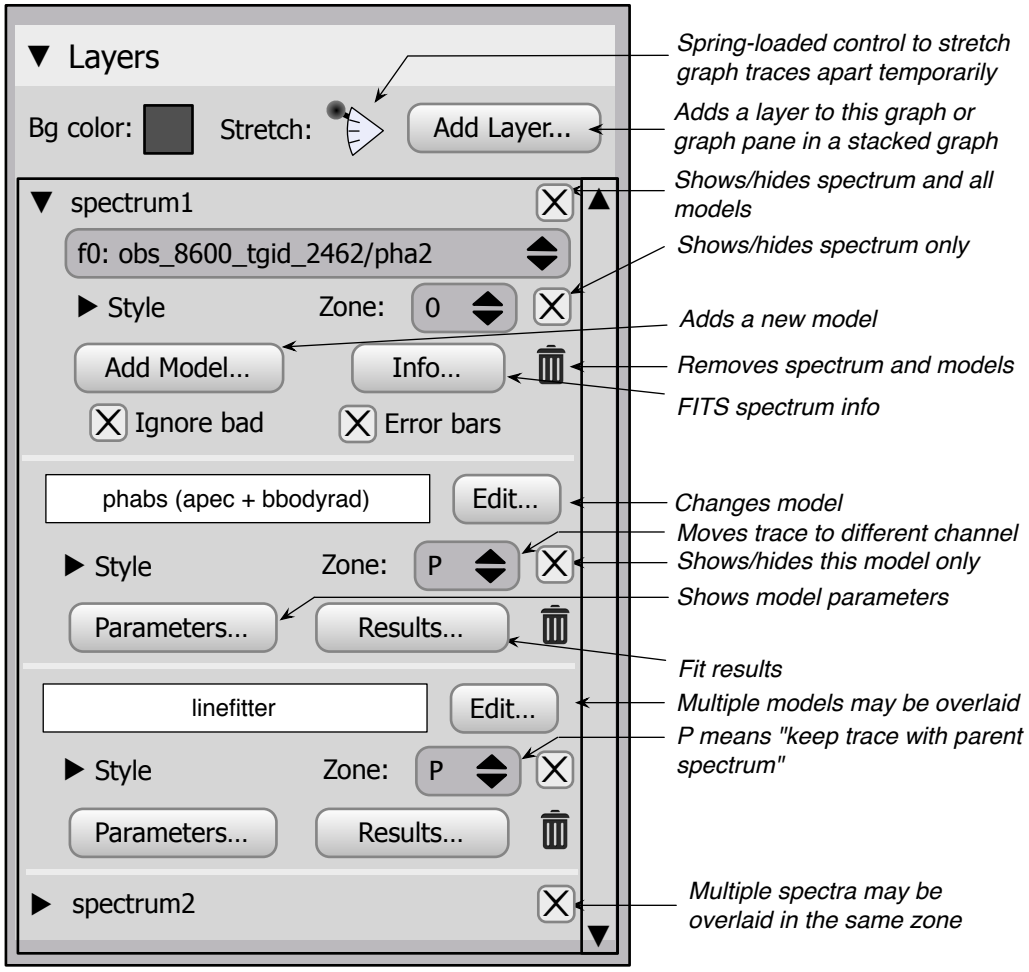


Figure 10-7. CPLayerGroup showing two spectra, the first of which has two models

Bg color sets the background for the entire graph. This defaults to "A" for auto (probably white). The Stretch control spreads graph traces apart vertically.[5] As the lever is dragged downward, multiple graph traces spread apart by a proportional amount. When the lever is released, they coalesce again. This makes it easier to disambiguate multiple overlaid traces, for example, to temporarily separate two nearly overlapping models. Add Layer... adds a layer, which is a spectrum file or an annotation layer.

The first layer in this example is a spectrum file. Note that the spectrum layer name is "spectrum1," although the spectrum file itself is named "pha2." The name "spectrum1" is the Python name of the layer, and since we may wish to run the same Python program with different input files, we usually don't want to refer to the file by name. Also note that the directory immediately containing the spectrum file is shown, because this is the extraction directory that also contains related files.

The Add Layer… command displays the Add Layer dialog:

[5] Yes, I know the present rendering looks silly—it is supposed to suggest a spring-loaded lever.

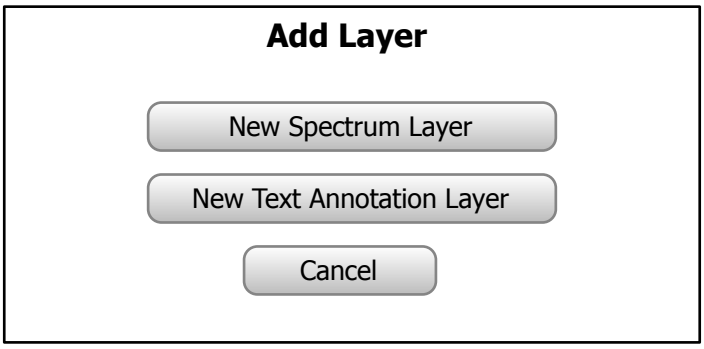


Figure 10-8. Add layer dialog

New Spectrum Layer displays a dialog for setting the parameters for the new spectrum:

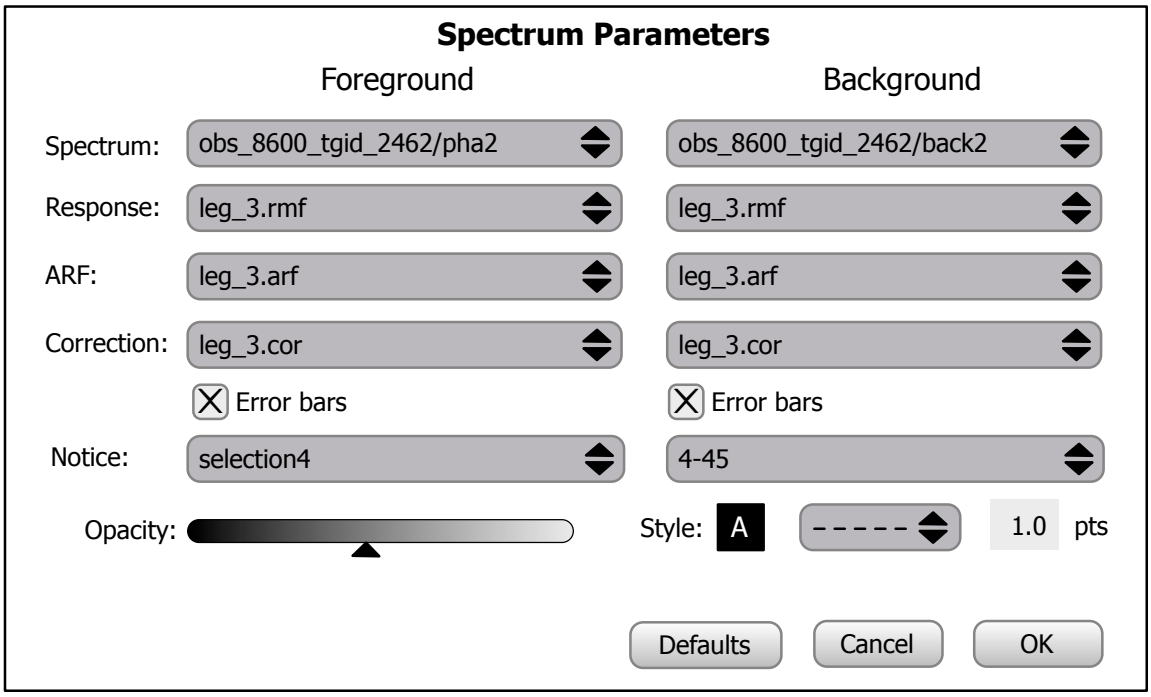


Figure 10-9. Spectrum parameters dialog

The spectrum file is chosen from the list of extractions enabled via the spectrum files tool. If keywords are set suitably, all of these parameters choose reasonable defaults once the spectrum itself is chosen, although the defaults may be overridden.

"Notice" allows a subregion of a spectrum to be "noticed," that is, counts outside of this region to be ignored. The noticed region may be expressed with a wavelength selection as shown above, or specified as a range of counts or a range of wavelengths.

After adding the spectrum "pha2" from the extraction "obs_8600_tgid_2462," the pane looks like this:

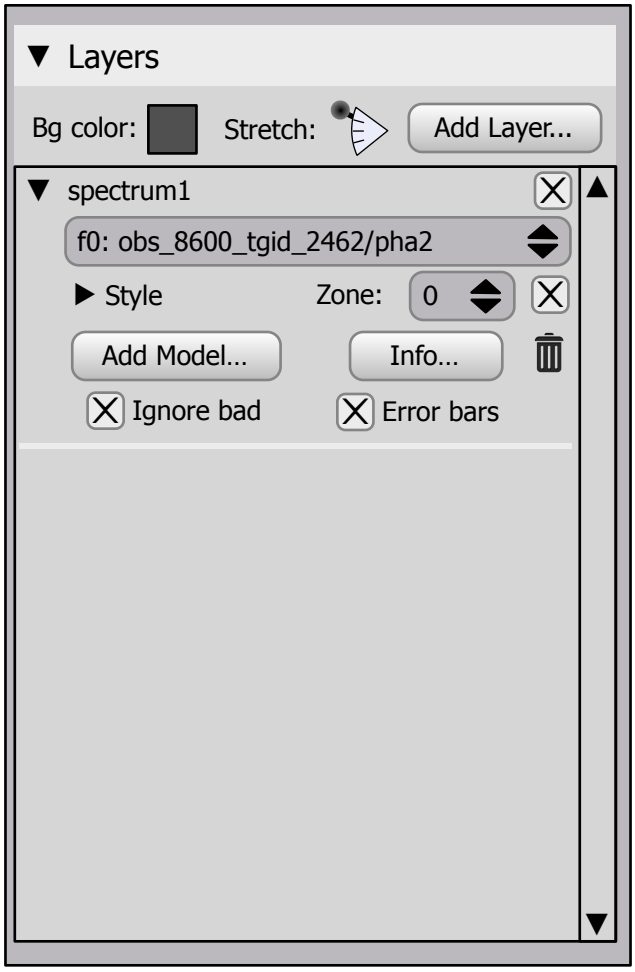


Figure 10-10. CPLayerManager after adding a spectrum

The spectrum also appears in the graph pane. By default, the new graph is placed in zone 0, and only zone 0 is actually displayed in the graph. Subsequent spectrum layers default to new zones. Adding a graph to a zone that is not displayed causes it to be displayed (along with any intervening zones if needed). Info… displays the FITS metadata. The uppermost checkbox shows or hides the spectrum and all models (there aren't any models yet in this example). The next checkbox shows/hides only the spectrum, independent of any models. The trash icon removes this spectrum and all its models from the plot, after asking for confirmation. "Ignore bad" ignores information in the spectrum that is known to be bad, and is set by default. "Error bars" shows the spectrum with error bars, also enabled by default. If this is turned off, a histogram display is shown instead.

Style settings are hidden by default, to avoid distraction while working with models. Clicking "Style" displays this subpanel.

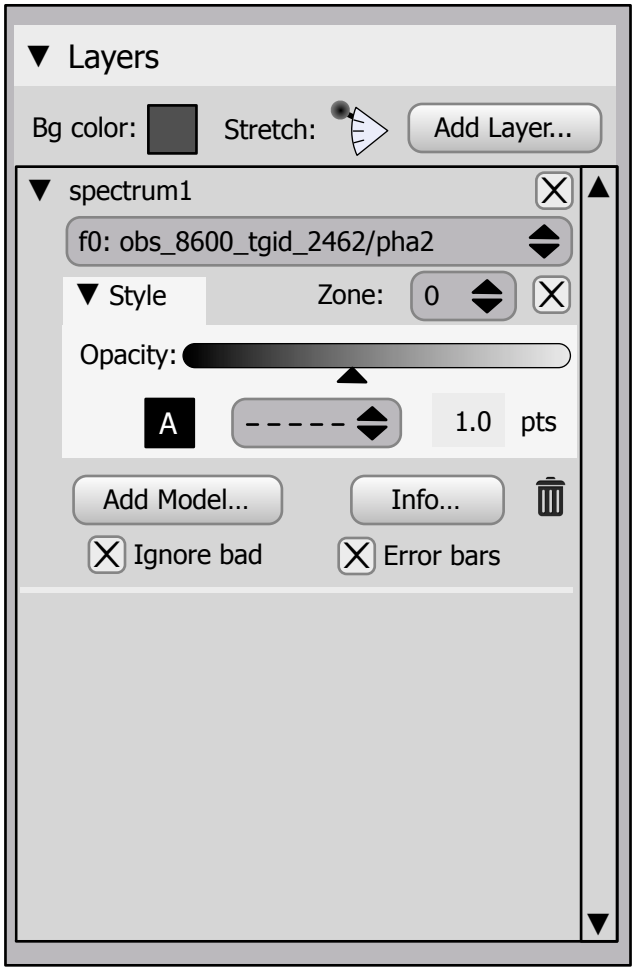


Figure 10-11. Spectrum style settings

The user selects Add Model... to select a model. The model dialog allows selection of XSPEC models, ATSAL-defined model combinations, or user-defined model combinations. This dialog is described in the following section.

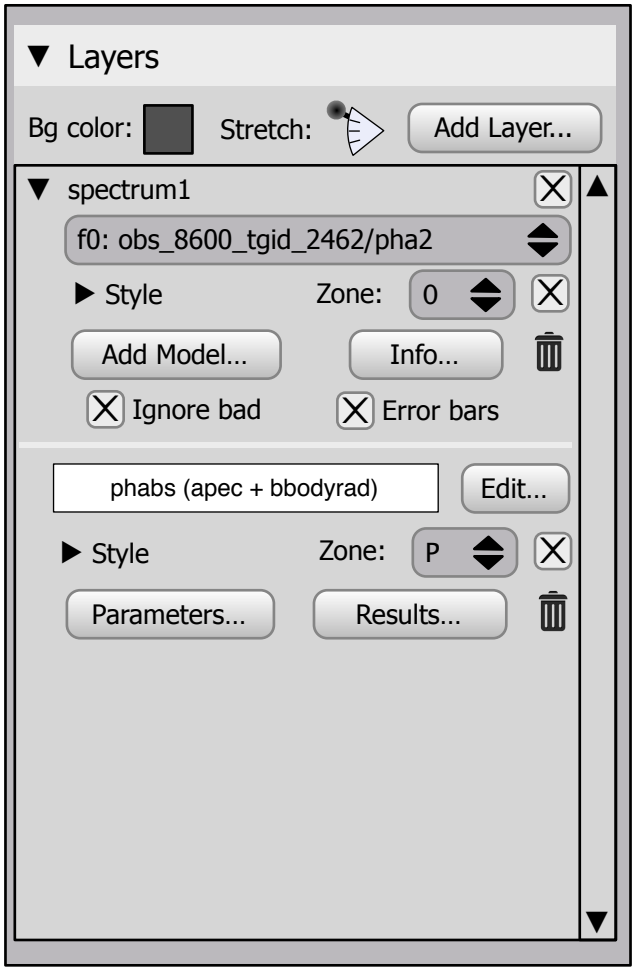


Figure 10-12. Layer after adding a model

Here, the user chose a combined model named "phabs (apec + bbodyrad)." The model expression can be edited here, or the Edit... button pressed to bring up the model editor. The model's fit appears on channel "P," that is, "parent," the same channel the associated spectrum is shown on.

The Parameters… button displays the parameters for the model, along with some graph style parameters. The Results… button displays a tabular view of the results for this fit. It is also possible to insert either the parameters or the results for a model directly into the notebook, to view them more easily. This is done via the parameter and model results tools.

The trash icon removes the model while the checkbox at the right simply hides it from the graph. It is often preferable to hide a model instead of removing it, since once it has been calculated, it will be retained and may be redisplayed later without recomputation. The button showing the model's name brings up a dialog in which to edit the model parameters.

Users can add multiple models, each of which are superimposed on the same graph by default. Their channel number is P (for parent), which displays the fit in the parent channel, but fits can be moved to different channels.

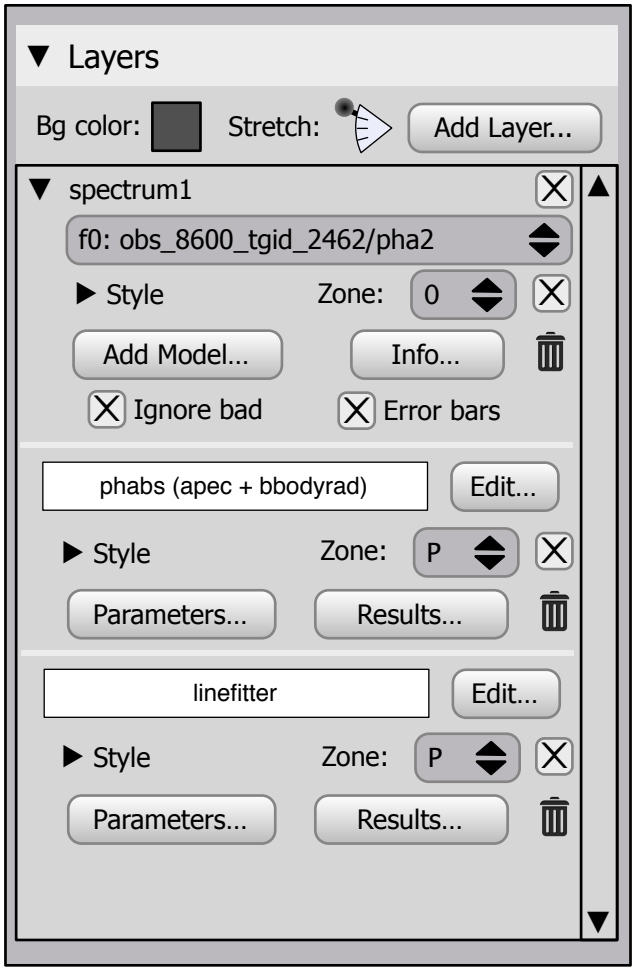


Figure 10-13. Layer after adding a line fitter

Here, the user added a second "model," a line fitter. This model evaluates the data in search of particular strong lines, and labels the lines according to settings for the fitter.

Layers uppermost in the list are displayed on top in the graph. If the checkbox beside the spectrum file name is turned off, the spectrum and all its fits are hidden. This might be desirable if several spectrum files are displayed in the same channel. The other checkboxes along the right turn off individual sub-layers. The order of spectrum layers, and the order of models within a layer, may be altered by dragging a panel up or down. Layers may be rearranged, or sub-layers within a layer, but all the sub-layers for the uppermost layer appear above all the sub-layers from a lower layer. This provides another way, in addition to the stretch control, to temporarily disambiguate each line when multiple lines are overlaid.

If the user prefers to separate the data and model graphs into separate graphs in the notebook, one way to do this is to set up the data and its overlays, then duplicate the graph a few times, setting each individually to display or hide portions of the graph. Computed fits are shared among the duplicated graphs if the parameters driving them are identical, so this operation is quite efficient from the perspective of memory or computes.

10.1.4.2.1 Add Model… Dialog

This dialog displays the model editor, used to browse models and compose new model expressions.

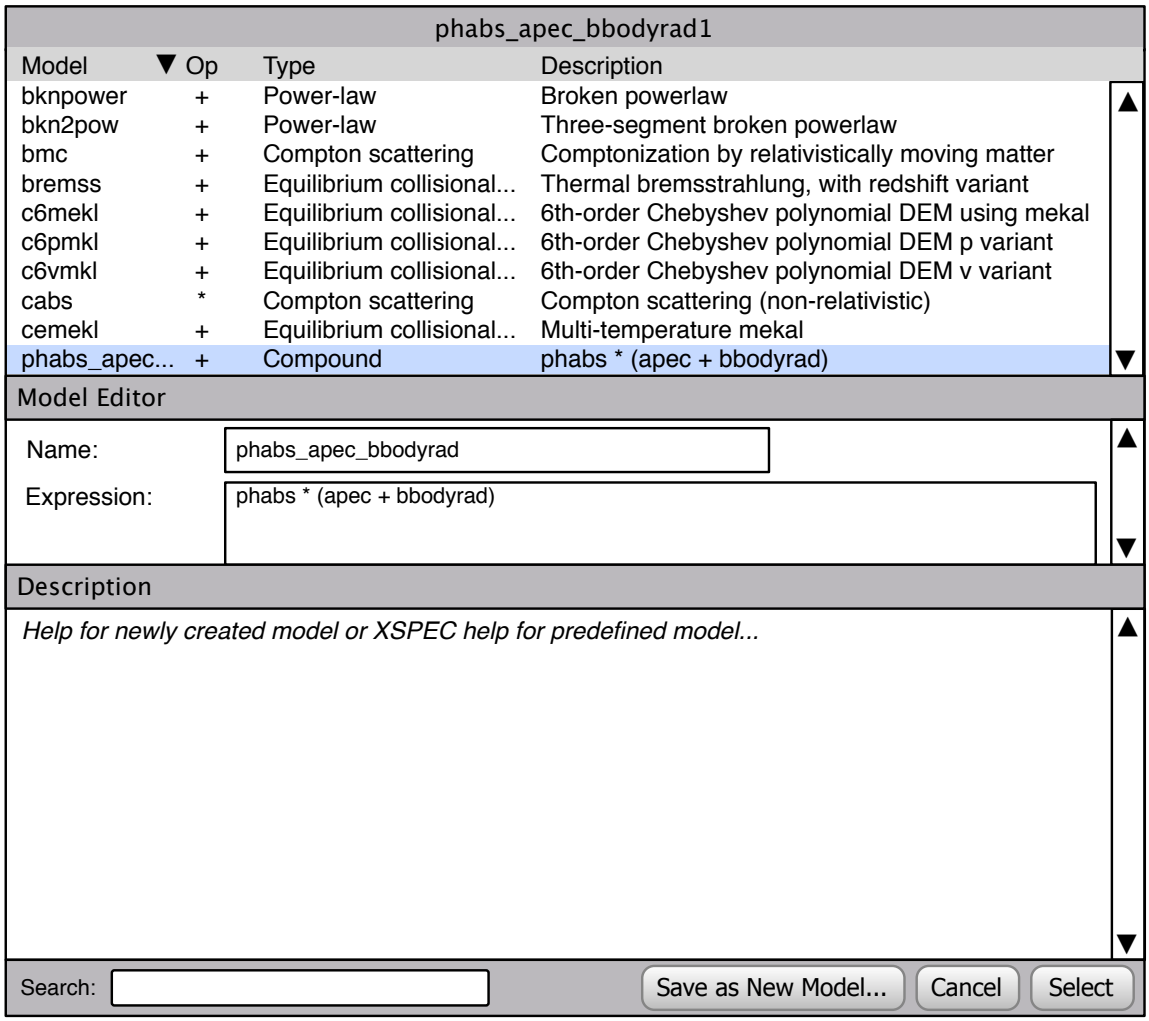


Figure 10-14. Model selection dialog with combination model

The model editor pane is used to build an expression that combines results of several models. Help is provided for ATSAL-supplied model combinations, and may be added or edited for user-created models.

When a model is saved for later use, its current parameters are saved as well. When a model is selected and added to an expression, one of the following happens:

- If the model to be inserted is a simple model, but it is inserted as it is named. For example, if model bknpower with new parameters is named "fred," it will be inserted with the name fred to indicate that the starting parameters are different.

- If the model to be inserted is a compound model, it is expanded into its components at insertion time, to permit subsequent editing. Hence if model "fred" is defined as "phabs * apec," it is inserted as "phabs * apec."

In both cases above, the inserted models are initialized with parameters stored in the templates. Newly created templates become a permanent part of the list of models, for future use with this notebook or others. ATSAL-supplied model templates cannot be deleted, but user-created templates are deleted with the delete key.

Design Note

New models may also be created in Python and registered with ATSAL for display in the model selection list. This provides a way to handle more complex types of compound models, or to add new types of basic models.

