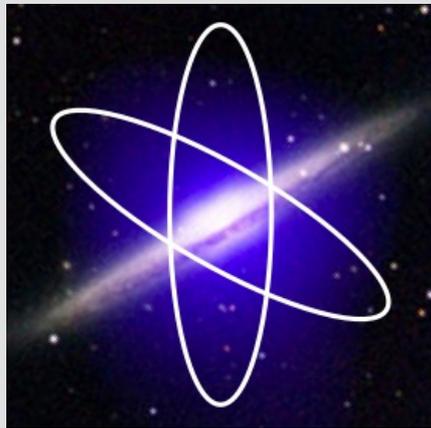


# ATSAL Developer Help



**Harvard-Smithsonian  
Center for Astrophysics**

ATSAL v0.1.7  
Generated on: 2017-05-31  
Build date: 2017-04-23T15:49:38Z  
Git SHA: 9c9c28bcad  
Git commit: 2017-04-23T15:36:03Z

## Table of Contents

Table of Contents	2
1 ATSAL	17
1.1 Developer Help	17
1.2 ATSAL Build Directory	17
1.3 Build Process	19
1.3.1 Mac Debug Builds	19
1.3.2 Mac Release Builds	19
1.3.2.1 qmake	20
1.3.3 atsal/private	20
1.3.4 atsal/proj	20
1.3.4.1 atsal/proj/atsal	21
1.3.4.1.1 Mac Deployment Command Line Build	22
1.3.4.1.1.1 macdylibbundler	23
1.3.4.1.1.1.1 About	23
1.3.4.1.1.1.2 Installation	23
1.3.4.1.1.1.3 Feedback/Contact	23
1.3.4.1.1.1.4 Using dylibbundler on the terminal	23
1.3.4.1.1.2 autoconf	24
1.3.4.1.1.2.1 Autoconf and Automake on Mac OS X Mountain Lion	25
1.4 XCode	25
1.4.1 XCode and Time Machine	25
1.4.2 Mac Icons	26
1.4.3 Mac Debugging	27
2 Development and Coding Conventions	34
2.1 Portability Issues	34
2.2 ATSAL Naming Conventions	34
2.3 Declarations Format	34
2.4 Maintainability	35
2.5 Standards	36
2.6 Documentation	36
2.7 Git Checkins	36
2.8 Python	37
2.8.1 Python Naming Conventions	37
2.8.2 Surfacing ATSAL to Python	37
2.8.3 Python Source Code	37
2.8.4 Documentation of ATSAL-surfaced Python	37
2.8.5 In summary:	37
2.9 C++11 Features	37
2.9.1 __cplusplus	38
2.9.2 auto—deduction of a type from an initializer	38
2.9.3 Range-for statement	39
2.9.4 Right-angle brackets	39
2.9.5 Control of defaults: default and delete	39
2.9.6 control of defaults: move and copy	40
2.9.7 Enum class—scoped and strongly typed enums	41
2.9.8 constexpr—generalized and guaranteed constant expressions	42
2.9.9 decltype—the type of an expression	44
2.9.10 Initializer lists	44
2.9.11 Preventing narrowing	45
2.9.12 Delegating constructors	46
2.9.13 In-class member initializers	47
2.9.14 Inherited constructors	48

2.9.15 Static (compile-time) assertions—static_assert	49
2.9.16 long long—a longer integer	50
2.9.17 nullptr—a null pointer literal	50
2.9.18 Suffix return type syntax	50
2.9.19 template alias (formerly known as "template typedef")	51
2.9.20 Variadic Templates	52
2.9.21 Uniform initialization syntax and semantics	55
2.9.22 Rvalue references	55
2.9.23 unions (generalized)	57
2.9.24 PODs (generalized)	58
2.9.25 Raw string literals	59
2.9.26 User-defined literals	60
2.9.27 Attributes	61
2.9.28 Lambdas	62
2.9.29 Local types as template arguments	63
2.9.30 noexcept—preventing exception propagation	64
2.9.31 Alignment	65
2.9.32 Override controls: override	65
2.9.33 Override controls: final	66
2.9.34 C99 features	67
2.9.35 Extended integer types	67
2.9.36 Dynamic Initialization and Destruction with Concurrency	67
2.9.37 Thread-local storage	67
2.9.38 Unicode characters	67
2.9.39 Copying and Rethrowing Exceptions	68
2.10 Extern templates	68
2.10.1 Inline namespace	68
2.10.2 Explicit conversion operators	69
2.10.3 Algorithms Improvements	70
2.10.4 Container Improvements	71
2.10.5 Scoped Allocators	73
2.10.6 std::array	74
2.10.7 std::forward_list	75
2.10.8 Unordered Containers	75
2.10.9 std::tuple	76
2.10.10 Metaprogramming and Type Traits	77
2.10.11 std::function and std::bind	77
2.10.12 unique_ptr	78
2.10.13 shared_ptr	79
2.10.14 weak_ptr	80
2.10.15 Garbage Collection ABI	81
2.10.16 Memory Model	82
2.10.17 Threads	83
2.10.18 Mutual Exclusion	86
2.10.19 Locks	87
2.10.20 Condition Variables	89
2.10.21 Time Utilities	89
2.10.22 Atomics	91
2.10.23 std::future and std::promise	91
2.10.24 std::async()	93
2.10.25 Abandoning a Process	93
2.10.26 Random Number Generation	93
2.10.27 Regular Expressions	96
2.10.28 Concepts	96

2.10.29 Concept Maps	97
2.10.30 Axioms	97
3 ATSAAL Classes	99
3.1 Qt Cross-platform Class Library	99
Qt Main Page	99
Qt Complete Class List	99
3.1.1 Qt Main Classes	99
3.1.2 Qt Classes Used Often by ATSAAL	100
QMake Manual	102
Deploying on Mac OS X	102
3.1.3 Qt Add-ons	102
MatPlotLib	102
4 Matplotlib Notes	104
4.1.1.1 Matplotlib and PythonQt	104
4.1.1.2 MeVis Matplotlib Files	104
4.2 FITS Files	107
4.2.1 OGIP Spectral FITS Format	107
4.2.1.1 Notes	107
FITS File I/O (cfitsio)	107
4.2.2 PythonQt Usage Notes	108
4.2.2.1 Publishing Enums	108
4.2.2.2 Publishing Optional Arguments	108
XMLRPC	108
4.3 EPSEngine	108
EPSEngine Web Site	109
4.4 Lossless File Compression	109
4.4.1 ZSTD	109
4.4.2 LZ4	109
4.4.3 Apple LZFS	109
Libcurl	109
5 Configuring a Development Environment	110
5.1 Setting Up a Mac for Development	110
5.1.1 XCode	110
5.1.2 Text Editors	110
5.1.3 Qt	110
5.1.4 Obtain ATSAAL	110
5.1.5 Setting up your Startup File	110
5.1.6 Run the Dev Installer	111
5.1.7 Mac Dev Installer Design	112
5.1.7.1 The private Archive	112
5.1.7.2 Nexus	112
5.1.7.3 README	112
5.1.7.4 macdevinstall	112
5.1.8 Virtual Machines (Deprecated)	112
5.1.8.1 Virtual Environment Manager	112
5.1.8.1.1 Parallels Desktop (Mac)	112
5.1.8.1.2 VMWare Fusion (Mac)	113
5.2 Setting Up for Linux Development	113
5.3 Software Components	113
5.3.1 Software Component Summary	114
5.4 Development Environment	114
5.4.1 XCode (Mac)	115
5.4.2 TBD (Linux)	115

5.5 X Window System	115
5.6 Editors	115
5.6.1 Macintosh Editors and Drawing Tools	115
5.6.1.1 Microsoft Office	115
5.6.1.2 OmniGraffle Pro	115
5.6.1.3 BBEEdit	115
5.6.2 Linux Tools	115
5.7 Homebrew	115
5.8 gdb	116
5.9 Qt	116
5.10 Python 3	117
5.10.1 Python on AT&T Development Machine	117
5.10.2 Python and PyQt Build Notes	117
5.11 CMake	118
5.12 wget	118
5.13 setuptools	118
5.14 pip	118
5.15 virtualenv	118
5.15.1.1 Restricting pip to virtual environments	119
5.15.1.2 Creating virtual environments	120
5.16 socketexec	120
ATOMDB: Atomic Data for Astrophysicists	121
FITS File I/O (cfitsio)	121
FITS Utilities	121
5.16.1 FITS File Notes	121
5.16.1.1 PHA Files	121
5.16.1.2 RMF Files	121
5.17 The .bash_profile (or equivalent)	122
5.18 XSPEC Server	123
5.18.1 XSPEC Extensions	123
5.18.2 XMLRPC	123
5.18.2.1 Mac Build	124
5.18.2.2 Debugging the Server with XCode	124
5.18.2.3 Choosing XMLRPC Libraries	124
5.18.2.4 Bidirectional IPC	125
5.18.2.4.1 File Transfers	125
5.18.2.4.2 Timing	126
5.18.3 Debugging XSPEC Server with XCode	126
5.19 Defunct (But Interesting) Approaches	128
5.19.1 PyQt and Sip	128
5.19.1.1 SIP	128
5.19.1.2 PyQt	128
5.19.1.3 Python/Qt Superhybrids	128
5.19.1.3.1 Software Components	128
5.19.1.3.2 Python and Qt "Superhybrid"	129
6 Update Issues	135
6.1 XCode 6.1, 2014-11-14	135
6.2 Icons	135
7 Design Issues	136
7.1 Sidebars	136
7.1.1 Sidebars and Signals	136
7.2 A Process for Each Notebook	136
7.2.1 Quitting in a Multiple Process Environment	137

7.2.2 Database Regeneration	138
7.2.3 Other Multiple Process Notes	139
7.2.3.1 LSUIElement	139
7.2.3.2 Switching Seamlessly Among Processes	139
7.2.3.2.1 Mac OS X Solution	139
7.2.3.2.2 Linux Solution	140
7.2.3.3 Threads	140
7.2.3.3.1 The Current Thread	140
7.3 Version Skew	140
7.4 Responsiveness and Parallelism	140
7.4.1 Responsiveness	141
7.4.2 Large-scale Parallelism	141
7.4.2.1 Use Cases	141
7.4.2.2 Setup	142
7.4.2.3 Deployment	142
7.4.2.4 Remote Processing	142
7.4.2.5 Results Processing	142
7.5 User Interface	142
7.5.1 ATSAL User Interface Goals	142
7.5.1.1 ATSAL User Interface Non-Goals	143
7.6 Python Design Issues	143
7.6.1 Python Debugger Notes	143
Pdb Debugger	143
Bdb Low-level Debugger	143
Inspect	143
Basic Types (incl. Frame and Traceback)	143
IO Module	143
7.6.2 Python Editor	143
7.6.2.1 Tabs vs. Spaces	144
7.6.3 Python Reloading	144
7.6.4 Surfacing ATSAL to Python	144
7.6.4.1 Embedding Python into Qt Applications	144
7.6.4.1.1 The Benefits of Scripting	145
7.6.4.1.2 About PythonQt	145
7.6.4.1.3 Getting started	146
7.6.4.1.4 Creating an Application Scripting API	146
7.6.4.1.5 About Python	146
7.6.4.1.6 GUI Scripting	147
7.6.4.1.7 The PythonQt Module	148
7.6.4.1.8 Decorators and C++ Wrappers	148
7.6.4.1.9 Other Features	149
7.6.4.1.10 Future Directions	149
7.6.5 Making ATSAL Classes and Variables Available to Python	149
7.6.5.1 Python for Development and Testing	150
7.6.5.2 Registering Classes and Variables	152
7.7 XSPEC Allocator	153
7.7.1 Pictorial Summary	154
7.7.2 XProxies	154
7.7.3 XSPEC Versions	155
7.7.4 Projections	155
7.7.5 User Interface Locking	155
7.7.6 Running XSPEC on Multiple Computers	155
7.7.7 Orphaned Notebooks	156
7.7.8 Abolishing the Load-leveler	156
7.7.8.1 Addendum 2017-04-25	157

7.8 Client/Server Architecture	157
7.8.1 Long Term Architecture	157
7.8.2 Short Term Architecture	158
7.8.3 Socket Connection Refused Errors	158
7.8.3.1 QTBUG-6305	159
7.8.3.1.1 02/Dec/09 4:39 PM	159
7.8.3.1.2 15/Apr/10 8:52 PM - edited	159
7.8.3.1.3 04/Jun/11 5:36 AM	160
7.8.3.2 What exactly does SO_REUSEADDR do?	160
7.8.3.3 ATSA Solution	160
7.8.3.4 Additional Comments	161
7.9 Observatory Tool	161
7.9.1 Search Queries	161
7.9.1.1 Quick Search	161
7.9.1.2 Detailed Search	162
7.9.2 Missions, Instruments, and Proxies	163
7.9.2.1 SearchQuerys	164
7.9.2.2 Defaults	164
7.10 Results Tool	164
7.10.1 Results Processing	165
7.10.2 Python Access	166
7.10.3 Multiple TableWidgets	166
7.10.4 Saving and Restoring Results Tools	166
7.11 Table Tool	166
7.11.1 Tables and Python	166
7.11.1.1 TableRanges	167
7.11.2 Arrays	170
7.11.3 TableCellFormats	170
7.11.4 Selections	171
7.11.5 Table Tool Editor	171
7.11.5.1 Table Cell Edit vs. View Properties	171
7.11.5.2 Navigator Keys	173
7.12 Tag Tool	173
7.12.1 Tag Sidebar	173
7.12.2 Tag Classes	174
7.12.3 Tag Ownership Diagram	174
7.12.4 Tag Tool Redesign	175
7.12.4.1 Design and Implementation	177
7.12.4.1.1 Micro-glossary	177
7.12.4.1.2 User Interface	177
7.12.4.1.3 Handling Parameter Updates	178
7.13 Parsers, Datasets, Tables, and Python: A Refactoring	179
7.13.1 Parser Extensions	180
7.13.2 Datasets	180
7.13.3 The Results Tool	180
7.13.4 Blocks	181
7.13.5 Summary	183
7.14 Physical Data Types	183
7.14.1 Quantities	183
7.14.1.1 Physical Units	184
7.14.1.2 Reading and Writing Quantity Values	185
7.14.2 QuVariables	185
7.14.2.1 Variable Names	186
7.14.2.2 Cloning Variables	186

7.14.3 Tables and Table Cells	186
7.14.4 Subtracting Curves	186
7.14.4.1 MatchSubtrahendPoints	186
7.14.4.2 MatchMinuendPoints	187
7.14.4.3 Residuals	188
7.15 XML Files	188
7.16 AtomDB	188
7.16.1 Low-level AtomDB Classes	189
7.17 The Refresh Cycle	190
7.17.1 Refresh State Tables	196
7.17.1.1 Plot Tool	196
7.17.1.2 Aborting a Refresh	198
7.17.1.3 Normal Process Exit	198
7.17.1.4 XSPEC Crashes	198
7.17.1.5 Error Propagation	199
7.17.1.6 MulticolorLEDs	200
7.17.1.7 Modify States	200
7.17.2 Refresh Phases	201
7.17.3 Refresh Loop (Old)	202
7.17.3.1 Concurrency	203
7.17.3.1.1 Plot Tool Command Completion	203
7.17.3.2 Parallelism, One Step Finer	203
7.17.3.2.1 Projections	203
7.17.3.3 Keeping ATSAI Busy	204
7.17.4 Process Monitoring and Logging	204
7.17.4.1 Deprecated (Overly Complicated) Approach	205
7.17.5 Fits of Pique	205
7.17.5.1 Performance Impact	206
7.17.5.2 Fit Progress	206
7.18 XFits and XSPEC Models	206
7.18.1 Model Editor	207
7.18.2 ToolEditorModelParameter	207
7.18.3 XSPEC Initialization Parameters	207
7.18.4 Model Classes	208
7.18.4.1 Model Pointers	209
7.18.4.2 Saving/Restoring	209
7.18.5 Spectrum Plot Sidebar Classes	210
7.18.6 Default Parameters	210
7.18.6.1 Parameter Editors and Defaults	211
7.18.7 Parsing Model Expressions	212
7.18.7.1 Phase 1: Parsing Model Expressions	212
7.18.7.2 Phase 2: Preserving Parameters	212
7.18.7.2.1 Some Examples	213
7.18.7.2.1.1 Changing an Operator	213
7.18.7.2.1.2 Repeated Models	213
7.18.7.2.1.3 User Models Containing User Models	213
7.18.7.2.2 The Parameter Preservation Parse	214
7.18.8 Line-based Analysis Design	214
7.18.8.1 Using Line-based Analysis	214
7.18.8.1.1 Peak Finder Controls	218
7.18.8.1.2 Emission Line Details	220
7.18.8.1.3 Fine-tuning Parameters	221
7.18.8.1.4 User Annotations	222
7.18.8.2 Overall LBA Design	223
7.18.8.2.1 LBA Dependency Tree	224

7.18.8.2.1.1 Line-based Analysis Classes	224
7.18.8.2.1.2 Line-based Analysis Ownership Graph	224
7.18.8.2.1.3 Phase 1: EmissionLinePeaks	225
7.18.8.2.1.4 Phase 2: EmissionLineMatches	225
7.18.8.2.1.5 Phase 3: AnchoredAnnotations	227
7.18.8.2.1.6 Phase 4: AnchoredLabels	228
7.18.8.2.1.7 Label Geometry	228
7.18.8.2.1.8 Panning and Zooming	231
7.18.8.2.1.9 User Annotations	231
7.18.8.2.1.10 Mouse Input	232
7.18.8.2.1.11 Selections	233
7.18.8.2.2 Calculating and Fine-tuning Gaussians	233
7.18.8.2.2.1 User Models and Line-based Analysis	233
7.18.8.2.2.2 Model Handling	234
7.18.8.2.2.2.1 Save/restore	235
7.18.8.2.2.3 Parameter Editing	235
7.18.8.3 Deprecated Line-based Analysis Design	236
7.18.8.3.1 Line-based Analysis	236
7.18.8.3.1.1 Merging	237
7.18.8.3.1.2 The Functions	237
7.18.8.3.1.3 Other Issues	237
7.18.8.3.1.4 Meeting 2016-05-03	237
7.18.8.3.1.4.1 Root Peak Finder	237
7.18.8.3.2 LBA States and Command Processing (Deprecated)	238
7.19 Persistent Pointers	241
7.20 Plots	242
7.20.1 Coordinate Systems	242
7.20.2 Monitor Resolution and Screen Size	242
Qt and High-resolution Displays	243
7.20.3 Zone Synchronization	244
7.20.4 Resize Protocol	244
7.20.4.1 Those Damned Tick Marks!	245
7.20.5 Plot Classes	245
7.20.5.1 Plot Class Hierarchy	245
7.20.5.2 Plot Tool Ownership Graph	246
7.20.5.3 Plot Widget Placement	247
7.20.6 Spectra, zones, and layers	248
7.20.7 Selections and the Selection Bar	248
7.20.7.1 Update, 2016-02-10	249
7.20.7.1.1 Ignore Enable/Disable	250
7.20.7.1.2 Storing Selection Sets	250
7.20.7.2 Selections and Channels	251
7.20.7.3 Ignore Regions	252
7.20.7.3.1 Include Regions Design Refinement (2017-03-18)	253
7.20.8 Plot Sidebar	253
7.20.8.1 Layers Section	253
7.20.8.2 Layers Section with Tag Groups	254
7.20.9 Plot Tool Use Cases	255
7.20.9.1 Analytical Mode	255
7.20.9.2 Annotation Mode	257
7.20.9.3 Publication Mode	258
7.20.9.4 Meeting, Oct. 14, 2015	258
7.20.9.4.1 More Thoughts on Large-scale Computing	258
7.20.9.4.2 Spectrum Plot Tool Walk-Through	259
7.20.9.4.2.1 Analysis Mode	259
7.20.9.4.2.2 Annotation Mode	261
7.20.9.4.2.3 Publication Mode	261

7.20.9.4.2.4 Questions	261
7.20.9.4.3 ... One more thing ...	262
7.20.10 Plot Types, Specialized	262
7.20.10.1 Types and Priorities	262
7.20.10.2 Fit Sub-plots	263
7.20.11 Channels, Zones, and Layers	264
7.20.11.1 Channels	264
7.20.11.2 Plot Layers and Zones	265
7.20.11.3 Isolationism	265
7.20.11.3.1 Show/hide and Isolate/deisolate Controls	266
7.20.11.3.1.1 Layer Controls	266
7.20.11.3.1.2 Spectrum Controls	266
7.20.11.3.1.3 Models	266
7.20.11.3.1.4 Plottables	266
7.20.11.3.1.5 Channel Display	267
7.20.11.3.2 Implementation	267
7.21 Qt Trivia	267
7.21.1 Copying QObjects	267
7.21.2 QSharedPointer	268
7.21.2.1 Initialization	268
7.21.2.2 Assignment	268
7.21.2.3 Propagation	268
7.21.2.4 Efficiency	269
7.21.3 Installing Event Filters	269
7.21.4 Qt and Threads ... and Sockets	270
7.21.4.1 QThreads and Clients	270
7.21.4.2 Controllers	271
7.21.4.3 Sockets	271
7.22 Version Numbering	272
7.23 Resource Files	272
7.23.1 Qt Resource Compiler	273
7.23.2 ATSAL Resource Manager	273
8 Documentation Tools	274
8.1 Nexus Document Tree Manager	274
8.1.1 Editing Help Files	274
8.1.1.1 The HTML Pane	275
8.1.2 Searching	276
8.1.2.1 Simultaneous Web and HTML Views	276
8.1.2.2 Remote Searching	276
8.1.2.3 View vs. Edit Mode	276
8.1.2.4 Local vs. Global Searches	277
8.1.2.5 Replacing	277
8.1.3 Tables	277
8.1.3.1 Convert tab-separated text	278
8.1.3.2 Change existing table/td/th commands	278
8.1.3.3 Strip existing table commands	279
8.1.3.4 Change all <td ...> (or <th ...>) commands	279
8.1.3.5 Convert one-per-line item list	279
8.1.3.6 Convert multicolumn table back to one-per-line item list	280
8.1.4 Equations	280
8.1.5 Style Control	280
8.1.6 Printing	280
8.1.7 Exporting the Help Tree	280
8.1.7.1 Exporting a PDF Book with Active Hyperlinks	281
8.1.7.2 Exporting a Complete Web Site	281

8.1.7.3	Generating a Concatenated HTML File	281
8.1.7.4	Support Files	281
8.1.7.5	Doxygen	282
8.1.7.6	HTML Blocks	282
8.1.8	Internal Notes	283
8.1.8.1	File Organization	283
8.1.8.2	ATSAL Build Process	283
8.1.8.3	To Do	283
8.1.9	Nexus Futures	283
8.2	ATSAL Web Site	283
8.3	Doxygen	284
	Doxygen Web Site	284
	MathJax	284
9	Python Subsystem	285
9.1	Build Issues	285
9.1.1	Python	285
9.1.2	PythonQt	285
	Python Web Site	285
	PythonQt Web Site	285
	PySide Web Site	285
	PyQt Web Site	286
10	Python ATLAS Library	287
10.1	The ATLAS Application	287
10.2	Notebook Class	287
10.2.1.1	XSpec Initialization Functions	287
10.3	Tools	288
10.3.1	Tool Class	288
10.3.1.1	ToolObservatory Class	288
10.3.1.2	ToolTag Class	289
10.3.1.3	ToolPlot Class	289
10.3.1.4	ToolPython Class	289
10.3.1.5	ToolMatplotlib Class	289
10.3.1.6	ToolResults Class	289
10.3.1.7	ToolText Class	289
10.3.1.8	exportAsHTML(path) Exports the text in HTML format.	289
10.3.1.9	ToolTable Class	289
10.3.2	Plot Tool	289
10.3.2.1	XSpectrum Class	290
10.3.2.2	XFit Class	290
10.3.2.3	XModelParamInstance Class	290
10.3.2.4	FileTag Class	291
10.4	Tables	291
10.4.1	TableRange Class	292
10.4.2	Table Class	294
10.4.3	TableCell Class	295
11	XSPEC Server	296
	XSPEC	296
	Frequently Asked Questions	296
11.1	Long Term Architecture	296
11.2	Short Term Architecture	297
11.3	Server Notes	297
11.4	ATSAL's XSPEC Command Parser	298
11.4.1	XSpec Parameters	301

11.4.2 Regular Expressions	302
11.4.3 Arrays	302
11.4.4 Skipping Lines	303
11.4.5 The Parserator	303
11.5 XSPEC Commands	305
11.5.1 Description of Syntax	305
11.5.2 Control Commands	306
11.5.2.1.1 autosave: set frequency of saving commands	306
11.5.2.1.2 chatter: set verbosity level	306
11.5.2.1.3 exit, quit: exit program	307
11.5.2.1.4 help: display manual or help for a specific command/theoretical model component	
11.5.2.1.5 log: log the session output	308307
11.5.2.1.6 parallel: enable parallel processing for particular tasks in XSPEC	308
11.5.2.1.7 query: set a default answer for prompts in scripts	309
11.5.2.1.8 save: save the current session commands	309
11.5.2.1.8.1 script: write commands to a script file	310
11.5.2.1.9 show: output current program state	310
11.5.2.1.10 syscall: execute a shell command	311
11.5.2.1.11 tclout: create tcl variables from current state	311
11.5.2.1.12 tclout: tclout with return value	315
11.5.2.1.13 time: print execution time	315
11.5.2.1.14 undo: undo the previous command	315
11.5.2.1.15 version: print the version string	316
11.5.3 Data Commands	316
11.5.3.1.1 arf: change the efficiency file for a given response	316
11.5.3.1.2 backgrnd: change the background file for a given spectrum	316
11.5.3.1.3 corfile: change the correction file for a given spectrum	317
11.5.3.1.4 cornorm: change the normalization of the correction file	318
11.5.3.1.5 data: read data, background, and responses	318
11.5.3.1.5.1 Spectrum numbering	319
11.5.3.1.5.2 Data groups	320
11.5.3.1.6 diagrsp: set a 'perfect' response for a spectrum	321
11.5.3.1.7 fakeit: simulate observations of theoretical models	322
11.5.3.1.7.1 Statistical Issues	322
11.5.3.1.7.2 Type I vs. Type II Output	322
11.5.3.1.7.3 Note on Grouped Spectra	323
11.5.3.1.7.4 Note for SPI/Integral Format	323
11.5.3.1.7.4.1 Type I files	324
11.5.3.1.7.4.2 Type II Files	324
11.5.3.1.8 ignore: ignore detector channels	325
11.5.3.1.9 notice: notice data channels	326
11.5.3.1.10 response: change the detector response for a spectrum	326
11.5.4 Fit Commands	327
11.5.4.1.1 bayes: set up for Bayesian inference	327
11.5.4.1.2 chain: run a Monte Carlo Markov Chain.	328
11.5.4.1.3 error, uncertain: determine confidence intervals of a fit	330
11.5.4.1.4 fit: fit data	331
11.5.4.1.5 freeze: set parameters as fixed	332
11.5.4.1.6 ftest: calculate the F-statistic from two chi-square values	332
11.5.4.1.7 goodness: perform a goodness of fit Monte-Carlo simulation	333
11.5.4.1.8 margin: MCMC probability distribution.	333
11.5.4.1.9 renorm: renormalize model to minimize statistic with current parameters	334
11.5.4.1.10 steppar: generate the statistic "surface" for 1 or more parameters	334
11.5.4.1.11 thaw: allow fixed parameters to vary	335
11.5.4.1.12 weight: change weighting used in computing statistic	335

11.5.5 Model Commands	336
11.5.5.1.1 addcomp: add component to a model	336
11.5.5.1.2 addline: add spectral lines to a model	337
11.5.5.1.3 delcomp: delete a model component	337
11.5.5.1.4 dummyrsp: create and assign dummy response	338
11.5.5.1.5 editmod: edit a model component	339
11.5.5.1.6 energies: specify new energy binning for model fluxes	340
11.5.5.1.7 eqwidth: determine equivalent width	341
11.5.5.1.8 flux: calculate fluxes	342
11.5.5.1.9 gain: modify a response file gain	343
11.5.5.1.9.1 Historical Notes	345
11.5.5.1.10 identify: identify spectral lines	345
11.5.5.1.11 initpackage: initialize a package of local models	346
11.5.5.1.12 lmod, localmodel: load a package of local models	347
11.5.5.1.13 lumin: calculate luminosities	347
11.5.5.1.14 mdefine: Define a simple model using an arithmetic expression	348
11.5.5.1.14.1 Operators	348
11.5.5.1.14.2 Functions	348
11.5.5.1.14.3 Proposed Unified Syntax	349
11.5.5.1.15 model: define a theoretical model	351
11.5.5.1.15.1 Syntax Rules	352
11.5.5.1.15.2 Examples	353
11.5.5.1.16 modid: write out possible IDs for lines in the model.	354
11.5.5.1.17 newpar: change parameter values	354
11.5.5.1.17.1 Parameter Links	355
11.5.5.1.17.2 Functions	356
11.5.5.1.18 systematic: add a model-dependent systematic term to the variance	357
11.5.5.1.19 untie/runtie: unlink previously linked parameters	357
11.5.6 Plot Commands	358
11.5.6.1.1 cpd: set current plotting device	358
11.5.6.1.1.1 PGPLOT devices	358
11.5.6.1.2 hardcopy: print plot	359
11.5.6.1.3 iplot: make a plot, and leave XSPEC in interactive plotting mode	359
11.5.6.1.4 plot: make a plot	359
11.5.6.1.4.1 background	360
11.5.6.1.4.2 chain	360
11.5.6.1.4.3 chisq	360
11.5.6.1.4.4 contour	360
11.5.6.1.4.5 counts	361
11.5.6.1.4.6 data	361
11.5.6.1.4.7 delchi	361
11.5.6.1.4.8 dem	361
11.5.6.1.4.9 eemodel	362
11.5.6.1.4.10 eeufspec	362
11.5.6.1.4.11 efficien	362
11.5.6.1.4.12 emodel	362
11.5.6.1.4.13 eqw	362
11.5.6.1.4.14 eufspec	362
11.5.6.1.4.15 goodness	362
11.5.6.1.4.16 icounts	362
11.5.6.1.4.17 insensitiv	362
11.5.6.1.4.18 lcounts	362
11.5.6.1.4.19 ldata	363
11.5.6.1.4.20 margin	363
11.5.6.1.4.21 model, emodel, eemodel	363
11.5.6.1.4.22 ratio	363