In lists, models may be sorted by name; operation (additive, multiplicative, or convolution); or by type:

Type	*Models*
Accretion disk	disk(bb|ir|m|o|pbb|pn)
Blackbody	(z)bbody, bbodyrad
Compton scattering	bmc, cabs, comp(LS|PS|ST|TT|bb), nthComp, simpl
Cooling flow	(mk)(vm)cflow
Cyclotron absorption	cyclabs
Dust scattering	dust
Emission line	(z)gaussian, diskline, kerrdisk, laor(2), lorentz
Equilibrium collisional plasma	(b)(v)apec, (z)(v)bremss, c6(p)(vm|me)kl, ce(vm|me)kl, equil, (v)meka(l), (v)raymond, smaug
Gamma-ray burst	grbm
Line-based analysis	linefitter
Neutron star atmosphere	nsa(grav|atmos), nsmax
Non-equilibrium collisional plasma	(v)(g)nei, (v)(n)pshock, (v)sedov
Pair plasma	nteea
Photoelectric absorption	absori, edge, partcov, pcfabs, (v)phabs, pwab, smedge, swind1, tb(var)abs, tbgrain, varabs, wndabs, zvfeabs, zxipf
Photoionized plasma	absori, redge, swindl, xion, zxipcf
Positronium continuum	posm
Power-law	bkn2pow, bknpower, cutoffpl, pegpwrlw, plcabs, (z)powerlaw
Reddening	redden, uvred, zdust, zsmdust
Reflection	(b|p)exr(a|i)v, (i)reflect, refsch
Synchrotron	srcut, sresc

Table 10-2. Physical processes and XSPEC models (from the Handbook of X-ray Astronomy*)*

10.1.4.2.2 Editing Model Parameters

After adding a new model as a sub-layer for a spectrum, users next supply parameters for the model, using the button with the model's name. This displays a dialog for setting the parameters that drive the model.

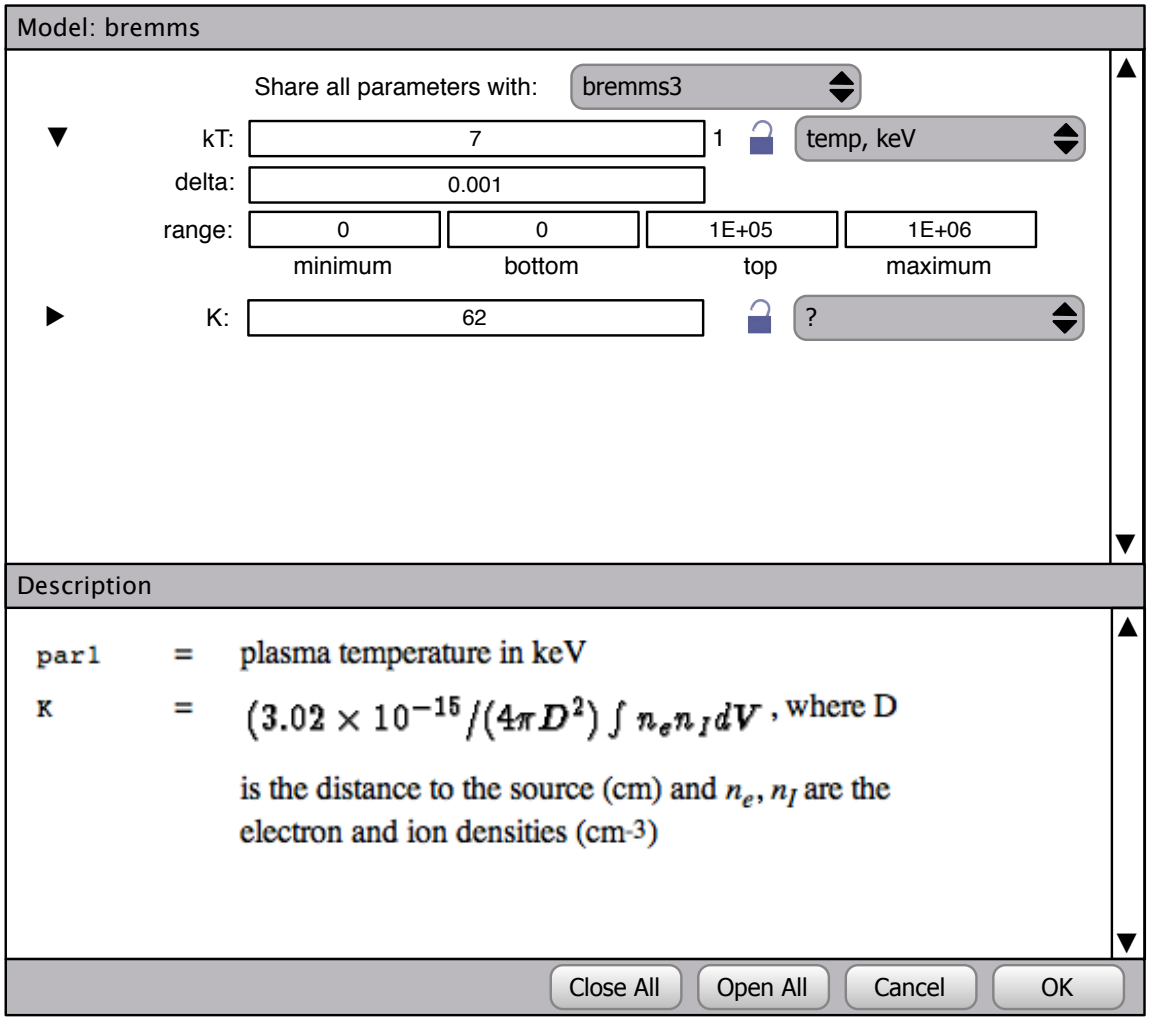


Figure 10-15. Params… dialog

The top pane of the params dialog lists each of the parameters that are inputs to the model. These may be expressed directly in numerical form, or retrieved from other Python objects. Where appropriate, they may be expressed in the user's preferred units. Each parameter consists of summary fields (the value, parameter number, frozen state, and units) and detail fields (delta and range parameters). Detail fields take on default values and are not shown by default. They are shown for individual parameters using the disclosure triangle at left. At the bottom of the window, "Close All" and "Open all" close/open all the detail fields at once.

The model's help pane is displayed to assist with parameter entry. If the model is a composed model, help for the composed model is shown first, followed by help for each of the component models. Similarly, for combination models, the par-

ameters are solicited in groups for each model component. The groups are listed in the same order as they occur in the combination expression.

The "Share all parameters" option shares this parameter block with one associated with a different model instance of the same model. Changes made to either instance apply to both.

[THK: This section does not yet address models that have other XSPEC steps as prerequisites. I think these can be handled by including the prerequisite steps as an implied part of the model. But I don't understand this well enough yet.]

When a spectrum plot tool is duplicated, the parameter settings for any models are also duplicated. Subsequent changes to the duplicate's model parameters only apply to the duplicate.

At the bottom, controls that determine the style of the displayed fit are shown.

Element abundance parameters are handled as a special case. Tables for common cases are supplied with ATSAL, and users may add tables by copying standard tables and editing them via Preferences. New tables are named by users and become available to all notebooks on the user's computer. Within the model parameters dialog, models that accept element abundances as a range of parameters are replaced by an interface like the one shown below. (Poetic license here—bremms doesn't actually accept a set of abundance parameters.)

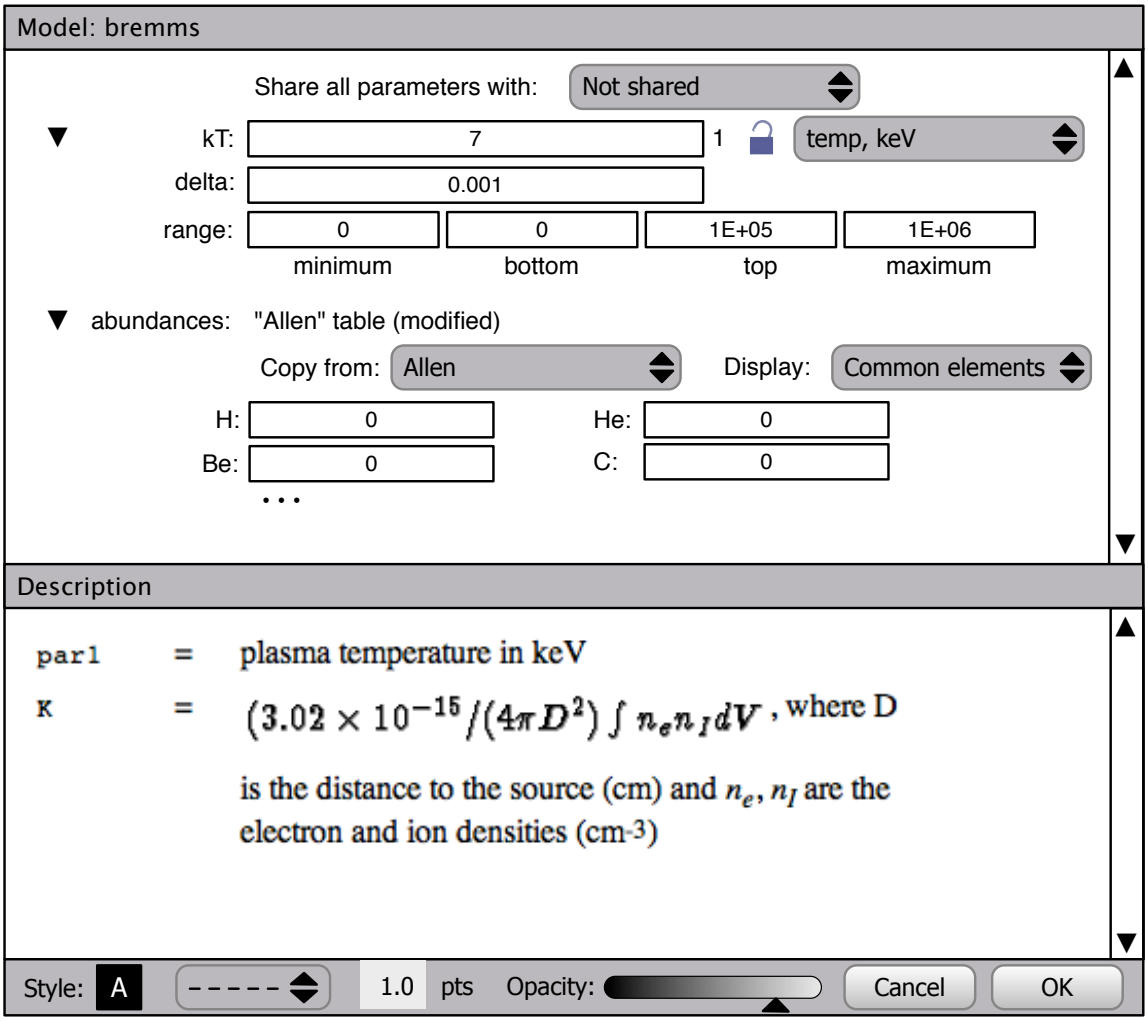


Figure 10-16. If bremms actually accepted abundance parameters, it would look something like this

The summary line for the abundances parameter indicates that the user started with the standard Allen table, but modified it for this model. The "Copy from" pop-up replaces all the abundance parameters with those from the specified table (tables are created in the Preferences dialog). The Display option shows common elements, less common elements, or rarer elements. Changes made in this parameter block apply only to this parameter block, not to the original table.

This interface allows abundances to be modified in groups, simplifying both initial settings and fine-tuning.

10.1.4.2.3 Annotation Layers

In addition to spectra, users can add annotation layers. Each such layer allows a user-created label to be associated with an x/y position in the graph.

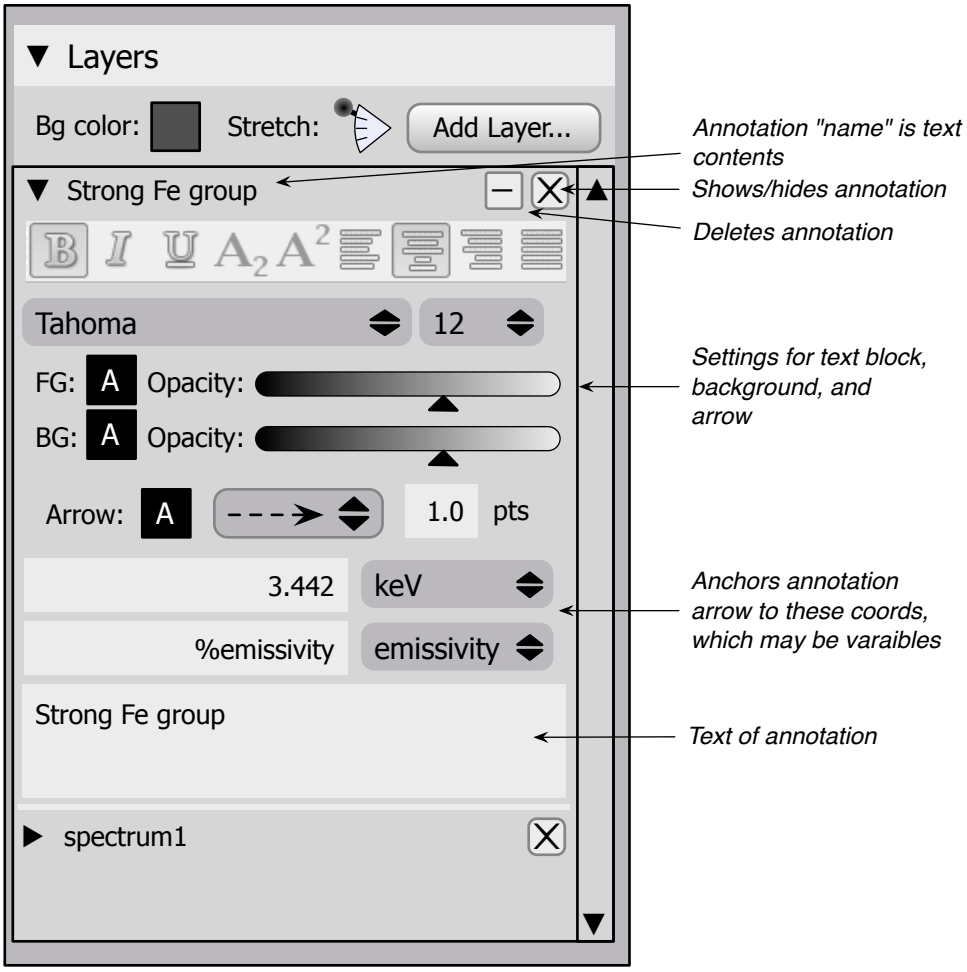


Figure 10-17. CPAnnotation

An annotation layer displays text blocks on the graph that are connected by lines or arrows to *x/y* locations in the graph. Individual annotations may be shown or hidden. Coordinates of an annotation may be set as Python variables. Python variables may also be embedded in the text of the annotation. For example "Strong %fitter.ion(1) line" could be used to label a particular line with the ion making the largest contribution.

If the emissivity is left blank, the *y* coordinate of the arrow points to the highest *y* value shown at this *x* location. When a new annotation is created, the default *x* coordinate is the current position of the spectrum cursor, and the default *y* coordinate is empty, defaulting as described above. Users may reposition either coordinate by dragging the arrowhead to a new position. The text block portion of an annotation appears within a resizing rectangle. The width of the rectangle is user-determined, and neither the width of the text block nor the size of the text changes when the graph is zoomed. Text is wrapped within the width onto as many lines as are needed.

Labels pan with the graph. As the graph is zoomed out, text blocks are repositioned along with their arrows to avoid overlap. When it is no longer possible to

display all the text blocks, some are suppressed. Those lower in the layer ordering are hidden first, so it is possible to prioritize the label display order.

10.1.4.3 GLTickMarks

This overlay displays tick marks as marks or lines across the graph. Tick marks have the following properties:

- TickMarkStyle: TMSGraticule displays lines across the entire graph, similar to the graticule on an oscilloscope. TMSTicks displays short tick marks along the top and bottom or left and right of the graph area. The default is TMSGraticule.
- The major line width is the width, in points, of major tick marks (or lines). The default is 1 point.
- The major line color defaults to light blue, with an opacity of 50%.
- The minor line width is the width, in points, of minor tick marks (or lines). A value of zero suppresses minor tick marks. The default is 0.5 point.
- The minor line color defaults to light blue, with an opacity of 50%.

The tick marks layer may be turned off using the checkbox at upper right.

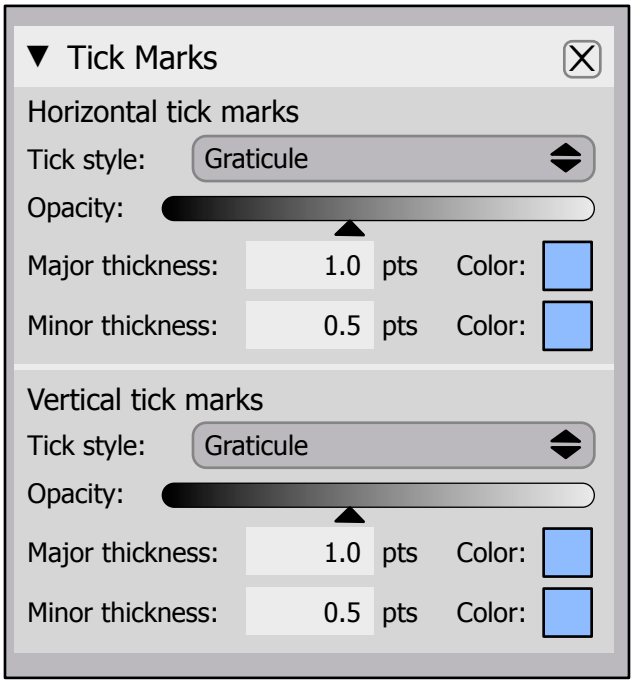


Figure 10-18. CPTickMarks

10.1.4.4 CPEmissionLines

This panel offers another way to keep track of stronger emission lines near the graphics cursor. Clicking on the graph repositions the cursor at a wavelength, and this pane shows the emission lines near the cursor, with the cursor's position shown in red. The minimum emissivity control limits the lines shown. Only the

strongest are shown at the leftmost position, and all are shown at the right. This control only applies to the lines in this box, not those on the graph itself. Adjusting the spectrum cursor recenters this list, but adjusting the list (using a scroll wheel) does not alter the spectrum cursor position. However, if the user clicks between two rows in the list, or clicks on the red line and drags it, the spectrum cursor is also altered.

The checkbox at upper right disables display of emission line labels on the graph.

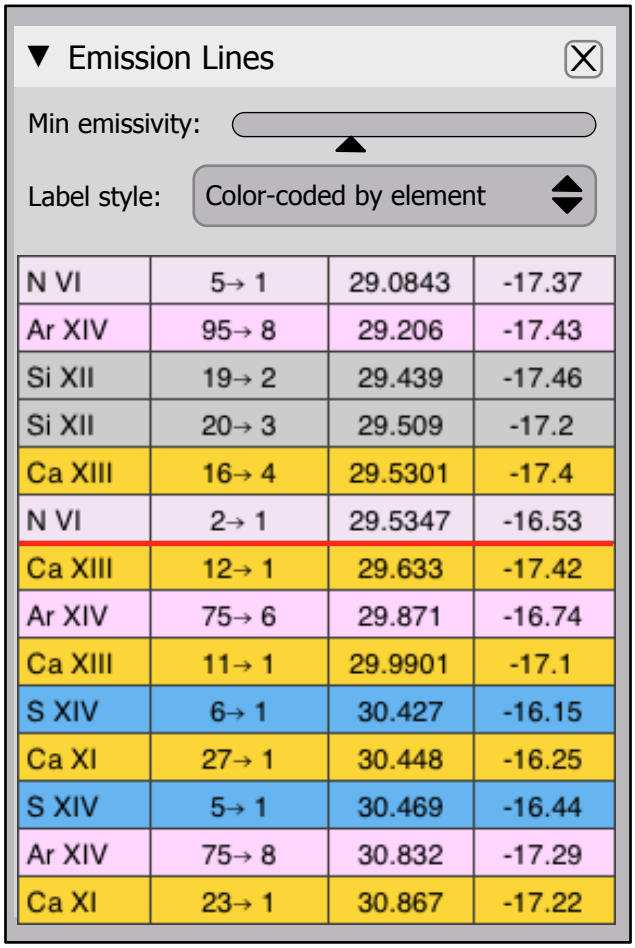


Figure 10-19. CPEmissionLines

10.1.4.5 GLSpectrum

A GLSpectrum displays a single instrument spectrum.

10.1.4.6 GLAtomDB

This displays theoretical data from the AtomDB database. The checkbox at upper right disables display of AtomDB emission lines. It is off by default.

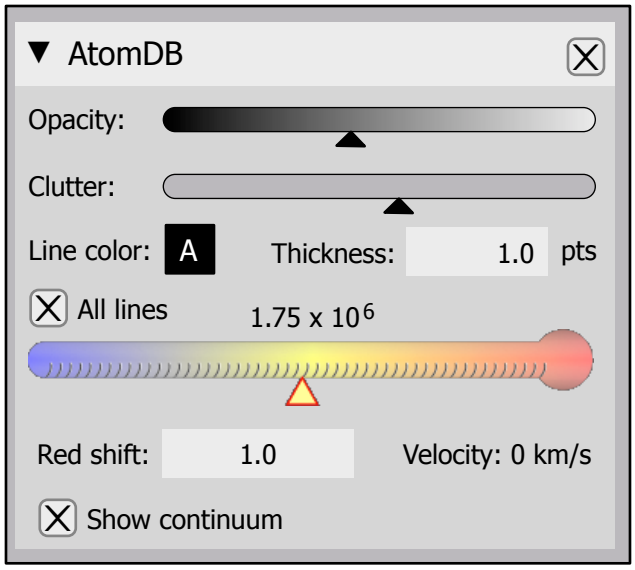


Figure 10-20. CPAtomDB

10.1.4.7 GLCurveFit

Displays a curve fit.

[THK: Need to define various types of fits and the corresponding user interface.]

10.1.5 Non-scrolling Areas

Plots have up to four regions where non-scrolling text may be shown. These areas are QTextEdits, and use a CPTextBlock to compose text and variables. The header and footer areas may be hidden on a graph. The background color of these text blocks matches the graph's background color.

Clicking inside a non-scrolling area activates the CPTextEdit collapsible pane. Note that while most collapsible panes apply to the graph as a whole, this one applies to the currently selected non-scrolling area.

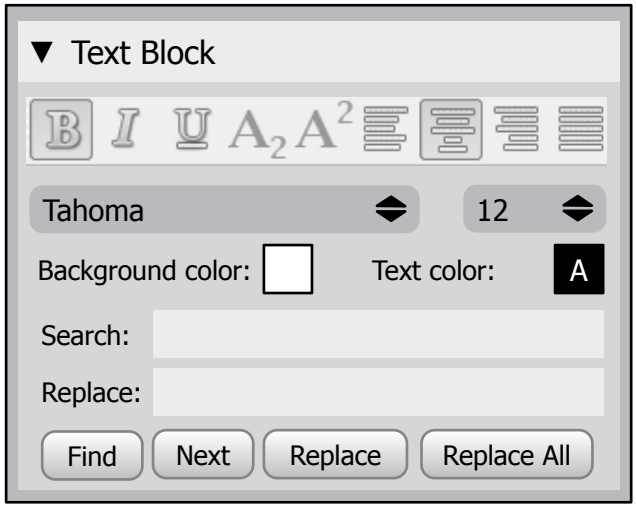


Figure 10-21. CPTextBlock

Named variables are included in text with a prepended '%'. Use two to get an actual percent sign. ATSAL standard variables begin with a leading underscore, so e.g. `%_runDate` evaluates to the date of the most recent run and `%_spectrumFile` evaluates to the filename of the spectrum being displayed (or that of the first such file if multiple files are being displayed). `%author` is an example of a user variable, and it will evaluate to "<undefined>" unless its value is set from a Python program:

```
spectrumGraph1.set("author", "Anders Ångström")
```

Note the distinction that system-defined variables are set automatically, while user-defined variables must be explicitly set via Python.

10.1.6 SelectionBar

The SelectionBar is a narrow bar above each graph zone which is used to show GLSelections. The bar represents these selections even at low zoom levels, where they might otherwise be invisible.