11.5.6.1.4.23 residuals	363
11.5.6.1.4.24 sensitvty	363
11.5.6.1.4.25 sum	363
11.5.6.1.4.26 ufspec, eufspec, eeufspec	363
11.5.6.1.5 setplot: modify plotting parameters	364
11.5.6.1.5.1 add	364
11.5.6.1.5.2 area, noarea	364
11.5.6.1.5.3 background, nobackground	364
11.5.6.1.5.4 channel	364
11.5.6.1.5.5 command	364
11.5.6.1.5.6 delete	364
11.5.6.1.5.7 device	365
11.5.6.1.5.8 PGPLOT devices	365
11.5.6.1.5.9 energy	366
11.5.6.1.5.10 group	366
11.5.6.1.5.11 id	367
11.5.6.1.5.12 list	367
11.5.6.1.5.13 noadd	367
11.5.6.1.5.14 noid	367
11.5.6.1.5.15 rebin	367
11.5.6.1.5.16 redshift	368
11.5.6.1.5.17 splashpage	368
11.5.6.1.5.18 ungroup	368
11.5.6.1.5.19 wave	368
11.5.6.1.5.20 xlog	369
11.5.6.1.5.21 ylog	369
11.5.7 Setting Commands	369
11.5.7.1.1 abund: set the Solar abundances	370
11.5.7.1.2 cosmo: set the cosmology	371
11.5.7.1.3 method: change the fitting method	371
11.5.7.1.3.1 leven	371
11.5.7.1.3.2 migrad	372
11.5.7.1.3.3 simplex	372
11.5.7.1.4 statistic: change the objective function (statistic) for the fit	372
11.5.7.1.5 xsect: set the photoionization cross-sections	373
11.5.7.1.6 xset: set variables for XSPEC models.	373
11.5.8 Tcl Scripts	375
11.5.8.1.1 lrt: likelihood ratio test between two models	375
11.5.8.1.2 multifake: perform multiple fakeit iterations and save to file.	375
11.5.8.1.3 rescalecov: rescale the covariance matrix.	376
11.5.8.1.4 simftest: estimate the F-test probability for adding a component.	376
11.5.8.1.5 writefits: write information about the current fit and errors to a FITS file	376
11.5.9 Deprecated Commands	376
11.5.9.1.1 dump	376
11.5.9.1.2 exec	377
11.5.9.1.3 extend	377
11.5.9.1.4 improve	378
11.5.9.1.5 quit	378
11.5.9.1.6 readline	378
11.5.9.1.7 recornrm	378
11.5.9.1.8 source	379
11.5.9.1.9 suggest	379
11.5.9.1.10 thleqw	379
11.5.9.1.11 uncertain	380
11.5.9.1.12 xhistory	380
12 ATSA Testing	382

12.1 Unit Tests	382
12.2 The Parserator	382
12.3 Python Notebook Extensions	382
12.3.1 Speeding up Debugging	382
12.3.2 Automated Testing	383
12.4 Squish Testing	385
13 Futures	387
13.1 Bits of History	387
13.1.1 Fun with Acronyms	387
13.2 General Enhancements	387
13.3 Notebook Enhancements	388
13.3.1 Copy/Paste	388
13.4 Observatory Tool	389
13.5 Plot Tool and Models	389
13.5.1 Line-based Analysis	389
13.6 Table Tool	389
13.7 Text Tool	390
13.7.1 Enhancements	390
13.7.2 Styles	390
13.8 Python Integrated Development Environment	391
13.9 Nexus Enhancements	391
14 Release Process	393
15 Build History	395
15.1 ATSA 0.1.1	395
15.1.1 Summary	395
15.1.2 Fixes/additions	395
15.2 ATSA 0.1.2	395
15.2.1 Changes:	395
15.3 ATSA 0.1.3	397
15.3.1 Python Issues	397
15.3.2 Notebook	397
15.3.2.1 Notebook Save/Restore	397
15.3.3 Table Tool	397
15.3.4 Python Tool	398
15.3.5 Text Tool	398
15.3.6 Tag Tool	398
15.3.7 XSpec	398
15.3.8 Bug Report #1	399
15.4 ATSA 0.1.4	400
15.4.1 Plots	400
15.4.2 Capella Files	400
15.4.2.1 Case 1: PHA file exists, RMF/ARF do not	400
15.4.2.2 Case 2: PHA file does not specify RMF/ARF, not specified by user either	400
15.4.2.3 PHA doesn't specify RMF/ARF but user does	401
15.4.2.4 Case 4: Another variant of manually supplied RMF/ARF	403
15.4.3 S54405.pha	403
15.4.4 Miscellaneous	403
15.5 ATSA 0.1.5	403
15.5.1 Fit Sub-plots	404
15.5.2 General	404
15.5.3 Line-based Analysis	405
15.5.4 W49b-sxs.pha	406
15.6 ATSA 0.1.6	406

15.6.1 Notebook Save/Restore	406
15.6.2 General	406
15.6.3 Spectrum-level XSpecs and Phases	407
15.6.4 XSpec	407
15.6.5 Long-running Fits	407
15.6.6 Developer Aids	407
15.7 ATSA 0.1.7	408
15.8 ATSA 0.1.8	409
15.8.1 General	409
15.8.2 Table Tool	411
15.8.3 Debug Support Example	412
16 Bugs and Features	416
16.1 High (Pre-demo) Priority	416
16.2 Medium Priority (Nice for demo, probably won't make it)	416
16.3 Low Priority (Post-demo)	416

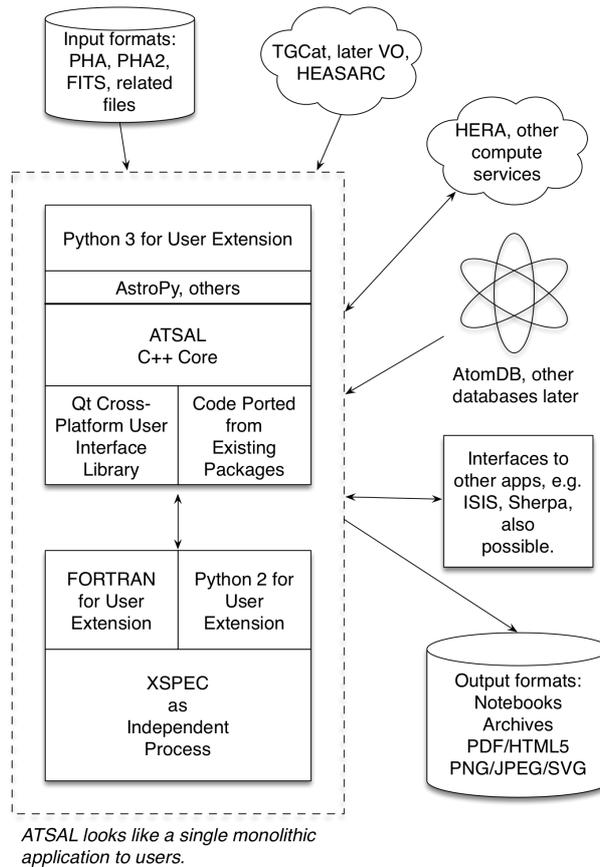


# 1 ATSA

*Nanos gigantum humeris insidentes*

## 1.1 Developer Help

ATSA is a user interface that sits atop a number of other components, notably XSpec, for performing spectrographic analysis. It employs a multiple process implementation in order to achieve several kinds of parallelism. ATSA is implemented almost entirely in C++, making heavy use of the Qt cross-platform class library. The initial version is implemented for MacOS, but later versions will support Linux and perhaps Windows. All components are open source.



This is the documentation tree for ATSA internals. It tracks both internal and external information of use to the project. The doc tree is stored as a series of HTML-subset files, managed by the Nexus documentation manager, so that git can handle conflict resolution when multiple collaborators update docs. The developer tree is stored in `ATSADevHelp` and edited or viewed via `ATSADevHelp.nexus`.

This help tree is **not intended for end users**, and is not included with ATSA. End user help is located in `help` and edited via `help.nexus`. Nexus can also export a free-standing web site or a PDF book with the same content.

## 1.2 ATSA Build Directory

Directory or File	Git	Description
Artwork	Y	Vector or master bitmap artwork from which icons are derived. Not part of the application. (Images derived from the master artwork and used in the application are stored in <code>resources/images</code> .)
ATFilesMaster	Y	Sample observations that are packaged with ATSA and used for tutorials. These are installed automatically if the user does not have such a directory already.

ATFilesMaster.tar.gz	N	Generated from the above, kept around to speed up builds.
ATNotebook.app	N	The most recent release build of ATSal's notebook component.
ATNotebook.build	N	Holds build products.
ATNotebook.pro	Y	Project dependency description for ATNotebook. This is processed by qmake to generate either an XCode project or a makefile.
ATSal.xcodeproj	*	Directory analogous to a makefile for building ATSal using XCode. This is regenerated by qmake, but contains some settings that are checked into git.
ATSalDevHelp	Y	ATSal developer help tree. Don't edit this directly, it is managed by Nexus. Double click on <code>ATSalDevHelp.nexus</code> to edit. This help is only for ATSal developers. The corresponding tree for end users is in directory <code>help</code> .
ATSalDevHelp.nexus	Y	See above.
cbuild	N	Intermediate build products from a command line build ( <code>macrebuild.sh</code> ).
CCFits	Y	C++ interface to FITS library. Currently not in use; we use the <code>cfitsio</code> library's C interface instead to avoid problems with incompatible versions of the C++ std library.
cfitsio	Y	C interface to FITS library.
Debug	N	Contains debug version of ATSal. Entire directory can be deleted.
defunct	Y	Stuff that is not presently used, but of possible interest.
doxygen	Y	Contains configuration files used by doxygen to generate doc trees into the <code>dox</code> directory. <code>macbuild.sh</code> builds them.
enumerator	Y	Contains source code for the <code>enum</code> utility, a simple program that generates enumerations and the user interface classes that display them in popups.
enums.pri	Y	Part of the build file used by qmake to build the <code>enum</code> utility.
enums.sh	Y	Reprocesses the enumerations file, <code>include/utilities/atsal.enum</code> , generating new source.
forms	Y	Contains the forms ( <code>.ui</code> ) files that display various dialogs in the user interface.
help	Y	Contains the HTML doc tree for end-user help for ATSal. Don't mess with this directly, use the Nexus doc tree manager. Double-click <code>help.nexus</code> to edit the tree.
Help.nexus	Y	Opens the <code>help</code> doc tree described above for editing via Nexus.
include	Y	Contains the include files used to build ATSal. (There are also a few in <code>nexus/include</code> that are shared between ATSal and Nexus.)
Info.plist	Y	Not sure if this is in use now.
ma.sh	Y	Uses qmake to generate a new Mac project file for XCode.
macbuild.sh	Y	Deprecated: use <code>macrebuild.sh</code> now.
macclean.sh	Y	Deletes build products.
macpackage_prv.sh	Y	Usually this is called from <code>macrebuild_prv.sh</code> to perform the packaging step for ATSal. If ATSal is already built, this script can also be invoked directly, with no arguments, to repeat only the link step and subsequent packaging steps. Direct invocation is mostly useful when debugging this script.
macrebuild_prv.sh	Y	Performs a from-scratch command line rebuild of darned near everything, except for a few libraries (especially Qt!) that do not change in normal use. Invoke <code>macrebuild.sh</code> , not this.
macrebuild.sh	Y	This is a simple wrapper around <code>macrebuild_prv.sh</code> . It does a timed build and directs the output to <code>build.log</code> . Do a complete rebuild with <code>sh macrebuild.sh &amp;</code> . Then <code>tail build.log</code> if desired.
Makefile	N	Created by <code>macrebuild_prv.sh</code> . Do not edit directly.
matplotlib	Y	The Matplotlib library.
mtouch.sh	Y	A utility script that makes it a little quicker to add new files to the ATSal project. Begin by <code>cd</code> -ing to the source directory that will contain a new source file. Then type e.g. <code>mtouch.sh foo.cpp</code> . This creates <code>foo.cpp</code> as well as <code>foo.h</code> in the corresponding include directory. Both paths are listed to make it easier to add them to the <code>.pro</code> file.

Nexus	Y	Contains sources needed to build the Nexus doc tree manager.
postDebugBuild.sh	Y	Steps performed immediately after each XCode debug project build.
postReleaseBuild.sh	Y	Steps performed immediately after each XCode release project build. This packages dynamic libraries and help and other resources into the application bundle on a Mac.
Python-3.3.3	Y	Python source tree.
pythonqt	Y	PythonQt source tree.
qdocs	Y	A doxygen-like facility for generating API docs. I like the output of this process better (more compact), but I have not evaluated it yet.
QInfo.plist	Y	<code>Info.plist</code> template for building Mac version of ATNotebook.
qrc_resources.cpp	N	Build product.
README	Y	Brief description of this directory. A similar file is in most subdirectories.
Release	N	Build products for the release version of ATNotebook.
resources	Y	Resource files that are compiled directly into ATSAL.
setversion	Y	Utility to keep version numbers in sync.
src	Y	ATSAL source code. There is a bit more in <code>nexus/src</code> , shared with Nexus.
supervisor	Y	Contains ATSAL, the ATSAL supervisor. This is a separate application that runs instances of ATNotebook for each open notebook.
testing	Y	Test code.
updatwebsite.sh	Y	Nexus can export a free-standing web site with a document tree. <code>updatewebsite.sh</code> uploads the generated site to a web server.

## 1.3 Build Process

This section contains a description of the overall build process. No automated mechanism exists yet to install the tools needed to do ATSAL development, though these tools are described under *Configuring a Development Environment*.

**Linux build procedures have not yet been developed.** Packaging issues that have been resolved for the Mac will need to be handled differently under Linux.

### 1.3.1 Mac Debug Builds

Typically, debug builds are done as follows:

```
cd ~/atsal/proj/atsal
sh ma.sh
```

This generates a new `.xcodeproj` file from the `.pro` file. Then I open the XCode project file with:

```
cd ~/atsal/proj/atsal
open ATNotebook.xcodeproj/
```

I have defined aliases to perform these commands in my `.bash_profile`: `ma` performs the first step, and `opena` performs the second. Similar aliases (`ms/opens` for building the supervisor, and `mn/openn` for Nexus) make it easier to move among the directories and open XCode sessions for each component.

Although the notebook process (ATNotebook) is designed to run under the supervisor (ATSAL), this complicates debugging, so ATNotebook is designed to run freestanding as well, with some constraints.

### 1.3.2 Mac Release Builds

Release builds are done by command line, to make automated build and test operations simpler.

Qt provides a tool, `qmake`, which processes a project description file (`.pro`) and generates either a makefile for command line builds on Mac or Linux, or an XCode project file for the Mac (`.xcodeproj`). Most directories contain a

file called `macbuild_prv.sh`, which rebuilds release software for that level and those below, via command line. Many directories also contain a simple wrapper for this, `macbuild.sh`, which times the build and directs the output to `build.log`.

The ATSAAL source tree looks like this:

<code>atsal*</code>	Root
<code>atsal/private*</code>	Documentation and other components that have not yet been organized for collaborative sharing.
<code>atsal/proj*</code>	Root of ATSAAL development tree.
<code>atsal/proj/atsal</code>	Root of ATSAAL notebook application, ATNotebook.
<code>atsal/proj/bin</code>	Development utilities and scripts.
<code>atsal/proj/docs</code>	Docs related to the project, both external (e.g. git) and internal.
<code>atsal/proj/heasoft*</code>	XSPEC server.
<code>atsal/proj/installers</code>	Software install packages for components needed to develop ATSAAL.
<code>atsal/proj/macdylibbundler</code>	Utility used to encapsulate dynamic libraries by redirecting them to point at each other.
<code>atsal/proj/mobile*</code>	AtomDB, cfitsio, and related software.
<code>atsal/web*</code>	Root of ATSAAL web site and related tools, which does not yet exist.

\*Indicates a separate git archive. Archives are bulky, so this allows some developers to check out a subset.

### 1.3.2.1 qmake

I use Apple's XCode integrated development system to create and debug ATSAAL on the Macintosh. But we also want build scripts that operate entirely from the command line, for testing and packaging of releases. Qt's qmake facility handles both of these cases:

```
qmake -spec macx-xcode ATNotebook.pro
```

accepts a description of the application's components and generates `ATSAAL.xcodeproj`, a file (really a directory tree) used by XCode to develop and debug the application.

```
qmake -spec macx-clang ATNotebook.pro
```

generates a makefile instead, for building from the command line. Releases are always built from the command line, while debug versions may be built either way.

### 1.3.3 atsal/private

This archive is "private" only insofar as it is a dumping ground for information not yet well enough organized for sharing with others. The idea here is that over time there will be two classes of ATSAAL developers: core developers who do this more or less full time, and other interested parties who check out the source tree and make occasional, probably isolated, contributions. The former group will check out the entire hierarchy of projects, while the latter group only needs `atsal/proj`.

Basic project documentation is in the docs subtree, and will later be rearranged a bit and moved into `atsal/proj`.

### 1.3.4 atsal/proj

This is the root ATSAAL project directory. To perform a complete command line rebuild of ATSAAL:

```
cd atsal/proj
sh macrebuild.sh &
```

To monitor build progress,

```
tail -f build.log
```

### 1.3.4.1 atsal/proj/atsal

This directory builds ATNotebook.app.

<i>Directory or File</i>	<i>Git</i>	<i>Description</i>
Artwork	Y	Vector or master bitmap artwork from which icons are derived. Not part of the application. (Images derived from the master artwork and used in the application are stored in <code>resources/images</code> .)
ATFilesMaster	Y	Sample observations that are packaged with ATSal and used for tutorials. These are installed automatically if the user does not have such a directory already.
ATFilesMaster.tar.gz	N	Generated from the above, kept around to speed up builds.
ATNotebook.app	N	The most recent release build of ATSal's notebook component.
ATNotebook.pro	Y	Project dependency description for ATNotebook. This is processed by <code>qmake</code> to generate either an XCode project or a makefile.
ATSAL.xcodeproj	*	Directory analogous to a makefile for building ATSal using XCode. This is regenerated by <code>qmake</code> , but contains some settings that are checked into git.
ATSALDevHelp	Y	ATSal developer help tree. Don't edit this directly, it is managed by Nexus. Double click on <code>ATSALDevHelp.nexus</code> to edit. This help is only for ATSal developers. The corresponding tree for end users is in directory <code>help</code> .
ATSALDevHelp.nexus	Y	See above.
cbuild	N	Intermediate build products from a command line build ( <code>macrebuild.sh</code> ).
CCFits	Y	C++ interface to FITS library. Currently not in use; we use the <code>cfitsio</code> library's C interface instead to avoid problems with incompatible versions of the C++ std library.
cfitsio	Y	C interface to FITS library.
Debug	N	Contains debug version of ATSal. Entire directory can be deleted.
dox	*	Contains HTML doc trees for several components, machine generated by doxygen. Also contains <code>other</code> , which is hand-generated. Only <code>other</code> is checked in.
doxygen	Y	Contains configuration files used by doxygen to generate doc trees into the <code>dox</code> directory. <code>macbuild.sh</code> builds them.
enumerator	Y	Contains source code for the <code>enum</code> utility, a simple program that generates enumerations and the user interface classes that display them in popups.
enums.pri	Y	Part of the build file used by <code>qmake</code> to build the <code>enum</code> utility.
enums.sh	Y	Reprocesses the enumerations file, <code>include/utilities/atsal.enum</code> , generating new source.
forms	Y	Contains the forms ( <code>.ui</code> ) files that display various dialogs in the user interface.
help	Y	Contains the HTML doc tree for end-user help for ATSal. Don't mess with this directly, use the Nexus doc tree manager. Double-click <code>help.nexus</code> to edit the tree.
Help.nexus	Y	Opens the <code>help</code> doc tree described above for editing via Nexus.
include	Y	Contains the include files used to build ATSal. (There are also a few in <code>nexus/include</code> that are shared between ATSal and Nexus.)
Info.plist	Y	Not sure if this is in use now.
ma.sh	Y	Uses <code>qmake</code> to generate a new Mac project file for XCode.
macbuild.sh	Y	Deprecated: use <code>macrebuild.sh</code> now.
macclean.sh	Y	Deletes build products.
macpackage_prv.sh	Y	Usually this is called from <code>macrebuild_prv.sh</code> to perform the packaging step for ATSal. If ATSal is already built, this script can also be invoked directly, with no arguments, to repeat only the link step and subsequent packaging steps. Direct invocation is mostly useful

		when debugging this script.
macrebuild_prv.sh	Y	Performs a from-scratch command line rebuild of darned near everything, except for a few libraries (especially Qt!) that do not change in normal use. Invoke <code>macrebuild.sh</code> , not this.
macrebuild.sh	Y	This is a simple wrapper around <code>macrebuild_prv.sh</code> . It does a timed build and directs the output to <code>build.log</code> . Do a complete rebuild with <code>sh macrebuild.sh &amp;</code> . Then <code>tail build.log</code> if desired.
Makefile	N	Created by <code>macrebuild_prv.sh</code> . Do not edit directly.
mtouch.sh	Y	A utility script that makes it a little quicker to add new files to the ATNSAL project. Begin by <code>cd</code> -ing to the source directory that will contain a new source file. Then type e.g. <code>mtouch.sh foo.cpp</code> . This creates <code>foo.cpp</code> as well as <code>foo.h</code> in the corresponding include directory. Both paths are listed to make it easier to add them to the <code>.pro</code> file.
Nexus	Y	Contains sources needed to build the Nexus doc tree manager.
postDebugBuild.sh	Y	Steps performed immediately after each XCode debug project build.
postReleaseBuild.sh	Y	Steps performed immediately after each XCode release project build. This packages dynamic libraries and help and other resources into the application bundle on a Mac.
Python-3.3.3	Y	Python source tree.
pythonqt	Y	PythonQt source tree.
qdocs	Y	A doxygen-like facility for generating API docs. I like the output of this process better (more compact), but I have not evaluated it yet.
QInfo.plist	Y	<code>Info.plist</code> template for building Mac version of <code>ATNotebook</code> .
qrc_resources.cpp	N	Build product.
README	Y	Brief description of this directory. A similar file is in most subdirectories.
resources	Y	Resource files that are compiled directly into ATNSAL.
src	Y	ATNSAL source code. There is a bit more in <code>nexus/src</code> , shared with Nexus.
testing	Y	Test code.

### 1.3.4.1.1 Mac Deployment Command Line Build

`macrebuild.sh` does a from-scratch command line build of the Mac version of ATNSAL. It exists at several levels in the hierarchy, starting at `atsal/proj`. The script comes in two forms: `macrebuild.sh` and `macrebuild_prv.sh`. The former simply does a timed build of the latter, directing output to `build.log`. Recursion to similarly named scripts in subdirectories is done manually at this point. This build script is not necessarily intended as a permanent solution, rather as a stopgap.

Run the script with `LDFLAGS` set to `-headerpad_max_install_names`, to get XSpec to build with room for redirecting dynamic library references into the application bundle. This will eventually get added to the XSpec build process (or perhaps XSpec's `--enable-static` build option will be made fully operational), but for now this workaround is used.

At the `proj` level, `macrebuild.sh`:

- builds `macdylibbundler`, a utility that helps encapsulate XSpecServer and ATNSAL so they don't interact with the rest of the user's system.
- builds `heasoft`—the XSpecServer and other components.
- builds ATNSAL:
  - `ATNSAL.app`, the supervisor process.
  - `ATNotebook.app`, the process that is instantiated for each open notebook.
  - `Nexus.app`, the process to create help trees.

On the Macintosh, the necessary resources are packaged within `ATNSAL.app`, so that it appears to be a single monolithic application and so that library or Python version skew cannot occur. `ATNSAL.app` becomes a container for all the dynamic libraries, XSpecServer, `ATNotebook.app`, `AtomDB`, etc. The final application bundle is placed on the desktop in a time-stamped folder.

It also:

- Reprocesses the enumerations files using the `enum` utility.
- Regenerates the `doxygen` docs.
- Regenerates the PythonQt libraries.

Because these components rarely need rebuilding, `macrebuild.sh` does *not*:

- Rebuild AtomDB, or its associated components.
- Rebuild `XMLRPC`.
- Rebuild the Python interpreter.

### 1.3.4.1.1.1 macdylibbundler

This utility is available on [github](#), and the docs are reproduced here in case they evaporate.

#### 1.3.4.1.1.1.1 About

Mac OS X introduced an innovative and very useful way to package applications: app bundles. While their design has all that is needed to ease distribution of resources and frameworks, it seems like dynamic libraries (`.dylib`) are very complicated to distribute. Sure, applications developed specifically for OS X won't make use of them, however applications ported from Linux or other Unices may have dependencies that will only compile as `dylibs`. By default, there exists no mechanism to bundle them but some command-line utilities provided by Apple - however it turns out that for a single program it is often necessary to issue dozens of commands! This often leads each porter to develop their own "home solution" which are often hacky, poorly portable and/or unoptimal.

`dylibbundler` is a small command-line programs that aims to make bundling `.dylibs` as easy as possible. It automatically determines which `dylibs` are needed by your program, copies these libraries inside the app bundle, and fixes both them and the executable to be ready for distribution... all this with a single command on the terminal! It will also work if your program uses plug-ins that have dependencies too.

It usually involves 2 actions:

- Creating a directory (by default called `libs`) that can be placed inside the Contents folder of the app bundle.
- Fixing the executable file so that it is aware of the new location of its dependencies.

#### 1.3.4.1.1.1.2 Installation

In the terminal, `cd` to the main directory of `dylibbundler` and type "make". You can install with "sudo make install".

#### 1.3.4.1.1.1.3 Feedback/Contact

You can contact me here on [github](#), for instance by creating a ticket or pull request

#### 1.3.4.1.1.1.4 Using dylibbundler on the terminal

Here is a list of flags you can pass to `dylibbundler`.

`-h, --help`

displays a summary of options

`-x, --fix-file executable-path`

Fixes given executable or plug-in file (A `.dylib` can work too. Anything on which ``otool -L`` works is accepted by ``-x``). `Dylibbundler` will walk through the dependencies of the specified file to build a dependency list. It will also fix said files' dependencies so that it expects to find the libraries relative to itself (e.g. in the app bundle) instead of at an absolute path (e.g. `/usr/local/lib`). To pass multiple files to fix, simply specify multiple ``-x`` flags.

`-b, --bundle-deps`

Copies libraries to a local directory, fixes their internal name so that they are aware of their new location, fixes

dependencies where bundled libraries depend on each other. If this option is not passed, no libraries will be prepared for distribution.

`-i, --ignore path`

Dylibs in `path` will be ignored. By default, `dylibbundler` will ignore libraries installed in `/usr/lib` since they are assumed to be present by default on all OS X installations. (It is usually recommend not to install additional stuff in `/usr`, always use `/usr/local` or another prefix to avoid confusion between system libs and libs you added yourself)

`-d, --dest-dir directory`

Sets the name of the directory in which distribution-ready dylibs will be placed, relative to the current working directory. (Default is `./libs`) For an app bundle, it is often convenient to set it to something like `./MyApp.app/Contents/libs`.

`-p, --install-path libraries-install-path`

Sets the "inner" installation path of libraries, usually inside the bundle and relative to executable. (Default is `@executable_path/./libs/`, which points to a directory named `libs` inside the `Contents` directory of the bundle.) The difference between `-d` and `-p` is that `-d` is the location `dylibbundler` will put files at, while `-p` is the location where the libraries will be expected to be found when you launch the app. Both are often related.

`-of, --overwrite-files`

When copying libraries to the output directory, allow overwriting files when one with the same name already exists.

`-od, --overwrite-dir`

If the output directory already exists, completely erase its current content before adding anything to it. (This option implies `--create-dir`)

`-cd, --create-dir`

If the output directory does not exist, create it.

`-sp, --search-path path`

Adds a directory to the search path. If a library cannot be found at the absolute path in the executable or library, an attempt is made to locate it in the search path. For example:

```
% dylibbundler -od -b -sp /usr/local/lib -sp /opt/lib -x
./HelloWorld.app/Contents/MacOS/helloworld
  -d ./HelloWorld.app/Contents/libs-ppc/ -p @executable_path/./libs-ppc/
```

searches `/usr/local/lib` and `/opt/lib` for any libraries not found at their link-time locations. **This switch is not part of the original, it was added for ATSA.**

If you want to create a universal binary by merging together two builds from PPC and Intel machines, you can ease it up by putting the ppc and intel libs in different directories, then to create the universal binary you only have to lipo the executable.

```
% dylibbundler -od -b -x ./HelloWorld.app/Contents/MacOS/helloworld
  -d ./HelloWorld.app/Contents/libs-ppc/ -p @executable_path/./libs-ppc/

% dylibbundler -od -b -x ./HelloWorld.app/Contents/MacOS/helloworld
  -d ./HelloWorld.app/Contents/libs-intel/ -p @executable_path/./libs-intel/
```

### 1.3.4.1.1.2 autoconf

From [Sebastien's blog](#):

Saturday, August 11, 2012

### 1.3.4.1.1.2.1 Autoconf and Automake on Mac OS X Mountain Lion

It's summer and I've been thinking about blogging more regularly again. I'm usually too busy to find time to blog, so I'm going to try a shorter format this time: sort of middle ground between a tweet and a full blown blog.