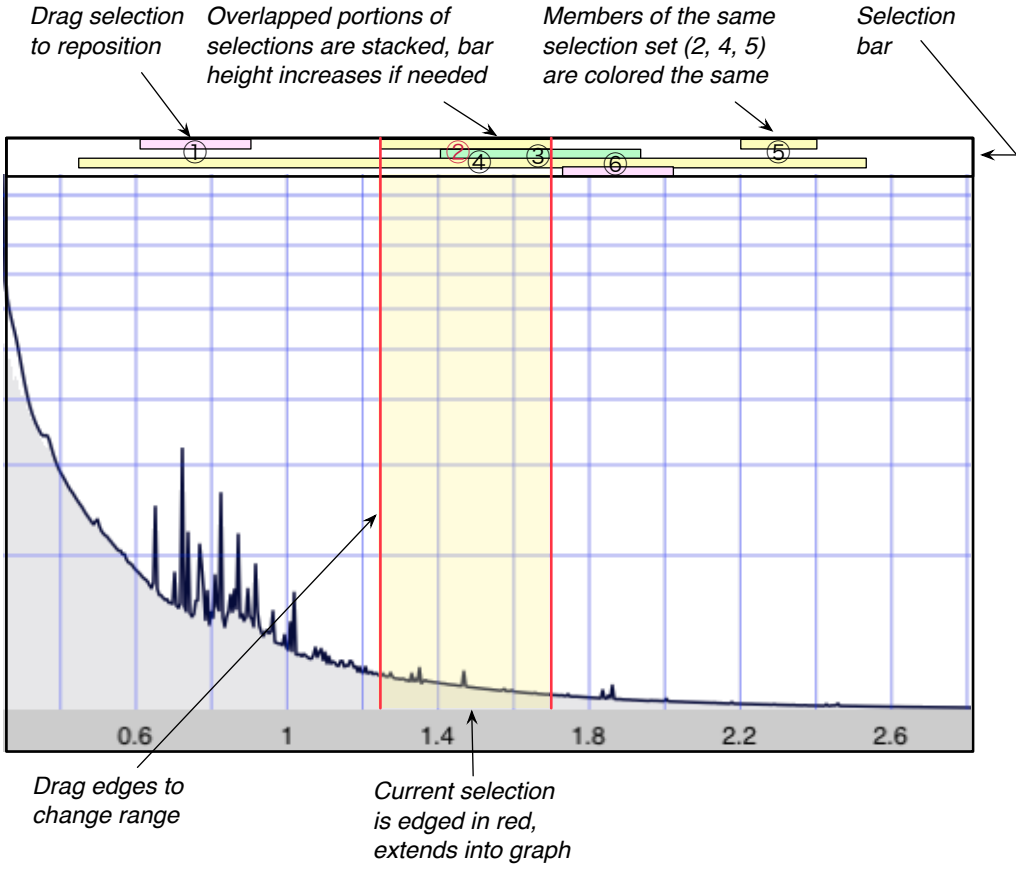


Figure 10-22. Selections example

Six selections are shown above, several of which partially overlap. If the zoom level is high enough, the width of the rectangle representing each selection represents the true range of the selection. But each selection has a minimum width (about a quarter inch) to indicate the presence of selection(s) too narrow to

display. The current selection is extended over the graph as a translucent overlay, and red lines show its boundaries.

Selections are colored according to the selection set to which they below, so selections 2, 4, and 5 are members of the same selection set.

Overlapping selections are stacked into different bands in the selection bar. The selection bar has a minimum height, and it increases in height if needed to accommodate higher degrees of overlap. Selections are numbered with numbers centered horizontally in the range, possibly moved off center when part of the selection is out of view or two numbers collide. Clicking a different selection bar selects it.

The current selection may be:

- Deleted with the delete key
- Dragged to a new location, repositioning both endpoints at once
- Double-clicked to open the dialog shown below
- The range may be adjusted via the dialog, or by dragging the left or right edges (dragging only works for the current selection).

To create a new selection, the user can click in the graph to position the graph cursor at either endpoint, then shift click to specify the other endpoint. A new selection is created with the shift click. Alternatively, New Selection creates a selection at an arbitrary location within view in the current zone. In either case, the new selection is added to the current selection set—the selection set that contains the last current selection.

The selection dialog looks like this:

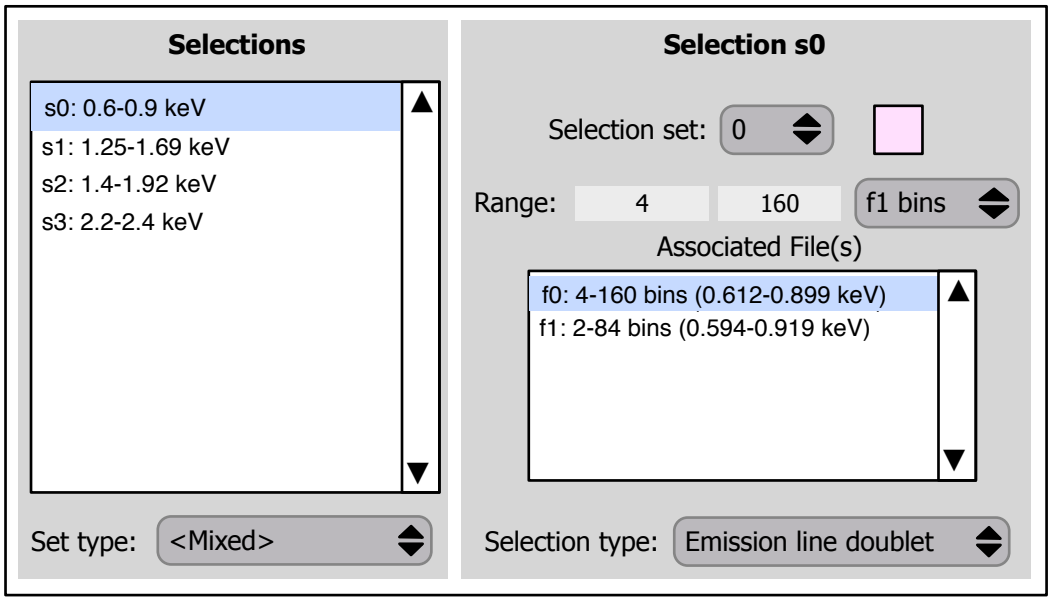


Figure 10-23. Selections dialog

All the selections for the current graph zone appear on the left, with the current selection highlighted. Current selection details appear on the right. By default,

each new selection is in selection set 1, but any selection may be assigned to a different selection set. The selection set's color is shown. The range may be viewed or set numerically. In addition to expressing the range in units of keV or Å, users can express bin ranges for a chosen associated file.

If a range is expressed in keV or Å, any associated file bin ranges are determined by rounding to the nearest bin boundary. Alternatively, users can enter bin ranges directly, which updates the wavelength range to match. If multiple observed spectra are shown in the same zone, there are multiple files associated with the selection. If this is the case, a bin range expressed for one file may not map cleanly to other file(s), but the closest available bin ranges for the other files are chosen.

In the example below, assume the user precisely chose a range of bins from a file, by entering the starting and ending bin number.

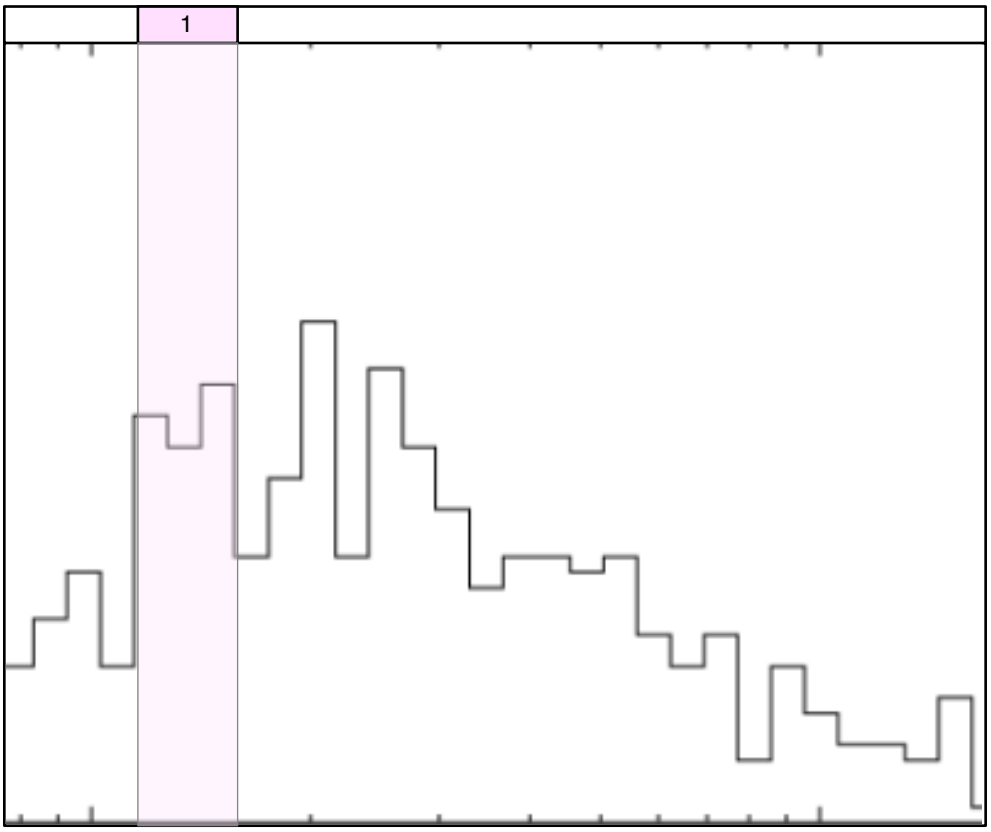


Figure 10-24. Selections made from bin ranges are stored as wavelength ranges

Internally, the selection is represented by the corresponding range of wavelengths, not the bin numbers. When a model references the selection, the wavelength range is converted back to bin numbers when appropriate. If a second file uses a different binning scheme, the wavelength range is mapped to the nearest matching bin range for that file.

From a Python program, selection.start and selection.end return the pure wavelengths. selection.fileRange("filename").start/.end return bin ranges for specific files.

Selection sets are used to refer to several discontiguous selections more conveniently. By default, all new selections in a graph zone are part of the same set. (Selections from other graph zones may not be included in a selection set.)

Selections are automatically associated with observed spectra that are shown in a zone in a graph. There are several cases:

1. There is no observed spectrum in a zone. In this case, the selection represents a range of wavelengths, but it cannot be used as an input to a model, which requires an associated file. It will be marked as an error in a model reference. However, such a selection may still be used as an input to a Python program that performs an operation on a range.
2. One observed spectrum is currently displayed in the graph zone. In this case, the selection may be used in place of a file as an argument to a model. That is, if selection s2 is placed in a zone containing file f3, instead of "gaussian(f3)," meaning "gaussian applied to entire range of spectrum f2," the user can say "gaussian(s2)," meaning "fit a Gaussian to range s2 of file f3."
3. More than one observed spectra are displayed in a zone. For example, if s2 is placed in a zone displaying f3 and f4, then "gaussian(s2)" means "apply Gaussian fits to selection s2 for files f3 and f4."
4. Similarly, a selection set may be used in most places where a selection may be used. "gaussian(ss1)" means "apply Gaussian fits to each of the selections in ss1, for all the files in the zone.

If the user adds or removes observed spectra from a zone, the selection applies to the current set of files. Additional cases are covered in the section on the model macro language.

10.1.6.1 Selection Types

A selection has an associated selection type, defaulting to Unknown. The type serves as a hint to some fitting algorithms, and may be ignored by others. It specifies the item(s) of interest within the selection. Selection types may be:

Absorption line (single)	A single absorption line
Absorption line (doublet)	A pair of nearby absorption lines
Absorption line (multiple)	Fits all prominent troughs within the selection (a fit parameter determines the threshold of "prominence")
Continuum	The selection consists primarily of continuum radiation
Emission line (single)	A single emission line
Emission line (doublet)	A pair of nearby emission lines
Emission line (multiple)	All the prominent peaks within the selection (a fit parameter determines the threshold of "prominence")
Ignore	Ignore this region
Manual	Tells the line fitter to find a centroid, but assigns a label of "Unknown." The user then assigns a label manually.
Unknown	Try to fit prominent peaks and troughs in this range

Table 10-3. Selection types

Selection types may be assigned to selections or selection sets. If assigned to a selection set, the type of each selection in the set changes. After assigning the set, the user may override individual selection types; the selection set's type changes to "<Mixed>" if any selection types do not match.

Selections and selection types make it easier to decompose count variations into progressively finer peaks and troughs. A strong peak that partially overlaps a modest trough might first be identified by selecting the peak and marking the selection as a single emission line. Noticing the apparent trough, a user might create a second, narrower, overlapping selection that encompasses the trough, marking it as a single absorption line. The line fitter would then attempt to label both.

Selection types do not necessarily determine the match that will be applied. Even if a peak is labeled as a single emission line, it will be assigned two labels if the probability is high that it consists of two closely spaced strong peaks.

10.1.7 GLSpectrumLabels

The label region displays emission line labels and selections. The AtomDB label overlay may be enabled to automatically label lines. A settable clutter factor displays labels only for the strongest emission lines. Alternatively, selections may be enabled, showing labels for user-defined selections or selections whose position and range is set by a program. Or both may be enabled.

The height of the label region is user-adjustable. As it increases in size, user-created labels have more space. Emission line labels are labeled minimally with element/ion, then with wavelength, then with emissivity.

For graphs shown with the wavelength axis oriented horizontally, labels are shown at a fixed angle, to increase readability. Vertically oriented graphs show horizontal labels.

10.1.8 XAxisNumbers/YAxisNumbers

These regions display the numbers corresponding to tick marks. Labels may be oriented at any angle between 0° (horizontal) and 90° (vertical).

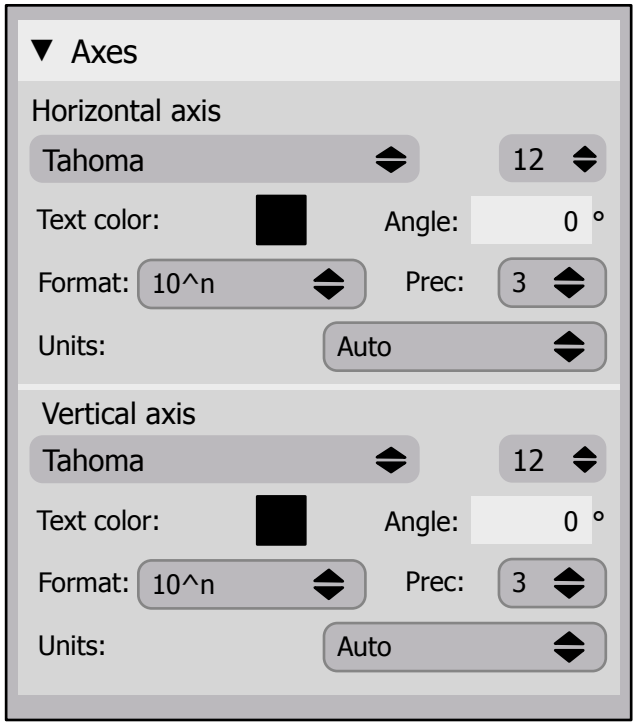


Figure 10-25. CPAxes

10.1.9 XAxisUnits/YAxisUnits

The current units are displayed in these regions. The units region may be clicked to select different units.

Axis units are part of the CPAxes collapsible pane.

10.1.10 Export

A CPGraphExport controls export of the graph.

Figure 10-26. CPGraphExport

Graphs marked for export are exported when the user chooses “Export all” from the menu. “Export all” is equivalent to “Refresh,” except that it also exports any graphs and text blocks marked for export.

11 Matplotlib Tool

The matplotlib tool is a window used to display plots generated using the Python matplotlib package. In the notebook, the plot is displayed, scaled to the size of the pane.

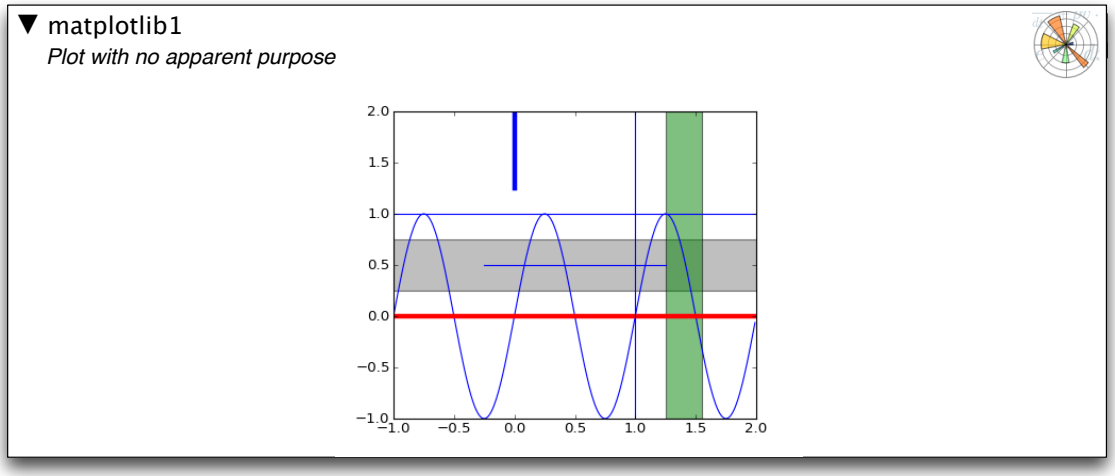
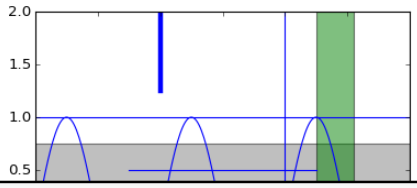


Table 11-1. Matplotlib tool in notebook

When opened for editing, the plot and the Python source code that generates it are shown. This tool does not sidestep the need to write a program to display custom results. It is intended as a bypass for plotting features that are not otherwise available within ATSAL.

[THK: Even in the first release, we may offer some matplotlib templates to graph common results from XPSEC that aren't handled by the other tools. The templates will include source code to display the basic graph, which can be modified by users to fine tune the graph.]

matplotlib1




```
t = np.arange(-1,2, .01)
s = np.sin(2*np.pi*t)

plt.plot(t,s)
# draw a thick red hline at y=0 that spans the xrange
l = plt.axhline(linewidth=4, color='r')

# draw a default hline at y=1 that spans the xrange
l = plt.axhline(y=1)

# draw a default vline at x=1 that spans the yrange
l = plt.axvline(x=1)

# draw a thick blue vline at x=0 that spans the upper quadrant of
```

Table 11-2. Matplotlib editor window

12 Model Parameter Blocks

The parameter sets that drive the computation of a model may be accessed directly from a plot's layer manager, so they remain hidden from view until needed. If desired, model parameter blocks may be inserted into the notebook for easier viewing. The user inserts a model parameter block anywhere in the notebook, then associates it with a previously defined model, or defines a new model for it.

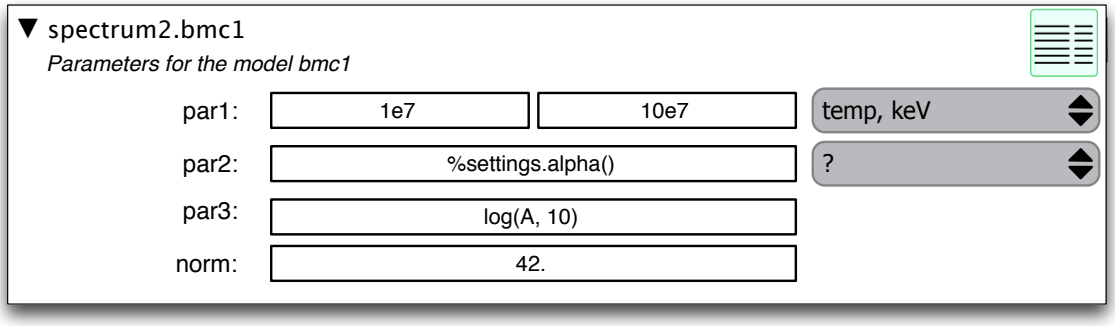


Figure 12-1. A parameter block

The name of the parameter block is that of the spectrum plot and model instance with which it is associated.

Double clicking brings up the model parameter editor. Fields are described elsewhere.

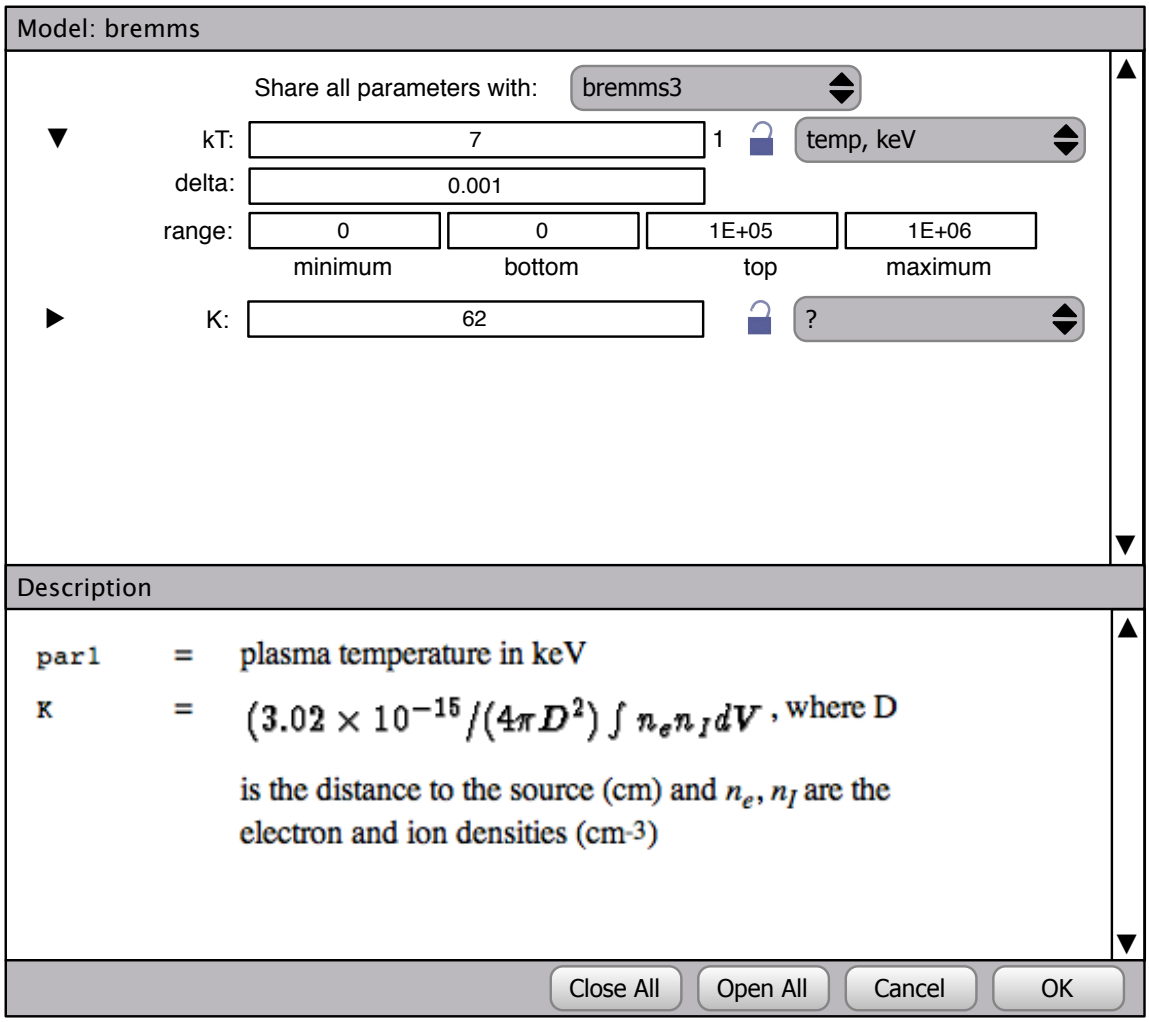


Figure 12-2. Parameter block editor

The parameter block selector allows switching to parameters for other spectra and models.

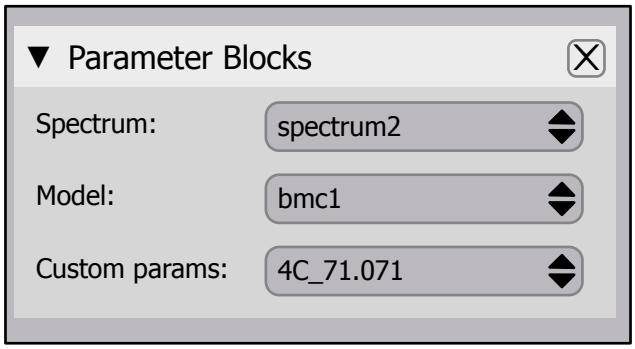


Figure 12-3. Parameter block selection

In release 2, this mechanism will support editing and creation of custom parameter blocks. The Custom params pop-up will list custom parameters, and includes an entry for New parameters.

If a custom parameter block is selected, its values may be edited as for standard parameter blocks. The user can also open the Custom Parameter Block pane to build or edit these blocks. "Enable editing" is off by default to guard against accidental parameter block edits.