So, here's my first entry in that shorter format.

If you're upgrading to Mac OS X Mountain Lion and Xcode 4.4.1, you'll find that Xcode does not include anymore the GNU Autoconf, Automake and Libtool build tools used by most open source projects to generate makefiles and dynamic libraries... That's not so great :(

I wanted to share what I did to build them myself from source, as it could help others too:

```
export build=-/devtools # or wherever you'd like to build
mkdir -p $build

cd $build
curl -OL http://ftpmirror.gnu.org/autoconf/autoconf-2.68.tar.gz
tar xzf autoconf-2.68.tar.gz
cd autoconf-2.68
./configure --prefix=$build/autotools-bin
make
make install
export PATH=$PATH:$build/autotools-bin/bin

cd $build
curl -OL http://ftpmirror.gnu.org/automake/automake-1.11.tar.gz
tar xzf automake-1.11.tar.gz
cd automake-1.11
./configure --prefix=$build/autotools-bin
make
make install

cd $build
curl -OL http://ftpmirror.gnu.org/libtool/libtool-2.4.tar.gz
tar xzf libtool-2.4.tar.gz
cd libtool-2.4
./configure --prefix=$build/autotools-bin
make
make install
```

## 1.4 XCode

### 1.4.1 XCode and Time Machine

Back in March 2015 I tried to retrieve an old file from the (Mac) Time Machine backup of the ATSAAL source tree. I discovered to my chagrin that the file was missing; in fact, everything under a directory named `dev` was missing. I tried to figure out what was going on and eventually turned up Time Machine's standard exclusion list, which, naturally enough, excludes `/dev`. So I reasoned that there was a bug, and it was refusing to back up *any* directory named `dev`. I tested this by renaming the directory to `proj`, and got a good backup. And immediately forgot all about it, until today (Oct 2015) when I needed to restore an old file.

**Now Time Machine has stopped backing up the entire project.**

I dug around and discovered this. (`atsal` is the root of the project tree.)

```
(py3)ATSAAL:proj$ xattr atsal
com.apple.XcodeGenerated
com.apple.metadata:com_apple_backup_excludeItem
```

I found articles that confirm that XCode is setting this attribute automatically, instructing Time Machine not to backup. That's perfectly understandable of course for temp directories that are regenerated constantly, but it is tagging the *containing* directory, not the regenerable directories. This seems incredible, but I verified by experiment that the attributes come back as soon as you run XCode. Specifically, running XCode (6.1 in this case) immediately marks the

directory containing the `.xcodeproj` file as do not back up. Apple, for reasons I don't claim to understand, doesn't consider this a bug, at least based on what I have found on the web.

I considered a workaround, maybe a `launchd` job that strips the attributes to allow backup, but the file tree is quite complex, with many nested `.xcodeproj`s, so that seems like a recipe for disaster. Finally I decided to write a script that tars the whole damned project off to another volume once daily at 1 AM, keeping the most recent 30 such files. Since I am the sole developer at this stage, I don't check in constantly with git, only every couple of weeks when I reach a plateau, so git isn't a reasonable substitute for daily backups.

I created a script, `~/backupATSAL.sh`, to create a time-stamped tar file and place it on another volume, keeping only the most recent 30 backups. Then I created a plist for `launchd`, `~/Library/LaunchAgents/com.florafinder.atsalbackup.plist` (florafinder? long story). Right now, it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.florafinder.atsalbackup</string>
  <key>ProgramArguments</key>
  <array>
  <string>/Users/tkent/backupATSAL.sh</string>
  </array>
  <key>StartCalendarInterval</key>
  <dict>
    <key>Minute</key>
    <integer>0</integer>
    <key>Hour</key>
    <integer>1</integer>
  </dict>
</dict>
</plist>
```

I found that I had to change the ownership to `rw-r--r--` and change the ownership to `root` to get the file to load via `launchctl`, but that `sudo` was not needed to do the actual launch. I also found that support for a user `launch.conf` was documented to be missing, so rather than load it on a system-wide basis, I added this to my `.bash_profile`:

```
launchctl load ~/Library/LaunchAgents/com.florafinder.atsalbackup.plist
```

This is **NOT** the way to do this, but it works, and I have other stuff to worry about.

## 1.4.2 Mac Icons

I have had a lot of problems with Mac icons, because the approach used by XCode is in constant flux and Qt is hard-pressed to track the changes. At the moment, I use an older system of representing icons. The long term goal is to switch to SVG icons.

Both of these are apparently already obsolete on El Capitan:

To force refresh icon put this into your terminal

```
/System/Library/Frameworks/ApplicationServices.framework/Frameworks/LaunchServices.framework/Support/lsregister -kill -r -domain local -domain system -domain user; killall Dock
```

Mavericks:

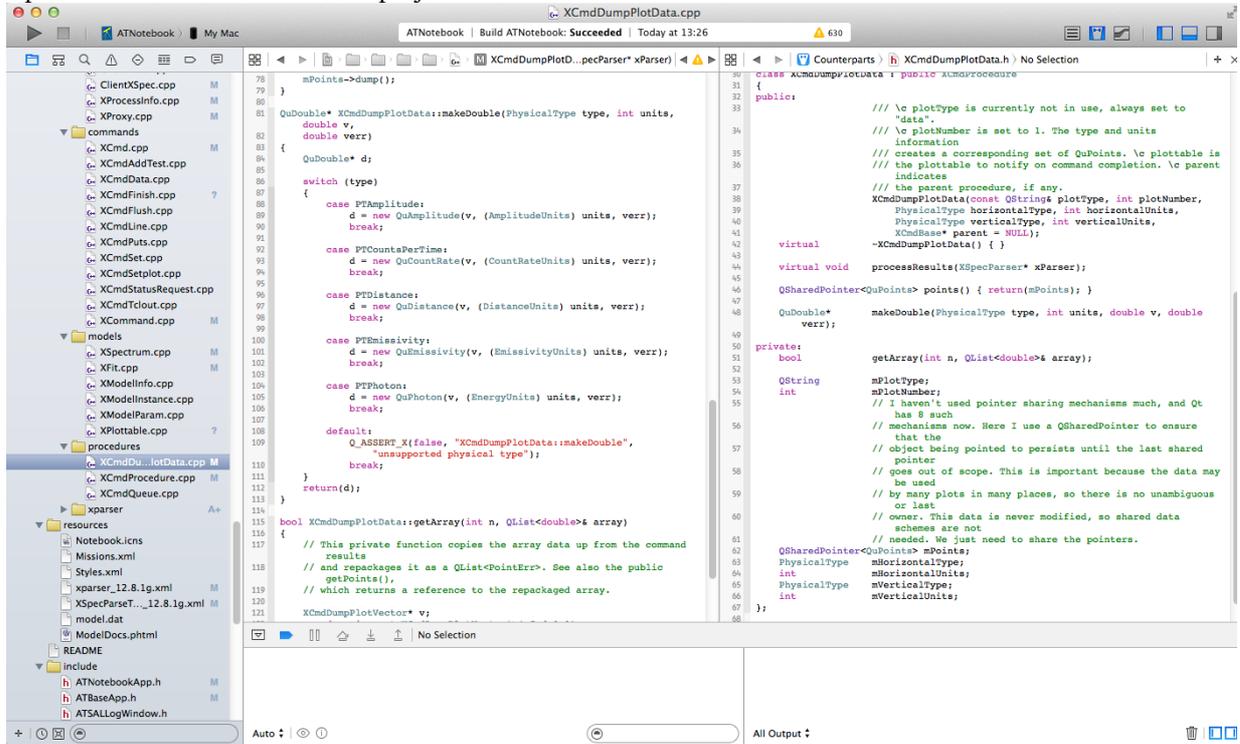
```
/System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/LaunchServices.framework/Versions/A/Support/lsregister; killall Dock
```

## 1.4.3 Mac Debugging

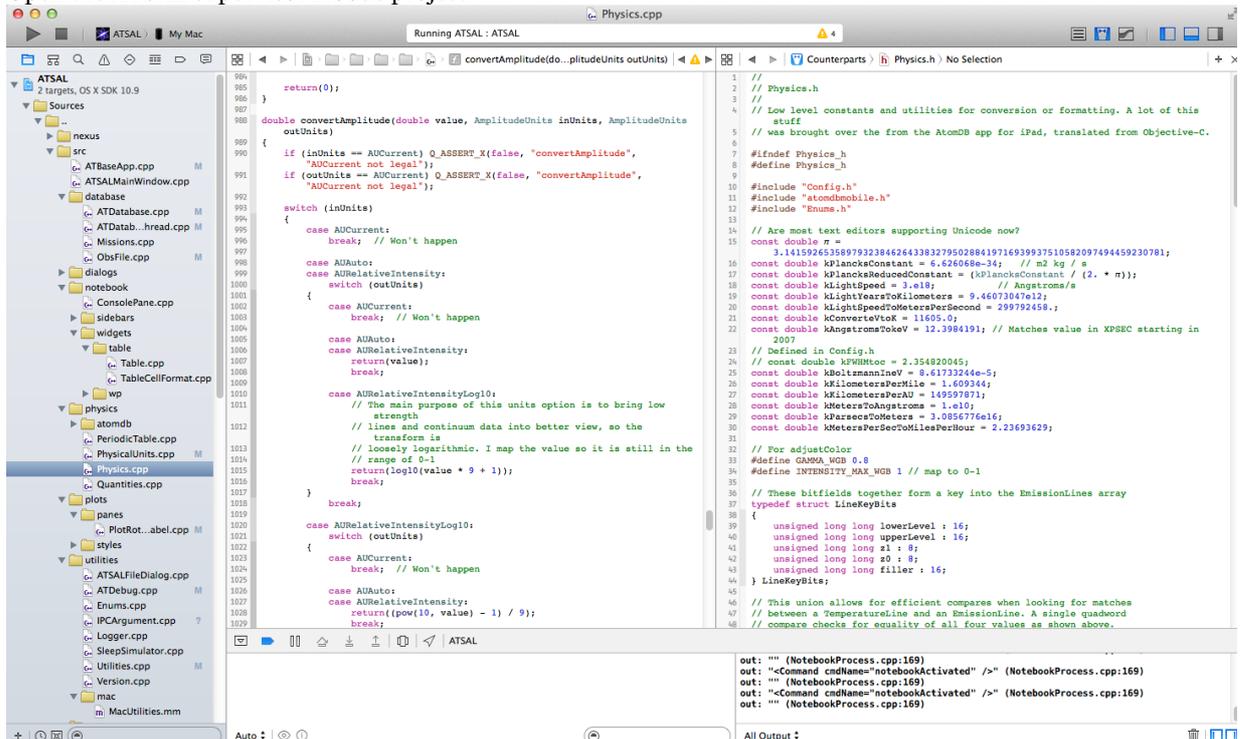
By Tom Kent  
Pepperell Sanitarium

Debugging ATNSAL is a little like playing three pianos at once.

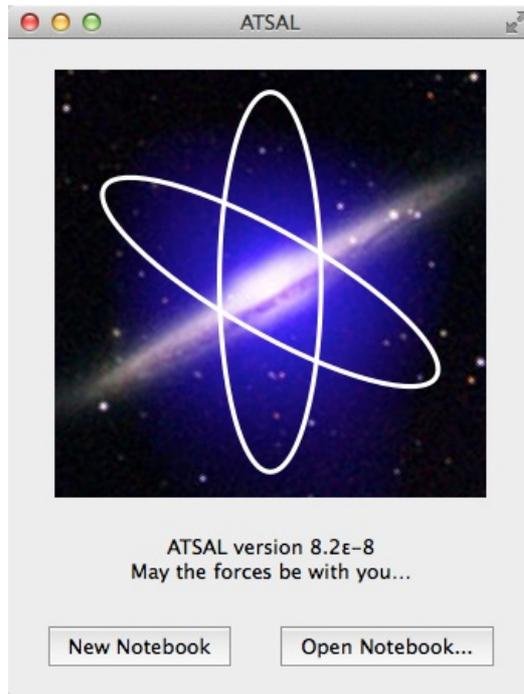
1. Open the ATNSAL notebook XCode project.



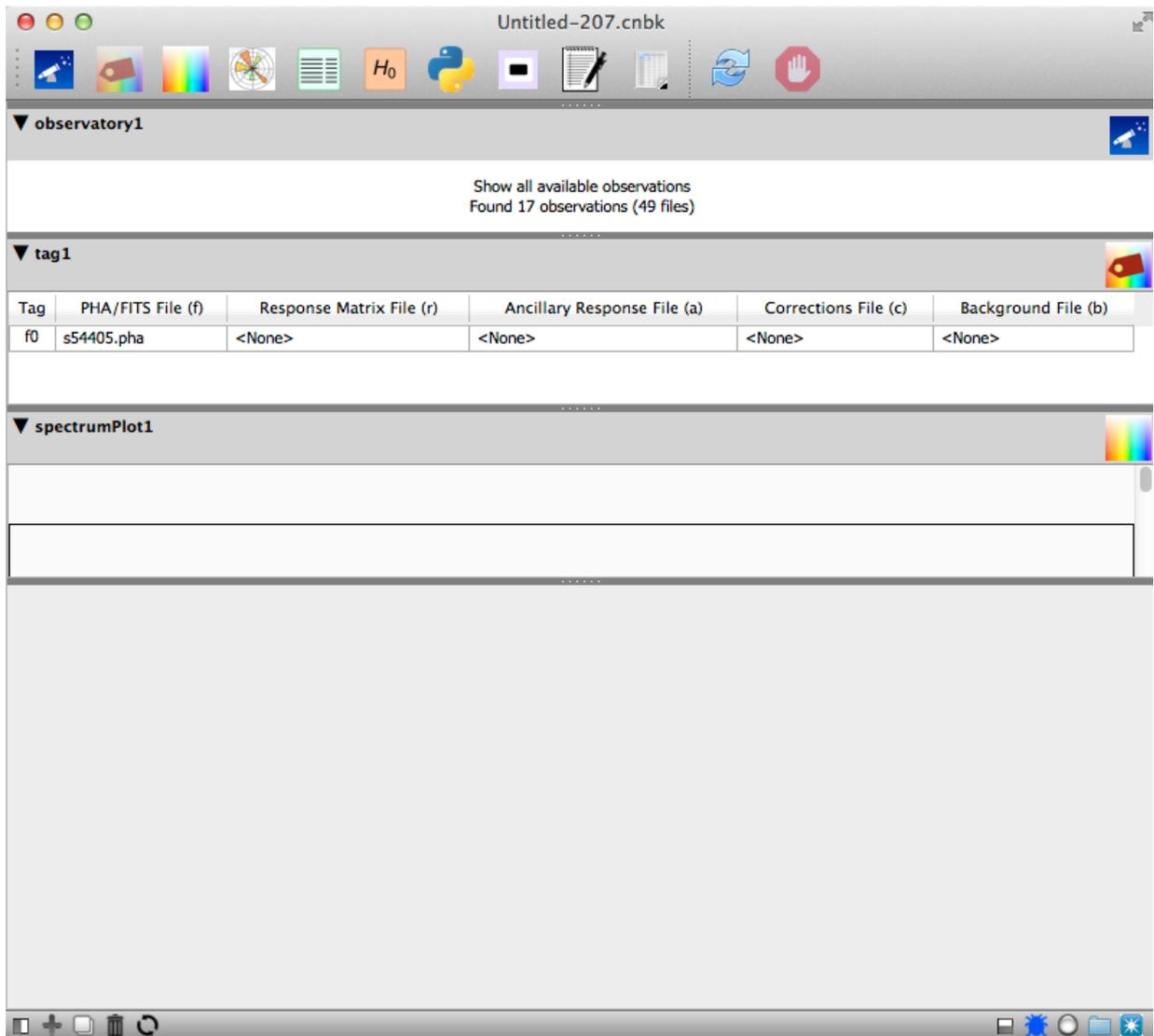
2. Open the ATNSAL supervisor XCode project.



3. Run the supervisor from the XCode project.

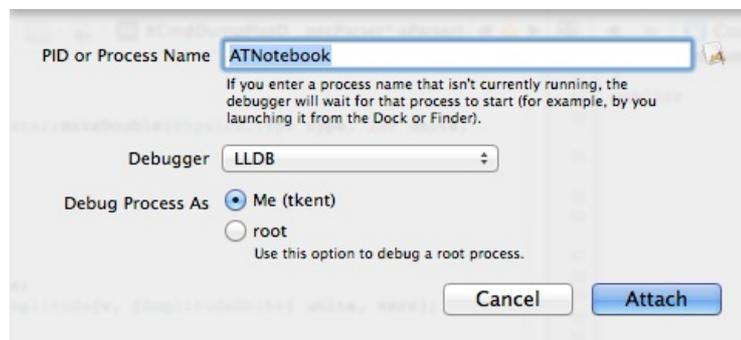


4. Create a notebook and add a few tools to it.



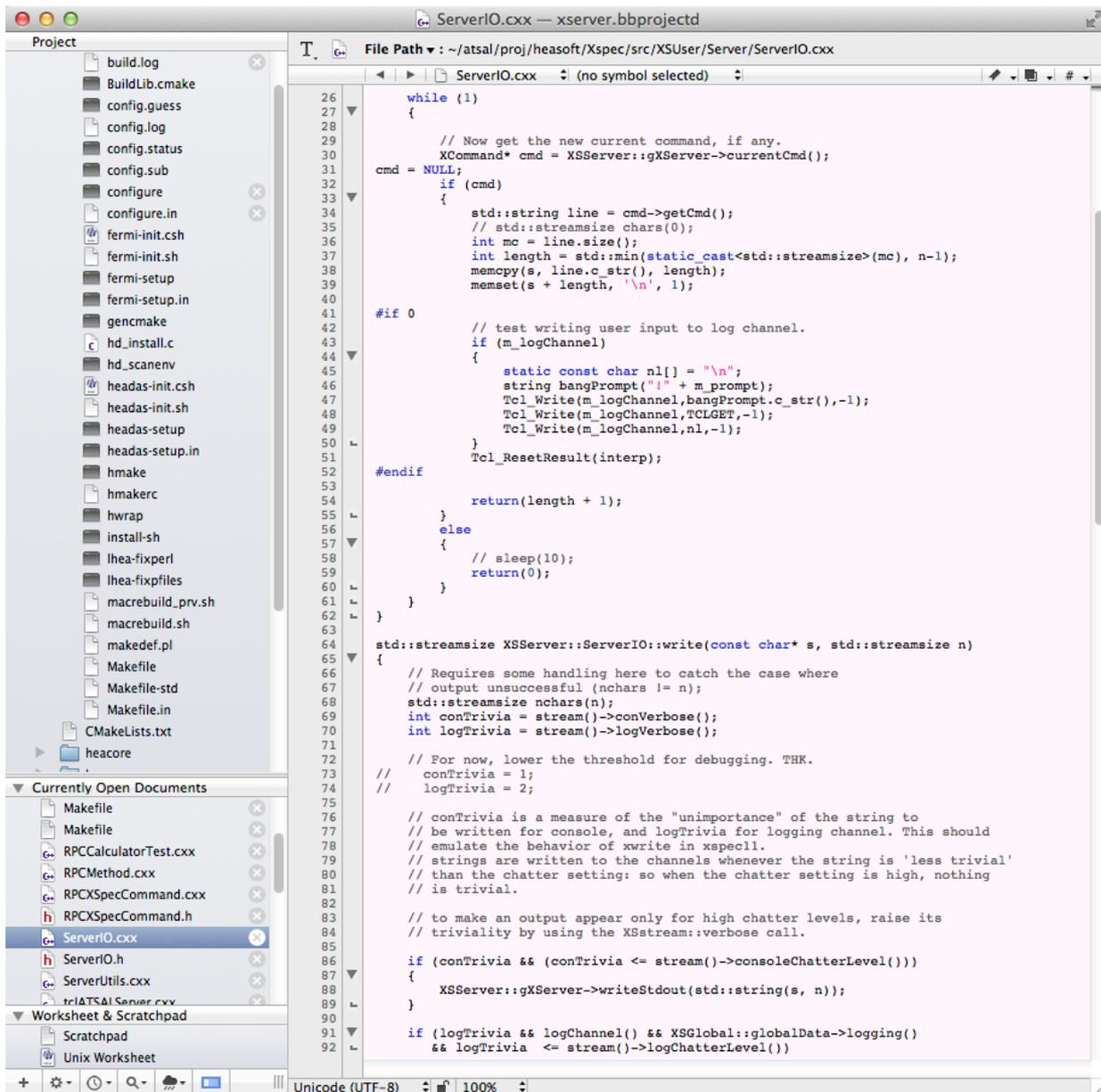
5. Attach to the notebook process.

Go to the notebook project. It is not attached to the notebook process, because the process was activated by the supervisor, not by XCode. Attach manually to the process.



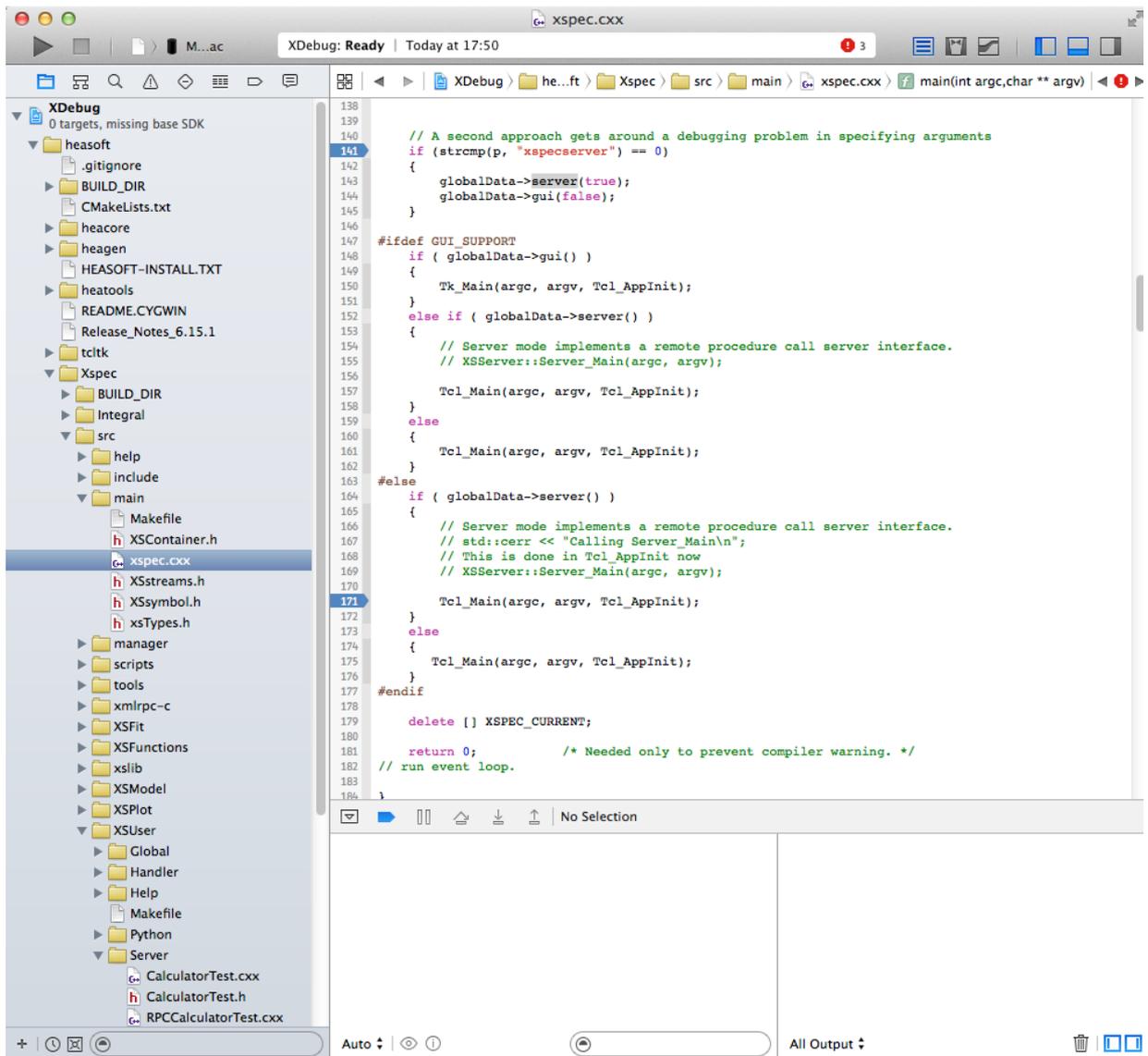
6. Open the XSpec Browser

If working with XSpec internals, open a BBEdit project that browses the XSpec source tree.



7. Open the XSpec debugger

This is a specially hacked XCode debugger project for XSpec (this is magic cobbled together from the web).



8. Open a terminal window and tail the notebook debug log.

```

model "project" not found (XFit.cpp:191)
Cannot find model "project" :: "e1" (XModelInfo.cpp:453)
Cannot find model "project" :: "e2" (XModelInfo.cpp:453)
Cannot find model "project" :: "e3" (XModelInfo.cpp:453)
model "xmmpsf" not found (XFit.cpp:191)
Cannot find model "xmmpsf" :: "alpha" (XModelInfo.cpp:453)
Cannot find model "xmmpsf" :: "beta" (XModelInfo.cpp:453)
Cannot find model "xmmpsf" :: "core" (XModelInfo.cpp:453)
"puts server1 $atsal_temp" (CommandTests.cpp:96)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
Warning: QObject::connect: No such slot ToolTag::removeUncheckedItems() in /Users/tkent/atsal/proj/atsal/src/notebook/si
debars/collapsibles/CPTag.cpp:49 (qobject.cpp:2248)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
XPlottableData: XPlottableData, ID 945030470983161213 (XPlottable.cpp:73)
Warning: QObject::connect: No such signal BlockPlot::trashModel(PlotID) in /Users/tkent/atsal/proj/atsal/src/notebook/si
debars/collapsibles/CPSpectrum.cpp:345 (qobject.cpp:2248)
ToolEditorPlot::newDataSet, ID 945030470983161213 (ToolEditorPlot.cpp:715)
PlotZoneInfo::insertDataSet, zone 0 called with 945030470983161213 (PlotZoneInfo.cpp:112)
PlotZoneInfo::insertDataSet, zone 1 called with 945030470983161213 (PlotZoneInfo.cpp:112)
XPlottableData: XPlottableData, ID 1043193501077063809 (XPlottable.cpp:73)
Warning: QObject::connect: No such signal BlockPlot::trashModel(PlotID) in /Users/tkent/atsal/proj/atsal/src/notebook/si
debars/collapsibles/CPSpectrum.cpp:345 (qobject.cpp:2248)
ToolEditorPlot::newDataSet, ID 1043193501077063809 (ToolEditorPlot.cpp:715)
PlotZoneInfo::insertDataSet, zone 1 called with 1043193501077063809 (PlotZoneInfo.cpp:112)
PlotZoneInfo::insertDataSet, zone 1 called with 1043193501077063809 (PlotZoneInfo.cpp:112)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
N->S: "<Command cmdName="notebookActivated" />"
" (Supervisor.cpp:127)
(py3)ATSAL:Logs$

```

9. ... and the supervisor log ...