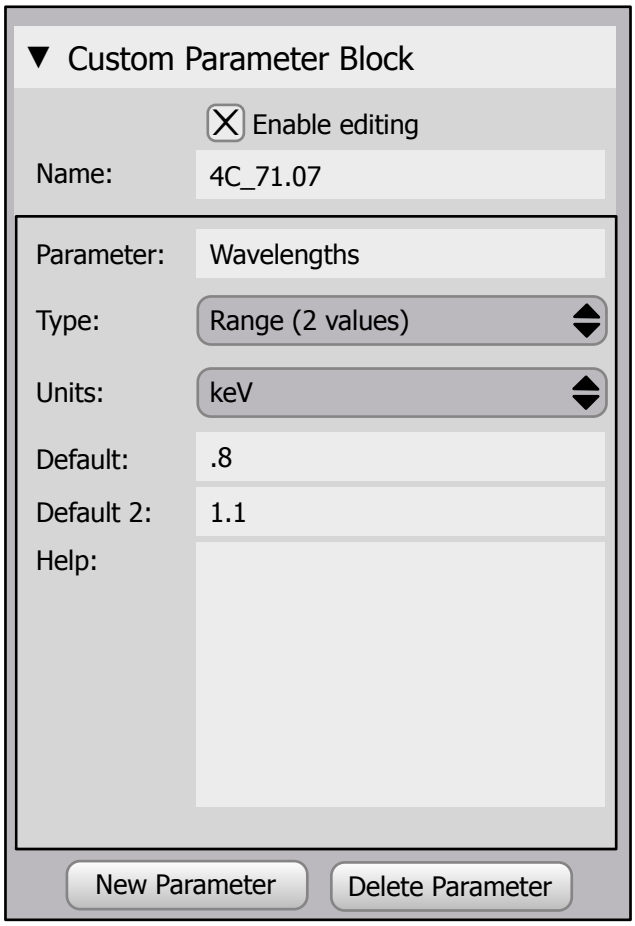


Figure 12-4. Custom parameter block pane②

This names the parameter block, and edits the currently selected parameter in the block. Choosing "New Parameter" inserts a new parameter at the bottom of the existing parameters, or a user selects an existing parameter to edit it. New parameters are added at the bottom, but may be rearranged by dragging.

Standard parameter blocks are accessed in Python via:

spectrumName.modelName.parameters(paramNumber, subNumber)

Custom parameter blocks are not associated with a particular spectrum tool, however, so they are accessed as:

customParameters("name", paramNumber, subNumber)

13 Fit Results Tool

A fit results tool is a special-purpose table tool that presents parameters calculated as part of a fit. Like model parameters, fit results may be displayed from a spectrum plot in which they appear, by clicking "Results..." Or users can insert a tabular display of the results in the notebook for viewing.

▼ spectrum2.model5

*Results for the model phabs[1] * powerlaw[2]*

H_0

	A	B	C	D	E
1	Chi-squared	Lvl	Param 1	Param 2	Param 3
2	204.136	-3	7.99E-02	1.564	4.4539E-03
3	84.5658	-4	0.3331	2.234	1.0977E-02
4	30.2511	-5	0.4422	2.174	1.1965E-02
5	30.1202	-6	0.4648	2.196	1.2264E-02
6	30.1189	-7	0.4624	2.195	1.2244E-02
7					
8	Variances and principle axes				
9	Variance	Axis 1	Axis 2	Axis 3	
10	4.14E-08	0	-0.01	1	
11	8.70E-02	-0.91	-0.41	-0.01	
12	2.32E-03	-0.41	0.91	0.01	

Figure 13-1. A fit summary in the notebook

As with other notebook tool panes, the height may be adjusted to show up to a full page of fit result data. Column headings, shown with gray backgrounds, may be used to address specific elements in a fit result. For examples:

spectrum2.model5.results.cell(11, "Axis 1") => -0.91

spectrum2.model5.results.cell("+2, Param 2") => 2.234

The second syntax specifies the row number as an offset from the column label, which makes the reference somewhat more immune to changes in the displayed data as a result of changed input parameters.

The fit results "editor"—a misnomer, since results blocks are read-only—appears like this:

spectrum2.model5

	A	B	C	D	E
1	Chi-squared	Lvl	Param 1	Param 2	Param 3
2	204.136	-3	7.99E-02	1.564	4.4539E-03
3	84.5658	-4	0.3331	2.234	1.0977E-02
4	30.2511	-5	0.4422	2.174	1.1965E-02
5	30.1202	-6	0.4648	2.196	1.2264E-02
6	30.1189	-7	0.4624	2.195	1.2244E-02
7					
8	Variances and principle axes				
9	Variance	Axis 1	Axis 2	Axis 3	
10	4.14E-08	0	-0.01	1	
11	8.70E-02	-0.91	-0.41	-0.01	
12	2.32E-03	-0.41	0.91	0.01	
13					
14	Model: phabs[1] * powerlaw[2]				
15	Model par	Fit par	Model comp	Component	Parameter
16	1	1	1	phabs	nH
17	2	2	2	powerlaw	PhoIndex
18	3	3	2	powerlaw	norm
19					
20	Chi-squared	Bins (PHA)	Reduced chi-sc	Null hypothesis	
21	30.1189	31	1.075675	0.358	
22					

Figure 13-2. Fits results "editor"

The Fit results pane selects results for a different spectrum plot and/or model.

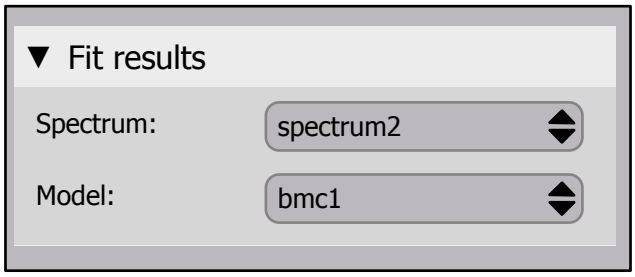


Figure 13-3. Fit results pane

14 Python Tool

When a notebook is refreshed, ATSAL requests each tool in succession to refresh its state. For example, consider a notebook with a spectrum to which a linefitter has been applied, as shown below. Two other tools, scrolled out of view prior to this one, are a spectrum files tool to select the input file and a text block that annotates the analysis.

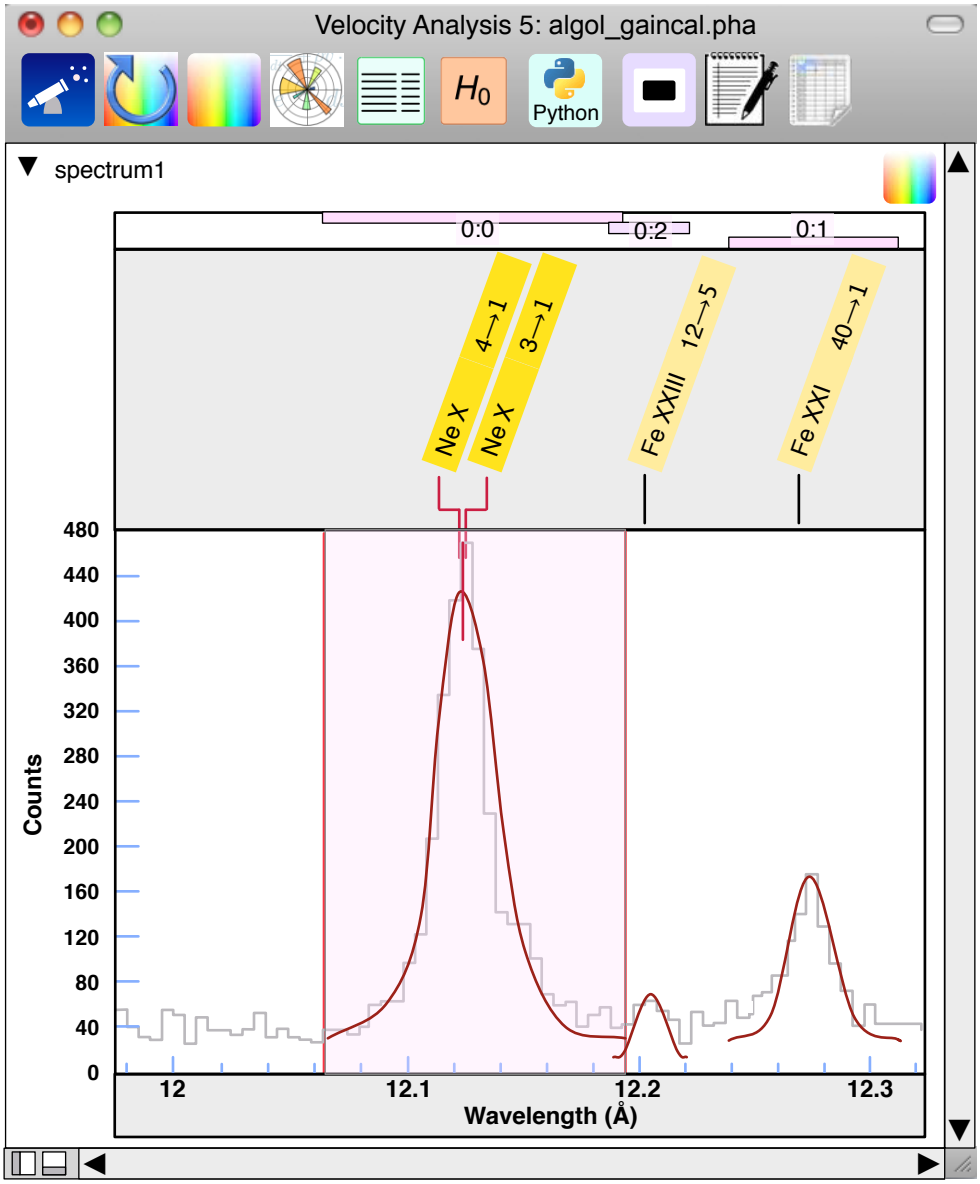


Figure 14-1. Adding a linefitter that will be accessed from Python

Here, the user added three selections and used a linefitter model to fit and label three peaks. To refresh this display, ATSAL creates and executes a small Python main program:

```
spectrumFiles1.refresh()
text1.refresh()
spectrum1.refresh()
```

Each tool handles the refresh request by performing any updates needed to bring the display windows in line with changes in the user interface. For example, spectrum1.refresh() calculates fits for the selections and assigns labels if the calculation has not been done previously.

Now suppose the user inserts a Python tool, here shown collapsed, and a second text tool in which to display some summary information:

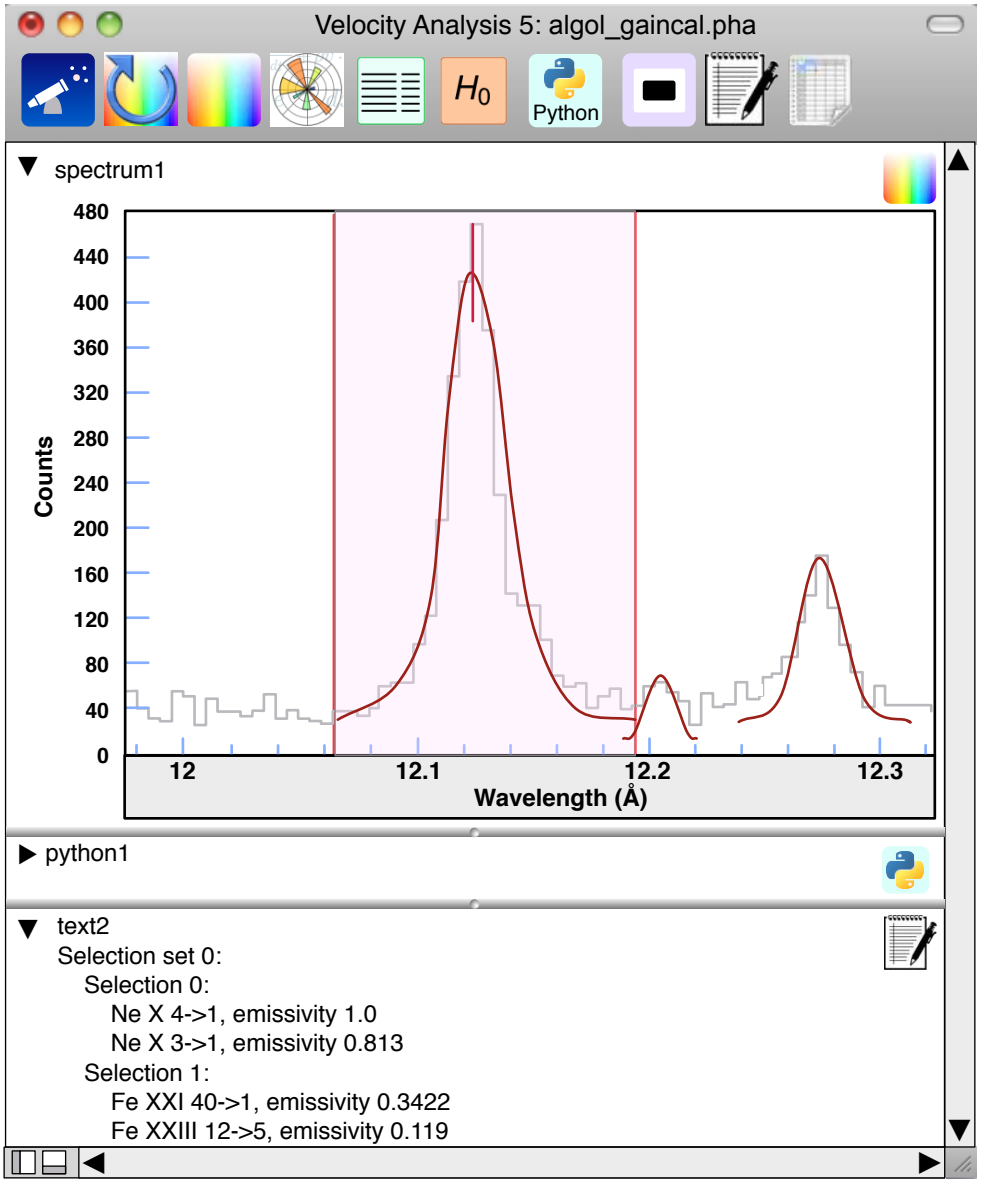


Figure 14-2. Adding a Python tool and a text tool

The second text tool simply displays a list of relative emissivities of the lines that were matched by the linefitter. The Python tool prepares this list, with a program fragment something like this:

```
result = ‘’
for (ss in xrange(spectrum1.selectionSets.count()):
    selectionSet = spectrum1.selectionsSets[ss]
    result += ‘Selection set ‘ + `ss` + ‘:\n’
    for (s in xrange(selectionSet.selections().count()):
        selection = selectionSet[s]
        result += ’    Selection ’ + `s` + ‘:\n’
        for (l in xrange(selection.lineInfos().count()):
            lineInfo = selection.lineInfos()[l];
            result += (‘        ‘ + lineInfo.symbol() + ‘ ‘ +
                lineInfo.ion() + ‘ ‘ + lineInfo.lower() + ‘->’ +
                lineInfo.upper() + ‘, emissivity ‘ +
                lineInfo.emissionLine.relativeEmissivity() + ‘\n’)
```

When this fragment of code is executed, the string “result” contains the lines that we wish to display. In the text tool, this string is displayed by entering the following:

```
%result
```

Internally, ATSAL executes this code fragment when its refresh() function is called. When the user clicks Refresh, it is equivalent to:

```
spectrumFiles1.refresh()
text1.refresh()
spectrum1.refresh()
pythonTool1.refresh()
text2.refresh()
```

The user-written code fragment has access to the full Python environment, except for constraints on importing additional Python components. To import packages that are not part of the default environment, the user calls functions defined in separate Python modules.

14.1 Python Dos and Don’ts

Python code inserted via the Python tool is executed every time the user hits Refresh. It is executed in the same sequence as the tools appear in the notebook. This has several implications:

- Tools prior to the Python tool will be fully up-to-date, having executed their own refresh commands.
- Accessing tools after a Python tool may return obsolete information.
- Method calls that have significant side effects may lead to unwanted results.
- Long-running Python programs (e.g. custom models) will be re-executed on each refresh, resulting in slow performance. This may be mitigated by storing earlier results and recomputing only if inputs have changed, or by subclassing ToolPython as described in the following section.

14.2 Subclassing ToolPython

ToolPython is the default Python tool class. Its refresh method executes the code that is typed into the tool. This provides a mechanism for gluing outputs from previous tools to inputs to new tools, perhaps with some data reduction.

Advanced users may need to subclass ToolPython to achieve more control over the execution environment, for example, by creating objects at initialization time that will remain present throughout an ATSAL session, or by presenting content other than source code in the notebook window, or by performing more complex analysis operations.

The procedure for subclassing ToolPython is shown below.

1. Display the file list pane in the notebook window if it is not visible.
2. Create a new Python file.
3. Create the subclass as shown. All such subclasses are automatically registered with ATSAL as new Python tool types. (TBD)
4. Override refresh() if desired to implement more sophisticated behavior upon a refresh. If an override is implemented, any code typed into the tool will be executed only if you call the inherited refresh().
5. Override drawNotebookPane() to replace the notebook pane's source code with contents of your own design.
6. Later versions of ATSAL will use this mechanism to extend the user interface by creating custom tools.

To use a subclass, the user selects it by name from a popup menu when deploying the Python tool.

Note: changes to source code in the default Python tool are incorporated immediately upon the next Refresh performed by the user. But changes to ToolPython subclasses themselves are not incorporated except when the notebook is opened or restarted using Restart Notebook. Since these steps must re-execute all XSPEC commands from scratch, it is easiest to debug new ToolPython subclasses before creating time-consuming models.

14.3 Python File Editor

The Python main program is the outer glue that ties together the operation of the tools in the notebook. It appears as the first step in the notebook. Opening it expands it to fill the notebook pane. Double click the icon to open it as a separate window. If you add more Python files later, they are not added to the notebook. Instead, they appear in the files list, an optional pane to the left of the notebook pane.

Features TBD.

Design Note

It looks like a Qt widget based on the Scintilla text editor and already bundled with PySide will provide advanced integrated text editor support for Python. If we get this for free, and it is robust, we will use it. If not, we will specify a bare bones text editor based on an existing Qt widget, and let more advanced users resort to their favorite external text editor. See this article on Scintilla limitations: http://www.scintilla.org/SWFlexibility.html.

Another alternative is JuffEd, but this is derived from Scintilla. http://qt-apps.org/content/show.php/JuffEd?content=59940

15 Text Tool

The text tool is a very simple word processor window. It supports a limited rich text environment, with a limited selection of fonts, control over basic text formatting, and support for pasted images. It does not directly support equations, many word processing functions, or tables. Unicode is supported.

Locating and entering desired Unicode characters relies on input entry and Unicode character selection tools that are supported by the OS, with the exception of common Greek and math symbols.

Text blocks created using this tool are stored internally as a subset of HTML 4, and may be exported in this format.

Design Note

HTML 5 is preferred, but the current version of the Qt class on which text tools are based supports HTML 4. We hope to switch to HTML 5 prior to the first release.

http://qt-project.org/doc/qt-4.8/demos-textedit.html

15.1 Displaying Object Values

Variable names are embedded in text blocks with:

```
%variableName
```

Examples:

```
%notebook.runDate()
%preferences.abundancesTable.name()
%notebook.author()
```

Example:

%notebook.name()

%notebook.author()

Analysis date:	%notebook.runDate() %notebook.runTime()
Input file(s):	%notebook.spectrumFiles()

In view final mode, or in the tool pane, this appears as:

Binary Star 2

George E. Hale

Analysis date:	2013-06-17 22:14:12
Input file(s):	M42_xxx.pha

Figure 15-1. Text block variables

Note that the HTML formatting is never shown explicitly, even when variable names are displayed. To view the text in its HTML markup form, copy it to the clipboard and paste it into a text editor.

15.2 Text Block Editor

The text block editor looks somewhat like this, but:

- the user can switch between variable entry mode, which shows variable names in square brackets, or display mode, which shows substituted values for variables
- there are icons for bold, italic, subscript, superscript, alignment, import, export, symbols, and execution settings
- there are a set of styles
- fonts are restricted to serif, sans serif, and typewriter, consistent with platform-independent HTML restrictions
- font sizes are available

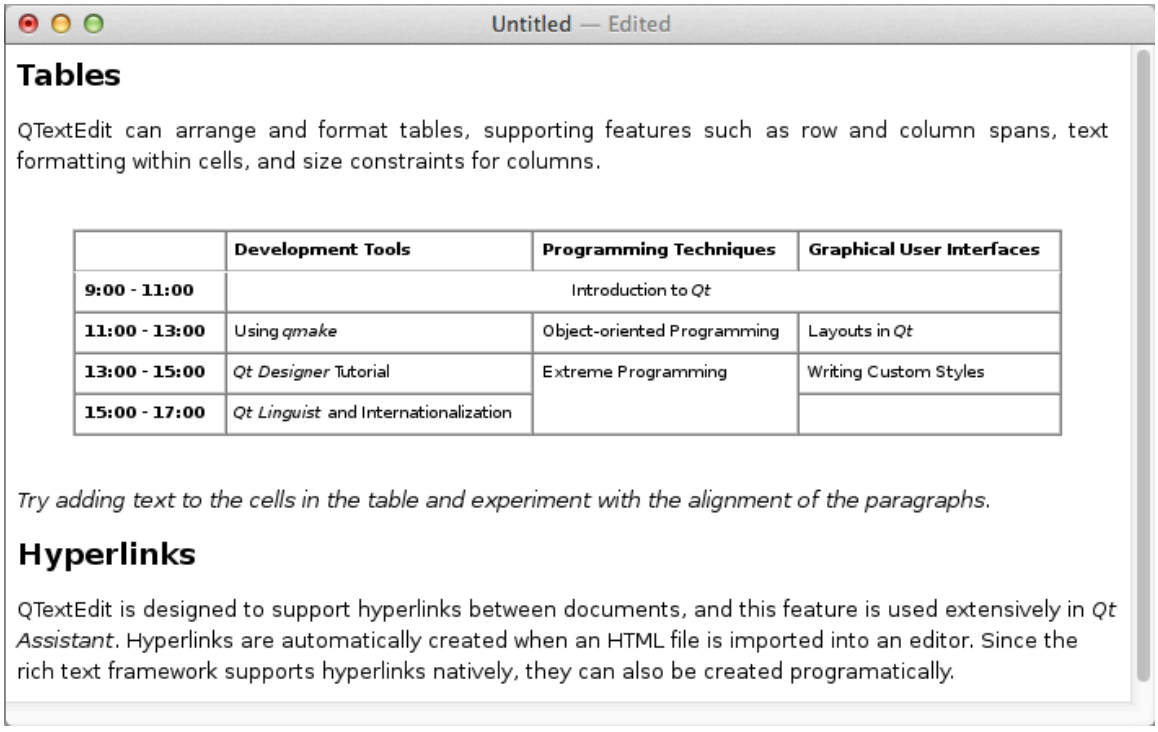


Figure 15-2. Text block tool editor. This image is from the Qt documentation for QTextEdit

15.2.1 Text Block Name

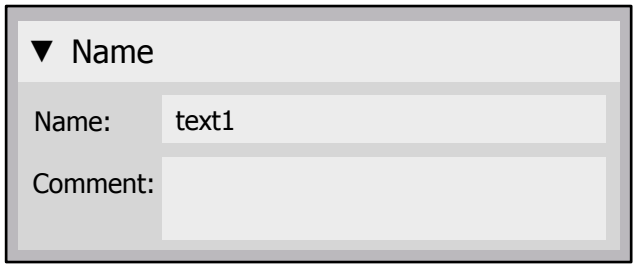


Figure 15-3. CPTextName

15.2.2 Import/Export Commands

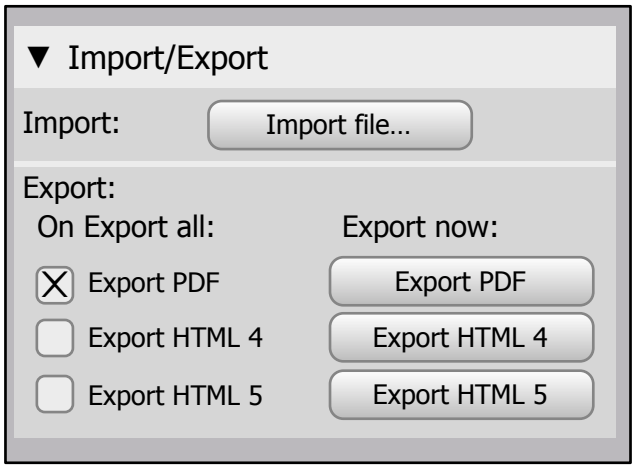


Figure 15-4. Text tool import/export

If any of the export checkboxes along the left are checked, the Export all menu command exports the text block in the selected formats. The files are named *toolname.ext*, for example, text1.html. HTML 4.0 files use the .htm extension, while HTML 5.0 files use .html.