```

BML: "/Users/tkent/ATFiles/Obs/Observations//r1_ol_pha.fits" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//s54405.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//s54405.rsp" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//xrs.fak" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//xrs.rmf" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//arlac_saxlecs.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//arlac_saxpds.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//arlac_xmrgs1.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//arlac_xte.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//capella_pha2.fits" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//capella_row3.arf" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//capella_row3.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//capella_row3.rmf" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//checksums.dat" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//lat.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//r1_ol.rmf" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//r1_ol_pha.fits" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//s54405.pha" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//s54405.rsp" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//xrs.fak" (ATDatabaseThread.cpp:145)
BML: "/Users/tkent/ATFiles/Obs/Observations//xrs.rmf" (ATDatabaseThread.cpp:145)
"/Users/tkent/atsal/proj/atsal/Debug/ATNotebook.app/Contents/MacOS/ATNotebook" (NotebookProcess.cpp:53)
out: "ATSAL Python 3.3 on darwin." (NotebookProcess.cpp:169)
out: "<Command cmdName="notebookActivated" />" (NotebookProcess.cpp:169)
out: "" (NotebookProcess.cpp:169)

```

10. ... and the XSpec server log ...

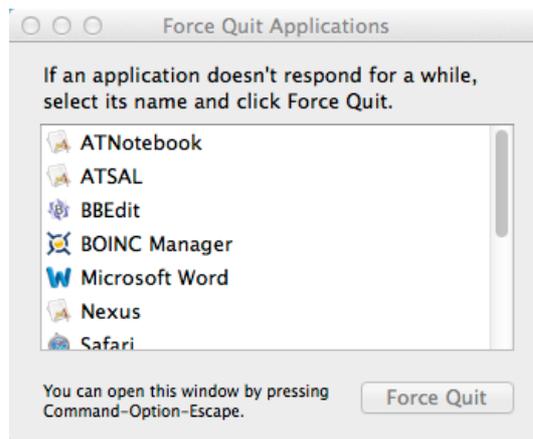
```

tkent — tail
R 0:
52: puts server1 $atsal_temp
Tcl_Eval(puts server1 $atsal_temp)
Calling cmdCompleted, Tcl_Eval returned 0/0
dumpStatusPacket:
O 52: 8.80453e-06 3.77995e-05 5.52445e-05 4.96201e-05 4.31485e-05 3.97507e-05 3.78577e-05 3.61664e-05 3.
48118e-05 3.43048e-05 3.3309e-05 3.22598e-05 3.1353e-05 3.01334e-05 2.95632e-05 2.89407e-05 2.87906e-05
2.86148e-05 2.86084e-05 2.87229e-05 2.88968e-05 2.8791e-05 2.85329e-05 2.84346e-05 2.83695e-05 2.80411e-05
2.78368e-05 2.76276e-05 2.75896e-05 2.7538e-05 2.78013e-05 2.81268e-05 2.88893e-05 2.96027e-05 3.00873e
-05 3.04613e-05 3.11267e-05 3.14823e-05 3.17107e-05 3.20264e-05 3.22922e-05 3.23479e-05 3.19675e-05 3.13
926e-05 3.04054e-05 2.91568e-05 2.77598e-05 2.68e-05 2.57659e-05 2.50187e-05 2.42187e-05 2.36099e-05 2.2
6545e-05 2.16975e-05 2.04817e-05 1.95683e-05 1.83706e-05 1.73004e-05 1.64056e-05 1.55271e-05 1.46954e-05
1.38957e-05 1.31679e-05 1.25812e-05 1.20801e-05 1.16997e-05 1.11897e-05 1.08014e-05 1.04309e-05 1.01295e
-05 9.83374e-06 9.5105e-06 9.09249e-06 8.95949e-06 8.60232e-06 8.37558e-06 8.1375e-06 7.81829e-06 7.58756
e-06 7.41814e-06 7.21773e-06 6.74912e-06 6.73395e-06 6.57663e-06 6.39694e-06 6.25129e-06 6.07816e-06 5.9
2467e-06 5.70972e-06 5.6263e-06 5.4698e-06 5.28985e-06 5.36329e-06 5.12628e-06 4.9592e-06 4.8532e-06 4.7
4034e-06 4.59427e-06 4.47641e-06 4.38857e-06 4.26428e-06 4.21847e-06 4.04455e-06 4.01524e-06 3.83242e-06
3.7406e-06 3.6814e-06 3.61586e-06 3.46509e-06 3.41455e-06 3.34296e-06 3.25004e-06 3.22498e-06 3.12646e-06
3.05845e-06 2.95228e-06 2.97265e-06 2.85321e-06 2.72337e-06 2.75333e-06 2.65527e-06 2.59988e-06 2.40517
e-06 2.02212e-06 1.23707e-06 \n
R 0:
53: # Finish
Tcl_Eval(# Finish)
Calling cmdCompleted, Tcl_Eval returned 0/0
dumpStatusPacket:
R 0:

```

11. Open a Force Quit window.

If you run processes without invoking them from the XCode debugger, or attaching to them, they are sometimes left hanging. I keep the Force Quit window open in order to dispense with such processes.



12. Debug.

Season to taste with a smattering of web page and local documentation windows. Set breakpoints. Debug.

13. Go to 1.

## 2 Development and Coding Conventions

This section describes portability issues, and the conventions presently in use for new ATSSAL code.

### 2.1 Portability Issues

ATSSAL was first developed for Macintosh computers, with Linux a definite goal and Windows a possible goal in the future. ATSSAL's design attempts to minimize dependence on platform-specific code. Qt does most of the heavy lifting here, but there are some issues that aren't easily abstracted away.

ATSSAL's user interface components are customized, with these goals:

- I selected fonts whose size metrics are similar across platforms, to reduce the incidence of layout problems.
- Many controls resize themselves to fit their text, helping with the above issue, and also an aid when translating to other languages.
- Controls (or widgets, or whatever) are redesigned for smaller sizes in order to achieve greater densities.
- The custom controls have another important purpose: they define a unique look for ATSSAL that remains largely consistent across platforms, simplifying documentation and reducing the amount of platform-specific code. (In a previous Qt project, layout inconsistencies across platforms consumed excessive time.)

Since Macintosh applications have a single global menu bar, the bar typically contains most of the menu options regardless of the application's currently active window. This ensures that options tend to remain in the same locations in menus. Linux and Windows windows have their own menu bars. At present, I resolve the disparity by creating the menu bar in a base class and subclassing all the application windows from it, preserving consistent menus for all modeless windows.

Mac applications remain open when all windows are closed, while Linux and Windows applications do not. I address this presently by employing a "last ditch" window, a splash screen with Open and New buttons, that appears at startup or when all other windows are closed.

Keyboard shortcuts are Mac-centric now, and will need to be rethought for each platform. At some point it will probably make sense to introduce user-controlled keymaps.

All file formats are portable. I haven't thought carefully about line-ending problems under Windows. ATSSAL should read files with Windows-style line endings, but may not generate suitable line endings.

### 2.2 ATSSAL Naming Conventions

<i>Naming Convention</i>	<i>Rationale</i>
Start classnames with uppercase	Qt + Python convention
Use base-class-first class names (omitting broader base classes like PythonPublic or QWidget)	Clusters related classes in alphabetic lists, e.g. ToolPlot and ToolTable are subclasses of Tool
Use camelcase names: EmissionLine, not Emission_line.	Qt + Python convention
Start function and member function names with lowercase	Qt + Python convention
Start variable names with a lowercase	Qt + Python convention
Start member variables names with 'm'	Scope designation
Start static variable and member function names with 's'	Scope designation
Start globals with 'g'	Scope designation
Start constants with 'k'	Scope/purpose designation
Use const, not #define for constants	Better typing
No namespaces used at ATSSAL level except AT:: for values that may collide.	Name collisions not anticipated
Start enums with the initials of the enum type name, for example, EnergyUnits includes EUAngstroms and EUElectronVolts	Simple disambiguation

### 2.3 Declarations Format

This summarizes some of the conventions for C++ class declarations used in ATSAAL source code.

```

/// PrvToolTable is the ATSAAL private class.

#ifndef ToolTable_h
#define ToolTable_h

#include "Tool.h"
#include "Table.h"

class TableWidget;

class PrvToolTable : public PrvTool
{
public:
    Q_OBJECT
    /// Creates an instance of a private table tool. This is called by
    /// ToolTable, the Python public class. \param parent is the parent
    /// notebook, and \param name is the instance name, which is also the
    /// Python variable name.
    PrvToolTable(Notebook* parent, const QString& name);
    PrvToolTable(const PrvToolTable& orig);
    ~PrvToolTable() { }
    PrvToolTable& operator=(const PrvToolTable& orig);

    virtual
    refresh();

    virtual bool
    refresh();

    virtual void
    writeSubclass(QTextStream& stream, int level);
    /// Write table tool to notebook file.
    /// \param stream is the stream to write to.
    /// \param level is the starting indentation level, in number of levels
    /// (used to improve readability in XML files).
    virtual void
    readSubclass(QXmlStreamReader& reader, const Version& notebookVersion);
    /// Read table tool from notebook file.
    /// \param reader is the stream reader to read from.
    /// \param notebookVersion is the version of ATSAAL that wrote the notebook,
    /// used to maintain backward compatibility.

public slots:
    void
    tableSelectionChanged();
    /// If the TableWidget's selection changes, the signal is delivered
    /// here first, then re-emitted from here to the sidebar. This is
    /// because when the sidebar is created, the TableWidget doesn't
    /// exist yet.

protected:
    Table*
    mTable;
};

class ToolTable : public Tool
{
public:
    Q_OBJECT
    ToolTable(Notebook* parent, const QString& name);
}

```

*Use Doxygen commenting* (points to the Doxygen-style comments in the code)

*Constructors & destructors near beginning* (points to the constructor and destructor)

*Line up function and variable names* (points to the alignment of function names and variables)

*Typically public functions are followed by protected, then private functions and variables* (points to the ordering of public, protected, and private sections)

*Related classes are often placed in the same file* (points to the ToolTable class definition)

In summary:

- General order is public, protected, private. The goal is to “hide the fine print in the legal contract.”
- Constructors, destructors, operator= at or near beginning.
- Group related functions when possible.
- Group accessors.
- Group signals and slots.
- Document member variables whose function may be unclear unless they have trivial accessors that are well-documented.
- Member variables are usually private, sometimes protected.
- Include helper classes, or several simple related classes, in a single pair of source files.

## 2.4 Maintainability

- Use `tr()` (or `QObject::tr()` for non-`QObject` classes) around strings that require translation to support eventual translation to other languages.
- Don’t use arcane C++ features without good cause. When you use C++11 and C++14 features, remember that they will look like संस्कृतम् (that would be Sanskrit) to some developers, so unless it is a commonly used feature, explain your rationale. See blog posts on Qt and C++11 [here](#) and [here](#), and on C++14 [here](#). Here is Bjarne Stroustrup’s

**C++11 feature list.** Newer versions of Qt are now dependent on C++11, so at the time of this writing (Nov 2016), it is okay to use C++11, but not yet recommended to use C++14. ATLAS build procedures no longer warn when C++11 features are used.

- Test builds under all compilers used by ATLAS (typically `g++` and `clang`) to ensure that warnings do not occur.
- Assume a port to Windows in the future; minimize use of code that won't port easily.
- Use a consistent string (e.g. your initials) to tag yourself as the author of comments other developers might wish to discuss with you, such as notes about incomplete design or implementation.
- Use Doxygen commenting conventions.
- Keep ATLAS internal Nexus docs up to date.
- Avoid the standard template library. Qt's own collection classes are simpler and handle most cases, resulting in more readable code and fewer shifts in C++ "dialect." (Note that XMLRPC has to be built using different versions of the STL for inclusion in the 32-bit version of XSpecServer and the 64-bit version of ATLAS.)
- List units for physical quantities explicitly, and/or use classes like `QuEnergy`, which forces specification of physical units when constructing or accessing values.
- 4-space tabs preferred for C++ source code.
- Some source code is common across three applications (ATLAS Supervisor, ATLAS Notebook, Nexus), so build all three to verify that code changes haven't broken any of them, prior to checkins. The master build procedure builds all of them.
- Write unit tests which run during nightly builds.
- Don't incorporate new open source code without fully examining the consequences. Is this code that we can support if it is no longer externally supported? Is the code documented and written clearly enough to be supportable? Does it play well with other ATLAS components? Is it truly portable?

## 2.5 Standards

- Unicode/UTF-8. Make sure all parts of the tool chain, and all software components, support Unicode. All text files are UTF-8. (Although Python 3.3 supports Unicode natively, I have encountered problems with Unicode. These could be due to PythonQt.) Source code presently makes sparing use of UTF-8 in order to verify that it works (e.g. `const double pi=3.14159...`)
- XML. Most ATLAS files, parse tables, and small scale databases are in XML-compliant formats.
- Doxygen. Documentation uses Doxygen-style comment conventions.
- FITS.
- Python 3.3+. Write Python source code assuming Python 3.3. (Exception: source code written for XSPEC must be compatible with an older version of Python.)
- HTML-5. Most of ATLAS's HTML is not yet HTML-5 compliant, but this is a goal.
- CSV. Comma-separated value format is not a formal standard. CSV import/export conforms to the Microsoft CSV implementation: commas, no spaces, and no double quotes except around strings.
- Supported bitmap formats include JPG and PNG.
- Supported Postscript formats include EPS and PDF.
- At the time of this writing (Nov 2016), SVG support is still inconsistent among some browsers, notably `QWebView`, so we cannot rely on it yet. Hence SVG artwork is now exported as PNG files.

## 2.6 Documentation

At least for the time being, developers are also documentors.

Docs include ATLAS internals, visible only to active developers, and end user docs. Both are created with the Nexus documentation utility, in conjunction with Doxygen. Nexus maintains a table of contents and a set of pages written in HTML, with some assist for common formatting operations. This format has the advantage of allowing multiple developers to update docs simultaneously, since there is a separate text file for each topic and merges can be managed by git.

End user docs are also created using Nexus. These docs are bundled with ATLAS releases, to avoid documentation skew. The Nexus help viewer is part of the ATLAS supervisor.

## 2.7 Git Checkins

- Checkin code that is reasonably stable and free of errors and warnings.
- BE CAREFUL TO CHECKIN ALL SUBPROJECTS.
- Do a complete command line build to check for unexpected side effects of changes.

## 2.8 Python

### 2.8.1 Python Naming Conventions

Where possible, AT SAL internals that are surfaced to the Python layer are as “Pythonesque” as possible, meaning that naming for identical methods in C++ and Python often differ. For example, a method called `toDouble()` in C++ would be called `toFloat()` in Python (which uses “float” as a precision-agnostic term for floating point numbers, defaulting to double precision). Similarly, Qt collection classes and other low-level datatypes such as strings are mapped to their Python equivalents. PythonQt makes it relatively easy to control the subset of methods surfaced to Python, their names, data types, and in some cases even their abstractions.

This is accomplished primarily by use of “decorator classes,” as dubbed by PythonQt. A decorator class is a singleton class that acts as a proxy between Python and a C++ class. Typically there is a single decorator class for each Python-public C++ class, though a decorator may also act as a proxy for several cooperating classes, in order to hide complexity. Methods in the decorator class call methods in the real class(es), allowing for renaming and data conversion as needed to produce natural-feeling Python interfaces. Since PythonQt relies on signals and slots to surface methods, and this mechanism cannot be used for constructors and destructors, decorators also allow for construction and destruction.

The following paragraph on `PythonPublic` classes is obsolete. Now `Notebook` and `Tool` classes are handled with decorator classes too, a simpler solution.

Some core AT SAL classes (`Notebook`, and `Tool` subclasses) use a Python public class (a subclass of `PythonPublic`) which has a private `prv()` pointer to the real C++ class. For example, Python public class `ToolText` contains methods surfaced to Python, and passes those calls on to an instance to `PrvToolText`, which implements the class. The instance of `ToolText` owns the instance of `PrvToolText`. (In retrospect, decorators would probably have been a better solution.) The difference is that there is a single decorator instance *per class*, while there is a single `PythonPublic` class *per instance*. This does not typically impact storage requirements, because the `PythonPublic` classes usually lack member variables, but the `PythonPublic` class model is a bit more awkward, since the `PythonPublic` instance is the “primary” pointer, and must reference the internal class with `prv()`.

### 2.8.2 Surfacing AT SAL to Python

The most important challenge here is supporting access that is reasonably safe (i.e. unlikely to crash AT SAL if misused), and backward-compatible as AT SAL’s internals evolve. As a general rule, creation/deletion of AT SAL objects are not (yet) permitted, only accessors to existing objects. Object ownership is therefore managed by AT SAL.

To minimize the risk of painting ourselves into a corner, review all Python access with other project members before releases.

### 2.8.3 Python Source Code

Use spaces instead of tabs in source files. This is a Python convention targeted at making user code more easily exchanged.

### 2.8.4 Documentation of AT SAL-surfaced Python

Doxygen won’t generate Python docs for AT SAL-surfaced code in Python-native form, so these docs are created and maintained manually, using Nexus. Doxygen can be used for documenting Python source code.

### 2.8.5 In summary:

- As a rule, use a single decorator per Python-public C++ class, or group of related classes.
- Use natural Python names.
- Perform any necessary data conversion.
- Document Python-to-C++ interfaces by hand (i.e. using Nexus).

## 2.9 C++11 Features

This is reprinted verbatim from Bjarne Stroustrup’s [C++11 FAQ](#). At the present time (Jan 2017), C++11 is the highest level of C++ features in use by AT SAL.

## 2.9.1 \_\_cplusplus

In C++11 the macro `__cplusplus` will be set to a value that differs from (is greater than) the current **199711L**.

## 2.9.2 auto—deduction of a type from an initializer

Consider

```
auto x = 7;
```

Here `x` will have the type `int` because that's the type of its initializer. In general, we can write

```
auto x = expression;
```

and the type of `x` will be the type of the value computed from "expression".

The use of `auto` to deduce the type of a variable from its initializer is obviously most useful when that type is either hard to know exactly or hard to write. Consider:

```
template<class T> void printall(const vector<T>& v)
{
    for (auto p = v.begin(); p!=v.end(); ++p)
        cout << *p << "\n";
}
```

In C++98, we'd have to write

```
template<class T> void printall(const vector<T>& v)
{
    for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
        cout << *p << "\n";
}
```

When the type of a variable depends critically on template argument it can be really hard to write code without `auto`. For example:

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu)
{
    // ...
    auto tmp = vt[i]*vu[i];
    // ...
}
```

The type of `tmp` should be what you get from multiplying a `T` by a `U`, but exactly what that is can be hard for the human reader to figure out, but of course the compiler knows once it has figured out what particular `T` and `U` it is dealing with.

The `auto` feature has the distinction to be the earliest to be suggested and implemented: I had it working in my Cfront implementation in early 1984, but was forced to take it out because of C compatibility problems. Those compatibility problems disappeared when C++98 and C99 accepted the removal of "implicit `int`"; that is, both languages require every variable and function to be defined with an explicit type. The old meaning of `auto` ("this is a local variable") is now illegal. Several committee members trawled through millions of lines of code finding only a handful of uses -- and most of those were in test suites or appeared to be bugs.

Being primarily a facility to simplify notation in code, `auto` does not affect the standard library specification.

See also

- the C++ draft section 7.1.6.2, 7.1.6.4, 8.3.5 (for return types)
- [N1984=06-0054] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: [Deducing the type of variable from its initializer expression \(revision 4\)](#).

## 2.9.3 Range-for statement

A range for statement allows you to iterate through a "range", which is anything you can iterate through like an STL-sequence defined by a `begin()` and `end()`. All standard containers can be used as a range, as can a `std::string`, an initializer list, an array, and anything for which you define `begin()` and `end()`, e.g. an `istream`. For example:

```
void f(vector<double>& v)
{
    for (auto x : v) cout << x << '\n';
    for (auto& x : v) ++x; // using a reference to allow us to change the value
}
```

You can read that as "for all `x` in `v`" going through starting with `v.begin()` and iterating to `v.end()`. Another example:

```
for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';
```

The `begin()` (and `end()`) can be a member to be called `x.begin()` or a free-standing function to be called `begin(x)`. The member version takes precedence.

See also

- the C++ draft section 6.5.4 (note: changed not to use concepts)
- [N2243==07-0103] Thorsten Ottosen: [Wording for range-based for-loop \(revision 2\)](#).
- [N3257=11-0027] Jonathan Wakely and Bjarne Stroustrup: [Range-based for statements and ADL](#) (Option 5 was chosen).

## 2.9.4 Right-angle brackets

Consider

```
list<vector<string>> lvs;
```

In C++98 this is a syntax error because there is no space between the two `>`s. C++11 recognizes such two `>`s as a correct termination of two template argument lists.

Why was this ever a problem? A compiler front-end is organized parses/stages. This is about the simplest model:

- lexical analysis (make up tokens from characters)
- syntax analysis (check the grammar)
- type checking (find the type of names and expressions)

These stages are in theory and sometimes in practice strictly separate, so the lexical analyzer that determines that `>>` is a token (usually meaning right-shift or input) has no idea of its meaning; in particular, it has no idea of templates or nested template argument lists. However, to get that example "correct" the three stages has somehow to cooperate. The key observation that led to the problem being resolved was that every C++ compiler already did understand the problem so that it could give decent error messages.

See also

- the C++ draft section ???
- [N1757==05-0017] Daveed Vandevoorde: [revised right angle brackets proposal \(revision 2\)](#).

## 2.9.5 Control of defaults: default and delete

The common idiom of "prohibiting copying" can now be expressed directly:

```
class X {
    // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```

Conversely, we can also say explicitly that we want to default copy behavior:

```
class Y {
    // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```

Being explicit about the default is redundant. However, comments about copy operations and (worse) a user explicitly defining copy operations meant to give the default behavior are not uncommon. Leaving it to the compiler to implement the default behavior is simpler, less error-prone, and often leads to better object code.

The "default" mechanism can be used for any function that has a default. The "delete" mechanism can be used for any function. For example, we can eliminate an undesired conversion like this:

```
struct Z {
    // ...

    Z(long long); // can initialize with an long long
    Z(long) = delete; // but not anything less
};
```

See also

- the C++ draft section ???
- [N1717==04-0157] Francis Glassborow and Lois Goldthwaite: [explicit class and default definitions](#) (an early proposal).
- Bjarne Stroustrup: [Control of class defaults](#) (a dead end).
- [N2326==07-0186] Lawrence Crowl: [Defaulted and Deleted Functions](#).
- [N3174=100164] B. Stroustrup: [To move or not to move](#). An analysis of problems related to generated copy and move operations. Approved.

## 2.9.6 control of defaults: move and copy

By default, a class has 5 operations:

- copy assignment
- copy constructor
- move assignment
- move constructor
- destructor

If you declare any of those you must consider all and explicitly define or default the ones you want. Think of copying, moving, and destruction as closely related operations, rather than individual operations that you can freely mix and match - you can specify arbitrary combinations, but only a few combinations make sense semantically.

If any move, copy, or destructor is explicitly specified (declared, defined, `=default`, or `=delete`) by the user, no move is generated by default. If any move, copy, or destructor is explicitly specified (declared, defined, `=default`, or `=delete`) by the user, any undeclared copy operations are generated by default, but this is deprecated, so don't rely on that. For example:

```
class X1 {
    X1& operator=(const X1&) = delete; // Disallow copying
};
```

This implicitly also disallows moving of `X1s`. Copy initialization is allowed, but deprecated.

```
class X2 {
    X2& operator=(const X2&) = delete;
};
```

This implicitly also disallows moving of **X2s**. Copy initialization is allowed, but deprecated.

```
class X3 {
    X3& operator=(X3&&) = delete; // Disallow moving
};
```

This implicitly also disallows copying of **X3s**.

```
class X4 {
    ~X4() = delete; // Disallow destruction
};
```

This implicitly also disallows moving of **X4s**. Copying is allowed, but deprecated.

I strongly recommend that if you declare one of these five function, you explicitly declare all. For example:

```
template<class T>
class Handle {
    T* p;
public:
    Handle(T* pp) : p{pp} {}
    ~Handle() { delete p; } // user-defined destructor: no implicit copy or move

    Handle(Handle&& h) :p{h.p} { h.p=nullptr; } // transfer ownership
    Handle& operator=(Handle&& h) { delete p; p=h.p; h.p=nullptr; return *this; } // transfer ownership

    Handle(const Handle&) = delete; // no copy
    Handle& operator=(const Handle&) = delete;

    // ...
};
```

See also

- the C++ draft section ???
- [N2326==07-0186] Lawrence Crowl: [Defaulted and Deleted Functions](#).
- [N3174=100164] B. Stroustrup: [To move or not to move](#). An analysis of problems related to generated copy and move operations. Approved.

## 2.9.7 Enum class—scoped and strongly typed enums

The **enum classes** ("new enums", "strong enums") address three problems with traditional C++ enumerations:

- conventional **enums** implicitly convert to **int**, causing errors when someone does not want an enumeration to act as an integer.
- conventional **enums** export their enumerators to the surrounding scope, causing name clashes.
- the underlying type of an **enum** cannot be specified, causing confusion, compatibility problems, and makes forward declaration impossible.

**enum class** ("strong enums") are strongly typed and scoped:

```
enum Alert { green, yellow, orange, red }; // traditional enum

enum class Color { red, blue }; // scoped and strongly typed enum
                                // no export of enumerator names into enclosing scope
                                // no implicit conversion to int

enum class TrafficLight { red, yellow, green };

Alert a = 7; // error (as ever in C++)
Color c = 7; // error: no int->Color conversion
```

```

int a2 = red;           // ok: Alert->int conversion
int a3 = Alert::red;   // error in C++98; ok in C++11
int a4 = blue;         // error: blue not in scope
int a5 = Color::blue;  // error: not Color->int conversion

Color a6 = Color::blue; // ok

```

As shown, traditional **enums** work as usual, but you can now optionally qualify with the **enum's** name.

The new enums are "enum class" because they combine aspects of traditional enumerations (names values) with aspects of classes (scoped members and absence of conversions).

Being able to specify the underlying type allow simpler interoperability and guaranteed sizes of enumerations:

```

enum class Color : char { red, blue }; // compact representation

enum class TrafficLight { red, yellow, green }; // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFFU }; // how big is an E?
// (whatever the old rules say;
// i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFFFU }; // now we can be specific

```

It also enables forward declaration of **enums**:

```

enum class Color_code : char; // (forward) declaration
void foobar(Color_code* p); // use of forward declaration
// ...
enum class Color_code : char { red, yellow, green, blue }; // definition

```

The underlying type must be one of the signed or unsigned integer types; the default is **int**.

In the standard library, enum classes are used

- For mapping systems specific error codes: In `<system_error>`: `enum class errc`;
- For pointer safety indicators: In `<memory>`: `enum class pointer_safety { relaxed, preferred, strict }`;
- For I/O stream errors: In `<iosfwd>`: `enum class io_errc { stream = 1 }`;
- For asynchronous communications error handling: In `<future>`: `enum class future_errc { broken_promise, future_already_retrieved, promise_already_satisfied }`;

Several of these have operators, such as `==` defined.

See also

- the C++ draft section 7.2
- [N1513=03-0096] David E. Miller: [Improving Enumeration Types](#) (original enum proposal).
- [N2347 = J16/07-0207] David E. Miller, Herb Sutter, and Bjarne Stroustrup: [Strongly Typed Enums \(revision 3\)](#).
- [N2499=08-0009] Alberto Ganesh Barbati: [Forward declaration of enumerations](#).

## 2.9.8 constexpr—generalized and guaranteed constant expressions

The **constexpr** mechanism

- provides more general constant expressions
- allows constant expressions involving user-defined types
- provides a way to guarantee that an initialization is done at compile time

Consider

```

enum Flags { good=0, fail=1, bad=2, eof=4 };

```

```
constexpr int operator|(Flags f1, Flags f2) { return Flags(int(f1)|int(f2)); }

void f(Flags x)
{
    switch (x) {
    case bad:          /* ... */ break;
    case eof:          /* ... */ break;
    case bad|eof:      /* ... */ break;
    default:           /* ... */ break;
    }
}
```

Here **constexpr** says that the function must be of a simple form so that it can be evaluated at compile time if given constant expressions arguments.

In addition to be able to evaluate expressions at compile time, we want to be able to *require* expressions to be evaluated at compile time; **constexpr** in front of a variable definition does that (and implies **const**):

```
constexpr int x1 = bad|eof; // ok

void f(Flags f3)
{
    constexpr int x2 = bad|f3; // error: can't evaluate at compile time
    int x3 = bad|f3; // ok
}
```

Typically we want the compile-time evaluation guarantee for global or namespace objects, often for objects we want to place in read-only storage.

This also works for objects for which the constructors are simple enough to be **constexpr** and expressions involving such objects:

```
struct Point {
    int x,y;
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point origo(0,0);
constexpr int z = origo.x;

constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
constexpr int x = a[1].x; // x becomes 1
```

Please note that **constexpr** is not a general purpose replacement for **const** (or vice versa):

- **const**'s primary function is to express the idea that an object is not modified through an interface (even though the object may very well be modified through other interfaces). It just so happens that declaring an object **const** provides excellent optimization opportunities for the compiler. In particular, if an object is declared **const** and its address isn't taken, a compiler is often able to evaluate its initializer at compile time (though that's not guaranteed) and keep that object in its tables rather than emitting it into the generated code.
- **constexpr**'s primary function is to extend the range of what can be computed at compile time, making such computation type safe. Objects declared **constexpr** have their initializer evaluated at compile time; they are basically values kept in the compiler's tables and only emitted into the generated code if needed.

See also

- the C++ draft 3.6.2 Initialization of non-local objects, 3.9 Types [12], 5.19 Constant expressions, 7.1.5 The constexpr specifier
- [N1521=03-0104] Gabriel Dos Reis: [Generalized Constant Expressions](#) (original proposal).
- [N2235=07-0095] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer: [Generalized Constant Expressions -- Revision 5](#).

## 2.9.9 decltype—the type of an expression

**decltype(E)** is the type ("declared type") of the name or expression **E** and can be used in declarations. For example:

```
void f(const vector<int>& a, vector<float>& b)
{
    typedef decltype(a[0]*b[0]) Tmp;
    for (int i=0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
        // ...
    }
    // ...
}
```

This notion has been popular in generic programming under the label "typeof" for a long time, but the **typeof** implementations in actual use were incomplete and incompatible, so the standard version is named **decltype**.

If you just need the type for a variable that you are about to initialize [auto](#) is often a simpler choice. You really need **decltype** if you need a type for something that is not a variable, such as a [return type](#).

See also

- the C++ draft 7.1.6.2 Simple type specifiers
- [Stro2] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002. (original suggestion).
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: [Decltype and auto](#) (original proposal).
- [N2343=07-0203] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: [Decltype \(revision 7\): proposed wording](#).