15.2.3 Clipboard

Text cut or copied is converted to HTML 4.0 format, and may be pasted into a text editor as HTML markup or into another text block. This means it is also legal to paste in HTML from another source, provided the HTML conforms to the published subset supported by this application.

Standard menu commands are used for clipboard operations.

15.2.4 Symbols

Symbols displays a commonly-used subset of Greek and math symbols for insertion into the text block.②

15.2.5 Basic Editing

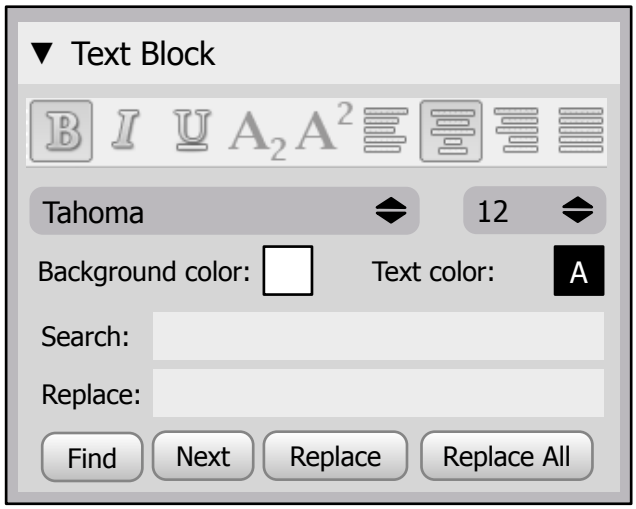


Figure 15-5. Text block settings and search/replace

15.2.6 Tables in Text Blocks

A table created by a table tool may be embedded in a text block. This is done by dragging the table tool icon into the text block, or using the import command.

This is implemented by converting the table to equivalent HTML and displaying it in the text block. Because of this, the table in the text block is display-only, not interactive.

When such a table is embedded, the formatting applied in the table editor is preserved in the text block, as closely as possible. Embedded tables are limited to about a page in size. Larger tables are legal, but will be displayed to indicate that they are incompletely shown.

Internally, the markup view is implemented using a Python file editor pane, while the formatted text uses a QTextDocument.

[THK: Maybe the built-in table support can be used for this, and accomplish I/O in the text block...]

15.3 Text Styles

A text styles implementation akin to that employed by Microsoft Word and many other word processors achieves several goals:

- It helps to handle the problem of producing similar-appearing results on operating systems with different fonts.
- It assists in the production of publication-ready results.
- It simplifies production of uniformly formatted results with material drawn from multiple sources.

A *paragraph style* is a named collection of paragraph-level attributes, such as margins, all or some of which are copied to a selection of text. Paragraph styles

effect all the paragraphs in a selection, even if some paragraphs are partially selected. A *character style* is a named collection of attributes, such as boldface or subscripting, that apply to a selected range of characters. *Direct styles* are attributes of either type that are applied directly to the text, rather than by using styles. Direct styles may be thought of as overrides to the defaults applied by paragraph and character styles.

QTextEdit, the Qt class upon which word processing and text block support is currently based, supports direct styles, but does not support paragraph or character styles. This section describes some design issues.

Paragraph and character styles have a parent style, and a mask that specifies which attributes are applied. For example, if the italic mask bit is set, the effected body text's italic state is set to match the style. If the bit is cleared, the attribute is inherited from any parent styles. If no parent style defines an override for the property, the text is left unchanged.

Both character and paragraph styles have an immutable, hidden root style, which defines some reasonable result for any platform. The root styles choose fonts and sizes for each platform that render fairly uniform results. In the case of character styles, some traits are undefined even by the root style, inheriting instead from the surrounding paragraph style.

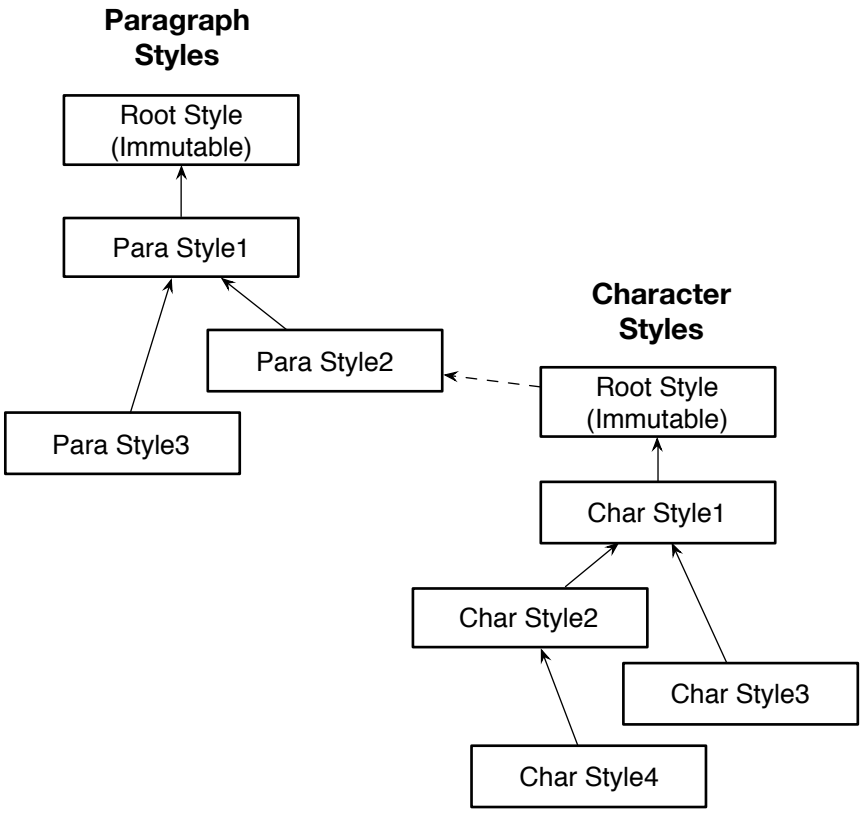


Figure 15-6. Text style inheritance

At least in the first release, the list of character and paragraph styles is predefined: styles can neither be created nor destroyed, only modified. This avoids a host of boundary cases in style handling.

15.3.1 Style Masks

A style mask determines which styles are applied vs. inherited. For example, if the text color attribute is to be applied, text color is set to match the style-defined color. If not, text color is inherited from parent styles. If no child styles apply text color, the attribute is inherited from the root paragraph style, which cannot be changed or even viewed by the user.

Bits in the style mask are not manipulated directly by the user. Instead, they are set as a side effect of the user's setting of the attribute itself. For example, if the user clicks the bold icon, this also sets the bold bit in the style mask. If they click the bold icon again to shut off the bold attribute, this style's attribute matches that of the parent style, so the mask bit is turned off. Thus the mask bit is set when the child style diverges from the parent and cleared when it matches. Turning off bold means "make the boldness of this text match the surrounding text," not "don't embolden this text even if the surrounding text is bold."

Internally, each style has two sets of properties and two masks, local and composite. The local settings are those manipulated by the user when modifying styles. The composite settings are those derived from the local and inherited settings, and are recomputed every time any style setting is altered—a cache. The local settings are presented to the user when modifying style properties, while the composite settings are those actually applied to the text.

15.3.2 Style Overrides

In addition to applying styles, users can directly set paragraph or character attributes, for example, by superscripting a character. In order to model this, a second pair of paragraph/character masks are employed. These are called override masks, and they are not stored in the styles, but in the block of user data associated with each text block. If, for example, the superscript trait is applied, the character superscript override bit is set, which blocks application of that trait to the text. The superscript setting itself is applied to the text block, so it is not stored in any character style.

Overrides are also used when the user pastes styled text that has no associated styles. The pasted text is assigned the same paragraph and character styles as the preceding block, but with all its override bits set. This preserves the original formatting of the pasted text. When the user explicitly sets the paragraph or character style, the override masks are disabled and the style once again dictates appearance. If the user chooses to paste as unformatted text, the style mask is cleared, allowing the text to inherit from the paragraph and character styles.

When overrides are in effect, the user interface indicates this by placing a plus sign next to the style name popup menu.

15.3.3 Style Precedence Rules

A default set of styles are stored with user preferences. Each document has a complete copy of this list. A new document starts with the defaults. When the user edits any existing document's style, the edit implicitly updates their defaults as well.

Existing document styles are always preserved.

When text is pasted...

- ...and it is pasted as unformatted, and it lacks style information, it takes on the styles currently in effect at the insertion point.
- ...and it is pasted as formatted, and lacks style information, its formatting is preserved and its style override masks are set.
- ...and it is pasted as formatted, and has style information, it inherits the receiving document's style rather than preserving the original styles. If, for example, the "Header-1" style is defined as Arial 18 pt in the receiving document and Rockwell 16 in the sending document, the text takes on Arial 18 formatting.

When text is cut or copied, the clipboard copy includes the user data that defines its styles. (Or if it doesn't, this will need to be added.)

15.3.4 User Interface

Styles are applied from a single popup menu that lists all character and paragraph styles merged into alphabetical order. When the user clicks in a document, the popup shows the current character or paragraph style, with the character style taking precedence. If the user clicks on an area with a blocking style, "<None>" appears. If the selection encompasses multiple styles, the menu is blanked. An icon beside each style indicates whether it is a paragraph style or a character style.

A "Set" button beside the style popup displays the style settings for editing.

15.3.5 The WordProcessor Subclass

WordProcessor is a subclass of QTextEdit that adds style support, text rulers, and Python variable substitution.

Style Support Discontinued for Now

QTextEdit is a layer atop QTextDocument. A QTextDocument is composed of QTextBlocks, usually one per paragraph; and QTextBlocks contain QTextFragments, one per block with uniform formatting. For the style system as currently designed to work, it is necessary to tag each block and fragment with user data. Blocks may be tagged with such data, but fragments may not. This complicates the implementation of styles considerably. Probably the easiest solution is to extend Qt to add a user data pointer for QTextFragments as well, but this is deferred at the time of this writing (Oct 2014), as it would mark the first time we diverged from using Qt unmodified.

A QTextEdit is a front end for editing QTextDocuments. QTextDocuments are composed of QTextBlocks, and each QTextBlock can point to user data. This is the mechanism used to associate styles with documents.

A TextBlockStyleInfo contains a pointer to the style associated with each text block. The WordProcessor subclasses manages these pointers through various editing operations.

When a document is saved, it is transformed to an HTML representation that lacks the style information. The style information is stored separately and re-applied when the document is re-opened. When styles are edited, their changes are propagated to all open WordProcessors in each notebook.

Saved style information is stored as a set of ordered tuples (starting QTextBlock number, para style name, char style name, para override, char override), with an entry each time at least one parameter changes state.

Master styles are saved with preferences, and notebook-specific styles are stored with each notebook.

Documents exported as HTML lack styles, although they preserve the most recent settings applied from styles.

When a different style is applied, all text blocks within the selection are tagged with the new style. When a text property is changed directly, rather than with a style, a blocking mask is applied.

16 Table Tool

The table tool is a 2D array of values. Some analysis steps produce tables as output, or you can produce such tables under program control. You can also input values in tables and use them as inputs to other tools.

A table may have any number of rows and columns, but the notebook pane may only be able to display a subset at a time to remain within a page in size.

	0	1	2	3	4	5
0		Ion	Transition	λ	Emissivity	
1		O VII	4->1	22.0977	-15.76	
2		O VII	2->1	21.6015	-15.24	
3		Fe XVII	2->1	17.096	-14.84	
4		Fe XVII	3->1	17.051	-14.91	
5						
6						
7						

Figure 16-1. Table tool editor

Tables resemble spreadsheets, but they do not have any interactive functions. They are simply arrays. Except for the absence of functions, tables mimic spreadsheet user interfaces.

> **Design Issue**
>
> Row 0 and column 0 are always reserved for label areas, although their display may be suppressed. This has the potentially confusing side effect that the data content of the table is effectively addressed in 1-based notation, not 0-based notation. We considered using –1 to address this row and column, but this is also confusing. Suggestions welcome.

A new table has a default set of six columns and eight rows. The leftmost column and topmost row serve as labels by default. Label regions are indicated with a different background color, as shown above. This row or column may be hidden if desired. In the example above, columns are labeled but rows are not.

A given location in the table may be accessed via row and column. You can also specify the row or column label instead. To access the row or column label itself, specify nil for the other dimension. For example, the numeric value 21.6015 in the table above may be referenced as:

```
table[2][3]   # 0-based row-major order for Python
table[3]["λ"] # 0-based (or "label-based") row-major order
```

[THK: the syntax for label-based cell access will probably have to be different from that shown here, such as table.cell(3, "λ").]

To access the cell containing the word "Emissivity":

```
table[0][4]
```

```
table[0]["Emissivity"]
```

It is illegal for two rows or two columns to have the same label. Exception: no label at all is legal, but items may not be indexed using a blank label.

Design Note

We considered adding spreadsheet cell address notation, such as table("D3") as well, but because spreadsheets are accessed in column-major (vs. row-major) order, and with one-based (or A-based) vs. zero-based indices, we decided against it.

16.1 Manipulating the Table

Selecting a row or column header selects a row or column.

Insert before/after row/column performs the specified operation relative to the selected row/column/cell. If multiple items are selected, the new row/column appears before the first or after the last.

Delete deletes a selected row or column(s). If a cell is selected, delete empties the cell and prepares for input of new data. If multiple cells are selected, a dialog appears to confirm before emptying all the cells.

Selecting a cell for editing displays an edit area at the top of the table window. Any markup used for the cell is displayed during editing of the cell.

Copying a table to the clipboard also copies cell properties. Paste pastes the data and properties. Paste and match style pastes only the data.

16.2 Table Properties

These properties apply to the table as a whole. Some may be overridden on a per-cell basis.

Show/hide row numbers	Controls display of row numbers in the tool pane. (Row and column numbers are always shown in the table editor.)
Show/hide column numbers	Controls display of column numbers in the tool pane.
Show/hide row labels	Controls labels in column –1.
Show/hide column labels	Controls labels in row –1.
Row width	Set by clicking on the line between columns. If several columns are selected, and one line is dragged, all the columns change in size. (Row height is set automatically to match the tallest cell in a row.)
Row padding	Sets extra width, in points, above and below each row.
Column padding	Sets extra height, in points, above and below each row.
Outer box thickness	Controls the thickness of the border enclosing the entire table. Set to 0 to disable.
Row header line thickness	Controls the line between the header row and the body of the table.
Column header line thick-	Same for the column header.

ness	
Body row line thickness	Determines the line between rows.
Column line thickness	Line between columns.
Text color	Default for table.
Background color	Sets background color for table.
Body alternating color	Sets background color for alternating rows of the table.
Font	Default font for entire table.
Font size	Default font size for entire table.

Table 16-1. Table properties

Background color may be set for the table as a whole, and one or two colors specified for row backgrounds. If two different colors are specified, the display alternates between the two colors.

The default text color, alignment, font, font style, and font size may be set for the table as a whole, or for a selected row or column range, or for individual cells. More locally scoped attribute changes override those more global in scope.

By default, if a cell value is too wide to fit, it is truncated and followed by an ellipsis ("...") to indicate the truncation. Adjacent cells may be merged by selecting them and choosing Merge/Separate cells. Any Python cell address within a merged cell selects the entire merged cell. (Merged cells may not be present in release 1.)

Wrapped text enables text wraparound, expanding the height of the row to accommodate the wrapped text.

16.3 Cell Properties

Cells can display strings, integers, floats, or dates.

Text color	Default for table.
Background color	Sets background color for table.
Alignment	Left, center, right, auto. Auto is right-justified for numeric values, otherwise left-justified.
Font	
Font size	
Font style	
Floating point format	Specifies via separate dialog a template for displaying numeric values. Default is %.3g.
Integer format	Specifies via separate dialog a template for displaying integer values. Default is %d.
Date format	Specifies via separate dialog a template for displaying date values. Default is YYYY-MM-DD g24:mm:ss.

Table 16-2. Table cell properties

[THK: Decimal point alignment probably isn't available in the HTML representation, so it is not included here. It is present in the spec, but most browsers do not support it.]

16.3.1 Floating Point Format

If a cell value is supplied in floating point format, it may be displayed with the following options:

Precision	Positive integer specifying number of decimal positions after decimal point.
Force/auto/avoid showing exponent	Corresponds to e, g, printf format options.
Exponent display as e/E/superscript	Examples: 6.5e18, 6.5E18 or 6.5×10^{18}
Use printf template	User can enter a printf-style format string

Table 16-3. Floating point decimal properties

16.3.2 Integer Format

Decimal/hexadecimal	
Use printf template	

Table 16-4. Integer properties

16.3.3 Date/Time Format

The date format dialog lists some common formats and allows users to construct their own.

YYYY-MM-DD g24:mm:ss	
YYYY-MM-DD	
g24:mm:ss	
(various others)	
Use date template format	Format described below.

Table 16-5. Date/time formats

The following options may be used to construct a custom date/time template.

YYYY	4-character year
YY	2-character year
MMM	Month, e.g. OCT
Mmm	Month, e.g. Oct
Month	Month, e.g. October
MM	Month, e.g. 10
DDD	Day, e.g. TUE
Ddd	Day, e.g. Tue
Day	Day, e.g. Tuesday
DD	Day of month, e.g. 16
gg	Hours part of time, in GMT 24-hour format
hh	Hours part of time, in local time zone 12-hour format
h24	Hours part of time, in local time zone 24-hour format
mm	Minutes part of time

ss	Seconds part of time
.uuuuuu	Fractional part of time, number of u's determine precision displayed
AM	Adds AM or PM
am	Adds am or pm
z	Current time zone as positive or negative offset in hours from GMT
T	If present between date and time, reproduced as is.

Table 16-6. Date/time format details

17 Help System

TBD.

18 Custom User Interfaces in Notebooks

Since ATSAL is based on Python, and ATSAL's flavor of Python is based on Qt, most of Qt's substantial repertoire of user interface construction tools are available to Python programs in ATSAL notebooks. That is, if the user is a Python/Qt programmer. This is a nontrivial learning curve, though, one that most users won't want to bother with.

[THK: We will need some rudimentary user interface construction tools.]

19 Style Tool

ATSAL offers detailed control over plot styles as an aid to preparing plots for publication. To this end, it provides a hierarchical style mechanism similar to that provided by CSS. A Style is a collection of style settings, each of which may be supplied explicitly or inherited from a parent Style. A style tool configures a Style.

ATSAL offers an application style, at set of hard-coded defaults shared by all ATSAL users. Next, there is a document style, which defaults to the application level style, but may be configured with a style tool. One or more user-created styles, separately named, may be created for special needs. Each such custom style is represented by its own style tool.

Style tools are explicit elements in a notebook because this allows them to but cut, copied, pasted, exported, imported, combined, and rearranged as needed to share styles with other documents.

A Style has a name. These operations are possible:

New style	Creates a new style tool. If no document style tool exists yet, user is prompted whether to create a document style tool or a new custom style tool.
Rename	Changes the name. If the new name conflicts with another style of the same name, a warning appears, offering to combine the styles, replace the old style with the new one, or cancel.
Cut	The cut style is replaced by a hidden alias that connects this style to its parent style. All objects using this style use this alias to inherit from the parent. If a new style is later defined with the same name, it replaces the alias and the objects using it switch to the new style. This permits temporary removal of a style without losing the binding to the style name.
Delete	Same as cut, above, except the style does not go on the clipboard.
Copy	The copied style is placed on the clipboard. Any styles from which it inherits are not copied as well.
Paste	If the pasted style name conflicts with one already in the document, the user may (1) replace the old style with the new; (2) combine the old and new styles, with the pasted style taking precedence; or (3) cancel.
Duplicate	The duplicated style is named with a sequence number.
Export	Exports an XML description of the style settings.
Import	Replaces the current style settings with the imported settings.

Order is not important in style tools. A style may inherit from another style defined later in the same document.

When pasting plots into a new document, the pasted plot continues to refer to the original styles by name. If the styles are not yet defined, the pasted plot refers to the document styles. If a style with the original name is later created or pasted, the plot will pick up the new style.

Font Style Issues

ATSAL predefines pseudo-fonts “Serif,” “San-serif,” and “Mono,” mapping them to appropriate choices on each platform.

Users can define additional font aliases to help span platforms. Font aliases are defined via Preferences, not the style mechanism. A font alias is a font name followed by a list of substitute font names. The first font name actually found on a given platform is used. A font alias also has a type, which is used to determine the substitution if nothing else is available.

Fonts

Since fonts vary across platforms, a CSS-like mechanism is needed to allow for platform-independent font definitions. ATSAL defines a fixed list of font alias names that users can redefine as needed:

Serif (cannot be redefined)
Serif-1
Serif-2
Serif-3
San-serif (cannot be redefined)
Sans-serif-1
Sans-serif-2
Sans-serif-3
Monospaced (cannot be redefined)
Monospaced-1
Monospaced-2
Monospaced-3
VAxis Units-1
VAxis Units-2
VAxis Numbers-1
VAxis Numbers-2
HAxis Units-1
HAxis Units-2
HAxis Numbers-1
HAxis Numbers-3
Annotations-1
Annotations-2
Annotations-3
Ions-1
Ions-2
Ions-3
Counts-1
Counts-2

These names are predefined in order to avoid problems exchanging them between documents.

Each font alias has a parent font alias, which must be specified, and an optional list of font names. If names are supplied, they are consulted in order on a

particular platform, and the first available font used. If no font is present on the platform, the parent font alias is used instead.

When styles and plots are copied and pasted into other documents, the aliases defined in the receiving document are always used.

20 User Documentation

ATSAL documentation will be supplied only in HTML format, though the HTML files may be supplied with ATSAL for local access in addition to online access. The following documentation will be provided:

- A tutorial that shows development of a two notebooks for real world analysis.
- An ATSAL manual for users to get acquainted with tools and creating notebooks.
- An overview of using Python in ATSAL, with pointers to other reading.
- Documentation for each ATSAL-specific class and method that is surfaced to Python, created via Doxygen and PySide. ATSAL will mimic Qt's well-regarded documentation style.
- A gallery of sample notebooks that provide starting points for common analysis functions.
- Pointers to documentation on any Python packages that are bundled with ATSAL (such as, probably, matplotlib).

Doxygen is a widely used tool that permits flexible documentation to be embedded in directly within source code, where it is (one would hope) easier to keep in sync with changes to source code. At a minimum, all classes and methods that are surfaced to Python will be documented at the Python level. We should also employ this to document at least public and protected methods in C++ code as an aid to ATSAL developers.

Design Issue

We do not presently understand how C++ level documentation is mapped to the Python level. It may be necessary to document separately at each level. If this is the case, priority will go to the Python layer.

21 Writing C++ for Python

Most of ATSAL's (and XSPEC's) internals are written in C++, for several reasons:

- High performance is a central design goal of ATSAL, because highly responsive graphics allow for more productive and more informative exploration of data.
- As application complexity increases, strongly typed languages become progressively more important in managing complexity and limiting bugs.
- A C++ implementation allows full control over the surfacing of capabilities to the Python layer, ensuring better compatibility across releases.
- Python's concurrency features do not support efficient use of multiple cores for CPU-bound applications, leaving much of the capacity of today's computers unused.

21.1 Surfacing C++ Code to Python

PySide is a tool that is part of Qt. It surfaces Qt libraries to Python by examining the C++ header files in conjunction with a *type system file,* an XML description that describes how to surface classes and methods. This process is not trivial, since there are various mapping issues that must be explicitly specified to bridge the language gap. This includes data type conversions in some cases, issues of object ownership, and handling of features that are present in only one of the languages.

The use of this mechanism for Qt is an existence proof that the mapping system is flexible enough to apply to ATSAL C++ code as well, though not necessarily a proof that the resulting Python "dialect" will be easy to use.

C++ code must enable RTTI (runtime type information), which PySide tools use to perform the mapping. C++ names may not begin with "Py" or "_Py" because Python reserves these names for its own use.

http://qt-project.org/wiki/PySide_Binding_Generator

http://qt-project.org/wiki/PySide_Binding_Generation_Tutorial

[Sample file:
/Users/tkent/Tom/Projects/ATSAL/PySide/apiextractor/build/generatorrunner/build/shiboken/build/pyside/PySide/QtGui/typesystem_gui_common.xml]

21.2 Which Python?

We selected Python 3 (vs. Python 2) for this project because it strips out some deprecated language features. This means that some older Python libraries will require a modicum of porting, but the benefit is a cleaner language that is easier to learn.