## 2.9.10 Initializer lists

Consider

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string,string>> languages = {
    {"Nygaard", "Simula"}, {"Richards", "BCPL"}, {"Ritchie", "C"}
};
map<vector<string>,vector<int>> years = {
    { {"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000} },
    { {"Martin", "Ritchards"}, {1982, 2003, 2007} },
    { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

Initializer lists are not just for arrays any more. The mechanism for accepting a **{}**-list is a function (often a constructor) accepting an argument of type **std::initializer\_list<T>**. For example:

```
void f(initializer_list<int>);
f({1,2});
f({23,345,4567,56789});
f({}); // the empty list
f{1,2}; // error: function call ( ) missing

years.insert({{"Bjarne", "Stroustrup"},{1950, 1975, 1985}});
```

The initializer list can be of arbitrary length, but must be homogeneous (all elements must be of a the template argument type, **T**, or convertible to **T**).

A container might implement an initializer-list constructor like this:

```

template<class E> class vector {
public:
    vector (std::initializer_list<E> s) // initializer-list constructor
    {
        reserve(s.size()); // get the right amount of space
        uninitialized_copy(s.begin(), s.end(), elem); // initialize elements (in elem[0:s.size()))
        sz = s.size(); // set vector size
    }

    // ... as before ...
};

```

The distinction between direct initialization and copy initialization is maintained for `{}`-initialization, but becomes relevant less frequently because of `{}`-initialization. For example, `std::vector` has an **explicit** constructor from `int` and an initializer-list constructor:

```

vector<double> v1(7); // ok: v1 has 7 elements
v1 = 9; // error: no conversion from int to vector
vector<double> v2 = 9; // error: no conversion from int to vector

void f(const vector<double>&);
f(9); // error: no conversion from int to vector

vector<double> v1{7}; // ok: v1 has 1 element (with its value 7.0)
v1 = {9}; // ok v1 now has 1 element (with its value 9.0)
vector<double> v2 = {9}; // ok: v2 has 1 element (with its value 9.0)
f({9}); // ok: f is called with the list { 9 }

vector<vector<double>> vs = {
    vector<double>(10), // ok: explicit construction (10 elements)
    vector<double>{10}, // ok: explicit construction (1 element with the value 10.0)
    10 // error: vector's constructor is explicit
};

```

The function can access the **initializer\_list** as an immutable sequence. For example:

```

void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p) cout << *p << "\n";
}

```

A constructor that takes a single argument of type `std::initializer_list` is called an initializer-list constructor.

The standard library containers, **string**, and **regex** have initializer-list constructors, assignment, etc. An initializer-list can be used as a range, e.g. in a [range for statement](#)

The initializer lists are part of the scheme for [uniform and general initialization](#).

See also

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [N1890=05-0150 ] Bjarne Stroustrup and Gabriel Dos Reis: [Initialization and initializers](#) (an overview of initialization-related problems with suggested solutions).
- [N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis: [Initializer lists](#).
- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis : [Initializer lists \(Rev. 3\)](#) .
- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde: [Initializer Lists -- Alternative Mechanism and Rationale \(v. 2\)](#) (final proposal).

## 2.9.11 Preventing narrowing

The problem: C and C++ implicitly truncates:

```
int x = 7.3; // Ouch!
void f(int);
f(7.3); // Ouch!
```

However, in C++11, `{}` initialization doesn't narrow:

```
int x0 {7.3}; // error: narrowing
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d}; // error: narrowing (double to int)
char x3{7}; // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 }; // error: double to int narrowing
```

The way C++11 avoids a lot of incompatibilities is by relying on the actual values of initializers (such as 7 in the example above) when it can (and not just type) when deciding what is a narrowing conversion. If a value can be represented exactly as the target type, the conversion is not narrowing.

```
char c1{7}; // OK: 7 is an int, but it fits in a char
char c2{77777}; // error: narrowing (assuming 8-bit chars)
```

Note that floating-point to integer conversions are always considered narrowing -- even 7.0 to 7.

See also

- the C++ draft section 8.5.4.
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: [Initialization and initializers](#) (an overview of initialization-related problems with suggested solutions).
- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis: [Initializer lists \(Rev. 3\)](#).
- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde: [Initializer Lists - Alternative Mechanism and Rationale \(v. 2\)](#) (primarily on "explicit").

## 2.9.12 Delegating constructors

In C++98, if you want two constructors to do the same thing, repeat yourself or call "an `init()` function." For example:

```
class X {
    int a;
    void validate(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
    X(string s) { int x = lexical_cast<int>(s); validate(x); }
    // ...
};
```

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. So, in C++11, we can define one constructor in terms of another:

```
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
    X() :X{42} { }
    X(string s) :X{lexical_cast<int>(s)} { }
    // ...
};
```

See also

- the C++ draft section 12.6.2
- N1986=06-0056 Herb Sutter and Francis Glassborow: [Delegating Constructors \(revision 3\)](#).

## 2.9.13 In-class member initializers

In C++98, only static const members of integral types can be initialized in-class, and the initializer has to be a constant expression. These restrictions ensure that we can do the initialization at compile-time. For example:

```
int var = 7;

class X {
    static const int m1 = 7;           // ok
    const int m2 = 7;                 // error: not static
    static int m3 = 7;                // error: not const
    static const int m4 = var;        // error: initializer not constant expression
    static const string m5 = "odd";  // error: not integral type
    // ...
};
```

The basic idea for C++11 is to allow a non-static data member to be initialized where it is declared (in its class). A constructor can then use the initializer when run-time initialization is needed. Consider:

```
class A {
public:
    int a = 7;
};
```

This is equivalent to:

```
class A {
public:
    int a;
    A() : a(7) {}
};
```

This saves a bit of typing, but the real benefits come in classes with multiple constructors. Often, all constructors use a common initializer for a member:

```
class A {
public:
    A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor run") {}
    int a, b;
private:
    HashingFunction hash_algorithm; // Cryptographic hash to be applied to all A instances
    std::string s;                 // String indicating state in object lifecycle
};
```

The fact that `hash_algorithm` and `s` each has a single default is lost in the mess of code and could easily become a problem during maintenance. Instead, we can factor out the initialization of the data members:

```
class A {
public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(D d) : a(7), b(g(d)) {}
    int a, b;
private:
    HashingFunction hash_algorithm{"MD5"}; // Cryptographic hash to be applied to all A instances
    std::string s{"Constructor run"};     // String indicating state in object lifecycle
};
```

If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```
class A {
public:
    A() {}
    A(int a_val) : a(a_val) {}
    A(D d) : b(g(d)) {}
    int a = 7;
    int b = 5;
private:
    HashingFunction hash_algorithm{"MD5"}; // Cryptographic hash to be applied to all A instances
    std::string s{"Constructor run"};     // String indicating state in object lifecycle
};
```

See also

- the C++ draft section "one or two words all over the place"; see proposal.
- [N2628=08-0138] Michael Spertus and Bill Seymour: [Non-static data member initializers](#).

## 2.9.14 Inherited constructors

People sometimes are confused about the fact that ordinary scope rules apply to class members. In particular, a member of a base class is not in the same scope as a member of a derived class:

```
struct B {
    void f(double);
};

struct D : B {
    void f(int);
};

B b;    b.f(4.5); // fine
D d;    d.f(4.5); // surprise: calls f(int) with argument 4
```

In C++98, we can "lift" a set of overloaded functions from a base class into a derived class:

```
struct B {
    void f(double);
};

struct D : B {
    using B::f; // bring all f()s from B into scope
    void f(int); // add a new f()
};

B b;    b.f(4.5); // fine
D d;    d.f(4.5); // fine: calls D::f(double) which is B::f(double)
```

I have said that "Little more than a historical accident prevents using this to work for a constructor as well as for an ordinary member function." C++11 provides that facility:

```
class Derived : public Base {
public:
    using Base::f; // lift Base's f into Derived's scope -- works in C++98
    void f(char); // provide a new f
    void f(int); // prefer this f to Base::f(int)

    using Base::Base; // lift Base constructors Derived's scope -- C++11 only
```

```

Derived(char);    // provide a new constructor
Derived(int);    // prefer this constructor to Base::Base(int)
// ...
};

```

If you so choose, you can still shoot yourself in the foot by inheriting constructors in a derived class in which you define new member variables needing initialization:

```

struct B1 {
    B1(int) { }
};

struct D1 : B1 {
    using B1::B1; // implicitly declares D1(int)
    int x;
};

void test()
{
    D1 d(6); // Oops: d.x is not initialized
    D1 e; // error: D1 has no default constructor
}

```

You might remove the bullet from your foot by using a [member-initializer](#):

```

struct D1 : B1 {
    using B1::B1; // implicitly declares D1(int)
    int x{0}; // note: x is initialized
};

void test()
{
    D1 d(6); // d.x is zero
}

```

See also

- the C++ draft section 12.9.
- [N1890=05-0150 ] Bjarne Stroustrup and Gabriel Dos Reis: [Initialization and initializers](#) (an overview of initialization-related problems with suggested solutions).
- [N1898=05-0158 ] Michel Michaud and Michael Wong: [Forwarding and inherited constructors](#) .
- [N2512=08-0022] Alisdair Meredith, Michael Wong, Jens Maurer: [Inheriting Constructors \(revision 4\)](#).

## 2.9.15 Static (compile-time) assertions—`static_assert`

A static (compile time) assertion consists of a constant expression and a string literal:

```

static_assert(expression, string);

```

The compiler evaluates the expression and writes the string as an error message if the expression is false (i.e., if the assertion failed). For example:

```

static_assert(sizeof(long)>=8, "64-bit code generation required for this library.");
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y), "unexpected padding in S");

```

A `static_assert` can be useful to make assumptions about a program and its treatment by a compiler explicit. Note that since `static_assert` is evaluated at compile time, it cannot be used to check assumptions that depends on run-time values. For example:

```
int f(int* p, int n)
{
    static_assert(p==0, "p is not null"); // error: static_assert() expression not a constant expression
    // ...
}
```

(instead, test and throw an exception in case of failure).

See also

- the C++ draft 7 [4].
- [N1381==02-0039] Robert Klarer and John Maddock: [Proposal to Add Static Assertions to the Core Language](#).
- [N1720==04-0160] Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant: [Proposal to Add Static Assertions to the Core Language \(Revision 3\)](#).

## 2.9.16 long long—a longer integer

An integer that's at least 64 bits long. For example:

```
long long x = 9223372036854775807LL;
```

No, there are no **long long longs** nor can **long** be spelled **short long long**.

See also

- the C++ draft ???.
- [05-0071==N1811] J. Stephen Adameczyk: [Adding the long long type to C++ \(Revision 3\)](#).

## 2.9.17 nullptr—a null pointer literal

**nullptr** is a literal denoting the null pointer; it is not an integer:

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0; // 0 still works and p==p2

void f(int);
void f(char*);

f(0); // call f(int)
f(nullptr); // call f(char*)

void g(int);
g(nullptr); // error: nullptr is not an int
int i = nullptr; // error nullptr is not an int
```

See also

- the C++ draft section ???
- [N1488==/03-0071] Herb Sutter and Bjarne Stroustrup: [A name for the null pointer: nullptr](#).
- [N2214 = 07-0074] Herb Sutter and Bjarne Stroustrup: [A name for the null pointer: nullptr \(revision 4\)](#).

## 2.9.18 Suffix return type syntax

Consider:

```
template<class T, class U>
??? mul(T x, U y)
{
    return x*y;
}
```

What can we write as the return type? It's "the type of  $x*y$ ", of course, but how can we say that? First idea, use [decltype](#):

```
template<class T, class U>
decltype(x*y) mul(T x, U y) // scope problem!
{
    return x*y;
}
```

That won't work because  $x$  and  $y$  are not in scope. However, we can write:

```
template<class T, class U>
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y) // ugly! and error prone
{
    return x*y;
}
```

However, calling that "not pretty" would be overly polite.

The solution is put the return type where it belongs, after the arguments:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y)
{
    return x*y;
}
```

We use the notation **auto** to mean "return type to be deduced or specified later."

The suffix syntax is not primarily about templates and type deduction, it is really about scope.

```
struct List {
    struct Link { /* ... */ };
    Link* erase(Link* p); // remove p and return the link before p
    // ...
};

List::Link* List::erase(Link* p) { /* ... */ }
```

The first **List::** is necessary only because the scope of **List** isn't entered until the second **List::**. Better:

```
auto List::erase(Link* p) -> Link* { /* ... */ }
```

Now neither **Link** needs explicit qualification.

See also

- the C++ draft section ???
- [Stro2] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002.
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: [Decltype and auto](#).
- [N2445=07-0315] Jason Merrill: [New Function Declarator Syntax Wording](#).
- [N2825=09-0015] Lawrence Crowl and Alisdair Meredith: [Unified Function Syntax](#).

## 2.9.19 template alias (formerly known as "template typedef")

How can we make a template that's "just like another template" but possibly with a couple of template arguments specified (bound)? Consider:

```
template<class T>
using Vec = std::vector<T, My_alloc<T>>; // standard vector using my allocator
```

```
Vec<int> fib = { 1, 2, 3, 5, 8, 13 }; // allocates elements using My_alloc

vector<int,My_alloc<int>> verbose = fib; // verbose and fib are of the same type
```

The keyword **using** is used to get a linear notation "name followed by what it refers to." We tried with the conventional and convoluted **typedef** solution, but never managed to get a complete and coherent solution until we settled on a less obscure syntax.

Specialization works (you can alias a set of specializations but you cannot specialize an alias) For example:

```
template<int>
struct int_exact_traits { // idea: int_exact_trait<N>::type is a type with exactly N bits
    typedef int type;
};

template<>
struct int_exact_traits<8> {
    typedef char type;
};

template<>
struct int_exact_traits<16> {
    typedef char[2] type;
};

// ...

template<int N>
using int_exact = typename int_exact_traits<N>::type; // define alias for convenient notation

int_exact<8> a = 7; // int_exact<8> is an int with 8 bits
```

In addition to being important in connection with templates, type aliases can also be used as a different (and IMO better) syntax for ordinary type aliases:

```
typedef void (*PFD)(double); // C style
using PF = void (*)(double); // using plus C-style type
using P = [](double)->void; // using plus suffix return type
```

See also

- the C++ draft: 14.6.7 Template aliases; 7.1.3 The typedef specifier
- [N1489=03-0072] Bjarne Stroustrup and Gabriel Dos Reis: [Templates aliases for C++](#).
- [N2258=07-0118] Gabriel Dos Reis and Bjarne Stroustrup: [Templates Aliases \(Revision 3\)](#) (final proposal).

## 2.9.20 Variadic Templates

Problems to be solved:

- How to construct a class with 1, 2, 3, 4, 5, 6, 7, 8, 9, or ... initializers?
- How to avoid constructing an object out of parts and then copying the result?
- How to construct a tuple?

The last question is the key: Think tuple! If you can make and access general tuples the rest will follow.

Here is an example (from "A brief introduction to Variadic templates" (see references)) implementing a general, type-safe, **printf()**. It would probably be better to use **boost::format**, but consider:

```
const string pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s).\n";
printf(m,pi,3.14159,"Indiana");
```

The simplest case of `printf()` is when there are no arguments except the format string, so we'll handle that first:

```
void printf(const char* s)
{
    while (s && *s) {
        if (*s=='%' && *++s!='%') // make sure that there wasn't meant to be more arguments
            // %% represents plain % in a format string
            throw runtime_error("invalid format: missing arguments");
        std::cout << *s++;
    }
}
```

That done, we must handle `printf()` with more arguments:

```
template<typename T, typename... Args> // note the "..."
void printf(const char* s, T value, Args... args) // note the "..."
{
    while (s && *s) {
        if (*s=='%' && *++s!='%') { // a format specifier (ignore which one it is)
            std::cout << value; // use first non-format argument
            return printf(++s, args...); // "peel off" first argument
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

This code simply “peels off” the first non-format argument and then calls itself recursively. When there are no more non-format arguments, it calls the first (simpler) `printf()` (above). This is rather standard functional programming done at compile time. Note how the overloading of `<<` replaces the use of the (possibly erroneous) “hint” in the format specifier.

The `Args...` defines what is called a “parameter pack.” That's basically a sequence of (type/value) pairs from which you can “peel off” arguments starting with the first. When `printf()` is called with one argument, the first definition (`printf(const char*)`) is chosen. When `printf()` is called with two or more arguments, the second definition (`printf(const char*, T value, Args... args)`) is chosen, with the first argument as `s`, the second as `value`, and the rest (if any) bundled into the parameter pack `args` for later use. In the call

```
printf(++s, args...);
```

The parameter pack `args` is expanded so that the next argument can now be selected as `value`. This carries on until `args` is empty (so that the first `printf()` is called).

If you are familiar with functional programming, you should find this an unusual notation for a pretty standard technique. If not, here are some small technical examples that might help. First we can declare and use a simple variadic template function (just like `printf()` above):

```
template<class ... Types>
void f(Types ... args); // variadic template function
// (i.e. a function that can take an arbitrary number of arguments of arbitrary types)
f(); // OK: args contains no arguments
f(1); // OK: args contains one argument: int
f(2, 1.0); // OK: args contains two arguments: int and double
```

We can build a variadic type:

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
```

```

: private tuple<Tail...> { // here is the recursion
  // Basically, a tuple stores its head (first (type/value) pair
  // and derives from the tuple of its tail (the rest of the (type/value) pairs.
  // Note that the type is encoded in the type, not stored as data
typedef tuple<Tail...> inherited;
public:
tuple() { } // default: the empty tuple

// Construct tuple from separate arguments:
tuple(typename add_const_reference<Head>::type v, typename add_const_reference<Tail>::type... vtail)
: m_head(v), inherited(vtail...) { }

// Construct tuple from another tuple:
template<typename... VValues>
tuple(const tuple<VValues...>& other)
: m_head(other.head()), inherited(other.tail()) { }

template<typename... VValues>
tuple& operator=(const tuple<VValues...>& other) // assignment
{
  m_head = other.head();
  tail() = other.tail();
  return *this;
}

typename add_reference<Head>::type head() { return m_head; }
typename add_reference<const Head>::type head() const { return m_head; }

inherited& tail() { return *this; }
const inherited& tail() const { return *this; }
protected:
  Head m_head;
}

```

Given that definition, we can make tuples (and copy and manipulate them):

```

tuple<string,vector<int>,double> tt("hello",{1,2,3,4},1.2);
string h = tt.head(); // "hello"
tuple<vector<int>,double> t2 = tt.tail(); // {{1,2,3,4},1.2};

```

It can get a bit tedious to mention all of those types, so often, we deduce them from argument types, e.g. using the standard library **make\_tuple()**:

```

template<class... Types>
tuple<Types...> make_tuple(Types&&... t) // this definition is somewhat simplified (see standard 20.5.2.2)
{
  return tuple<Types...>(t...);
}

string s = "Hello";
vector<int> v = {1,22,3,4,5};
auto x = make_tuple(s,v,1.2);

```

See also:

- Standard 14.6.3 Variadic templates
- [N2151==07-0011] D. Gregor, J. Jarvi: [Variadic Templates for the C++0x Standard Library](#).
- [N2080==06-0150] D. Gregor, J. Jarvi, G. Powell: [Variadic Templates \(Revision 3\)](#).
- [N2087==06-0157] Douglas Gregor: [A Brief Introduction to Variadic Templates](#).
- [N2772==08-0282] L. Joly, R. Klarer: [Variadic functions: Variadic templates or initializer lists? -- Revision 1](#).
- [N2551==08-0061] Sylvain Pion: [A Variadic std::min\(T, ...\) for the C++ Standard Library \(Revision 2\)](#).
- Anthony Williams: [An Introduction to Variadic Templates in C++0x](#). DevX.com, May 2009.

## 2.9.21 Uniform initialization syntax and semantics

C++ offers several ways of initializing an object depending on its type and the initialization context. When misused, the error can be surprising and the error messages obscure. Consider:

```
string a[] = { "foo", " bar" };           // ok: initialize array variable
vector<string> v = { "foo", " bar" };     // error: initializer list for non-aggregate vector
void f(string a[]);
f( { "foo", " bar" } );                  // syntax error: block as argument
```

and

```
int a = 2;                               // "assignment style"
int aa[] = { 2, 3 };                     // assignment style with list
complex z(1,2);                          // "functional style" initialization
x = Ptr(y);                              // "functional style" for conversion/cast/construction
```

and

```
int a(1); // variable definition
int b(); // function declaration
int b(foo); // variable definition or function declaration
```

It can be hard to remember the rules for initialization and to choose the best way.

The C++11 solution is to allow `{}`-initializer lists for all initialization:

```
X x1 = X{1,2};
X x2 = {1,2}; // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
    D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
    int a[3];
    S(int x, int y, int z) :a{x,y,z} { /* ... */ }; // solution to old problem
};
```

Importantly, `X{a}` constructs the same value in every context, so that `{}`-initialization gives the same result in all places where it is legal. For example:

```
X x{a};
X* p = new X{a};
z = X{a}; // use as cast
f({a}); // function argument (of type X)
return {a}; // function return value (function returning X)
```

See also

- the C++ draft section ???
- [N2215==07-0075] Bjarne Stroustrup and Gabriel Dos Reis: [Initializer lists \(Rev. 3\)](#) .
- [N2640==08-0150] Jason Merrill and Daveed Vandevor: [Initializer Lists -- Alternative Mechanism and Rationale \(v. 2\)](#) (final proposal).

## 2.9.22 Rvalue references

The distinction between lvalues (what can be used on the left-hand side of an assignment) and rvalues (what can be

used on the right-hand side of an assignment) goes back to Christopher Strachey (the father of C++'s distant ancestor CPL and of denotational semantics). In C++, non-const references can bind to lvalues and const references can bind to lvalues or rvalues, but there is nothing that can bind to a non-const rvalue. That's to protect people from changing the values of temporaries that are destroyed before their new value can be used. For example:

```
void incr(int& a) { ++a; }
int i = 0;
incr(i); // i becomes 1
incr(0); // error: 0 is not an lvalue
```

If that **incr(0)** were allowed either some temporary that nobody ever saw would be incremented or - far worse - the value of 0 would become 1. The latter sounds silly, but there was actually a bug like that in early Fortran compilers that set aside a memory location to hold the value 0.

So far, so good, but consider

```
template<class T> swap(T& a, T& b) // "old style swap"
{
    T tmp(a); // now we have two copies of a
    a = b; // now we have two copies of b
    b = tmp; // now we have two copies of tmp (aka a)
}
```

If **T** is a type for which it can be expensive to copy elements, such as string and vector, swap becomes an expensive operation (for the standard library, we have specializations of string and vector **swap()** to deal with that). Note something curious: We didn't want any copies at all. We just wanted to move the values of **a**, **b**, and **tmp** around a bit.

In C++11, we can define "move constructors" and "move assignments" to move rather than copy their argument:

```
template<class T> class vector {
    // ...
    vector(const vector&); // copy constructor
    vector(vector&&); // move constructor
    vector& operator=(const vector&); // copy assignment
    vector& operator=(vector&&); // move assignment
}; // note: move constructor and move assignment takes non-const &&
// they can, and usually do, write to their argument
```

The **&&** indicates an "rvalue reference". An rvalue reference can bind to an rvalue (but not to an lvalue):

```
X a;
X f();
X& r1 = a; // bind r1 to a (an lvalue)
X& r2 = f(); // error: f() is an rvalue; can't bind

X&& rr1 = f(); // fine: bind rr1 to temporary
X&& rr2 = a; // error: bind a is an lvalue
```

The idea behind a move assignment is that instead of making a copy, it simply takes the representation from its source and replaces it with a cheap default. For example, for strings **s1=s2** using the move assignment would not make a copy of **s2**'s characters; instead, it would just let **s1** treat those characters as its own and somehow delete **s1**'s old characters (maybe by leaving them in **s2**, which presumably are just about to be destroyed).

How do we know whether it's ok to simply move from a source? We tell the compiler:

```
template<class T>
void swap(T& a, T& b) // "perfect swap" (almost)
{
    T tmp = move(a); // could invalidate a
    a = move(b); // could invalidate b
```

```

    b = move(tmp); // could invalidate tmp
}

```

**move(x)** means "you can treat **x** as an rvalue". Maybe it would have been better if **move()** had been called **rval()**, but by now **move()** has been used for years. The **move()** template function can be written in C++11 (see the "brief introduction") and uses rvalue references.

Rvalue references can also be used to provide perfect forwarding.

In the C++11 standard library, all containers are provided with move constructors and move assignment and operations that insert new elements, such as **insert()** and **push\_back()** have versions that take rvalue references. The net result is that the standard containers and algorithms quietly - without user intervention - improve in performance because they copy less.

See also

- the C++ draft section ???
- N1385 N1690 N1770 N1855 N1952
- [N2027==06-0097] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki: [A brief introduction to rvalue references](#)
- [N1377=02-0035] Howard E. Hinnant, Peter Dimov, and Dave Abrahams: [A Proposal to Add Move Semantics Support to the C++ Language](#) (original proposal).
- [N2118=06-0188] Howard Hinnant: [A Proposal to Add an Rvalue Reference to the C++ Language Proposed Wording \(Revision 3\)](#) (final proposal).

## 2.9.23 unions (generalized)

In C++98 (as in the earlier versions of C++), a member with a user-defined constructor, destructor, or assignment cannot be a member of a union:

```

union U {
    int m1;
    complex<double> m2; // error (silly): complex has constructor
    string m3; // error (not silly): string has a serious invariant
                // maintained by ctor, copy, and dtor
};

```

In particular

```

U u; // which constructor, if any?
u.m1 = 1; // assign to int member
string s = u.m3; // disaster: read from string member

```

Obviously, it's illegal to write one member and then read another but people do that nevertheless (usually by mistake).

C++11 modifies the restrictions of unions to make more member types feasible; in particular, it allows a member of types with constructors and destructors. It also adds a restriction to make the more flexible unions less error-prone by encouraging the building of discriminated unions.

Union member types are restricted:

- No virtual functions (as ever)
- No references (as ever)
- No bases (as ever)
- If a union has a member with a user-defined constructor, copy, or destructor then that special function is [deleted](#); that is, it cannot be used for an object of the union type. This is new.

For example:

```

union U1 {
    int m1;
    complex<double> m2; // ok

```

```
};

union U2 {
    int m1;
    string m3; // ok
};
```

This may look error-prone, but the new restriction helps. In particular:

```
U1 u; // ok
u.m2 = {1,2}; // ok: assign to the complex member
U2 u2; // error: the string destructor caused the U2 destructor to be deleted
U2 u3 = u2; // error: the string copy constructor caused the U2 copy constructor to be deleted
```

Basically, **U2** is useless unless you embed it in a struct that keeps track of which member (variant) is used. So, build discriminate unions, such as:

```
class Widget { // Three alternative implementations represented as a union
private:
    enum class Tag { point, number, text } type; // discriminant
    union { // representation
        point p; // point has constructor
        int i;
        string s; // string has default constructor, copy operations, and destructor
    };
    // ...
    Widget& operator=(const Widget& w) // necessary because of the string variant
    {
        if (type==Tag::text && w.type==Tag::text) {
            s = w.s; // usual string assignment
            return *this;
        }

        if (type==Tag::text) s.~string(); // destroy (explicitly!)

        switch (w.type) {
            case Tag::point: p = w.p; break; // normal copy
            case Tag::number: i = w.i; break;
            case Tag::text: new(&s)(w.s); break; // placement new
        }
        type = w.type;
        return *this;
    }
};
```

See also:

- li>the C++ draft section 9.5
- [N2544=08-0054] Alan Talbot, Lois Goldthwaite, Lawrence Crowl, and Jens Maurer: [Unrestricted unions \(Revision 2\)](#)

## 2.9.24 PODs (generalized)

A POD ("Plain Old Data") is something that can be manipulated like a C struct, e.g. copies with **memcpy()**, initializes with **memset()**, etc. In C++98 the actual definition of POD is based on a set of restrictions on the use of language features used in the definition of a struct:

```
struct S { int a; }; // S is a POD
struct SS { int a; SS(int aa) : a(aa) { } }; // SS is not a POD
struct SSS { virtual void f(); /* ... */ };
```

In C++11, S and SS are "standard layout types" (a.k.a. POD) because there is really nothing "magic" about SS: the constructor does not affect the layout (so memcpy() would be fine), only the initialization rules (memset() would be bad - not enforcing the invariant). However, SSS will still have an embedded vptr and will not be anything like "plain old data." C++11 defines POD, trivially copyable types, trivial types, and standard-layout types to deal with various technical aspects of what used to be PODs. POD is defined recursively

- If all your members and bases are PODs, you're a POD
- As usual (details in section 9 [10])
  - No virtual functions
  - No virtual bases
  - No references
  - No multiple access specifiers

The most important aspect of C++11 PODs are that adding or subtracting constructors do not affect layout or performance.

See also:

- the C++ draft section 3.9 and 9 [10]
- [N2294=07-0154] Beman Dawes: [POD's Revisited; Resolving Core Issue 568 \(Revision 4\)](#).

## 2.9.25 Raw string literals

In many cases, such as when you are writing regular expressions for the use with the standard [regex](#) library, the fact that a backslash (\) is an escape character is a real nuisance (because in regular expressions backslash is used to introduce special characters representing character classes). Consider how to write the pattern representing two words separated by a backslash (\w\\w):

```
string s = "\\w\\\\\\w"; // I hope I got that right
```

Note that the backslash character is represented as two backslashes in a regular expression. Basically, a "raw string literal" is a string literal where a backslash is just a backslash so that our example becomes:

```
string s = R"(\w\\w)"; // I'm pretty sure I got that right
```

The original proposal for raw strings presents this as a motivating example

```
"('(?:[^\\"']|\\\\".)*|\\"(?:[^\\""]|\\\\".)*\")" // Are the five backslashes correct or not?  
// Even experts become easily confused.
```

The **R"(...)"** notation is a bit more verbose than the "plain" "..." but "something more" is necessary when you don't have an escape character: How do you put a quote in a raw string? Easy, unless it is preceded by a **)**:

```
R>("quoted string") // the string is "quoted string"
```

So, how do we get the character sequence **)** into a raw string? Fortunately, that's a rare problem, but **"(...)"** is only the default delimiter pair. We can add delimiters before and after the **(...)** in **"(...)"**. For example

```
R***("quoted string containing the usual terminator (")")*** // the string is "quoted string containing t
```

The character sequence after **)** must be identical to the sequence before the **(**. This way we can cope with (almost) arbitrarily complicated patterns.

The initial **R** of a raw string can be preceded by an encoding-prefix: **u8**, **u**, **U**, or **L**. For example **u8R"(fdfdfa)"** is an UTF-8 string literal.

See

- Standard 2.13.4
- [N2053=06-0123] Beman Dawes: [Raw string literals](#). (original proposal)
- [N2442=07-0312] Lawrence Crowl and Beman Dawes: [Raw and Unicode String Literals; Unified Proposal \(Rev.](#)

- 2). (final proposal combined with the [User-defined literals](#) proposal).
- [N3077==10-0067] Jason Merrill: [Alternative approach to Raw String issues](#). (replacing [ with ();

## 2.9.26 User-defined literals

C++ provides literals for a variety of built-in types (2.14 Literals):

```
123 // int
1.2 // double
1.2F // float
'a' // char
1ULL // unsigned long long
0xD0 // hexadecimal unsigned
"as" // string
```

However, in C++98 there are no literals for user-defined types. This can be a bother and also seen as a violation of the principle that user-defined types should be supported as well as built-in types are. In particular, people have requested:

```
"Hi!"s // string, not "zero-terminated array of char"
1.2i // imaginary
123.4567891234df // decimal floating point (IBM)
101010111000101b // binary
123s // seconds
123.56km // not miles! (units)
1234567890123456789012345678901234567890x // extended-precision
```

C++11 supports “user-defined literals” through the notion of *literal operators* that map literals with a given suffix into a desired type. For example:

```
constexpr complex<double> operator "" i(long double d) // imaginary literal
{
    return {0,d}; // complex is a literal type
}

std::string operator""s (const char* p, size_t n) // std::string literal
{
    return string(p,n); // requires free store allocation
}
```

Note the use of **constexpr** to enable compile-time evaluation. Given those, we can write

```
template<class T> void f(const T&);
f("Hello"); // pass pointer to const char*
f("Hello"s); // pass (5-character) string object
f("Hello\n"s); // pass (6-character) string object

auto z = 2+1i; // complex(2,1)
```

The basic (implementation) idea is that after parsing what could be a literal, the compiler always check for a suffix. The user-defined literal mechanism simply allows the user to specify a new suffix and what is to be done with the literal before it. It is not possible to redefine the meaning of a built-in literal suffix or augment the syntax of literals. A literal operator can request to get its (preceding) literal passed “cooked” (with the value it would have had if the new suffix hadn’t been defined) or “uncooked” (as a string).

To get an “uncooked” string, simply request a single **const char\*** argument:

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}
```

```
void f(Bignum);
f(12345678901234567890123456789012345678901234567890x);
```

Here the C-style string "12345678901234567890123456789012345678901234567890" is passed to `operator"" x()`. Note that we did not explicitly put those digits into a string.

There are four kinds of literals that can be suffixed to make a user-defined literal

- integer literal: accepted by a literal operator taking a single **unsigned long long** or **const char\*** argument.
- floating-point literal: accepted by a literal operator taking a single **long double** or **const char\*** argument.
- string literal: accepted by a literal operator taking a pair of (**const char\***, **size\_t**) arguments.
- character literal: accepted by a literal operator taking a single **char** argument.

Note that you cannot make a literal operator for a string literal that takes just a **const char\*** argument (and no size). For example:

```
string operator"" S(const char* p); // warning: this will not work as expected

"one two"S; // error: no applicable literal operator
```

The rationale is that if we want to have “a different kind of string” we almost always want to know the number of characters anyway.

Suffixes will tend to be short (e.g. **s** for string, **i** for imaginary, **m** for meter, and **x** for extended), so different uses could easily clash. Use namespaces to prevent clashes:

```
namespace Numerics {
// ...
class Bignum { /* ... */ };
namespace literals {
operator"" X(char const*);
}
}

using namespace Numerics::literals;
```

See also:

- Standard 2.14.8 User-defined literals
- [N2378==07-0238] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Mauer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde: [User-defined Literals \(aka. Extensible Literals \(revision 3\)\)](#).

## 2.9.27 Attributes

“Attributes” is a new standard syntax aimed at providing some order in the mess of facilities for adding optional and/or vendor specific information into source code (e.g. `__attribute__`, `__declspec`, and `#pragma`). C++11 attributes differ from existing syntaxes by being applicable essentially everywhere in code and always relating to the immediately preceding syntactic entity. For example:

```
void f [[ noreturn ]] () // f() will never return
{
throw "error"; // OK
}

struct foo* f [[carries_dependency]] (int i); // hint to optimizer
int* g(int* x, int* y [[carries_dependency]]);
```

As you can see, an attribute is placed within double square brackets: `[[ ... ]]`. **noreturn** and **carries\_dependency** are the two attributes defined in the standard.

There is a reasonable fear that attributes will be used to create language dialects. The recommendation is to use attributes to only control things that do not affect the meaning of a program but might help detect errors (e.g. `[[noreturn]]`) or help optimizers (e.g. `[[carries_dependency]]`).

One planned use for attributes is improved support for OpenMP. For example:

```
for [[omp::parallel()]] (int i=0; i<v.size(); ++i) {
    // ...
}
```

As shown, attributes can be qualified.

See also:

- Standard: 7.6.1 Attribute syntax and semantics, 7.6.3-4 noreturn, carries\_dependency 8 Declarators, 9 Classes, 10 Derived classes, 12.3.2 Conversion functions
- [N2418=07-027] Jens Maurer, Michael Wong: [Towards support for attributes in C++ \(Revision 3\)](#)

## 2.9.28 Lambdas

A lambda expression is a mechanism for specifying a function object. The primary use for a lambda is to specify a simple action to be performed by some function. For example:

```
vector<int> v = {50, -10, 20, -30};

std::sort(v.begin(), v.end()); // the default sort
// now v should be { -30, -10, 20, 50 }

// sort by absolute value:
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a)<abs(b); });
// now v should be { -10, 20, -30, 50 }
```

The argument `[](int a, int b) { return abs(a)<abs(b); }` is a "lambda" (or "lambda function" or "lambda expression"), which specifies an operation that given two integer arguments **a** and **b** returns the result of comparing their absolute values.

A lambda expression can access local variables in the scope in which it is used. For example:

```
void f(vector<Record>& v)
{
    vector<int> indices(v.size());
    int count = 0;
    generate(indices.begin(), indices.end(), [&count]() { return count++; });

    // sort indices in the order determined by the name field of the records:
    std::sort(indices.begin(), indices.end(), [&](int a, int b) { return v[a].name<v[b].name; });
    // ...
}
```

Some consider this "really neat!"; others see it as a way to write dangerously obscure code. IMO, both are right.

The `[&]` is a "capture list" specifying that local names used will be passed by reference. We could have said that we wanted to "capture" only `v`, we could have said so: `[&v]`. Had we wanted to pass `v` by value, we could have said so: `[=v]`. Capture nothing is `[]`, capture all by references is `[&]`, and capture all by value is `[=]`.

If an action is neither common nor simple, I recommend using a named function object or function. For example, the example above could have been written:

```
void f(vector<Record>& v)
{
    vector<int> indices(v.size());
```

```

int count = 0;
generate(indices.begin(), indices.end(), [&]() { return ++count; });

struct Cmp_names {
    const vector<Record>& vr;
    Cmp_names(const vector<Record>& r) :vr(r) { }
    bool operator()(int a, int b) const { return vr[a].name<vr[b].name; }
};

// sort indices in the order determined by the name field of the records:
std::sort(indices.begin(), indices.end(), Cmp_names(v));
// ...
}

```

For a tiny function, such as this Record name field comparison, the function object notation is verbose, though the generated code is likely to be identical. In C++98, such function objects had to be non-local to be used as template argument; in C++11 [this is no longer necessary](#).

To specify a lambda you must provide

- its capture list: the list of variables it can use (in addition to its arguments), if any (**[&]** meaning "all local variables passed by reference" in the Record comparison example). If no names needs to be captured, a lambda starts with plain **[]**.
- (optionally) its arguments and their types (e.g, **(int a, int b)**)
- The action to be performed as a block (e.g., **{ return v[a].name<v[b].name; }**).
- (optionally) the return type using the [new suffix return type syntax](#); but typically we just deduce the return type from the return statement. If no value is returned **void** is deduced.

See also:

- Standard 5.1.2 Lambda expressions
- [N1968=06-0038] Jeremiah Willcock, Jaakko Jarvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine: [Lambda expressions and closures for C++](#) (original proposal with a different syntax)
- [N2550=08-0060] Jaakko Jarvi, John Freeman, and Lawrence Crowl: [Lambda Expressions and Closures: Wording for Monomorphic Lambdas \(Revision 4\)](#) (final proposal).
- [N2859=09-0049] Daveed Vandevoorde: [New wording for C++0x Lambdas](#).

## 2.9.29 Local types as template arguments

In C++98, local and unnamed types could not be used as template arguments. This could be a burden, so C++11 lifts the restriction:

```

void f(vector<X>& v)
{
    struct Less {
        bool operator()(const X& a, const X& b) { return a.v<b.v; }
    };
    sort(v.begin(), v.end(), Less()); // C++98: error: Less is local
    // C++11: ok
}

```

In C++11, we also have the alternative of using a [lambda expression](#):

```

void f(vector<X>& v)
{
    sort(v.begin(), v.end(),
        [] (const X& a, const X& b) { return a.v<b.v; }); // C++11
}

```

It is worth remembering that naming action can be quite useful for documentation and an encouragement to good design. Also, non-local (necessarily named) entities can be reused.

C++11 also allows values of unnamed types to be used as template arguments:

```
template<typename T> void foo(T const& t){}
enum X { x };
enum { y };

int main()
{
    foo(x); // C++98: ok; C++11: ok
    foo(y); // C++98: error; C++11: ok
    enum Z { z };
    foo(z); // C++98: error; C++11: ok
}
```

See also:

- Standard: Not yet: CWG issue 757
- [N2402=07-0262] Anthony Williams: [Names, Linkage, and Templates \(rev 2\)](#).
- [N2657] John Spicer: [Local and Unnamed Types as Template Arguments](#).

## 2.9.30 noexcept—preventing exception propagation

If a function cannot throw an exception or if the program isn't written to handle exceptions thrown by a function, that function can be declared **noexcept**. For example:

```
extern "C" double sqrt(double) noexcept; // will never throw

vector<double> my_computation(const vector<double>& v) noexcept // I'm not prepared to handle memory exhaus
{
    vector<double> res(v.size()); // might throw
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
    return res;
}
```

If a function declared **noexcept** throws (so that the exception tries to escape, the **noexcept** function) the program is terminated (by a call to **terminate()**). The call of **terminate()** cannot rely on objects being in well-defined states (i.e. there is no guarantees that destructors have been invoked, no guaranteed stack unwinding, and no possibility for resuming the program as if no problem had been encountered). This is deliberate and makes **noexcept** a simple, crude, and very efficient mechanism (much more efficient than the old dynamic **throw()** mechanism).

It is possibly to make a function conditionally **noexcept**. For example, an algorithm can be specified to be **noexcept** if (and only if) the operations it uses on a template argument are **noexcept**:

```
template<class T>
void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0)))) // can throw if f(v.at(0)) can
{
    for(int i; i<v.size(); ++i)
        v.at(i) = f(v.at(i));
}
```

Here, I first use **noexcept** as an operator: **noexcept(f(v.at(0)))** is true if **f(v.at(0))** can't throw, that is if the **f()** and **at()** used are **noexcept**.

The **noexcept()** operator is a constant expression and does not evaluate its operand.

The general form of a **noexcept** declaration is **noexcept(expression)** and “plain **noexcept**” is simply a shorthand for **noexcept(true)**. All declarations of a function must have compatible **noexcept** specifications.

A destructor shouldn't throw; a generated destructor is implicitly **noexcept** (independently of what code is in its body) if all of the members of its class have **noexcept** destructors.

It is typically a bad idea to have a move operation throw, so declare those **noexcept** wherever possible. A generated copy or move operation is implicitly **noexcept** if all of the copy or move operations it uses on members of its class have **noexcept** destructors.

**noexcept** is widely and systematically used in the standard library to improve performance and clarify requirements. See also:

- Standard: 15.4 Exception specifications [except.spec].
- Standard: 5.3.7 noexcept operator [expr.unary.noexcept].
- [N3103==10-0093] D. Kohlbrenner, D. Svoboda, and A. Wesie: Security impact of noexcept. (Noexcept **must** terminate, as it does).
- [N3167==10-0157] David Svoboda: [Delete operators default to noexcept](#).
- [N3204==10-0194] Jens Maurer: [Deducing "noexcept" for destructors](#)
- [N3050==10-0040] D. Abrahams, R. Sharoni, and D. Gregor: [Allowing Move Constructors to Throw \(Rev. 1\)](#).

## 2.9.31 Alignment

Occasionally, especially when we are writing code that manipulates raw memory, we need to specify a desired alignment for some allocation. For example:

```
alignas(double) unsigned char c[1024]; // array of characters, suitably aligned for doubles
alignas(16) char[100]; // align on 16 byte boundary
```

There is also an **alignof** operator that returns the alignment of its argument (which must be a type). For example

```
constexpr int n = alignof(int); // ints are aligned on n byte boundaries
```

See also:

- Standard: 5.3.6 Alignof [expr.alignof]
- Standard: 7.6.2 Alignment specifier [dcl.align]
- [N3093==10-0083] Lawrence Crowl: [C and C++ Alignment Compatibility](#). Aligning the proposal to C's later proposal.
- [N1877==05-0137] Attila (Farkas) Fehér: [Adding Alignment Support to the C++ Programming Language](#). The original proposal.

## 2.9.32 Override controls: override

No special keyword or annotation is needed for a function in a derived class to override a function in a base class. For example:

```
struct B {
    virtual void f();
    virtual void g() const;
    virtual void h(char);
    void k(); // not virtual
};

struct D : B {
    void f(); // overrides B::f()
    void g(); // doesn't override B::g() (wrong type)
    virtual void h(char); // overrides B::h()
    void k(); // doesn't override B::k() (B::k() is not virtual)
};
```

This can cause confusion (what did the programmer mean?), and problems if a compiler doesn't warn against suspicious code. For example,

- Did the programmer mean to override **B::g()**? (almost certainly yes).
- Did the programming mean to override **B::h(char)**? (probably not because of the redundant explicit **virtual**).
- Did the programmer mean to override **B::k()**? (probably, but that's not possible).

To allow the programmer to be more explicit about overriding, we now have the "contextual keyword" **override**:

```
struct D : B {
    void f() override; // OK: overrides B::f()
    void g() override; // error: wrong type
    virtual void h(char); // overrides B::h(); likely warning
    void k() override; // error: B::k() is not virtual
};
```

A declaration marked **override** is only valid if there is a function to override. The problem with **h()** is not guaranteed to be caught (because it is not an error according to the language definition) but it is easily diagnosed.

**override** is only a *contextual* keyword, so you can still use it as an identifier:

```
int override = 7; // not recommended
```

See also:

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]
- [N3234==11-0004] Ville Voutilainen: [Remove explicit from class-head](#).
- [N3151==10-0141] Ville Voutilainen: [Keywords for override control](#). Earlier, more elaborate design.
- [N3163==10-0153] Herb Sutter: [Override Control Using Contextual Keywords. Alternative earlier more elaborate design](#).
- [N2852==09-0042] V. Voutilainen, A. Meredith, J. Maurer, and C. Uzdavinis: [Explicit Virtual Overrides](#). Earlier design based on [attributes](#).
- [N1827==05-0087] C. Uzdavinis and A. Meredith: [An Explicit Override Syntax for C++](#). The original proposal.

## 2.9.33 Override controls: final

Sometimes, a programmer wants to prevent a virtual function from being overridden. This can be achieved by adding the specifier **final**. For example:

```
struct B {
    virtual void f() const final; // do not override
    virtual void g();
};

struct D : B {
    void f() const; // error: D::f attempts to override final B::f
    void g(); // OK
};
```

There are legitimate reasons for wanting to prevent overriding, but I'm afraid that most examples I have been shown to demonstrate the need for **final** have been based on mistaken assumptions on how expensive **virtual** functions are (usually based on experience with other languages). So, if you feel the urge to add a **final** specifier, please double check that the reason is logical: Would semantic errors be likely if someone defined a class that overwrote that virtual function? Adding **final** closes the possibility of a future user of the class might provide a better implementation of the function for some class you haven't thought of. If you don't want to keep that option open, why did you define the function to be **virtual** in the first place? Most reasonable answers to that question that I have encountered have been along the lines: This is a fundamental function in a framework that the framework builders needed to override but isn't safe for general users to override. My bias is to be suspicious towards such claims.

If it is performance (inlining) you want or you simply never want to override, it is typically better not to define a function to be **virtual** in the first place. This is not Java.

**final** is only a *contextual* keyword, so you can still use it as an identifier:

```
int final = 7; // not recommended
```

See also:

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]

## 2.9.34 C99 features

To preserve a high degree of compatibility, a few minor changes to the language were introduced in collaboration with the C standards committee:

- [long long](#).
- [Extended integral types](#) (i.e. rules for optional longer int types).
- UCN changes [N2170==07-0030] “lift the prohibitions on control and basic source universal character names within character and string literals.”
- concatenation of narrow/wide strings.
- **Not** VLAs (Variable Length Arrays; thank heaven for small mercies).

Some extensions of the preprocessing rules were added:

- `__func__` a macro that expands to the name of the lexically current function
- `__STDC_HOSTED__`
- **Pragma:** `__Pragma( X )` expands to `#pragma X`
- vararg macros (overloading of macros with different number of arguments)

```
#define report(test, ...) ((test)?puts(#test):printf(_ _VA_ARGS_ _))
```

- empty macro arguments

A lot of standard library facilities were inherited from C99 (essentially all changes to the C99 library from its C89 predecessor):

See:

- Standard: 16.3 Macro replacement.
- [N1568=04-0008] P.J. Plauger: [PROPOSED ADDITIONS TO TR-1 TO IMPROVE COMPATIBILITY WITH C99](#).

## 2.9.35 Extended integer types

There are a set of rules for how an extended (precision) integer type should behave if one exists.

See

- [06-0058==N1988] J. Stephen Adamczyk: [Adding extended integer types to C++ \(Revision 1\)](#).

## 2.9.36 Dynamic Initialization and Destruction with Concurrency

Sorry, I have not had time to write this entry. See

- [N2660 = 08-0170] Lawrence Crowl: [Dynamic Initialization and Destruction with Concurrency](#) (Final proposal).

## 2.9.37 Thread-local storage

Sorry, I have not had time to write this entry. See

- [N2659 = 08-0169] Lawrence Crowl: [Thread-Local Storage](#) (Final proposal).

## 2.9.38 Unicode characters

Sorry, I have not had time to write this entry. Please come back later.

- ?

## 2.9.39 Copying and Rethrowing Exceptions

How do you catch an exception and then rethrow it on another thread? Use a bit of library magic as described in the standard 18.8.5 Exception Propagation:

- **exception\_ptr current\_exception();** Returns: An *exception\_ptr* object that refers to the currently handled exception (15.3) or a copy of the currently handled exception, or a null *exception\_ptr* object if no exception is being handled. The referenced object shall remain valid at least as long as there is an *exception\_ptr* object that refers to it. ...
- **void rethrow\_exception(exception\_ptr p);**
- **template<class E> exception\_ptr copy\_exception(E e);** Effects: as if

```
try {
    throw e;
} catch(...) {
    return current_exception();
}
```

This is particularly useful for [transmitting an exception from one thread to another](#)

---

## 2.10 Extern templates

A template specialization can be explicitly declared as a way to suppress multiple instantiations. For example:

```
#include "MyVector.h"

extern template class MyVector<int>; // Suppresses implicit instantiation below --
// MyVector<int> will be explicitly instantiated elsewhere

void foo(MyVector<int>& v)
{
    // use the vector in here
}
```

The “elsewhere” might look something like this:

```
#include "MyVector.h"

template class MyVector<int>; // Make MyVector available to clients (e.g., of the shared library)
```

This is basically a way of avoiding significant redundant work by the compiler and linker.

See

- Standard 14.7.2 Explicit instantiation
- [N1448==03-0031] Mat Marcus and Gabriel Dos Reis: [Controlling Implicit Template Instantiation](#).

### 2.10.1 Inline namespace

The **inline namespace** mechanism is intended to support library evolution by providing a mechanism that support a form of versioning. Consider:

```
// file V99.h:
inline namespace V99 {
    void f(int); // does something better than the V98 version
    void f(double); // new feature
    // ...
}
```

```

// file V98.h:
namespace V98 {
    void f(int); // does something
    // ...
}

// file Mine.h:
namespace Mine {
#include "V99.h"
#include "V98.h"
}

```

We here have a namespace **Mine** with both the latest release (**V99**) and the previous one (**V98**). If you want to be specific, you can:

```

#include "Mine.h"
using namespace Mine;
// ...
V98::f(1); // old version
V99::f(1); // new version
f(1); // default version

```

The point is that the **inline** specifier makes the declarations from the nested namespace appear exactly as if they had been declared in the enclosing namespace.

This is a very “static” and implementer-oriented facility in that the **inline** specifier has to be placed by the designer of the namespaces—thus making the choice for all users. It is not possible for a user of **Mine** to say “I want the default to be **V98** rather than **V99**”.

See

- Standard 7.3.1 Namespace definition [7]-[9].

## 2.10.2 Explicit conversion operators

C++98 provides implicit and explicit constructors; that is, the conversion defined by a constructor declared **explicit** can be used only for explicit conversions whereas other constructors can be used for implicit conversions also. For example:

```

struct S { S(int); }; // "ordinary constructor" defines implicit conversion
S s1(1); // ok
S s2 = 1; // ok
void f(S);
f(1); // ok (but that's often a bad surprise -- what if S was vector?)

struct E { explicit E(int); }; // explicit constructor
E e1(1); // ok
E e2 = 1; // error (but that's often a surprise)
void f(E);
f(1); // error (protects against surprises -- e.g. std::vector's constructor from int is explicit)

```

However, a constructor is not the only mechanism for defining a conversion. If we can't modify a class, we can define a conversion operator from a different class. For example:

```

struct S { S(int) { } /* ... */ };

struct SS {
    int m;
    SS(int x) :m(x) { }
    operator S() { return S(m); } // because S don't have S(SS); non-intrusive
};

```

```

SS ss(1);
S s1 = ss; // ok; like an implicit constructor
S s2(ss); // ok ; like an implicit constructor
void f(S);
f(ss); // ok; like an implicit constructor

```

Unfortunately, there is no **explicit** conversion operators (because there are far fewer problematic examples). C++11 deals with that oversight by allowing conversion operators to be **explicit**. For example:

```

struct S { S(int) { } };

struct SS {
    int m;
    SS(int x) :m(x) { }
    explicit operator S() { return S(m); } // because S don't have S(SS)
};

SS ss(1);
S s1 = ss; // error; like an explicit constructor
S s2(ss); // ok ; like an explicit constructor
void f(S);
f(ss); // error; like an explicit constructor

```

See also:

- Standard: 12.3 Conversions
- [N2333=07-0193] Lois Goldthwaite, Michael Wong, and Jens Maurer: [Explicit Conversion Operator \(Revision 1\)](#).

## 2.10.3 Algorithms Improvements

The standard library algorithms are improved partly by simple addition of new algorithms, partly by improved implementations made possible by new language features, and partly by new language features enabling easier use:

- *New algorithms:*

```

bool all_of(Iter first, Iter last, Pred pred);
bool any_of(Iter first, Iter last, Pred pred);
bool none_of(Iter first, Iter last, Pred pred);

Iter find_if_not(Iter first, Iter last, Pred pred);

OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred);
OutIter copy_n(InIter first, InIter::difference_type n, OutIter result);

OutIter move(InIter first, InIter last, OutIter result);
OutIter move_backward(InIter first, InIter last, OutIter result);

pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last, OutIter1 out_true, OutIter2 out_false,
Iter partition_point(Iter first, Iter last, Pred pred);

RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last);
RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last, Compare comp);
bool is_sorted(Iter first, Iter last);
bool is_sorted(Iter first, Iter last, Compare comp);
Iter is_sorted_until(Iter first, Iter last);
Iter is_sorted_until(Iter first, Iter last, Compare comp);

bool is_heap(Iter first, Iter last);
bool is_heap(Iter first, Iter last, Compare comp);
Iter is_heap_until(Iter first, Iter last);
Iter is_heap_until(Iter first, Iter last, Compare comp);

```

```

T min(initializer_list<T> t);
T min(initializer_list<T> t, Compare comp);
T max(initializer_list<T> t);
T max(initializer_list<T> t, Compare comp);
pair<const T&, const T&> minmax(const T& a, const T& b);
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
pair<const T&, const T&> minmax(initializer_list<T> t);
pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);
pair<Iter, Iter> minmax_element(Iter first, Iter last);
pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);

void iota(Iter first, Iter last, T value); // For each element referred to by the iterator i in the ra

```

- *Effects of move*: Moving can be much more efficient than copying (see [Move semantics](#)). For example, move-based `std::sort()` and `std::set::insert()` has been measured to be 15 times faster than copy based versions. This is less impressive than it sounds because such standard library operations for standard library types, such as **string** and **vector**, are usually hand-optimized to gain the effects of moving through techniques such as replacing copies with optimized swaps. However, if *your* type has a move operation, you gain the performance benefits automatically from the standard algorithms.

Consider also that the use of moves allows simple and efficient sort (and other algorithms) of containers of “smart” pointers, especially [unique\\_ptr](#):

```

template<class P> struct Cmp<P> { // compare *P values
    bool operator() (P& a, P& b) const { return *a<*b; }
}

vector<std::unique_ptr<Big>> vb;
// fill vb with unique_ptr's to Big objects

sort(vb.begin(), vb.end(), Cmp<unique_ptr<Big>()); // don't try that with an auto_ptr

```

- *Use of lambdas*: For ages, people have complained about having to write functions or (better) function objects for use as operations, such as **Cmp<T>** above, for standard library (and other) algorithms. This was especially painful to do if you wrote large functions (don't) because in C++98 you could not define a local function object to use as an argument; [now you can](#). However, [lambdas](#) allows us to define operations “inline:”

```

sort(vb.begin(), vb.end(), [](unique_ptr<Big> a, unique_ptr<Big> b) { return *a<*b; });

```

I expect lambdas to be a bit overused initially (like all powerful mechanisms).

- *Use of initializer lists*: Sometimes, [initializer lists](#) come in handy as arguments. For example, assuming **string** variables and **Nocase** being a case-insensitive comparison:

```

auto x = max({x, y, z}, Nocase());

```

See also:

- 25 Algorithms library [algorithms]
- 26.7 Generalized numeric operations [numeric.ops]
- Howard E. Hinnant, Peter Dimov, and Dave Abrahams: [A Proposal to Add Move Semantics Support to the C++ Language](#). N1377=02-0035.

## 2.10.4 Container Improvements

Given the new language features and a decade's worth of experience, what has happened to the standard containers? First, of course we got a few new ones: [array](#) (a fixed-sized container), [forward\\_list](#) (a singly-linked list), and [unordered containers](#) (the hash tables). Next, new features, such as [initializer lists](#), [rvalue references](#), [variadic templates](#), and [constexpr](#) were put to use. Consider **std::vector**.

- *Initializer lists*: The most visible improvement is the use of initializer-list constructors to allow a container to take

an initializer list as its argument:

```
vector<string> vs = { "Hello", "", "", "World!", "\n" };
for (auto s : vs ) cout << s;
```

- *Move operators*: Containers now have move constructors and move assignments (in addition to the traditional copy operations). The most important implication of this is that we can efficiently return a container from a function:

```
vector<int> make_random(int n)
{
    vector<int> ref(n);
    for(auto& x : ref) x = rand_int(0,255); // some random number generator
    return ref;
}

vector<int> v = make_random(10000);
for (auto x : make_random(1000000)) cout << x << '\n';
```

The point here is that no vectors are copied. Rewrite this to return a free-store-allocated vector and you have to deal with memory management. Rewrite this to pass the vector to be filled as an argument to **make\_random()** and you have a far less obvious code (plus an added opportunity for making an error).

- *Improved push operations*: My favorite container operation is **push\_back()** that allows a container to grow gracefully:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.push_back({s,i});
```

This will construct a **pair<string,int>** out of **s** and **i** and move it into **vp**. Note “move” not “copy;” There is a **push\_back** version that takes an **rvalue reference** argument so that we can take advantage of **string**'s move constructor. Note also the use of the **unified initializer syntax** to avoid verbosity.

- *Emplace operations*: The **push\_back()** using a move constructor is far more efficient in important cases than the traditional copy-based one, but in extreme cases we can go further. Why copy/move anything? Why not make space in the vector and then construct the desired value in that space? Operations that do that are called “emplace” (meaning “putting in place”). For example **emplace\_back()**:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.emplace_back(s,i);
```

An **emplace** takes a **variadic template** argument and uses that to construct an object of the desired type. Whether the **emplace\_back()** really is more efficient than the **push\_back()** depends on the types involved and the implementation (of the library and of variadic templates). If you think it matters, measure. Otherwise, choose based on aesthetics: **vp.push\_back({s,i});** or **vp.emplace\_back(s,i);**. For now, I prefer the **push\_back()** version, but that might change over time.