> "This article explains the new features in Python 3.0, compared to 2.6. Python 3.0, also known as "Python 3000" or "Py3K", is the first ever *intentionally backwards incompatible* Python release. There are

more changes than in a typical release, and more that are important for all Python users. Nevertheless, after digesting the changes, you'll find that Python really hasn't changed all that much – by and large, we're mostly fixing well-known annoyances and warts, and removing a lot of old cruft." [By Guido van Rossum at http://docs.python.org/3.0/whatsnew/3.0.html.]

21.3 C++/Python Datatype Conversion

C++ classes that are surfaced to Python have the same names in Python. References to objects created automatically, such as tools, and their contents, are typically owned by ATSAL's C++ layer, and not collected by Python's garbage collector. Objects created in the C++ layer are typically composed entirely of C++ basic types such as Qt QLists. This allows these objects to be passed among various tools without intermediate conversion to Python. Accessors intended for use in Python code convert where necessary to native Python types, usually by creating a copy of the object. The copy will later be garbage-collected by Python, while the original copy remains owned by C++ object(s).

When both Qt and Python define identical data types, conversions are necessary. For base data types, conversions are as follows:

C++	*Python*	*Notes*
long	integer	integer At least 32 bits long (probably 64-bits, TBD)
n/a	long integer	Arbitrary precision integer is not supported in C++ (not yet needed)
bool	PySide.QtCore.bool	
double	real	Double precision floating point. Floats (single precision) in C++ are converted to/from double precision.
Complex	complex	Probably use the m++ basic type? TBD.
QString	string	Usually conversion is performed when calling across C++/Python boundary, but large collections of strings may retain QString format, so that conversion occurs only when explicitly requested. (Example: a list of hundreds of URLs might be left in QString format for efficiency.)
QFile	?	TBD.
QList<type>	sequence	In some cases, conversion is implicit; in other cases, users can choose between conversion and no conversion for efficiency reasons (see below).
QDictionary	dictionary	Similar to above

Table 21-1. C++/Python basic data types

[THK: Need to settle on an approach here, primitive types in Qt are syntactically inconvenient. Example: PySide.QtCore.bool.]

For example, a Spectrum object might offer the following methods:

C++	*Python*	*Description*
`QString filename()`	`[string] filename()`	Returns filename as a string in native format.
`long maxCount()`	`int maxCount()`	Returns maximum count in spectrum
`long minCount()`	`int minCount()`	Minimum count
`QString missionName()`	`[string] missionName`	Name of mission
`QList<Bin> spectrum`	`[sequence] spectrum`	Returns a list (immutable sequence) of a copy of the elements making up an instrument spectrum.
	`PyObject* spectrumCpp`	Like above, but returns a reference, not a copy, to the native C++ object. Treated where possible as an opaque type.

Table 21-2. Sample tool methods

The last two cases are significant. The first returns a copy of the list (or sequence in Python terminology). There is obvious overhead in making this copy, but it is in a convenient format for Python calculations, and the copy is "owned" by Python's garbage collector. If the user's Python program requires access to the actual spectrum, this is usually the best way to get it.

The second case returns a reference to the C++ object. The reference is not owned by the garbage collector, and as an instance of class QList, it must be accessed using QList semantics. This representation is useful when it will be treated as an opaque type by Python, and simply passed to another C++ object at some point.

The option of retrieving a datum in either C++ or Python format applies primarily to collection classes such as QLists/sequences or matrices, which may contain large amounts of data and therefore encounter relatively high conversion overhead.

Naming conventions:

Access	*C++*	*Python native data format*	*Python in C++ format*
`Setter`	`set`*`Foo`*	`set`*`Foo`*	`set`*`Foo`*`Cpp`
`Getter`	*`foo`*	*`foo`*	*`foo`*`Cpp`

Table 21-3. Setters and getters

21.4 Asynchronous Design Issues

ATSAL juggles operations that are potentially slow, such as retrieving groups of files from the network or performing long-running analyses, with operations that benefit from animation rate responsiveness, such as working with multiple overlapped spectra or superimposing multiple types of curve fits. ATSAL attempts to allow users to continue to work while these operations are underway. It uses several techniques to keep multiple cores busy. There are two key constraints: (1) XSPEC does not fully utilize multiple processor cores; and (2) Python, though it

supports multiple threads, does not support multiple concurrent processor cores efficiently.

ATSAL's main thread handles all Python and user interface updates. To achieve more effective use of computing resources, we employ the following:

1. A process queue manager handles some tasks, such as conversion of FITS spectra to a binary cached format that may be mapped directly into virtual memory.
2. An XSPEC process queue manager keeps at least one and possibly multiple copies of XSPEC busy simultaneously. This overlaps user interface and analysis operations.
3. Qt's multi-threading will be used where feasible to spin off long-running actions.
4. We also considered using the graphics processing unit to perform some computes, but we haven't yet found a case where this would significantly improve performance.

21.4.1 Signals

Qt signals are used in some cases to convey results to user programs. But writing asynchronous programs is a major inconvenience, so ATSAL tries to hide this, at least for commoner applications. For example, many XSPEC computations are requested interactively via the graph editors, and the interchange of commands with XSPEC, with async communications and timeouts, is hidden.

For communications with independent processes that are not handled by the user interface, users may call a signal handler to respond to incoming data. But to preserve the simple sequential nature of the Python main, they can also issue a synchronous-appearing call to a function object to retrieve a value calculated by XSPEC or another external process. As an example, consider a tool which computes a function by asking XSPEC:

```
function.compute()  # Compute function configured via user interface
x = function.result() # Retrieve result value
```

The second line above will block if XSPEC has not yet returned a result, but the call is implemented so that it will not consume computes while waiting. This sometimes permits user interface updates while waiting.

22 Graphics Approach

We evaluated both the OpenGL 3D graphics subsystem and Qt's graphics subsystem. The former is high in performance but optimized for game programs rather than high quality 2D graphics rendering, while the latter generates Postscript, the *de facto* gold standard in publication quality graphics. Because of the publication quality requirement, Postscript would be required in either case; so we were really considering whether we had to write everything twice, once for interactive speed, again for Postscript.

In doing an iPad application for the AtomDB database, we chose Postscript graphics, gaining a better understanding of performance issues.[6] We concluded that Postscript is fast enough on modern processors to be a suitable choice for ATSAL.

There is some use of conditional code to optimize screen display on devices of relatively limited resolution (≤150 ppi or so), including most (but not all) current monitors. But for the most part, the same drawing code can also be used to generate Postscript for high resolution printing.

22.1 SVG vs. Postscript

SVG, Adobe's Scalable Vector Graphics standard, is an open standard for representing device-independent graphics. Postscript is a "display-only" representation, in that it does not preserve enough context about relationships between graphical elements to permit some kinds of editing after the fact, while SVG is a representation that is intended to support later edits. SVG is human-readable and computer-parsable, making it possible to automate certain types of downstream processing as well.

Hence ATSAL will generate either Postscript, which is older and widely supported as a display-only format, and SVG, which can be further edited later, a convenience for preparing image data for publication. SVG files may be edited by Adobe Illustrator and OmniGraffle Professional on the Macintosh, or InkScape and several other editors on Linux. (InkScape used to run on the Macintosh as well, but does not at the time of this writing.)

22.2 JPEG and PNG Export

Graphs may also be exported in these two widely used bitmap formats.

[6] The iOS operating system employs Objective-C as a development language, a language that is less efficient than Python or C++; and tablet processors are slower than their desktop counterparts.

23 External Dependencies

23.1 Dependencies for ATSAL Users (vs. Developers)

Tool	*Version*	*Description*
XSPEC	11	http://heasarc.gsfc.nasa.gov/xanadu/xspec/ X-ray spectral fitting
Qt	4.8.1	http://qt-project.org User interface and many other libraries
PySide		http://qt-project.org/wiki/PySide_Binding_Generator Binding generator used to link Qt or other C++ libraries for Python access.
cmake		http://www.cmake.org Software build tool, needed to build ATSAL for Linux, and by ATSAL developers
Python	3.3	http://www.python.org Python programming language
AstroPy②	0.2.1	http://www.astropy.org Includes many Python components of use to astronomers, listed in following section
CCFits	2.4	http://heasarc.gsfc.nasa.gov/fitsio/CCfits/ C++ wrapper for FITS library
pyregion		http://leejjoon.github.io/pyregion/ Way to import DS9 regions into Python program

Table 23-1. Software required by ATSAL

Design Issue

Ideally, all these components will be bundled with distributions of ATSAL, in order to minimize version skew. However, it is not clear whether it will be possible to do this on user systems that have installed some of these components independently, for other purposes.

23.1.1 AstroPy

AstroPy aggregates several Python packages of use to astronomers. By adopting this package as a part of ATSAL's programming environment, we add a strong base of tools to ATSAL's Python programming layer.

Design Issue

This is a double-edged sword, because we cannot verify the operation of these tools, and a bug anywhere in the entire library has the potential to bring ATSAL to its knees. A second issue is potential for data conversion issues when moving data between various components. We need to take a holistic view to presenting a reasonably seamless programming environment to users who wish to delve more deeply into Python programming.

Hence this is delayed until release 2. We'll see how long our resolve lasts...

Tool	*Version*	*Description*
NumPy		http://www.numpy.org Matrix arithmetic with n-dimensional arrays; much other math
SciPy		http://www.scipy.org Integration, interpolation, Fourier transforms, signal processing, linear algebra, sparse Eigenvalue problems, compressed sparse graph routines, statistics, multidimensional image processing, more...
Matplotlib		http://matplotlib.org Plotting
Constants		Constants
nddata		N-dimensional datasets
units		Units
time		Times and dates
coordinates		Astronomical coordinate systems
table		Data tables
cosmology		Cosmological calculations
io.fits		FITS files
io.ascii		ASCII tables
io.votable		VOTable XML handling
io.misc		Miscellaneous I/O
wcs		World Coordinate System
stats		Astrostatics tools
utils		AstroPy core package
config		Configuration system
io.registry		I/O registry

Table 23-2. Software bundled with AstroPy

23.1.2 Additional Dependencies for ATSAL Developers

Tool	*Version*	*Description*
cmake		http://www.cmake.org Software build tool, needed to build ATSAL for Linux, and by ATSAL developers
git		http://git-scm.com Source code management tool
Doxygen		http://www.stack.nl/~dimitri/doxygen/ Developers documentation from information embedded in source code
Qt	4.8.1	http://qt-project.org User interface and many other libraries
PySide		http://qt-project.org/wiki/PySide_Binding_Generator Binding generator used to link Qt or other C++ libraries for Python access

Table 23-3. Additional software required for ATSAL developers

24 Sharing and Version Skew

Definitions: a *Python library* is a collection of related Python source files, documentation, and tests. In the context of this document at least, a *Python package* is a Python library that has been registered with PyPI (Python Package Index, pypi.python.org/pypi), so that is has a unique name and a uniform system for version tracking.

This section describes sharing of ATSAL notebooks, archives, and Python libraries. A major design goal for ATSAL is to permit exchange of notebooks among ATSAL users, while minimizing version skew problems. ATSAL uses several strategies to try to minimize skew:

- Each release of ATSAL is "locked" to a particular release of the tools upon which it depends, such as XSPEC, Python, Python libraries, etc. Dependent tools are bundled to ensure this correspondence.
- Major releases have a relatively infrequent release cycle, approximately once per year.
- Minor releases, which correct bugs, or track new operating system releases when necessary, remain with the same releases of dependent tools wherever possible. Thus if ATSAL v2.0 uses matplotlib v7.3, ATSAL 2.1 probably will too.
- Key Python libraries such as matplotlib and AstroPy are bundled with ATSAL so that all users receive the same version.
- External components downloaded from PyPI (Python Package Index) are "version locked" for notebook archives. That is, a recipient gets the same version of a package that was used by the notebook's creator. The recipient has the option of running with the original version or upgrading.
- Libraries that are not registered with PyPI are also bundled with notebook archives to minimize version skew, but updates are not managed automatically.

Design Issue

This doesn't come close to solving all the skew problems. In particular, there is no formal guarantee (other than unit tests) that new releases of ATSAL will exactly reproduce results from previous releases. There is not even a guarantee that older Python libraries will run on newer releases of Python, and it is a foregone conclusion that we will have to adopt newer Python releases over time. Since older releases of ATSAL may no longer run on newer OS releases, it is not feasible simply to keep multiple versions of ATSAL around for running older notebooks. Any ideas appreciated.

So sharing an ATSAL notebook does not guarantee version-level compatibility with external Python packages. But ATSAL archives do make this guarantee, to

the extent possible. The downside of this is that ATSAL archives may be very large, since they bundle these libraries as well.

24.1 External Python Packages and Libraries

Python packages fall into one of these categories:

Bundled	These packages are included with ATSAL itself. They are updated only at major release cycles of ATSAL, unless there is no alternative.
PyPI	Packages from the PyPI, or Python Package Index, are "version-locked." If you create a notebook archive, it includes current versions of libraries used by the notebook in the archive. When a recipient opens an archive, PyPI packages are checked against the online resource (http://pypi.python.org/pypi) to see if newer versions are available. If so, the recipient can choose to run the notebook against the version supplied with the notebook or the current version. The decision is reversible.
Other external libraries	Use of unregistered external libraries is not recommended because version skew management cannot be done reliably. If you use such libraries, they too are bundled with notebook archives, but automatic update management is not possible.
Notebook libraries	Libraries that are specific to an analysis procedure, or are under development in a notebook but intended for later sharing, are stored in the notebook itself. Hence they are saved with the notebook as well as with archives.

Table 24-1. Python package types

24.1.1 Bundled Packages

Bundled packages are an intrinsic part of ATSAL. Users accessing these packages can be certain they will behave uniformly for all users of a given release of ATSAL. ATSAL developers will make a best effort to ensure backward compatibility for these packages as well. From the perspective of version skew, these packages are "safest" to use.

24.1.2 PyPI Packages

The Python Package Index lists about 1800 scientific and engineering packages at the time of this writing, some of which may be applicable to ATSAL programs. A package downloaded from this source is incorporated into any notebook archives that use it, but not saved with the notebook.

A recipient of a notebook archive receives a notification if a newer version of the package exists at PyPI. The user can run with the original version (default), or try running with the current version. It is possible to revert to the original version at any time. ATSAL does this by checking with the PyPI web site and, optionally, downloading the newer version automatically.

If the notebook is re-exported as a new archive, the currently selected version of the PyPI package becomes the new default for the notebook.

24.1.3 Other Libraries

Other Python libraries do not conform to a uniform standard for version representation, though there are some informal standards. Hence ATSAL does not attempt to manage version updates. However, ATSAL bundles these libraries as part of notebook archives too. Recipients must handle any upgrade issues manually.

24.1.4 Notebook Libraries

When a user adds a directory containing a Python library to a notebook, the default is that the files themselves are not transferred from their original location, and they appear with a lock icon in the file pane. Alternatively, the user may elect to copy the files into the notebook, and the local copy may be modified as needed. (A locked external library may be converted to an unlocked notebook library at any time—the files are copied into the notebook.)

When users add new Python files to a notebook, they are created inside the notebook and are local to the notebook.

ATSAL preserves these files in the notebook.

24.1.4.1 Exporting Notebook Libraries

If a collection of Python programs would make a good library, users can share the library by bundling the files into a directory, and adding documentation and tests. We recommend that you register the package with PyPI. Notify ATSAL developers about the package and we will list it on the web site and/or assist in PyPI registration.

Once a package is registered, you can convert a notebook to use the package instead of the notebook library by deleting the notebook library and adding the package.

25 File Organization

ATSAL files associated with the application, or shared among all notebooks, reside in ~/ATSAL, where "~" indicates the user's home directory. Each notebook has a directory tree with the name of the notebook. Caches and temporaries are stored in standard locations for these files as dictated by the operating system.

25.1 Extraction File Caching

Extraction files may be downloaded from any notebook using the Spectrum Download tool. These files are aggregated into a directory tree on the local system, where they become available to searches from all notebooks. Users can maintain this directory themselves, for example, by deleting files that are no longer needed. When a user creates an archive from a notebook, either to store results of an analysis or to give a copy to a colleague, they can choose to copy the extractions used by the analysis into the archive.

Extraction files are located via search paths, so users can choose other storage schemes. For example, collaborators might choose to store extractions on a shared network volume. ATSAL does not automatically download to the shared area, but users can move files by hand to the area and include it in their search paths.

Extractions (at least those from TGCat) are stored with two levels of compression. Individual .PHA, .RMF, .ARF, etc. files are compressed; so are groups. When downloaded for local use, the outer level of compression is removed, but the individual files remain in compressed form. .PHA files are temporarily decompressed and their FITS metadata entered into a hidden MySQL database. The database allows very fast searches of local extractions for use. It also ensures that extractions are not downloaded if they are already present.

When users select files for analysis using the Spectrum Files tool, they are decompressed (and possibly converted to a more efficient form) and stored in each user's ATSAL cache. This hidden directory is not backed up, can be flushed at any time, and has a user-settable maximum size. The database tracks the cache directory too, to prevent redundant caching. While the original files may be shared among collaborates, the cached files are local to each machine.

25.2 Sessions

An analysis session contains the results of a single session of using ATSAL. The session includes activity logs and output files generated from a run. During a session, any automatically generated files such as exported graphs overwrite their previous versions, so at the end of the session only the most recent versions of each file exist.

To preserve a snapshot of a current session and begin a new one, the user chooses New Session. This creates a new folder, Session_*date_time*, e.g. Session_2014-07-07_14-15-58, in the Outputs folder. Log files are terminated in the previous session and re-opened in the new one, and all new outputs are directed to the new session folder. Thus New Session provides a simple way to obtain a

clean record of a run from start to finish. If a run is already in progress, New Session warns of this and offers to abort the current run.

25.3 Files on Mac OS X

File Type	*Location*
Extractions	~/ATSAL/Spectra/*Archive/Source/* *Archive* is the archive from which the file was downloaded and *Source* is the approximate region the data observes
Examples	~/ATSAL/Examples Contains sample notebooks
Caches	~/Library/Caches/org.atsal/ Standard location for application-specific cache files—a preference determines the upper size limit
Extractions database	~/Library/Caches/Extractions.db A MySQL database of the extractions currently available on the local system.
AtomDB	~/ATSAL/Atomic Databases/ATOMDB/ This version has been converted for very fast loading. Users can separately download the ATOMDB in FITS format.
Python packages	~/ATSAL/Python/ Python packages supplied with ATSAL are placed here.
ATSAL application	/Applications/ATSAL v*n*.app/
Notebook Templates	~/ATSAL/Notebook Templates/ Notebook templates are a readonly part of ATSAL, so they are stored in the application bundle.
Temporaries	Stored in a temporary directory generated by the operating system.

25.4 Files on Linux

File Type	*Location*
Extractions	~/ATSAL/*Archive/Source/* *Archive* is the archive from which the file was downloaded and *Source* is the approximate region the data observes
Examples	~/ATSAL/Examples Contains sample notebooks
Caches	? Standard location for application-specific cache files—a preference determines the upper size limit
Extractions database	? A MySQL database of the extractions currently available on the local system.
AtomDB	~/ATSAL/Atomic Databases/ATOMDB/ This version has been converted for very fast loading. Users can separately download the ATOMDB in FITS format.
Python packages	~/ATSAL/Python/ Python packages supplied with ATSAL are placed here.
ATSAL application	/usr/bin/atsal
Notebook Templates	~/ATSAL/Notebook Templates/ Notebook templates are a readonly part of ATSAL, so they are stored in the application bundle.
Temporaries	Stored in a temporary directory generated by the operating system.

25.5 ATSAL Notebooks

An ATSAL notebook is a directory containing files associated with the notebook.

File Type	*Location*
Notebook	*Notebook*/Notebook/Notebook.atl The notebook and its directory share the same name by default.
Manifest	*Notebook*/Notebook/Manifest.xml Contains a list of the input files associated with this notebook, including version information, as needed to reproduce the analysis with the same files by downloading them from archives.
Documentation	*Notebook*/Notebook/Docs/ Stores other files associated with this analysis notebook, such as PDFs.
Sources	*Notebook*/Notebook/Sources Stores Python programs specific to this notebook.
Sessions	*Notebook*/Outputs/Session_*date_time*/ Output files produced during analysis of an extraction are stored in subdirectories by extraction name. Date is the date and time when the ATSAL session was initiated for this extraction.
Python packages	*Notebook*/Python/ Contains Python packages used by this notebook that are not part of the standard ATSAL distribution.
Logs	*Notebook*/Outputs/Session_*date_time*/Logs/ Logfiles generated during an analysis are stored here.
Extractions	*Notebook*/*Archive*/*Source*/ *Archive* is the archive from which the file was downloaded and *Source* is the approximate region the data observes. Extractions are present within a notebook only in archives that explicitly include them. If desired, they can be moved equivalent locations in ~/ATSAL to make them available to future analyses.

Table 25-1. Notebook file tree. Shaded files are optional

The Notebook subdirectory contains all the files that are an essential part of the notebook. The Notebook manifest, Manifest.xml, lists the input files necessary to reproduce an analysis, with enough additional information to ensure that the files, once downloaded again, are identical to those used in the original analysis. The Documentation folder contains files that are associated with the notebook, but play no role beyond supplying useful documentation. The Outputs folder contains results from analysis sessions. The Extractions folder may be present in ATSAL archives if the user elects to store extractions in the archive.

25.6 ATSAL Archives

Notebooks may be exported as archives. The simplest form of an archive is a gzipped notebook, containing only the core Notebook subdirectory. Users can include extractions, Python packages, and other optional components as well. ATSAL archives increases the likelihood that a recipient will be able to reproduce an analysis by including files the recipient may not have or be able to download.

26 User Interface Extensions

This section describes a series of user interface components designed for use in ATSAL. The components address (to the extent possible) creation of consistent-appearing user interfaces on systems with different font metrics, since in past projects this problem has required a lot of custom per-platform layout.

Users may select only the portions of the user interface that are most suitable to their current work, by closing sections that are less frequently used.

26.1 Cross-platform User Interface Issues

This section addresses some of the issues involved in cross-platform user interface design.

26.1.1 Font Metrics

Two common problems result from changes in font metrics on different systems: text truncation or other effects from varying fits, and baseline alignment variations. Compensating for the former problem often forces overly loose layout of user interface elements, undesirable with complex interfaces in restricted screen real estate. The second issue, while only cosmetic, is easily avoided.

We address the former by choosing system font(s) for each platform that have nearly identical font metrics. The latter problem is addressed by defining user interface elements that adapt themselves to font variations on different platforms, so that hand adjustment is not necessary.

26.1.2 Special Characters

The availability of Unicode simplifies the use of specialized glyphs, but leaves many problems unsolved. Special glyphs are defined for only a subset of fonts, or perhaps none at all, so their availability and appearance is uncertain. Characters widely used by ATSAL, such as Greek, will be selected from available fonts. ATSAL makes no guarantee that less widely used characters will be preserved from one platform to another, only that Unicode will be used to encode them.

26.1.3 Resolution Independence

There are three graphics design approaches in use on all desktop operating systems, each adapted to particular types of tasks:

- User interface widgets are most often implemented with coordinate systems in integers, with a one-to-one mapping between logical pixels and physical pixels, so that interface element size varies with the actual resolution of the screen. This approach is rooted in low resolution, low performance displays which produce unattractive aliasing when scaled.
- Postscript-style graphics express primitives in a coordinate system that is independent of device resolution, and support 2D imaging options that are lacking from many other packages (e.g. affine transforms, paths, path clipping, gradient fills, translucency). The logical coordinate system is floating point. Postscript works best with high resolution rendering

devices such as printers. Historically its performance was sluggish on screen, but newer systems render it well and Postscript-style graphics are a superior choice for contemporary user interfaces.

- 3D graphics chips and cards have extremely fast polygon rendering for realistic scene generation. They are popular for games and some kinds of data visualization, but they are not designed for high quality rendering for traditional applications, though they assist in many such rendering operations because of their high speed. OpenGL provides the 3D programming interface.

Apple's Core Graphics subsystem is Postscript-native, but Linux has not yet adopted a similar convention. Qt, the cross-platform class library ATSAL employs for its user interface, offers a user interface implementation that is integer-based, based on the QWidget class. It also offers full Postscript-style support in the form of QGraphicsView and related classes, but the user interface itself does not yet use this, nor does, as far as we are aware, any other cross-platform user interface library.