- *Scoped allocators*: Containers can now hold “real allocation objects (with state)” and use those to control nested/scoped allocation (e.g. allocation of elements in a container)

Obviously, the containers are not the only parts of the standard library that has benefitted from the new language features. Consider:

- *Compile-time evaluation*: **constexpr** is used to ensure compiler time evaluation in **bitset**, **duration**, **char\_traits**, **array**, atomic types, random numbers, **complex<double>**., etc. In some cases, it means improved performance; in others (where there is no alternative to compile-time evaluation), it means absence of messy low-level code and macros.
- *Tuples*: Tuples would not be possible without variadic templates.

## 2.10.5 Scoped Allocators

For compactness of container objects and for simplicity, C++98 did not require containers to support allocators with state: Allocator objects need not be stored in container objects. This is still the default in C++11, but it is possible to use an allocator with state, say an allocator that holds a pointer to an arena from which to allocate. For example:

```
template<class T> class Simple_alloc { // C++98 style
    // no data
    // usual allocator stuff
};

class Arena {
    void* p;
    int s;
public:
    Arena(void* pp, int ss);
    // allocate from p[0..ss-1]
};

template<class T> struct My_alloc {
    Arena& a;
    My_alloc(Arena& aa) : a(aa) { }
    // usual allocator stuff
};

Arena my_arena1(new char[100000],100000);
Arena my_arena2(new char[1000000],1000000);

vector<int> v0; // allocate using default allocator

vector<int,My_alloc<int>> v1(My_alloc<int>{my_arena1}); // allocate from my_arena1

vector<int,My_alloc<int>> v2(My_alloc<int>{my_arena2}); // allocate from my_arena2

vector<int,Simple_alloc<int>> v3; // allocate using Simple_alloc
```

Typically, the verbosity would be alleviated by the use of typedefs.

It is not guaranteed that the default allocator and **Simple\_alloc** takes up no space in a vector object, but a bit of elegant template metaprogramming in the library implementation can ensure that. So, using an allocator type imposes a space overhead only if its objects actually has state (like **My\_alloc**).

A rather sneaky problem can occur when using containers and user-defined allocators: Should an element be in the same allocation area as its container? For example, if you use **Your\_allocator** for **Your\_string** to allocate its elements and I use **My\_allocator** to allocate elements of **My\_vector** then which allocator should be used for string elements in **My\_vector<Your\_allocator>**? The solution is the ability to tell a container which allocator to pass to elements. For example, assuming that I have an allocator **My\_alloc** and I want a **vector<string>** that uses **My\_alloc** for both the **vector** element and **string** element allocations. First, I must make a version of **string** that accepts **My\_alloc** objects:

```
using xstring = basic_string<char, char_traits<char>, My_alloc<char>>; // a string with my allocator
```

Then, I must make a version of **vector** that accepts those strings, accepts a **My\_alloc** object, and passes that object on to the string:

```
using svec = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
```

Finally, we can make an allocator of type **My\_alloc<xstring>**:

```
svec v(svec::allocator_type(My_alloc<xstring>{my_arena1}));
```

Now **svec** is a **vector** of strings using **My\_alloc** to allocate memory for strings. What's new is that the standard library "adaptor" ("wrapper type") **scoped\_allocator\_adaptor** is used to indicate that string also should use **My\_alloc**. Note that the adaptor can (trivially) convert **My\_alloc<xstring>** to the **My\_alloc<char>** that **xstring** needs.

So, we have four alternatives:

```
// vector and string use their own (the default) allocator:
using svec0 = vector<string>;
svec0 v0;

// vector (only) uses My_alloc and string uses its own (the default) allocator:
using svec1 = vector<string,My_alloc<string>>;
svec1 v1(My_alloc<string>{my_arena});

// vector and string use My_alloc (as above):
using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
using svec2 = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
svec2 v2(scoped_allocator_adaptor<My_alloc<xstring>>{my_arena});

// vector uses My_alloc and string uses My_string_alloc:
using xstring2 = basic_string<char, char_traits<char>, My_string_alloc<char>>;
using svec3 = vector<xstring2,scoped_allocator_adaptor<My_alloc<xstring>, My_string_alloc<char>>>;
svec3 v3(scoped_allocator_adaptor<My_alloc<xstring2>, My_string_alloc<char>>{my_arena,my_string_arena});
```

Obviously, the first variant, **svec0**, will be by far the most common, but for systems with serious memory-related performance constraints, the other versions (especially **svec2**) can be important. A few typedefs would make that code a bit more readable, but it is good it is not something you have to write every day. The **scoped\_allocator\_adaptor2** is a variant of **scoped\_allocator\_adaptor** for the case where the two non-default allocators differ.

See also:

- Standard: 20.8.5 Scoped allocator adaptor [allocator.adaptor]
- Pablo Halpern: [The Scoped Allocator Model \(Rev 2\)](#). N2554=08-0064.

## 2.10.6 std::array

The standard container **array** is a fixed-sized random-access sequence of elements defined in **<array>**. It has no space overheads beyond what it needs to hold its elements, it does not use free store, it can be initialized with an initializer list, it knows its size (number of elements), and doesn't convert to a pointer unless you explicitly ask it to. In other words, it is very much like a built-in array without the problems.

```
array<int,6> a = { 1, 2, 3 };
a[3]=4;
int x = a[5]; // x becomes 0 because default elements are zero initialized
int* p1 = a; // error: std::array doesn't implicitly convert to a pointer
int* p2 = a.data(); // ok: get pointer to first element
```

Note that you can have zero-length **arrays** but that you cannot deduce the length of an **array** from an initializer list:

```
array<int> a3 = { 1, 2, 3 }; // error: size unknown/missing
array<int,0> a0; // ok: no elements
int* p = a0.data(); // unspecified; don't try it
```

The standard **array**'s features makes it attractive for embedded systems programming (and similar constrained, performance-critical, or safety critical tasks). It is a sequence container so it provides the usual member types and functions (just like **vector**):

```
template<class C> C::value_type sum(const C& a)
{
    return accumulate(a.begin(),a.end(),0);
}
```

```

}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
int x1 = sum(a10);
double x2 = sum(a1000);
int x3 = sum(v);

```

Also, you don't get (potentially nasty) derived to base conversions:

```

struct Apple : Fruit { /* ... */ };
struct Pear : Fruit { /* ... */ };

void nasty(array<Fruit*,10>& f)
{
    f[7] = new Pear();
};

array<Apple*,10> apples;
// ...
nasty(apples); // error: can't convert array<Apple*,10> to array<Fruit*,10>;

```

If that was allowed, **apples** would now contain a **Pear**.

See also:

- Standard: 23.3.1 Class template array

## 2.10.7 std::forward\_list

The standard container **forward\_list**, defined in `<forward_list>`, is basically a singly-linked list. It supports forward iteration (only) and guarantees that elements don't move if you insert or erase one. It occupies minimal space (an empty list is likely to be one word) and does not provide a **size()** operation (so that it does not have to store a size member):

```

template <ValueType T, Allocator Alloc = allocator<T> >
    requires NothrowDestructible<T>
class forward_list {
public:
    // the usual container stuff
    // no size()
    // no reverse iteration
    // no back() or push_back()
};

```

See also:

- Standard: 23.3.3 Class template forward\_list

## 2.10.8 Unordered Containers

A unordered container is a kind of hash table. C++11 offers four standard ones:

- unordered\_map
- unordered\_set
- unordered\_multimap
- unordered\_multiset

They should have been called **hash\_map** etc., but there are so many incompatible uses of those names that the

committee had to choose new names and the **unordered\_map**, etc. were the least bad we could find. The "unordered" refers to one of the key differences between **map** and **unordered\_map**: When you iterate over a **map** you do so in the order provided by its less-than comparison operator (by default <) whereas the value type of **unordered\_map** is not required to have a less-than comparison operator and a hash table doesn't naturally provide an order. Conversely, the element type of a **map** is not required to have a hash function.

The basic idea is simply to use **unordered\_map** as an optimized version of **map** where optimization is possible and reasonable. For example:

```
map<string,int> m {
    {"Dijkstra",1972}, {"Scott",1976}, {"Wilkes",1967}, {"Hamming",1968}
};
m["Ritchie"] = 1983;
for(auto x : m) cout << '{' << x.first << ', ' << x.second << '}';
```

```
unordered_map<string,int> um {
    {"Dijkstra",1972}, {"Scott",1976}, {"Wilkes",1967}, {"Hamming",1968}
};
um["Ritchie"] = 1983;
for(auto x : um) cout << '{' << x.first << ', ' << x.second << '}';
```

The iterator over **m** will present the elements in alphabetical order; the iteration over **um** will not (except through a freak accident). Lookup is implemented very differently for **m** and **um**. For **m** lookup involves  $\log_2(m.size())$  less-than comparisons whereas for **um** lookup involves a single call of a hash function and one or more equality operations. For a few elements (say a few dozen), it is hard to tell which is faster. For larger numbers of elements (e.g. thousands), lookup in an **unordered\_map** can be much faster than for a **map**.

More to come.

See also:

- Standard: 23.5 Unordered associative containers.

## 2.10.9 std::tuple

The standard library **tuple** (an N-tuple) is a ordered sequence of N values where N can be a constant from 0 to a large implementation-defined value, defined in **<tuple>**. You can think of an tuple as an unnamed struct with members of the specified tuple element types. In particular, the elements of a **tuple** is stored compactly; a tuple is not a linked structure.

The element types of a tuple can explicitly specified or be deduced (using **make\_tuple()**) and the elements can be access by (zero-based) index using **get()**:

```
tuple<string,int> t2("Kylling",123);

auto t = make_tuple(string("Herring"),10, 1.23); // t will be of type tuple<string,int,double>
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

Tuples are used (directly or indirectly) whenever we want a heterogeneous list of elements at compile time but do not want to define a named class to hold them. For example, **tuple** is used internally in **std::function** and **std::bind** to hold arguments.

The most frequently useful tuple is the 2-tuple; that is, a pair. However, pair is directly supported in the standard library through **std::pair** (20.3.3 Pairs). A **pair** can be used to initialize a **tuple**, but the opposite isn't the case.

The comparison operators (==, !=, <, <=, >, and >=) are defined for tuples of comparable element types.

See also:

- Standard: 20.5.2 Class template tuple

- Variadic template paper
- Boost::tuple

## 2.10.10 Metaprogramming and Type Traits

Sorry. Come back later.

### 2.10.11 std::function and std::bind

The **bind** and **function** standard function objects are defined in `<functional>` (together with a lot of other function objects); they are used to handle functions and function arguments. **bind** is used to take a function (or a function object or anything you can invoke using the `(...)` syntax) and produce a function object with one or more of the arguments of the argument function “bound” or rearranged. For example:

```
int f(int,char,double);
auto ff = bind(f,_1,'c',1.2); // deduce return type
int x = ff(7); // f(7,'c',1.2);
```

This binding of arguments is usually called “Currying.” The `_1` is a place-holder object indicating where the first argument of `ff` is to go when `f` is called through `ff`. The first argument is called `_1`, the second `_2`, and so on. For example:

```
int f(int,char,double);
auto frev = bind(f,_3,_2,_1); // reverse argument order
int x = frev(1.2,'c',7); // f(7,'c',1.2);
```

Note how `auto` saves us from having to specify the type of the result of **bind**.

It is not possible to just bind arguments for an overloaded function, we have to explicitly state which version of an overloaded function we want to bind:

```
int g(int);
double g(double); // g() is overloaded

auto g1 = bind(g,_1); // error: which g()?
auto g2 = bind((double*)(double))g,_1); // ok (but ugly)
```

**bind()** comes in two variants: the one shown above and a “legacy” version where you explicitly specify the return type:

```
auto f2 = bind<int>(f,7,'c',_1); // explicit return type
int x = f2(1.2); // f(7,'c',1.2);
```

This second version was necessary and is widely used because the first (and for a user simplest) version cannot be implemented in C++98.

**function** is a type that can hold a value of just about anything you can invoke using the `(...)` syntax. In particular, the result of **bind** can be assigned to a **function**. **function** is very simple to use. For example:

```
function<float (int x, int y)> f; // make a function object

struct int_div { // take something you can call using ()
    float operator()(int x, int y) const { return ((float)x)/y; };
};

f = int_div(); // assign
cout << f(5, 3) << endl; // call through the function object
std::accumulate(b,e,1,f); // passes beautifully
```

Member functions can be treated as free functions with an extra argument

```

struct X {
    int foo(int);
};

function<int (X*, int)> f;
f = &X::foo; // pointer to member

X x;
int v = f(&x, 5); // call X::foo() for x with 5
function<int (int)> ff = std::bind(f,&x,_1); // first argument for f is &x
v=ff(5); // call x.foo(5)

```

**functions** are useful for callbacks, for passing operations as argument, etc. It can be seen as a replacement for the C++98 standard library function objects **mem\_fun\_t**, **pointer\_to\_unary\_function**, etc. Similarly, **bind()** can be seen as a replacement for **bind1()** and **bind2()**.

See also:

- Standard: 20.7.12 Function template bind, 20.7.16.2 Class template function
- Herb Sutter: [Generalized Function Pointers](#). August 2003.
- Douglas Gregor: [Boost.Function](#).
- Boost::bind

## 2.10.12 unique\_ptr

- The **unique\_ptr** (defined in **<memory>**) provides a semantics of strict ownership.
  - owns the object it holds a pointer to
  - is not CopyConstructible, nor CopyAssignable; however, it is MoveConstructible and MoveAssignable.
  - stores a pointer to an object and deletes that object using the associated deleter when it is itself destroyed (such as when leaving block scope (6.7)).
- The uses of **unique\_ptr** include
  - providing exception safety for dynamically allocated memory,
  - Passing ownership of dynamically allocated memory to a function
  - returning dynamically allocated memory from a function
  - storing pointers in containers
- "what **auto\_ptr** should have been" (but that we couldn't write in C++98)

**unique\_ptr** relies critically on [rvalue references](#) and move semantics.

Here is a conventional piece of exception unsafe code:

```

X* f()
{
    X* p = new X;
    // do something - maybe throw an exception
    return p;
}

```

A solution is to hold the pointer to the object on the free store in a **unique\_ptr**:

```

X* f()
{
    unique_ptr<X> p(new X); // or {new X} but not = new X
    // do something - maybe throw an exception
    return p.release();
}

```

Now, if an exception is thrown, the **unique\_ptr** will (implicitly) destroy the object pointed to. That's basic [RAII](#). However, unless we really need to return a built-in pointer, we can do even better by returning a **unique\_ptr**:

```

unique_ptr<X> f()
{
    unique_ptr<X> p(new X); // or {new X} but not = new X
    // do something - maybe throw an exception
    return p; // the ownership is transferred out of f()
}

```

We can use this **f** like this:

```

void g()
{
    unique_ptr<X> q = f(); // move using move constructor
    q->memfct(2); // use q
    X x = *q; // copy the object pointed to
    // ...
} // q and the object it owns is destroyed on exit

```

The **unique\_ptr** has "move semantics" so the initialization of **q** with the rvalue that is the result of the call **f()** simply transfers ownership into **q**.

One of the uses of **unique\_ptr** is as a pointer in a container, where we might have used a built-in pointer except for exception safety problems (and to guarantee destruction of the pointed to elements):

```

vector<unique_ptr<string>> vs { new string{"Doug"}, new string{"Adams"} };

```

**unique\_ptr** is represented by a simple built-in pointer and the overhead of using one compared to a built-in pointer are miniscule. In particular, **unique\_ptr** does not offer any form of dynamic checking.

See also

- the C++ draft section 20.7.10
- Howard E. Hinnant: [unique\\_ptr Emulation for C++03 Compilers](#).

### 2.10.13 shared\_ptr

A **shared\_ptr** is used to represent shared ownership; that is, when two pieces of code needs access to some data but neither has exclusive ownership (in the sense of being responsible for destroying the object). A **shared\_ptr** is a kind of counted pointer where the object pointed to is deleted when the use count goes to zero. Here is a highly artificial example:

```

void test()
{
    shared_ptr<int> p1(new int); // count is 1
    {
        shared_ptr<int> p2(p1); // count is 2
        {
            shared_ptr<int> p3(p1); // count is 3
        } // count goes back down to 2
    } // count goes back down to 1
} // here the count goes to 0 and the int is deleted.

```

A more realistic example would be pointers to nodes in a general graph where someone wanting to remove a pointer to a node wouldn't know if anyone else held a pointer to that node. If a node can hold resources that require an action by a destructor (e.g. a file handle so that a file needs to be closed when the node is deleted). You could consider **shared\_ptr** to be for what you might consider plugging in a [garbage collector](#) for, except that maybe you don't have enough garbage for that to be economical, your execution environment doesn't allow that, or the resource managed is not just memory (e.g. that file handle). For example:

```

struct Node { // note: a Node may be pointed to from several other nodes.
    shared_ptr<Node> left;
    shared_ptr<Node> right;
    File_handle f;
};

```

```
// ...
};
```

Here **Node**'s destructor (the implicitly generated destructor will do fine) deletes its sub-nodes; that is, **left** and **right**'s destructors are invoked. Since **left** is a **shared\_ptr**, the **Node** pointed to (if any) is deleted if **left** was the last pointer to it; **right** is handled similarly and **f**'s destructor does whatever is required for **f**.

Note that you should not use a **shared\_ptr** just to pass a pointer from one owner to another; that's what **unique\_ptr** is for and **unique\_ptr** does that cheaper and better. If you have been using counted pointers as return values from factory functions and the like, consider upgrading to **unique\_ptr** rather than **shared\_ptr**.

Please don't thoughtlessly replace pointers with **shared\_ptrs** in an attempt to prevent memory leaks; **shared\_ptrs** are not a panacea nor are they without costs:

- a circular linked structure of **shared\_ptrs** will cause a memory leak (you'll need some logical complication to break the circle, e.g. using a **weak\_ptr**),
- "shared ownership objects" tend to stay "live" for longer than scoped objects (thus causing higher average resource usage),
- shared pointers in a multi-threaded environment can be expensive (because of the need to avoid data races on the use count),
- a destructor for a shared object does not execute at a predictable time, and
- the algorithms/logic for the update of any shared object is easier to get wrong than for an object that's not shared.

. A **shared\_ptr** represents *shared ownership* but shared ownership isn't my ideal: It is better if an object has a definite owner and a definite, predictable lifespan.

See also

- the C++ draft: `Shared_ptr` (20.7.13.3)

## 2.10.14 weak\_ptr

Weak pointers are often explained as what you need to break loops in data structures managed using **shared\_ptrs**. I think it is better to think of a **weak\_ptr** as a pointer to something that

1. you need access to (only) if it exists, and
2. may get deleted (by someone else), and
3. must have its destructor called after its last use (usually to delete anon-memory resource)

Consider an implementation of the old "asteroid game". All asteroids are owned by "the game" but each asteroids must keep track of neighboring asteroids and handle collisions. A collision typically leads to the destruction of one or more asteroids. Each asteroid must keep a list of other asteroids in its neighborhood. Note that being on such a neighbor list should not keep an asteroid "alive" (so a **shared\_ptr** would be inappropriate). On the other hand, an asteroid must not be destroyed while another asteroid is looking at it (e.g. to calculate the effect of a collision). And obviously, an asteroids destructor must be called to release resources (such as a connection to the graphics system). What we need is a list of asteroids that *might* still be intact and a way of "getting hold of one if it exist" for a while. A **weak\_ptr** does just that:

```
void owner()
{
// ...
vector<shared_ptr<Asteroid>> va(100);
for (int i=0; i<va.size(); ++i) {
// ... calculate neighbors for new asteroid ...
va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor])));
launch(i);
}
// ...
}
```

**reset()** is the function to make a **shared\_ptr** refer to a new object.

Obviously, I radically simplified "the owner" and gave each new **Asteroid** just one neighbor. The key is that we give the **Asteroid** a **weak\_ptr** to that neighbor. The owner keeps a **shared\_ptr** to represent the ownership that's shared whenever an **Asteroid** is looking (but not otherwise). The collision calculation for an **Asteroid** will look something

like this:

```
void collision(weak_ptr<Asteroid> p)
{
    if (auto q = p.lock()) { // p.lock returns a shared_ptr to p's object
        // ... that Asteroid still existed: calculate ...
    }
    else {
        // ... oops: that Asteroid has already been destroyed: just forget about it (delete the weak_ptr to it ..
    }
}
```

Note that even if the owner decides to shut down the game and deletes all **Asteroids** (by destroying the **shared\_ptr**s representing ownership) every **Asteroid** that is in the middle of calculating a collision still finishes correctly (because after the **p.lock()** it holds a **shared\_ptr** that won't just become invalid).

I expect to find **weak\_ptr** use much rarer than "plain" **shared\_ptr** use and I hope that **unique\_ptr** will become much more popular than **shared\_ptr** use because **unique\_ptr** represents a simpler (and more efficient) notion of ownership and (therefore) allows better local reasoning.

See also

- the C++ draft: `Weak_ptr` (20.7.13.3)

## 2.10.15 Garbage Collection ABI

Garbage collection (automatic recycling of unreferenced regions of memory) is optional in C++; that is, a garbage collector is not a compulsory part of an implementation. However, C++11 provides a definition of what a GC can do if one is used and an ABI (Application Binary Interface) to help control its actions.

The rules for pointers and lifetimes are expressed in terms of "safely derived pointer" (3.7.4.3); roughly: "pointer to something allocated by `new` or to a sub-object thereof." Here are some examples of "not safely derived pointers" aka "disguised pointers" aka what not to do in a program you want to be considered well behaved and comprehensible to ordinary mortals:

- Make a pointer point "elsewhere" for a while

```
int* p = new int;
p+=10;
// ... collector may run here ...
p-=10;
*p = 10; // can we be sure that the int is still there?
```

- Hide the pointer in an int

```
int* p = new int;
int x = reinterpret_cast<int>(p); // non-portable
p=0;
// ... collector may run here ...
p = reinterpret_cast<int*>(x);
*p = 10; // can we be sure that the int is still there?
```

- There are many more and even nastier tricks Think I/O, think "scattering the bits around in different words", ...

There are legitimate reasons to disguise pointers (e.g. the xor trick in exceptionally memory-constrained applications), but not as many as some programmers think.

A programmer can specify where there are no pointers to be found (e.g. in an image) and what memory can't be reclaimed even if the collector can't find a pointer into it:

```
void declare_reachable(void* p); // the region of memory starting at p
// (and allocated by some allocator
```

```

    // operation which remembers its size)
    // must not be collected
template<class T> T* undeclare_reachable(T* p);

void declare_no_pointers(char* p, size_t n);    // p[0..n] holds no pointers
void undeclare_no_pointers(char* p, size_t n);

```

A programmer can inquire which rules for pointer safety and reclamation is in force:

```

enum class pointer_safety {relaxed, preferred, strict };
pointer_safety get_pointer_safety();

```

3.7.4.3[4]: If a pointer value that is not a safely-derived pointer value is dereferenced or deallocated, and the referenced complete object is of dynamic storage duration and has not previously been declared reachable (20.7.13.7), the behavior is undefined.

- **relaxed**: safely-derived and not safely-derived pointers are treated equivalently; like C and C++98, but that was not my intent - I wanted to allow GC if a user didn't keep a valid pointer around for an object.
- **preferred**: like relaxed; but a garbage collector may be running as a leak detector and/or detector of dereferences of "bad pointers"
- **strict**: safely-derived and not safely-derived pointers may be treated differently, i.e. a garbage collector may be running and will ignore pointers that's not safely derived

There is no standard way of saying which alternative you prefer. Considered that a "quality of implementation" and a "programming environment" issue.

See also

- the C++ draft 3.7.4.3
- the C++ draft 20.7.13.7
- Hans Boehm's [GC page](#)
- Hans Boehm's [Discussion of Conservative GC](#)
- [final proposal](#)
- Michael Spertus and Hans J. Boehm: [The Status of Garbage Collection in C++0X](#). ACM ISMM'09.

## 2.10.16 Memory Model

A memory model is an agreement between the machine architects and the compiler writers to ensure that most programmers do not have to think about the details of modern computer hardware. Without a memory model, few things related to threading, locking, and lock-free programming would make sense.

The key guarantee is: Two threads of execution can update and access separate memory locations without interfering with each other. But what is a "memory location?" A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. For example, here **S** has exactly four separate memory locations:

```

struct S {
    char a;    // location #1
    int b:5,  // location #2
    int c:11,
    int :0,   // note: :0 is "special"
    int d:8;  // location #3
    struct {int ee:8;} e; // location #4
};

```

Why is this important? Why isn't it obvious? Wasn't this always true? The problem is that when several computations can genuinely run in parallel, that is several (apparently) unrelated instructions can execute at the same time, the quirks of the memory hardware can get exposed. In fact, in the absence of compiler support, issues of instruction and data pipelining and details of cache use *will* be exposed in ways that are completely unmanageable to the applications programmer. This is true even if no two threads have been defined to share data! Consider, two separately compiled "threads:"

```

// thread 1:
char c;
c = 1;
int x = c;

// thread 2:
char b;
b = 1;
int y = b;

```

For greater realism, I could have used the separate compilation (within each thread) to ensure that the compiler/optimizer wouldn't be able to eliminate memory accesses and simply ignore **c** and **b** and directly initialize **x** and **y** with 1. What are the possible values of **x** and **y**? According to C++11 the only correct answer is the obvious one: 1 and 1. The reason that's interesting is that if you take a conventional good pre-concurrency C or C++ compiler, the possible answers are 0 and 0 (unlikely), 1 and 0, 0 and 1, and 1 and 1. This has been observed "in the wild." How? A linker might allocate **c** and **b** right next to each other (in the same word) -- nothing in the C or C++ 1990s standards says otherwise. In that, C++ resembles all languages not designed with real concurrent hardware in mind. However, most modern processors cannot read or write a single character, it must read or write a whole word, so the assignment to **c** really is "read the word containing **c**, replace the **c** part, and write the word back again." Since the assignment to **b** is similar, there are plenty of opportunities for the two threads to clobber each other even though the threads do not (according to their source text) share data!

So, C++11 guarantees that no such problems occur for "separate memory locations." More precisely: A memory location cannot be safely accessed by two threads without some form of locking unless they are both read accesses. Note that different bitfields within a single word are not separate memory locations, so don't share structs with bitfields among threads without some form of locking. Apart from that caveat, the C++ memory model is simply "as everyone would expect."

However, it is not always easy to think straight about low-level concurrency issues. Consider:

```

// start with x==0 and y==0

if (x) y = 1; // Thread 1

if (y) x = 1; // Thread 2

```

Is there a problem here? More precisely, is there a data race? (No there isn't).

Fortunately, we have already adapted to modern times and every current C++ compiler (that I know of) gives the one right answer and have done so for years. They do so for most (but unfortunately not yet for all) tricky questions. After all, C++ has been used for serious systems programming of concurrent systems "forever." The standard should further improve things.

See also

- Standard: 1.7 The C++ memory model [intro.memory]
- Paul E. McKenney, Hans-J. Boehm, and Lawrence Crowl: [C++ Data-Dependency Ordering: Atomics and Memory Model](#). N2556==08-0066.
- Hans-J. Boehm: [Threads basics](#), HPL technical report 2009-259. "what every programmer should know about memory model issues."
- Hans-J. Boehm and Paul McKenney: [A slightly dated FAQ on C++ memory model issues](#).

## 2.10.17 Threads

A thread is a representation of an execution/computation in a program. In C++11, as in much modern computing, a thread can—and usually does—share an address space with other threads. In this, it differs from a process, which generally does not directly share data with other processes. C++ have had a host of threads implementations for a variety of hardware and operating systems in the past, what's new is a standard-library threads library, a standard thread ABI.

Many thick books and tens of thousands of papers have been writing about concurrency, parallelism, and threading, this FAQ entry barely scratch the surface. It is *hard* to think clearly about concurrency. If you want to do concurrent

programming, at least read a book. Do not rely just on a manual, a standard, or an FAQ.

A thread is launched by constructing a `std::thread` with a function or a function object (incl. a [lambda](#)):

```
#include<thread>

    void f();

struct F {
    void operator()();
};

int main()
{
    std::thread t1{f}; // f() executes in separate thread
    std::thread t2{F()}; // F()() executes in separate thread
}
```

Unfortunately, this is unlikely to give any useful results -- whatever **f()** and **F()** might do. The snag is that the program may terminate before or after **t1** executes **f()** and before or after **t2** executes **F()**. We need to wait for the two tasks to complete:

```
int main()
{
    std::thread t1{f}; // f() executes in separate thread
    std::thread t2{F()}; // F()() executes in separate thread

    t1.join(); // wait for t1
    t2.join(); // wait for t2
}
```

The **join()**s ensure that we don't terminate until the threads have completed. To “join” means to “wait for the thread to terminate.”

Typically, we'd like to pass some arguments to the task to be executed (I call something executed by a thread a task). For example:

```
    void f(vector<double>&);

struct F {
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();
};

int main()
{
    std::thread t1{std::bind(f,some_vec)}; // f(some_vec) executes in separate thread
    std::thread t2{F(some_vec)}; // F(some_vec)() executes in separate thread

    t1.join();
    t2.join();
}
```

Basically, the standard library [bind](#) makes a function object of its arguments.

In general, we'd also like to get a result back from an executed task. With plain tasks, there is no notion of a return value; I recommend [std::future](#) for that. Alternative, we can pass an argument to a task telling it where to put its result: For example:

```

    void f(vector<double>&, double* res); // place result in res

struct F {
    vector<double>& v;
    double* res;
    F(vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()(); // place result in res
};

int main()
{
    double res1;
    double res2;

    std::thread t1{std::bind(f,some_vec,&res1)}; // f(some_vec,&res1) executes in separate thread
    std::thread t2{F(some_vec,&res2)}; // F(some_vec,&res2)() executes in separate thread

    t1.join();
    t2.join();

    std::cout << res1 << ' ' << res2 << '\n';
}

```

But what about errors? What if a task throws an exception? If a task throws an exception and doesn't catch it itself **std::terminate()** is called. That typically means that the program finishes. We usually try rather hard to avoid that. A **std::future** can transmit an exception to the parent/calling thread; that's one reason I like futures. Otherwise, return some sort of error code.

When a **thread** goes out of scope the program is **terminate()**d unless its task has completed. That's obviously to be avoided.

There is no way to request a **thread** to terminate (i.e. request that it exit as a soon as possible and as gracefully as possible) or to force a thread to terminate (i.e. kill it). We are left with the options of

- designing our own cooperative “interruption mechanism” (with a piece of shared data that a caller thread can set for a called thread to check (and quickly and gracefully exit when it is set)),
- “going native” (using **thread::native\_handle()** to gain access to the operating system's notion of a thread),
- kill the process (**std::quick\_exit()**),
- kill the program (**std::terminate()**).

This was all the committee could agree upon. In particular, representatives from POSIX were vehemently against any form of “thread cancellation” however much C++'s model of resources rely on destructors. There is no perfect solution for every systems and every possible application.

The basic problem with threads is data races; that is, two threads running in a single address space can independently access an object in ways that cause undefined results. If one (or both) writes to the object and the other (or both) reads the object they have a “race” for who gets its operation(s) done first. The results are not just undefined; they are usually completely unpredictable. Consequently, C++11 provides some rules/guarantees for the programmer to avoid data races:

- A C++ standard library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including **this**.
- A C++ standard library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's nonconst arguments, including **this**.
- C++ standard library implementations are required to avoid data races when different elements in the same sequence are modified concurrently.

Concurrent access to a stream object, stream buffer object, or C Library stream by multiple threads may result in a data race unless otherwise specified. So don't share an output stream between two threads unless you somehow control the access to it.

You can

- wait for a thread for [a specified time](#)
- control access to some data [by mutual exclusion](#)
- control access to some data [using locks](#)
- wait for an action of another task [using a condition variable](#)
- return a value from a thread [through a future](#)

See also

- Standard: 30 Thread support library [thread]
- 17.6.4.7 Data race avoidance [res.on.data.races]
- ???
- H. Hinnant, L. Crawl, B. Dawes, A. Williams, J. Garland, et al.: [Multi-threading Library for Standard C++ \(Revision 1\)](#) N2497==08-0007
- H.-J. Boehm, L. Crawl: [C++ object lifetime interactions with the threads API](#) N2880==09-0070.
- L. Crawl, P. Plauger, N. Stoughton: [Thread Unsafe Standard Functions](#) N2864==09-0054.
- WG14: [Thread Cancellation](#) N2455=070325.

## 2.10.18 Mutual Exclusion

A **mutex** is a primitive object use for controlling access in a multi-threaded system. The most basic use is

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// manipulate shared data:
sh+=1;
m.unlock();
```

Only one thread at a time can be in the region of code between the **lock()** and the **unlock()** (often called a critical region). If a second thread tries **m.lock()** while a first thread is executing in that region, that second thread is blocked until the first executes the **m.unlock()**. This is simple. What is not simple is to use mutexes in a way that doesn't cause serious problems: What if a thread “forgets” to **unlock()**? What if a thread tries to **lock()** the same mutex twice? What if a thread waits a very long time before doing an **unlock()**? What if a thread need to **lock()** two mutexes to do its job? The complete answers fill books. Here (and in the [Locks section](#)) are just the raw basics.