Hence, ATSAL will not be resolution independent. We will take the following approaches:

- Most user interface elements, with the exception of a few that we create specifically for ATSAL, will be implemented in integer-based coordinates and written for approximately a 120 dpi display. This is consistent with most applications written today.
- A "trick" in Qt enables support for Apple's high-resolution "Retina" displays, which have double the resolution. This will make user interfaces on Retina displays look reasonable. Since this solution is Apple-specific, it is not ideal, but there is not yet an alternative.
- All *graphing* code written for ATSAL will employ QGraphicsView, so it will be fully resolution-independent and will image well on Postscript printers.

It would be preferable to support resolution independence from the outset of the project, given ATSAL's long lifespan. This would require that all user interface elements be rewritten as QGraphicsView subclasses, a considerable and probably prohibitive extra expense.

26.2 User Interface Panels

Some of ATSAL's tools involve fairly complex control panels, especially those that fine-tune graph settings. The complexity requires that elements be fairly compact in size, and that users can adjust their display to favor heavily used controls. Many of ATSAL's user interface elements are similar in some respects to those used by Adobe Lightroom, a commercial product for managing and adjusting settings of collections of photos.

Illustrations on the two following pages show some characteristics of Lightroom's interface that have analogs in ATSAL's user interface.

User interface elements are quite compact, and employ reduced contrast to avoid competing with the subject

Heavily used interface elements are non-scrolling, like this histogram pane

Interface panes may be displayed or hidden as needed

Where possible, numeric entry is replaced with sliders. Numeric values are shown though

Related elements are grouped...

...and sub-grouped

Much of control panel is scrollable, allowing for later additions

Figure 26-1. Sample Adobe Lightroom control panel, part 1

Entire user interface control panel may be expanded or collapsed *(triangle at far right)*

Subtle UI feedback, like the histogram behind this tone curve, is shown with low contrast to avoid distraction

Effects of a panel of controls may be turned off temporarily to compare before/after settings *("switch" at far left)*

Color is used only when it plays a meaningful role in the operation of a control

Fonts and sizes are chosen for consistency across supported platforms, making layout simpler

Figure 26-2. Sample Adobe Lightroom control panel, part 2

Lightroom employs a dark background with lighter foreground elements in order to focus attention on the photos that are the subject. ATSAL will not do this,

though, because printable output is a goal and light backgrounds are preferable for printing.

26.3 UIPanel User Interface Components

The following components are designed for use in UIPanels. They initialize to conform to a PanelDisplayStyle, which provides a uniform style.

26.3.1 UIPanel

A panel that may appear on the left or right side of a window, which may be expanded or contracted. It consists of an optional unscrolled pane at the top, and a scrollable region at the bottom. Panel height matches window height. Panel width is settable over a limited range. Each panel component calculates its minimum width, and this is used to set the minimum width of the UIPanel.

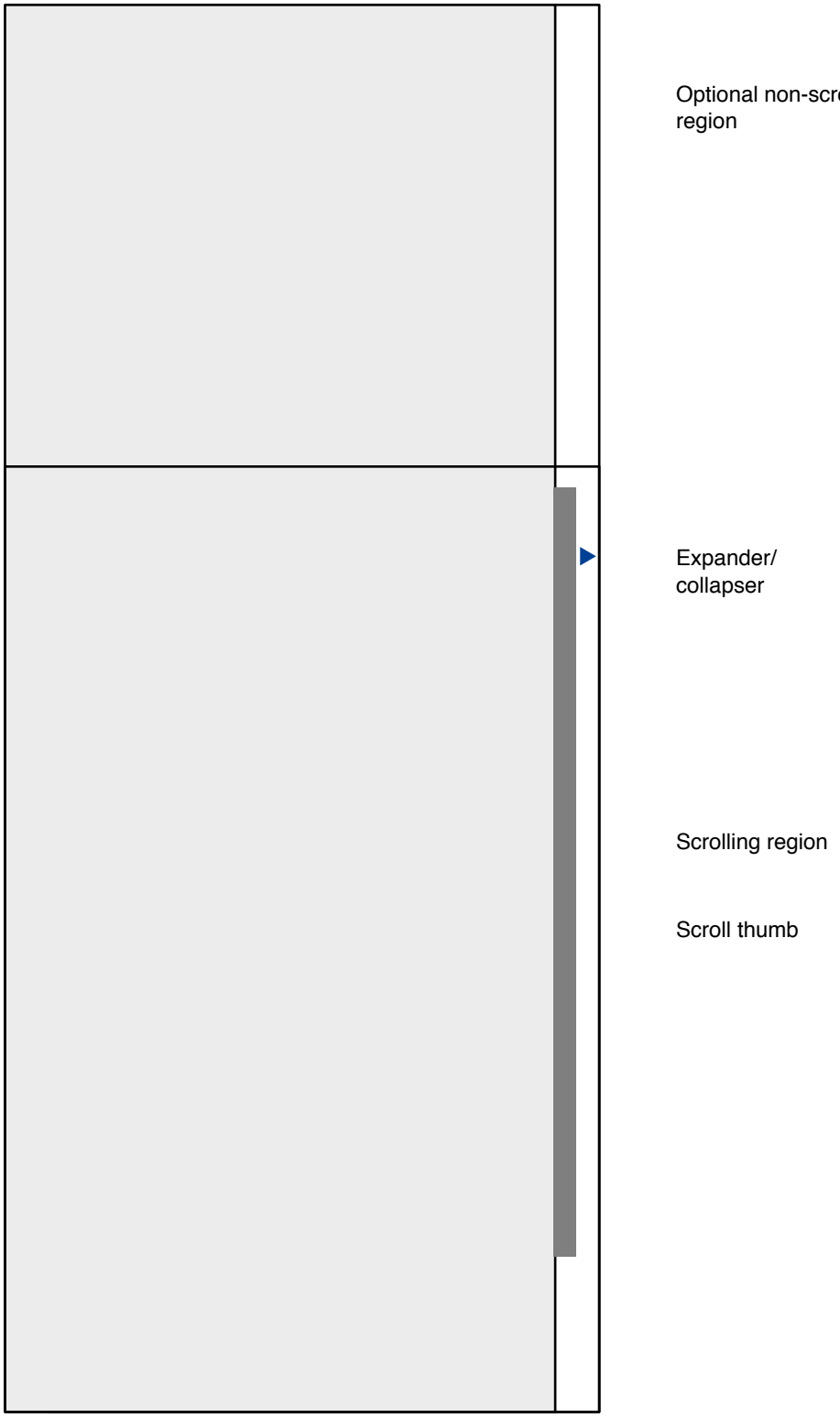


Figure 26-3. UIPanel

The expanded or collapsed state of a panel is remembered on a per-window basis.

26.3.2 UICollapsiblePane

This element collapses into a name and disclosure triangle, or expands to display one or more QWidgets. The expanded or collapsed state of the pane is remembered on a per-window basis.

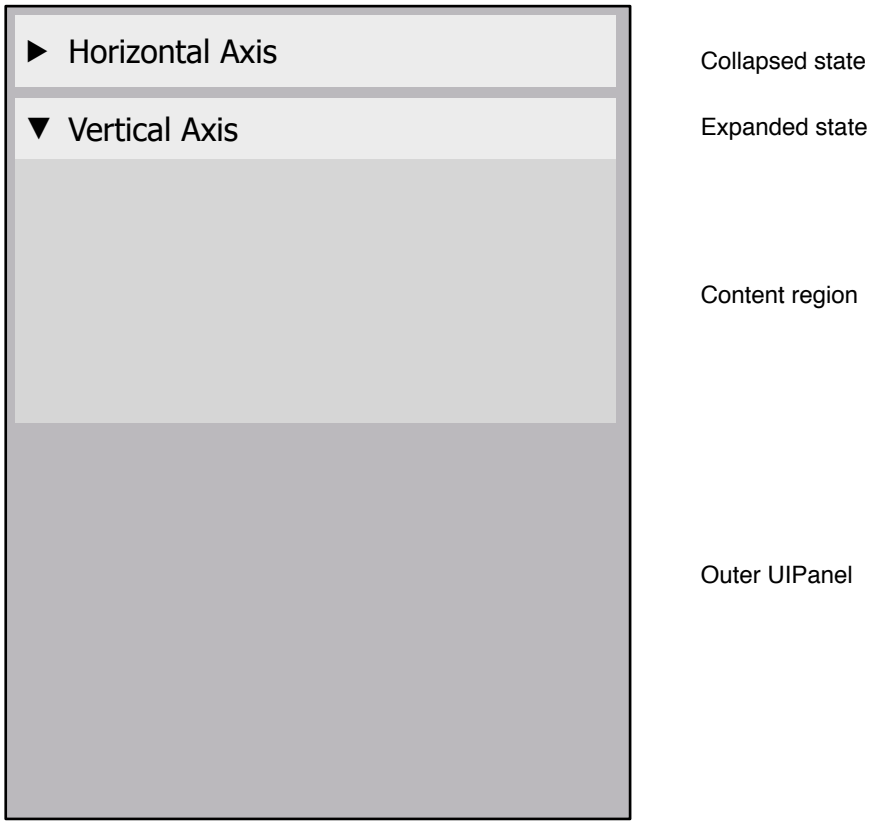


Figure 26-4. UICollapsiblePane

26.3.3 UIHierarchy

Displays a hierarchical view, for example, of files in a filesystem. Each entry in the hierarchy consists of a disclosure triangle, an optional icon, a text label, and an optional right-justified datum.

26.3.4 UIMiniTabs

Mini-tabs are a group of selection tabs, each of which has an associated QWidget. These tabs have the following constraints:

All tabs must fit within the minimum allowable width of the owning QWidget.

The content QWidget for each tab in a UIMiniTab must be the same size.

Figure 26-5. UIMiniTabs

26.3.5 UISeparatorBar

This is a simple rectangle used to separate regions of a QWidget. The bar style distinguishes major vs. minor separations.

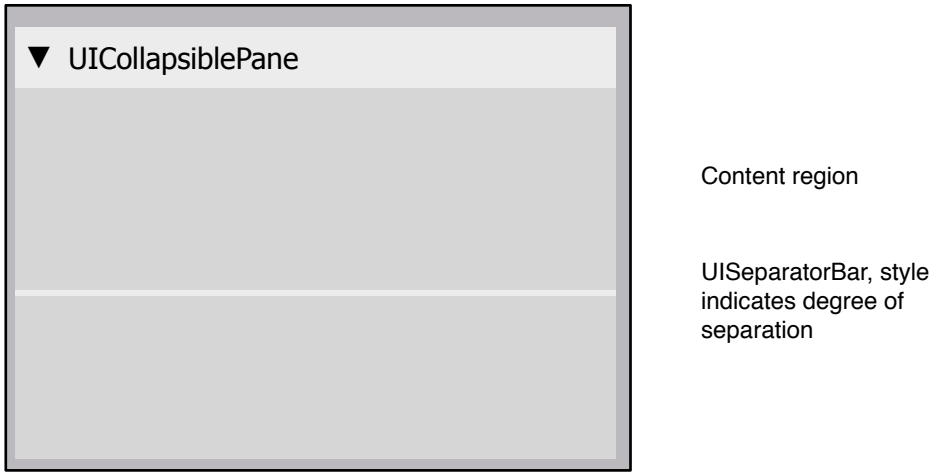


Figure 26-6. UISeparatorBar

26.3.6 UIToolBar

A UIToolBar displays a set of toolbar icons. The only special behavior here is to align the icons with either the left or right edge, or distribute them evenly across the bar. UIToolBars may not have more than one row of icons—use multiple UIToolBars if needed.

26.3.7 UIButton

A standard button.

26.3.8 UIPopupMenu

Displays a pop-up menu using a portable font. Includes an optional label above or to the left of the menu.

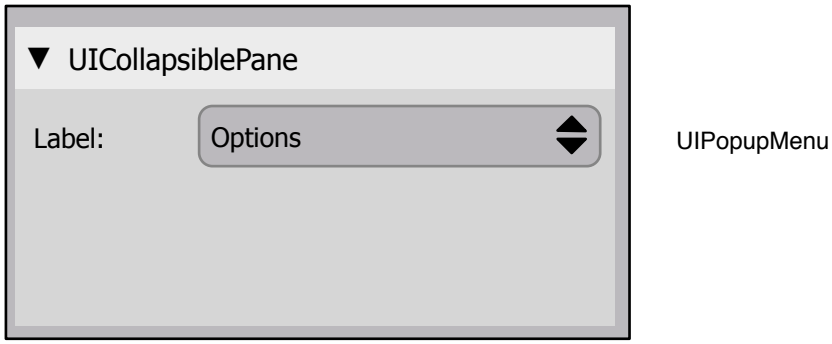


Figure 26-7. UIPopupMenu

26.3.9 UISlider

A UISlider consists of a label, a slider, and a numeric value display area. The numeric value may be display-only, or editable. A slider has an upper and lower limit, and may be integer only or floating point.

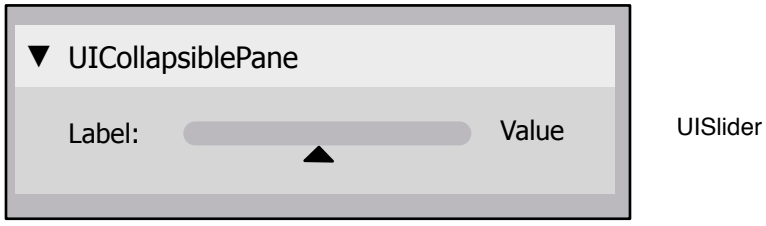


Figure 26-8. UISlider

The label is left-adjusted. The value (if shown) is right-adjusted. The slider is sized to fill the remaining space. If the value is editable, it is shown with a bounding box.

26.3.10 UISliderGroup

A UISliderGroup is a group of UISliders that are uniform in size and evenly spaced.

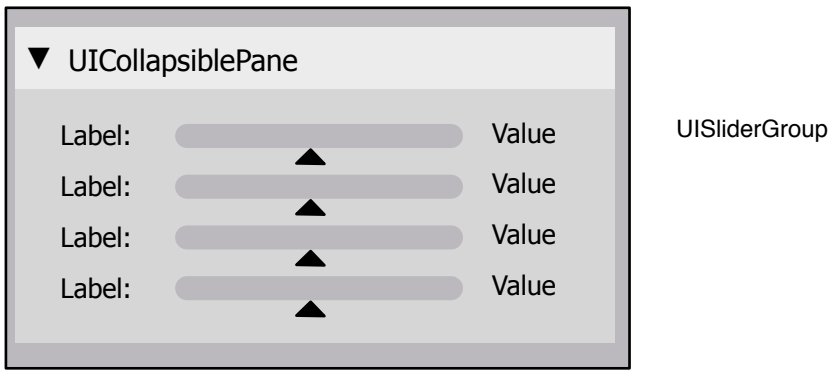


Figure 26-9. UISliderGroup

26.3.11 UILabel

A UILabel displays a single line of text in the standard ATSAL user interface font. The selected point size is mapped if necessary to produce text closely similar in size on each platform.

26.3.12 UISymbolPane

This pane displays symbols commonly used by astrophysicists, for inclusion in notebooks. It is simply an array of glyphs.

26.3.13 UIValue, UIValueWithUnits

A UIValue is a box for entry of a text or numeric value. The box has an optional UILabel that may be aligned above or to the left of the edit box.

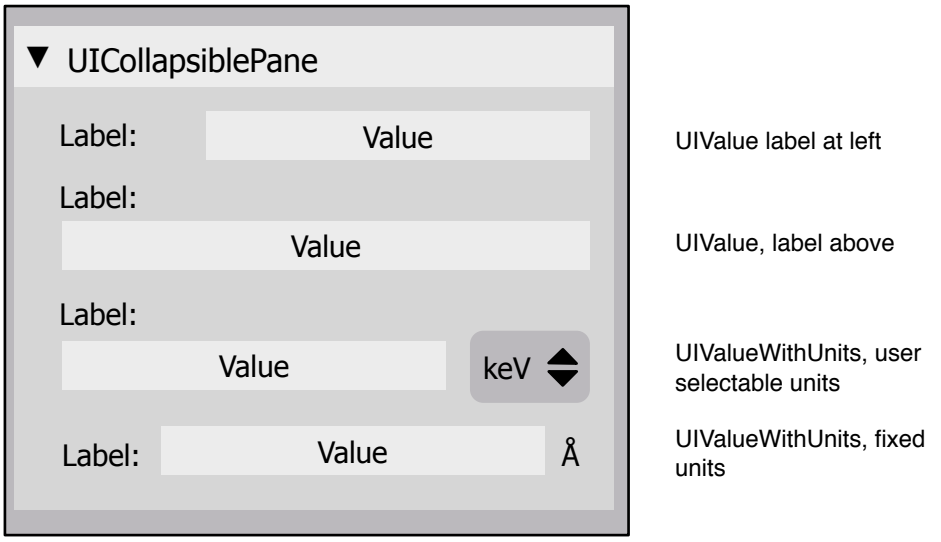


Figure 26-10. UIValue, UIValueWithUnits

The text baselines of all elements are aligned. When labels are displayed to the left, the label width is set explicitly, and the value width is adjusted to fill the remaining space. If the label is displayed above, the value width fills the space.

A UIValueWithUnits also includes a popup menu for selecting units, or a label for displaying fixed units. The units are always right-adjusted, and display to the right or below the value. The returned data type is a class that represents a quantity independent of units, such as a Lambda or a Velocity. A datum entered in a selected units is converted to other units when the units popup is used. Internally, no precision is lost during these conversions.

26.3.14 UICheckbox

A UICheckbox is a standard checkbox and associated text.

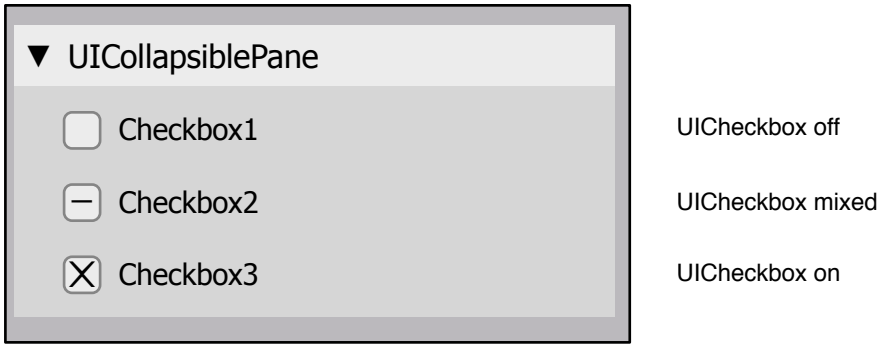


Figure 26-11. UICheckbox

26.3.15 UITextSelector

This consists of an optional label, shown to the left or on top, followed by a few mutually exclusive choices shown as text. The choices must fit within the minimum width of the UIPanel.

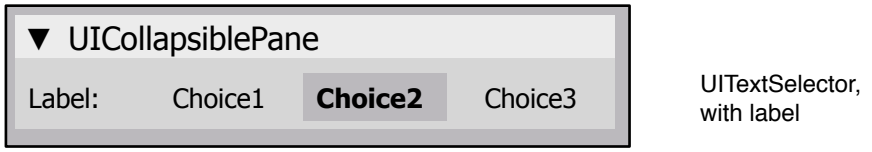


Figure 26-12. UITextSelector

26.3.16 UIIconSelector

A UIIconSelector is identical to the above, but selects among a set of icons.

26.3.17 PanelDisplayStyle

A PanelDisplayStyle is not a user interface element. Instead, it contains information used to lay out UIPanels in a fashion that is stylistically consistent and as uniform as possible across platforms. Standard UIPanel classes use this information by default.

27 Plot Classes

A Plot is a graphical depiction of data, including all the elements associated with its display, such as titles, labels, axes, etc. A PlotLayer is the graph itself, and any embedded labels.

27.1 Plots

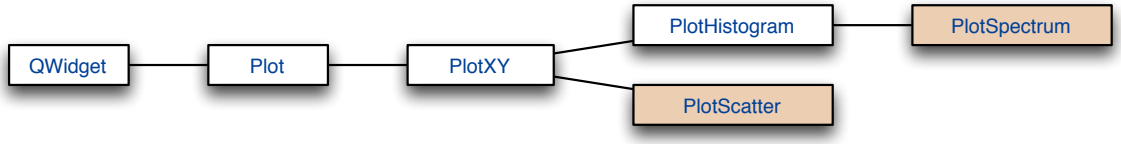


Figure 27-1. Plot classes. Colored classes are concrete (designed to be instantiated).

27.2 Axis Classes

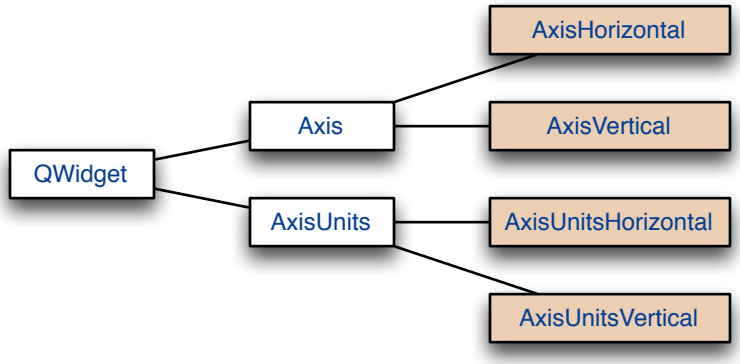


Figure 27-2. Axis classes

An Axis displays the numbering of units. An AxisUnits displays the units. The concrete classes may be used as-is, or subclassed to add features.

27.3 Plot Layers

A plot is composed of a series of one or more stacked plot layers, each of which presents a certain type of information.

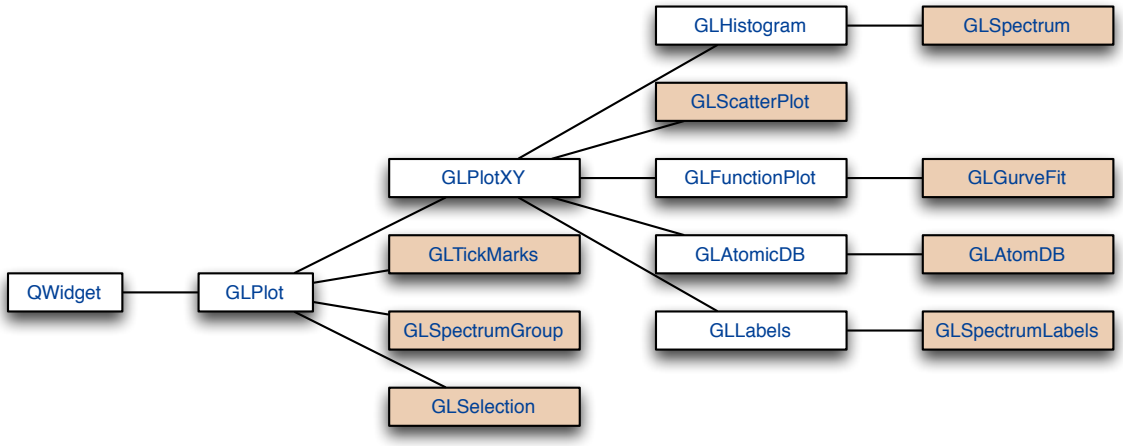


Figure 27-3. Graph Layer class hierarchy

These classes are described below:

GLPlot	Base class for a layer in a plot.
GLPlotXY	Base class for X/Y layer: any X/Y plot.
GLHistogram	Base class for histogram classes that show individual histogram cells when zoomed.
GLLayer	Concrete class for showing a spectrum, fit, or annotation.
GLScatterPlot	Concrete class for showing a collection of x/y values.
GLFunctionPlot	Base class for plots that evaluate a function to produce a plot.
GLCurveFit	Concrete class that displays a curve fit.
GLAtomicDB	Base class for plots that show an atomic database, including continua, emission lines, and absorption lines.
GLAtomDB	Concrete class that displays the AtomDB database.
GLLabels	Base class that creates and positions labels.
GLSpectrumLabels	Concrete class that creates and positions atomic database and selection labels.
GLTickMarks	Tick marks or graticule.
GLLayerManager	Contains a group of GLLayers
GLSelection	This layer contains selections as they are overlaid on the plot.

Table 27-1. Graph layer classes

27.3.1 GLPlot

A QWidget with a transparent background, a text color, and a line color. It has a pointer to a PlotSettings, which contains information that pertains to the layout of all elements of a plot.

27.3.2 GLPlotXY

This class, a subclass of GLPlot, displays an X/Y plot. Includes a pointer to the data to be plotted.

[THK: Any behavior here?]

27.3.3 GLHistogram

A GLHistogram is a GLPlotXY that interprets its data as a series of histogram cells. The cells resolve into separate bins at high zoom, and show explicit counts. At lower zoom levels, the cells are merged into a scatter plot.

27.3.4 GLSpectrum

A GLSpectrum is a GLHistogram. GLSpectrum and its subclasses know about specific sensor properties.

27.3.5 GLScatterPlot

A GLScatterPlot is a GLPlotXY that shows a series of x/y values as points on a graph. Optionally, the points may be connected by line segments. More sophisticated fits are done with GLCurveFits.

A generic GLScatterPlot has a fixed set of units for x and y. To change this under program control requires that a new data set be generated with different units.

27.3.6 GLFunctionPlot

This base class, a subclass of GLPlotXY, displays an x/y plot by evaluating a function. Subclasses of GLFunctionPlot override:

```
double GLFunctionPlot::function(double x)
```

The function must return values at any zoom level.

27.3.7 GLCurveFit

This GLFunctionPlot override provides any of several common curve fits.

27.3.8 GLAtomicDB

This GLPlotXY subclass displays a plot from an atomic database. It is overridden for each specific database.

27.3.9 GLAtomDB

This subclass of GLAtomicDB displays emission lines from the AtomDB atomic database, either the general set, or subsets for a particular temperature range.

27.3.10 GLLabels

This class generates the labels that appear in a plot. Because labels must make room for each other, each subclass of GLLabels must know about all the data sources from which labels may be created, and make decisions about placement.

27.3.11 GLSpectrumLabels

This subclass of GLLabels does automatic labeling by merging data from AtomDB and selections.

27.3.12 GLTickMarks

Displays tick marks or a graticule grid.

27.3.13 GLLayerManager

Displays a group of layers (GLLayer). Implements behaviors that apply to the group as a whole.

27.3.14 GLSelection

Displays selections as they appear in the graph area.

28 XSPEC Server

ATSAL communicates with one or more instantiations of XSPEC that are running on the same machine, or possibly on remote machines. It employs HTTP protocol and a remote procedure call (RPC) interface that is based on a fairly widely used RPC interface library that uses XML-formatted data packets for data exchange: http://xmlrpc-c.sourceforge.net. The RPC interface replicates the C-level XSPEC class library interface as a client. It is a freestanding library that may be used by other applications as well.

Design Note

We originally planned to write the XSPEC driver in PyXspec, but the presence of Python's Global Interpreter Lock makes this awkward, and implementing it at the C level provides the client process with the option of using either the C library interface or the PyXspec Python interface. A C-level interface is also somewhat more efficient.

XSPEC is frequently hard compute-bound for long periods, preventing a driver thread from operating, so the architecture, shown below, addresses this.

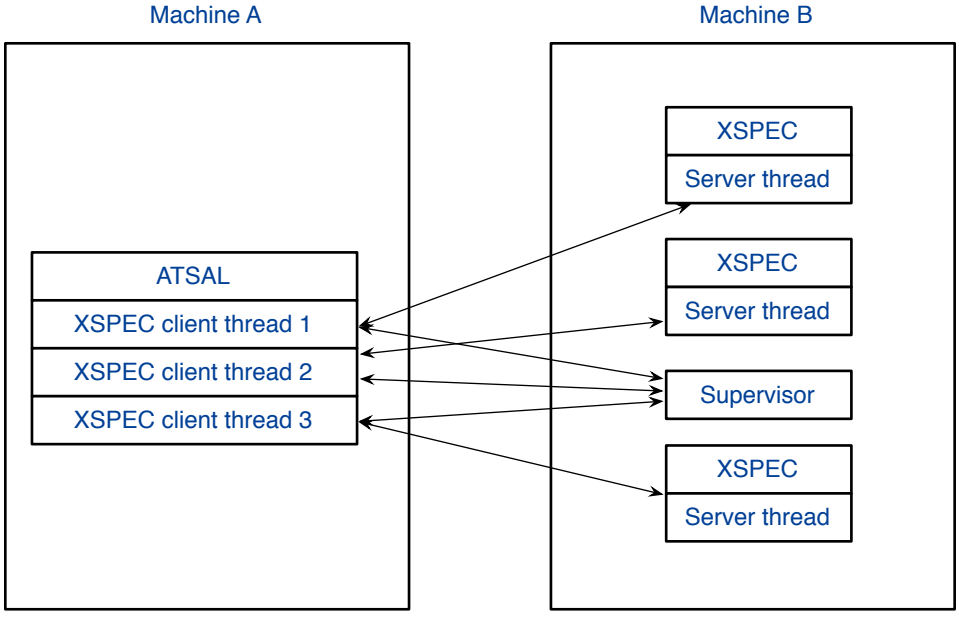


Figure 28-1. XSPEC/ATSAL RPC design

ATSAL runs on Machine A. It includes a driver thread for each instance of XSPEC that it controls. On machine B, there is a process for each XSPEC server, which includes a synchronous RPC driver thread. ("Synchronous" in that it is only able to respond to commands or send results when XSPEC is not compute-bound.) There is also a supervisor process, one per machine, which listens asynchronously for interrupt requests. When an interrupt request is received for a particular instance of XSPEC, the supervisor interrupts the XSPEC process by sending it a SIGINT signal, at which point its driver thread begins responding to

commands. The supervisor process also doubles as a listener or daemon for instantiating new XSPEC instances.

Although shown as two machines, all these processes may be present on a single machine, or spread over multiple machines.

The protocol exchanged via the driver communications channels is structured in XML format. The driver is written in C++ on both sides of the link, and it is designed to replicate the C level XSPEC library interface on the client side. Since the C level interface is replicated, this makes it possible to use the PyXspec code on the client side to provide Python access to a remote XSPEC as well.

28.1.1 ATSAL/XSPEC Communications Protocol

Commands, replies and status are formatted in XML. XML packets are always sent in full, so they are atomic. Packet traffic is logged for debugging, but it is not directly viewable by users. The ATSAL driver converts these packets to a more human readable format, which may be optionally viewed in the helper pane of the notebook window.

Since ATSAL needs to maintain information necessary to operate XSPEC by sending the minimum number of commands needed, as a side effect, it has enough information to be able to bring a new instantiation of XSPEC into the same state as a prior one. Hence ATSAL can recover from an XSPEC crash or hang by restarting XSPEC and resending all necessary commands.

Each time the user requests a refresh, ATSAL determines a sequence of commands to be executed by XSPEC to bring everything up to date. These commands are prepared and packaged before the first is executed. As each command executes and receives a reply, the dependency tree that tracks the actual state of XSPEC is updated. If an error occurs, the rest of the command series is discarded, and the error presented to the user. After correcting the error, the user hits refresh again, and a new set of commands are created.

28.1.2 XSPEC Client

The XSPEC server will make the same services available that are presently available via the PyXspec Python interface available as part of XSPEC. The PyXspec interface is implemented with calls to a set of functions listed in the _PyXspecMethods array initialized near the top of pyXspecmodule.cxx. The same array of functions will be implemented on the client side.

Since the client may communicate with more than one instance of XSPEC, the desired instance is first selected with the global function SetCurrentXspec. All the global functions apply to the currently selected instance. This implementation makes it possible to support PyXspec on the client side. A similar C++ level interface will be created for use by ATSAL.

The client code will be packaged as a library for use by any application.

28.1.3 XSPEC Supervisor

The XSPEC supervisor is a background process that is launched at boot time. There is a single copy per computer, regardless of the number of XSPEC

instantiations. It listens on a fixed port number for requests from ATSAL processes. It supports four request types:

- An **instantiation request**, once it is verified as legitimate, causes instantiation of a new XSPEC process and assignment of unique port number(s).
- A **shutdown request** may be normal or mandatory. A normal request performs a structured exit by calling XSPEC's function to shut down. A mandatory request begins with a structured exit, but follows with a SIGKILL to ensure that the process is terminated.
- A **status request** returns information on whether XSPEC is idle, compute-bound, or hung.
- An **interruption request** causes the supervisor to deliver a SIGINT signal to XSPEC, to abort a long-running analysis.

If the supervisor is shut down, all XSPECs under it are also shut down.

The XSPEC supervisor is installed or uninstalled on a given machine as part of the XSPEC server installation. A simple app sets the maximum number of XSPEC instantiations that may be activated at the same time, and perhaps fine tunes the priority of the XSPEC processes.

28.1.3.1 Validation

We will use a simple password-based validation scheme for instantiating and accessing XSPEC servers. Although an attacker could perhaps commandeer a copy of XSPEC server and use it to disrupt a server machine's environment, XSPEC is so specialized and will run on so few machines that an attacker would choose a more easily exploited target. "Security through obscurity."

29 Preferences and Notebook Settings

Preferences are settings that apply to all notebooks on a given user's computer. Notebook settings apply to a particular notebook.

29.1 Preferences

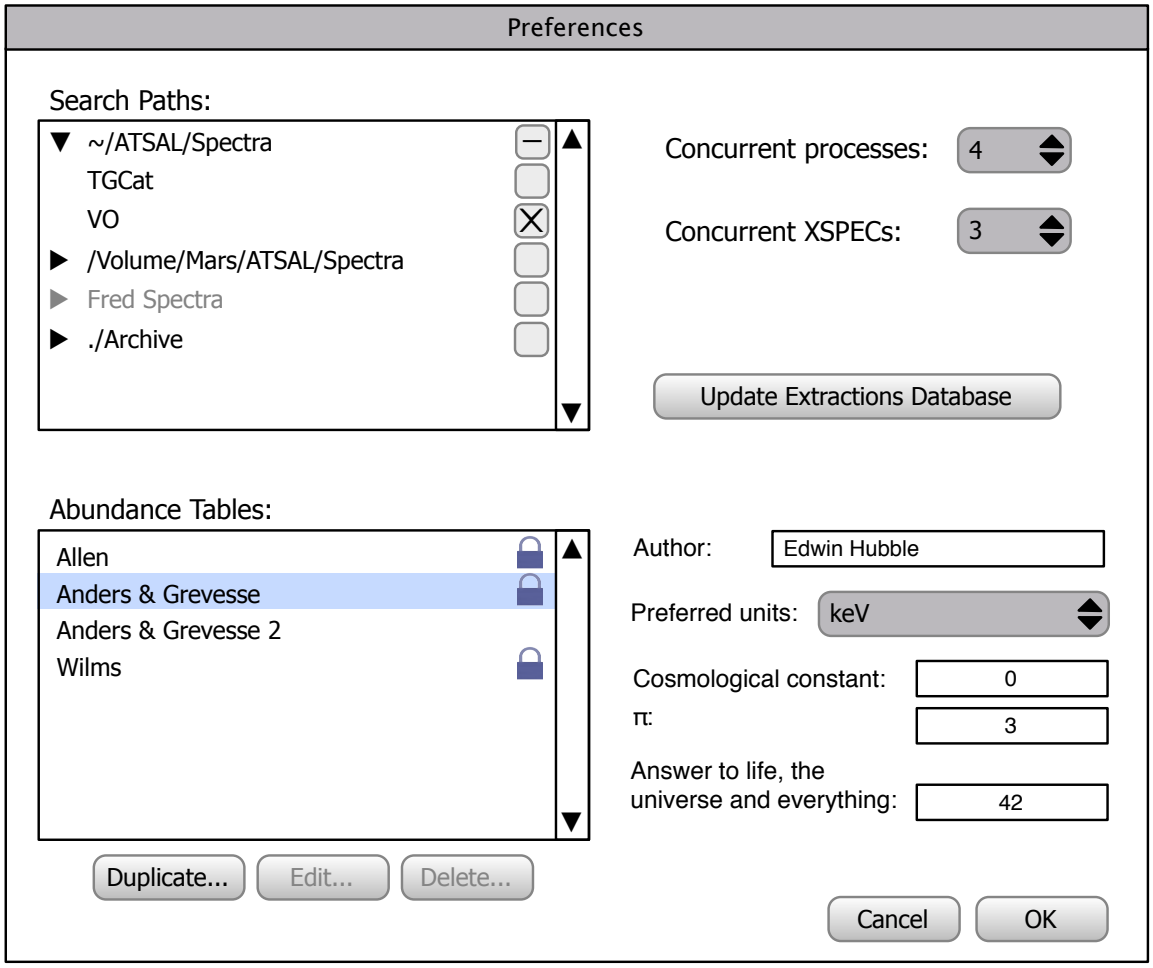


Figure 29-1. Preferences dialog

Search paths specify the locations on local and mounted volumes that will be searched for extraction files. Update Extractions Database updates a MySQL database with data accessible via any global search paths.

The concurrent options specify the maximum number of background processes or XSPEC instantiations that may be started during a session.

The Abundance Tables lists standard tables supplied with ATSAL, as well as user tables. Standard tables may not be edited or deleted, but they can be saved under new names and edited as needed. Double-clicking an editable table, such as "Anders & Grevesse 2" above, displays the elements and their abundances for editing. These tables may be copied into the parameters of models that accept element abundances.

Author is an example of settings that describe the user, and might also include e-mail address and other contact info. This information is available to the notebook. "Preferred units" selects among Ångstroms or keV; other default units selections would also appear here.

The next section involves settings for other parameters that have global scope.

Preference settings are available to Python programs via the singleton preferences object, `preferences`, of class Preferences.

29.2 Notebook Settings

This section details settings that apply to a given notebook.

- Energies command
- Notebook-specific search paths

They are accessible to Python programs via the instance `notebook`, which represents the current notebook. (Python programs cannot reference objects associated with other notebooks that are open at the same time.)

30 Risks

ATSAL's design is fairly complex. This describes major risks, mitigation strategies, and justifications.

30.1 Mulitple Software Dependencies

ATSAL depends on a large array of software components, most very complex in their own right, and many developed without any overarching architectural uniformity. Virtually all are on independent development schedules. This results in testing issues, overall software reliability issues, and interoperability issues.

Mitigation. ATSAL will reduce dependency by restricting external dependencies as much as possible, and by adding ATSAL-level testing on top of testing performed by suppliers of external software. However, software components that are not heavily used by ATSAL itself will not receive additional testing, nor will they receive the same priority for support issues.

30.2 Version Skew

There are two forms of version skew that ATSAL attempts to address:

- Skew due to combining different releases of external components.
- Skew that results from sharing notebooks or notebook archives among ATSAL users who are running different versions of ATSAL.

30.2.1 External Skew

Ideally, a given release of ATSAL will support only a specific release of each component upon which it depends. However, because users may need to install other versions of some of these components, this probably isn't possible.

Mitigation. Where possible, we will address external skew issues by packaging dependent components with ATSAL and building it with static libraries, so that skew is minimized. This is not a complete solution, however.

30.2.2 ATSAL Skew

Because ATSAL notebooks can include user-developed Python libraries, ATSAL cannot promise compatibility across releases, and the sharing of Python libraries is potentially complex.

Mitigation. This is dealt with in the chapter on version skew.

30.3 XSPEC State Synchronization

ATSAL maintains a representation of XSPEC's state in order to issue the minimum set of commands needed to have XSPEC update the user interface. It is possible that subtleties in XSPEC's internal state exist that might make such a representation difficult to maintain. For example, some error conditions might leave XSPEC in a state that differs from the assumption made by ATSAL.

Mitigation. The mitigation strategy is to describe the state representation in enough detail so that XSPEC's developers can help to identify potential weak spots.

30.4 XSPEC as an Independent Process

ATSAL's communication with XSPEC as an independent process has benefits listed at the beginning of this document, but it also has notable drawbacks:

- It assumes that the C++ interface exposed to PyXspec, the Python library for accessing XSPEC, can access all necessary information (commands, results, out-of-band errors, progress information) to allow for remote operation by ATSAL.

  Mitigation. Work with XSPEC's developers to identify any areas where necessary access is lacking, since limitations that apply to ATSAL probably apply to PyXspec as well.

- It gives up direct access to XSPEC or its underlying libraries, on the assumption they are not needed at the ATSAL level—a possible performance penalty.

  Mitigation. No significant performance penalties have been found at this time. If necessary, additional external libraries could be linked into ATSAL.

- To add a new feature to XSPEC, it must be (1) implemented in XSPEC; (2) implemented in PyXspec; (3) supported by the PyXspec driver; (4) supported by the ATSAL driver; (5) supported by the C++ layer of ATSAL; (6) surfaced to the Python layer of ATSAL and documented. This process is not technically difficult, but labor-intensive.

  Mitigation. The extension mechanism will be well documented, reducing time necessary for adding features. C++ is necessary in ATSAL for many performance-sensitive components, and for better (Qt-based) user interface tools. Python is an excellent language for end users, one that already enjoys wide popularity in the astrophysics community. Hence the extra effort involved in supporting a dual-language ATSAL layer is justified.

30.5 Discontinuation of External Software

All of the components upon which ATSAL depends are open source software, so discontinuation of any component does not prevent us from maintaining the software ourselves. However, many of these components are complex enough to make maintenance costs prohibitive. Hence ATSAL restricts its use to the tools that are critically important and presently perceived to have long term viability in the software market.

In the following table, entries in boldface are critical dependencies: events that force these components to be updated will probably force ATSAL to incorporate the updates as well.

Component	*Mitigation*
XSPEC	Extremely successful, and a core component of other tools; no risk.
Qt	Qt development began in 1991. It is widely regarded as the most complete, mature open source cross-platform development library available. It provides an abstraction layer that protects against both platform-specific variations as well as changes within a platform (for example, Apple's change from Carbon to Cocoa user interface libraries). From this perspective, Qt offers risk reduction. On the other hand, stewardship of Qt has passed through several hands in recent years, and increasing focus on mobile apps in all software development markets may impact Qt's future quality. But Qt is in effect the only viable package available with suitable flexibility, quality, and performance.[7] New releases of Qt will be adopted by ATSAL updates only when necessary to support new desktop/laptop hardware or major perturbations in vendor software.
PySide	PySide is a binding generator that makes Qt C++ classes available to Python. ATSAL will use the version of PySide that is consistent with a given release of Qt.
AstroPy (including NumPy, SciPy, and Matplotlib)	AstroPy enjoys an enthusiastic and vocal user base. Early versions of ATSAL probably won't have strict dependencies on AstroPy though, since this is intended for user extension. Hence this will be included as an option in early releases.
Observation archives	Archives available online (TGCat, VO, etc.) are expected to be in continuous flux. ATSAL will track changes to web interfaces, but it will also be possible to bypass ATSAL for obtaining extractions if needed while waiting for ATSAL to support newer interfaces.
Atomic databases	Unless it is easily accomplished, ATSAL may not track atomic database updates promptly.
Python	Each release of ATSAL will be close-coupled with a specific Python release. ATSAL support of newer Python releases may lag significantly. This will not prevent use of newer Python releases for other purposes, however.
Web technologies	ATSAL uses web technologies mostly for documentation, and will keep web use relatively simple to avoid compatibility issues.

Table 30-1. Software dependency risks

Tracking changes in dependent software is unavoidable, because the platforms themselves are changing so quickly. Here are some changes that have occurred within the last ten years:

- Computer languages continue to increase in variety and design criteria. We chose C++ as the most mature, performance-sensitive, cross-platform language.

[7] ATSAL could be restricted to a single platform to reduce development costs, but if that platform is Linux, Qt would be used anyway; if Mac, the native API could be used. Native APIs are in continuous flux, though, and Qt provides insulation against this.

- The industry has transitioned from 32-bit to 64-bit architectures. Qt helps to insulate against this change.
- Apple completely revamped their user interface toolkits, replacing a C-based package with a very different Objective-C-based package. On a prior project, Qt completely insulated against this change.
- Apple switched from PowerPC to Intel hardware architectures. Apple made this change almost invisible to developers, except at the driver level.
- Increased attention to security has altered the software landscape in a variety of ways. ATSAL will have to continue to adapt to changing security constraints.
- The popularity of mobile computing is creating backpressure to make desktop and laptop computing environments look more similar to their mobile counterparts. ATSAL stresses continuity, not changing user interface styles, except where the latter truly add value.
- Tools for web development remain relatively poor, restricted by compatibility issues, slow progress on web standards, and the stateless model of application development. Steady changes here lead to version skew problems. ATSAL's use of web technologies will be "lowest common denominator."

31 Development Issues

31.1 Development Systems

Since ATSAL runs on Linux and Macintosh systems, at least one developer will need to work on each of these systems.

Unit tests can be run on test or developer systems initially. If the test suites become long-running, we may dedicate a machine to this purpose.

XCode, Apple's integrated development environment, will be employed for Macintosh development. Under Linux, the Qt-supplied development environment will be used.

[THK: Need to select two Linux flavors that we formally support.]

31.2 Source Code Management

http://stackoverflow.com/questions/5507489/git-server-like-github

ATSAL source trees will be stored in git archives. Git archives implement the entire archive on local development systems as well as a central host. Because of this duplication on each client system, any other archive can serve as the "system of record" in the event the central host becomes unavailable. This provides distributed backup and protection from a third party host that fails for some reason.

31.3 HEASARC Standards

ATSAL will probably be distributed via HEASARC, so it must conform to the standards in use at HEASARC for building and distributing software.

This means:

- Builds that are compatible with compilers used for other HEASARC components.
- Conformance to acceptable levels of spurious warnings during builds.
- Conformance to HEASARC standards for documentation preparation and distribution.
- Use of the standard configure/make/install mechanism for building applications.
- Testing on HEASARC-supported systems.
- Packaging of source trees as optional components of HEASARC downloads.

Early releases of ATSAL will be done directly from the Harvard-Smithsonian Center for Astrophysics. Integration with HEASARC distribution will probably take place at the first formal release of ATSAL.

31.4 Testing

ATSAL will employ two basic test methodologies: unit testing and Squish testing. Units tests employ a test framework that is part of Qt, providing for a uniform approach. Unit tests will run as a part of each nightly build. This mechanism permits testing of some user interface features in addition to "data layer" testing.

http://qt-project.org/doc/qt-4.8/qtestlib-tutorial1.html

Squish, a third party test facility designed for Qt applications, provides a framework for more sophisticated user interface testing. These tests exercise the user interface by simulating user interaction and comparing windows against saved window snapshots. These tests are more comprehensive, since they pick up even subtle changes in the presentation of data to the user.

Both Qt unit testing and Squish are capable of driving the user interface directly, so the choice of how to implement a particular test is somewhat arbitrary. However, tests that involve window comparison fail as a result of even trivial adjustments to layout, precision, graphing, etc. When window comparison-based tests fail due to changes, testers must examine the changes manually to determine whether the newer result is acceptable. This tends to hamper the progress of user interface refinements. Also, these tests can consume a great deal of elapsed time, since they proceed at a relatively slow rate.

Hence the following strategy is proposed for testing:

- Testers and developers will work together to develop unit tests that can run quickly and are relatively independent of user interface refinements. These tests will be run at regular intervals, perhaps daily, so that developers are quickly alerted to failures.
- Testers will develop and debug Squish user interface tests for various windows and dialogs, but these will not immediately become part of the daily test regimen. Instead, when a particular user interface component is considered relatively stable, Squish tests for that component will be enabled. This allows more volatile components to continue to evolve quickly.
- If the time required to run Squish tests becomes excessive, we will dedicate one or more computers to performing these tests.

Testers will develop a means of notifying developers of test failures.

31.5 Distribution

ATSAL will be distributed in binary and source form for Macintosh users, and in source form for Linux users.

On either platform, authorized users will be able to access git archives for source code development or review.

31.6 Bug Tracking

Testers will evaluate and select a web-based bug tracking mechanism to be used throughout the project. Testers will also identify a strategy for registering and handling user bug reports.

Bugs are assigned a priority of low, medium, or high. Feature requests are also registered using this mechanism, and marked as such.

31.7 Release Policy

There are three types of releases:

- A major release occurs approximately once per year, and offers significant new features. Major releases are numbered 1.0, 2.0, ... After a major release, the source tree is branched into maintenance and new development branches.
- A minor release contains high priority bug fixes for the current release. Release 1.1, for example, lacks new features but corrects major bugs that have surfaced since release 1.0.
- A test release, e.g. 1.1.7, is not guaranteed to be stable or even fully tested, but contains features some users will wish to evaluate.

31.8 Coding Standard

The coding standard for this project is described in the *ATSAL Coding Standard*.

31.9 Multilingual Support

ATSAL will employ a Qt-supported mechanism that represents strings in a manner that may be later used to facilitate support for multiple human languages. Early releases will be English only.