In addition to **lock()**, a **mutex** has a **try\_lock()** operation which can be used to try to get into the critical region without the risk of getting blocked:

```
std::mutex m;
int sh; // shared data
// ...
if (m.try_lock()) {
    // manipulate shared data:
    sh+=1;
    m.unlock();
} else {
    // maybe do something else
}
```

A **recursive\_mutex** is a **mutex** that can be acquired more than once by a thread:

```
std::recursive_mutex m;
int sh; // shared data
// ...
void f(int i)
{
    // ...
    m.lock();
```

```

// manipulate shared data:
sh+=1;
if (--i>0) f(i);
m.unlock();
// ...
}

```

Here, I have been blatant and let **f()** call itself. Typically, the code is more subtle. The recursive call will be indirect along the line of **f()** calls **g()** that calls **h()** that calls **f()**.

What if I need to acquire a **mutex** within the next ten seconds? The **timed\_mutex** class is offered for that. Its operations are specialized versions of **try\_lock()** with an associated time limit:

```

std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_for(std::chrono::seconds(10))) {
// manipulate shared data:
sh+=1;
m.unlock();
}
else {
// we didn't get the mutex; do something else
}

```

The **try\_lock\_for()** takes a relative time, a **duration** as its argument. If instead you want to wait until a fixed point in time, a **time\_point** you can use **try\_lock\_until()**:

```

std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_until(midnight)) {
// manipulate shared data:
sh+=1;
m.unlock();
}
else {
// we didn't get the mutex; do something else
}

```

The **midnight** is a feeble joke: for a mechanism as low level as mutexes, the timescale is more likely to be milliseconds than hours.

There is of course also a **recursive\_timed\_mutex**.

A mutex is considered a resource (as it is typically used to represent a real resource) and must be visible to at least two threads to be useful. Consequently, it cannot be copied or moved (you couldn't just make another copy of a hardware input register).

It can be surprisingly difficult to get the **lock()**s and **unlock()**s to match. Think of complicated control structures, errors, and exceptions. If you have a choice, use **locks** to manage your mutexes; that will save you and your users a lot of sleep.

See also

- Standard: 30.4 Mutual exclusion [thread.mutex]
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: [Multi-threading Library for Standard C++ \(Revision 1\)](#)
- ???

## 2.10.19 Locks

A **lock** is an object that can hold a reference to a **mutex** and may **unlock()** the **mutex** during the **lock**'s destruction (such as when leaving block scope). A **thread** may use a **lock** to aid in managing **mutex** ownership in an exception safe manner. In other words, a lock implements [Resource Acquisition Is Initialization](#) for mutual exclusion. For example:

```
std::mutex m;
int sh; // shared data
// ...
void f()
{
    // ...
    std::unique_lock lck(m);
    // manipulate shared data:
    sh+=1;
}
```

A lock can be moved (the purpose of a lock is to represent local ownership of a non-local resource), but not copied (which copy would own the resource/**mutex**?).

This straightforward picture of a lock is clouded by **unique\_lock** having facilities to do just about everything a mutex can, but safer. For example, we can use a lock to do try lock:

```
std::mutex m;
int sh; // shared data
// ...
void f()
{
    // ...
    std::unique_lock lck(m, std::defer_lock); // make a lock, but don't acquire the mutex
    // ...
    if (lck.try_lock()) {
        // manipulate shared data:
        sh+=1;
    }
    else {
        // maybe do something else
    }
}
```

Similarly, **unique\_lock** supports **try\_lock\_for()** and **try\_lock\_until()**. What you get from using a lock rather than the mutex directly is exception handling and protection against forgetting to **unlock()**. In concurrent programming, we need all the help we can get.

What if we need two resources represented by two mutexes? The naive way is to acquire the mutexes in order:

```
std::mutex m1;
std::mutex m2;
int sh1; // shared data
int sh2
// ...
void f()
{
    // ...
    std::unique_lock lck1(m1);
    std::unique_lock lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```

This has the potentially deadly flaw that some other thread could try to acquire **m1** and **m2** in the opposite order so that each had one of the locks needed to proceed and would wait forever for the second (that's called deadlock). With

many locks in a system, that's a real danger. Consequently, the standard locks provide two functions for (safely) trying to acquire two or more locks:

```
void f()
{
    // ...
    std::unique_lock lck1(m1, std::defer_lock); // make locks but don't yet try to acquire the mutexes
    std::unique_lock lck2(m2, std::defer_lock);
    std::unique_lock lck3(m3, std::defer_lock);
    lock(lck1, lck2, lck3);
    // manipulate shared data
}
```

Obviously, the implementation of **lock()** has to be carefully crafted to avoid deadlock. In essence, it will do the equivalent to careful use of **try\_lock()**s. If **lock()** fails to acquire all locks it will throw an exception. Actually, **lock()** can take any argument with **lock()**, **try\_lock()**, and **unlock()** member functions (e.g. a **mutex**), so we can't be specific about which exception **lock()** might throw; that depends on its arguments.

If you prefer to use **try\_lock()**s yourself, there is an equivalent to **lock()** to help:

```
void f()
{
    // ...
    std::unique_lock lck1(m1, std::defer_lock); // make locks but don't yet try to acquire the mutexes
    std::unique_lock lck2(m2, std::defer_lock);
    std::unique_lock lck3(m3, std::defer_lock);
    int x;
    if ((x = try_lock(lck1, lck2, lck3)) == -1) { // welcome to C land
        // manipulate shared data
    }
    else {
        // x holds the index of a mutex we could not acquire
        // e.g. if lck2.try_lock() failed x==1
    }
}
```

See also

- Standard: 30.4.3 Locks [thread.lock]
- ???

## 2.10.20 Condition Variables

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached.

Sorry, I have not had time to write this entry. Please come back later.

See also

- Standard: 30.5 Condition variables [thread.condition]
- ???

## 2.10.21 Time Utilities

We often want to time things or to do things dependent on timing. For example, the standard-library [mutexes](#) and [locks](#) provide the option for a [thread](#) to wait for a period of time (a **duration**) or to wait until a given point in time (a **time\_point**).

If you want to know the current **time\_point** you can call **now()** for one of three **clocks**: **system\_clock**, **monotonic\_clock**, **high\_resolution\_clock**. For example:

```

monotonic_clock::time_point t = monotonic_clock::now();
// do something
monotonic_clock::duration d = monotonic_clock::now() - t;
// something took d time units

```

A **clock** returns a **time\_point**, and a **duration** is the difference of two **time\_points** from the same **clock**. As usual, if you are not interested in details, **auto** is your friend:

```

auto t = monotonic_clock::now();
// do something
auto d = monotonic_clock::now() - t;
// something took d time units

```

The time facilities here are intended to efficiently support uses deep in the system; they do not provide convenience facilities to help you maintain your social calendar. In fact, the time facilities originated with the stringent needs for high-energy physics. To be able to express all time scales (such as centuries and picoseconds), avoid confusion about units, typos, and rounding errors, **durations** and **time\_points** are expressed using a compile-time rational number package. A **duration** has two parts: a numbers clock “tick” and something (a “period”) that says what a tick means (is it a second or a millisecond?); the period is part of a **durations** type. This table from the standard header `<ratio>`, defining the periods of the SI system (also known as MKS or metric system) might give you an idea of the scope of use:

```

// convenience SI typedefs:
typedef ratio<1, 1000000000000000000000000> yocto; // conditionally supported
typedef ratio<1, 100000000000000000000000> zepto; // conditionally supported
typedef ratio<1, 10000000000000000000000> atto;
typedef ratio<1, 1000000000000000000000> femto;
typedef ratio<1, 100000000000000000000> pico;
typedef ratio<1, 10000000000000000000> nano;
typedef ratio<1, 1000000000000000000> micro;
typedef ratio<1, 100000000000000000> milli;
typedef ratio<1, 10000000000000000> centi;
typedef ratio<1, 1000000000000000> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000, 1> tera;
typedef ratio<1000000000000000, 1> peta;
typedef ratio<1000000000000000000, 1> exa;
typedef ratio<1000000000000000000000, 1> zetta; // conditionally supported
typedef ratio<1000000000000000000000000, 1> yotta; // conditionally supported

```

The compile time rational numbers provide the usual arithmetic (+, -, \*, and /) and comparison (==, !=, <, <=, >, >=) operators for whatever combinations **durations** and **time\_points** makes sense (e.g. you can't add two **time\_points**). These operations are also checked for overflow and divide by zero. Since this is a compile-time facility, don't worry about run-time performance. In addition you can use ++, --, +=, -=, \*=, and /= on **durations** and **tp+=d** and **tp-=d** for a **time\_point tp** and a **duration d**.

Here are some examples of values using standard **duration** types as defined in `<chrono>`:

```

microseconds mms = 12345;
milliseconds ms = 123;
seconds s = 10;
minutes m = 30;
hours h = 34;

auto x = std::chrono::hours(3); // being explicit about namespaces
auto x = hours(2)+minutes(35)+seconds(9); // assuming suitable "using"

```

You cannot initialize a **duration** to a fraction. For example, don't try 2.5 seconds; instead use 2500 milliseconds. This is because a **duration** is interpreted as a number of “ticks.” Each tick represent on unit of the **duration**'s “period,” such as **milli** and **kilo** as defined above. The default unit is **seconds**; that is, for a **duration** with a period of 1 a tick is interpreted as a second. We can be explicit about the representation of a **duration**:

```
duration<long> d0 = 5; // seconds (by default)
duration<long,kilo> d1 = 99; // kiloseconds!
duration<long,ratio<1000,1>> d2 = 100; // d1 and d2 have the same type ("kilo" means "*1000")
```

If we actually want to do something with a **duration**, such as writing it out, we have to give a unit, such as **minutes** or **microseconds**. For example:

```
auto t = monotonic_clock::now();
// do something
nanoseconds d = monotonic_clock::now() - t; // we want the result in nanoseconds
cout << "something took " << d << "nanoseconds\n";
```

Alternatively, we could convert the **duration** to a floating point number (to get rounding):

```
auto t = monotonic_clock::now();
// do something
auto d = monotonic_clock::now() - t;
cout << "something took " << duration_cast<double>(d).count() << "seconds\n";
```

The **count()** is the number of “ticks.”

See also

- Standard: 20.9 Time utilities [time]
- Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno: [A Foundation to Sleep On](#). N2661=08-0171. Including “A Brief History of Time” (With apologies to Stephen Hawking).

## 2.10.22 Atomics

Sorry, I have not had time to write this entry. Please come back later.

See also

- Standard: 29 Atomic operations library [atomics]
- ???

## 2.10.23 std::future and std::promise

Concurrent programming can be *hard*, especially if you try to be clever with [threads](#), and [locks](#). It is harder still if you must use [condition variables](#) or use [std-atomics](#) (for lock-free programming). C++11 offers **future** and **promise** for returning a value from a task spawned on a separate thread and **packaged\_task** to help launch tasks. The important point about **future** and **promise** is that they enable a transfer of a value between two tasks without explicit use of a lock; “the system” implements the transfer efficiently. The basic idea is simple: When a task wants to return a value to the **thread** that launched it, it puts the value into a **promise**. Somehow, the implementation makes that value appear in the **future** attached to the **promise**. The caller (typically the launcher of the task) can then read the value. For added simplicity, see [async\(\)](#).

The standard provides three kinds of futures, **future** for most simple uses, and **shared\_future** and **atomic\_future** for some trickier cases. Here, I'll just present **future** because it's the simplest and does all I need done. If we have a **future<X>** called **f**, we can **get()** a value of type **X** from it:

```
x v = f.get(); // if necessary wait for the value to get computed
```

If the value isn't there yet, our thread is blocked until it arrives. If the value couldn't be computed, the result of **get()** might be to throw an exception (from the system or transmitted from the task from which we were trying to **get()** the value.

We might not want to wait for a result, so we can ask the **future** if a result has arrived:

```
if (f.wait_for(0)) { // there is a value to get()
    // do something
}
else {
    // do something else
}
```

However, the main purpose of **future** is to provide that simple **get()**.

The main purpose of **promise** is to provide a simple “put” (curiously, called “set”) to match **future**'s **get()**. The names “future” and “promise” are historical; please don't blame me. They are also a fertile source of puns.

If you have a **promise** and need to send a result of type **X** (back) to a **future**, there are basically two things you can do: pass a value and pass an exception:

```
try {
    X res;
    // compute a value for res
    p.set_value(res);
}
catch (...) { // oops: couldn't compute res
    p.set_exception(std::current_exception());
}
```

So far so good, but how do I get a matching **future/promise** pair, one in my **thread** and one in some other **thread**? Well, since **futures** and **promises** can be moved (not copied) around there is a wide variety of possibilities. The most obvious idea is for whoever wants a task done to create a thread and give the promise to it while keeping the corresponding future as the place for the result. Using **async()** is the most extreme/elegant variant of the latter technique.

The **packaged\_task** type is provided to simplify launching a thread to execute a task. In particular, it takes care of setting up a **future** connected to a **promise** and to provides the wrapper code to put the return value or exception from the task into the **promise**. For example:

```
double comp(vector<double>& v)
{
    // package the tasks:
    // (the task here is the standard accumulate() for an array of doubles):
    packaged_task<double(double*,double*,double)> pt0{std::accumulate<double*,double*,double>};
    packaged_task<double(double*,double*,double)> pt1{std::accumulate<double*,double*,double>};

    auto f0 = pt0.get_future(); // get hold of the futures
    auto f1 = pt1.get_future();

    pt0(&v[0],&v[v.size()/2],0); // start the threads
    pt1(&v[v.size()/2],&v[size()],0);

    return f0.get()+f1.get(); // get the results
}
```

See also

- Standard: 30.6 Futures [futures]
- Anthony Williams: [Moving Futures - Proposed Wording for UK comments 335, 336, 337 and 338](#). N2888==09-0078.
- Detlef Vollmann, Howard Hinnant, and Anthony Williams [An Asynchronous Future Value \(revised\)](#) N2627=08-0137.
- Howard E. Hinnant: [Multithreading API for C++0X - A Layered Approach](#). N2094=06-0164. The original proposal for a complete threading package..

## 2.10.24 std::async()

The `async()` simple task launcher function is the only facility in this FAQ that has not yet been voted into the draft standard. I expect it to be voted in in October after reconciling two slightly different proposals (feel free to tell your local committee member to be sure to vote for it).

Here is an example of a way for the programmer to rise above the messy threads-plus-lock level of concurrent programming:

```
template<class T, class V> struct Accum { // simple accumulator function object
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& v) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
// spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

This is a very simple-minded use of concurrency (note the “magic number”), but note the absence of explicit **threads**, **locks**, buffers, etc. The type of the **f**-variables are determined by the return type of the standard-library function `async()` which is a [future](#). If necessary, `get()` on a **future** waits for a [thread](#) to finish. Here, it is `async()`'s job to spawn **threads** as needed and the **future**'s job to `join()` the **threads** appropriately. “Simple” is the most important aspect of the `async()/future` design; futures can also be used with threads in general, but don't even *think* of using `async()` to launch tasks that do I/O, manipulates mutexes, or in other ways interact with other tasks. The idea behind `async()` is the same as the idea behind the range-`for` statement: Provide a simple way to handle the simplest, rather common, case and leave the more complex examples to the fully general mechanism.

An `async()` can be requested to launch in a new **thread**, in any **thread** but the caller's, or to launch in a different **thread** only if `async()` “thinks” that it is a good idea. The latter is the simplest from the user's perspective and potentially the most efficient (for *simple* tasks(only)).

See also

- Standard: ???
- Lawrence Crowl: [An Asynchronous Call for C++](#). N2889 = 09-0079.
- Herb Sutter : [A simple async\(\)](#) N2901 = 09-0091 .

## 2.10.25 Abandoning a Process

Sorry, I have not had time to write this entry. Please come back later.

- [Abandoning a process](#)

## 2.10.26 Random Number Generation

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of application areas is reflected in the wide selection of random number generators provided by the standard library. A random number generator consists of two parts an *engine* that produces a sequence of random or pseudo-random values and a *distribution* that maps those values in to a mathematical distribution in a range. Examples of distributions

are **uniform\_int\_distribution** (where all integers produced are equally likely) and **normal\_distribution** (“the bell curve”); each for some specified range. For example:

```
uniform_int_distribution<int> one_to_six {1,6}; // distribution that maps to the ints 1..6
default_random_engine re {}; // the default engine
```

To get a random number, you call a distribution with an engine:

```
int x = one_to_six(re); // x becomes a value in [1:6]
```

Passing the engine in every call can be considered tedious, so we could bind that argument to get a function object that we can call without arguments:

```
auto dice {bind(one_to_six,re)}; // make a generator

int x = dice(); // roll the dice: x becomes a value in [1:6]
```

Thanks to its uncompromising attention to generality and performance one expert deemed the standard-library random number component “what every random number library wants to be when it grows up.” However, it can hardly be deemed “novice friendly.” I have never found the random number interface to be a performance bottle neck, but I have never taught novices (of any background) without needing a very simple random number generator. This would be sufficient

```
int rand_int(int low, int high); // generate a random number from a uniform distribution in [low:high]
```

So, how could we get that? We have to get something like **dice()** inside **rand\_int()**:

```
int rand_int(int low, int high)
{
    static default_random_engine re {};
    using Dist = uniform_int_distribution<int>;
    static Dist uid {};
    return uid(re, Dist::param_type{low,high});
}
```

That definition is still “expert level” but the *use* of **rand\_int()** manageable in the first week of a C++ course.

Just to show a non-trivial example, here is a program that generates and prints a normal distribution:

```
default_random_engine re; // the default engine
normal_distribution<int> nd(31 /* mean */,8 /* sigma */);

auto norm = std::bind(nd, re);

vector<int> mn(64);

int main()
{
    for (int i = 0; i<1200; ++i) ++mn[round(norm())]; // generate

    for (int i = 0; i<mn.size(); ++i) {
        cout << i << '\t';
        for (int j=0; j<mn[i]; ++j) cout << '*';
        cout << '\n';
    }
}
```

The result was:

0  
1  
2  
3  
4 \*  
5  
6  
7  
8  
9 \*  
10 \*\*\*  
11 \*\*\*  
12 \*\*\*  
13 \*\*\*\*\*  
14 \*\*\*\*\*  
15 \*\*\*\*  
16 \*\*\*\*\*  
17 \*\*\*\*\*  
18 \*\*\*\*\*  
19 \*\*\*\*\*  
20 \*\*\*\*\*  
21 \*\*\*\*\*  
22 \*\*\*\*\*  
23 \*\*\*\*\*  
24 \*\*\*\*\*  
25 \*\*\*\*\*  
26 \*\*\*\*\*  
27 \*\*\*\*\*  
28 \*\*\*\*\*  
29 \*\*\*\*\*  
30 \*\*\*\*\*  
31 \*\*\*\*\*  
32 \*\*\*\*\*  
33 \*\*\*\*\*  
34 \*\*\*\*\*  
35 \*\*\*\*\*  
36 \*\*\*\*\*  
37 \*\*\*\*\*  
38 \*\*\*\*\*  
39 \*\*\*\*\*  
40 \*\*\*\*\*  
41 \*\*\*\*\*  
42 \*\*\*\*\*  
43 \*\*\*\*\*  
44 \*\*\*\*\*  
45 \*\*\*\*\*  
46 \*\*\*\*\*  
47 \*\*\*\*\*  
48 \*\*\*\*\*  
49 \*\*\*\*\*  
50 \*\*\*\*  
51 \*\*\*  
52 \*\*\*  
53 \*\*  
54 \*  
55 \*  
56  
57 \*  
58  
59  
60

61  
62  
63

See also

- Standard 26.5: Random number generation

## 2.10.27 Regular Expressions

- 28 Regular expressions library

Sorry, I have not had time to write this entry. Please come back later.

See also

- Standard: ???
- ???

## 2.10.28 Concepts

**Warning: "concepts" did not make it into C++11 and a radical redesign is in progress**

"Concepts" is a mechanism for describing requirements on types, combinations of types, and combinations of types and integers. It is particularly useful for getting early checking of uses of templates. Conversely, it also helps early detection of errors in a template body. Consider the standard library algorithm **fill**:

```
template<ForwardIterator Iter, class V>           // types of types
requires Assignable<Iter::value_type,V>        // relationships among argument types
void fill(Iter first, Iter last, const V& v);    // just a declaration, not definition

fill(0, 9, 9.9); // Iter is int; error: int is not a ForwardIterator
//                int does not have a prefix *
fill(&v[0], &v[9], 9.9); // Iter is int*; ok: int* is a ForwardIterator
```

Note that we only declared **fill()**; we did not define it (provide its implementation). On the other hand, we explicitly stated what **fill()** requires from its argument:

- The arguments **first** and **last** must be of a type that is a **ForwardIterator** (and they must be of the same type).
- The third argument **v** must be of a type that can be assigned to the **ForwardIterator's value\_type**.

We knew that, of course, having read the standard. However, compilers do not read requirement documents, so we had to tell it in code using the concepts **ForwardIterator** and **Assignable**. The result is that errors in the use of **fill()** are caught immediately at the point of use and that error messages are greatly improved. The compiler now has the information about the programmers' intents to allow good checking and good diagnostics.

Concepts also help template implementers. Consider:

```
template<ForwardIterator Iter, class V>
requires Assignable<Iter::value_type,V>
void fill(Iter first, Iter last, const V& v)
{
while (first!=last) {
*first = v;
first=first+1; // error: + not defined for ForwardIterator
// (use ++first)
}
}
```

This error is caught immediately, eliminating the need for much tedious testing (though of course not all testing).

Being able to classify and distinguish different types of types, we can overload based on the kind of types passed. For

example

```
// iterator-based standard sort (with concepts):
template<Random_access_iterator Iter>
requires Comparable<Iter::value_type>
void sort(Iter first, Iter last); // use the usual implementation

// container-based sort:
template<Container Cont>
requires Comparable<Cont::value_type>
void sort(Cont& c)
{
    sort(c.begin(),c.end()); // simply call the iterator version
}

void f(vector<int>& v)
{
    sort(v.begin(), v.end()); // one way
    sort(v); // another way
    // ...
}
```

You can define your own concepts, but for starters the standard library provides a variety of useful concepts, such as **ForwardIterator**, **Callable**, **LessThanComparable**, and **Regular**.

Note: the C++0x standard libraries were specified using concepts.

See also

- the C++ draft 14.10 Concepts
- [N2617=08-0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: [Proposed Wording for Concepts \(Revision 5\)](#) (Final proposal).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: [Concepts: Linguistic Support for Generic Programming in C++](#). OOPSLA'06, October 2006.

## 2.10.29 Concept Maps

An `int*` is a **ForwardIterator**; we said so when presenting [concepts](#), the standard has always said so, and even the first version of the STL used pointers as iterators. However, we also talked about **ForwardIterator's value\_type**. But an `int*` does not have a member called `value_type`; in fact, it has no members. So how can an `int*` be a **ForwardIterator**? It is because we say it is. Using a **concept\_map**, we say that when a `T*` is used where a **ForwardIterator** is required, we consider the `T` its `value_type`:

```
template<Value_type T>
concept_map ForwardIterator<T*> { // T*'s value_type is T
    typedef T value_type;
};
```

A **concept\_map** allows us to say how we want to see a type, saving us from having to modify it or to wrap it into a new type. "Concept maps" is a very flexible and general mechanism for adapting independently developed software for common use.

- the C++ draft 14.10.2 Concept maps
- [N2617=08-0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: [Proposed Wording for Concepts \(Revision 5\)](#) (Final proposal).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine: [Concepts: Linguistic Support for Generic Programming in C++](#). OOPSLA'06, October 2006.

## 2.10.30 Axioms

An axiom is a set of predicates specifying the semantics of a concept. The primary use cases for axioms are external tools (e.g. not the common compiler actions), such as tools for domain-specific optimizations (languages for specifying program transformations were a significant part of the motivation for axioms). A secondary use is simply precise

specification of semantics in the standard (as is used in many parts of the standard library specification). Axioms may also be useful for some optimizations (done by compilers and traditional optimizers), but compilers are *not* required to take notice of user-supplied axioms; they work based on the semantics defined by the standard.

An axiom lists pairs of computations that may be considered equivalent. Consider:

```
concept Semigroup<typename Op, typename T> : CopyConstructible<T> {
    T operator()(Op, T, T);
    axiom Associativity(Op op, T x, T y, T z) {
        op(x, op(y, z)) <=> op(op(x, y), z); // T's operator may be assumed to be associative
    }
}

concept Monoid<typename Op, typename T> : Semigroup<Op, T> { // a monoid is a semigroup with an identity el
    T identity_element(Op);
    axiom Identity(Op op, T x) {
        op(x, identity_element(op)) <=> x;
        op(identity_element(op), x) <=> x;
    }
}
```

The `<=>` is the equivalence operator, which is used only in axioms. Note that you cannot (in general) prove an axiom; we use axioms to state what we cannot prove, but what a programmer can state to be an acceptable assumption. Note that both sides of an equivalence statement may be illegal for some values, e.g. use of a NaN (not a number) for a floating-point type: if both sides of an equivalence uses a NaN both are (obviously) invalid and equivalent (independently of what the axiom says), but if only one side uses a NaN there may be opportunities for taking advantage of the axiom.

An axiom is a sequence of equivalence statements (using `<=>`) and conditional statements (of the form "if (something) then we may assume the following equivalence"):

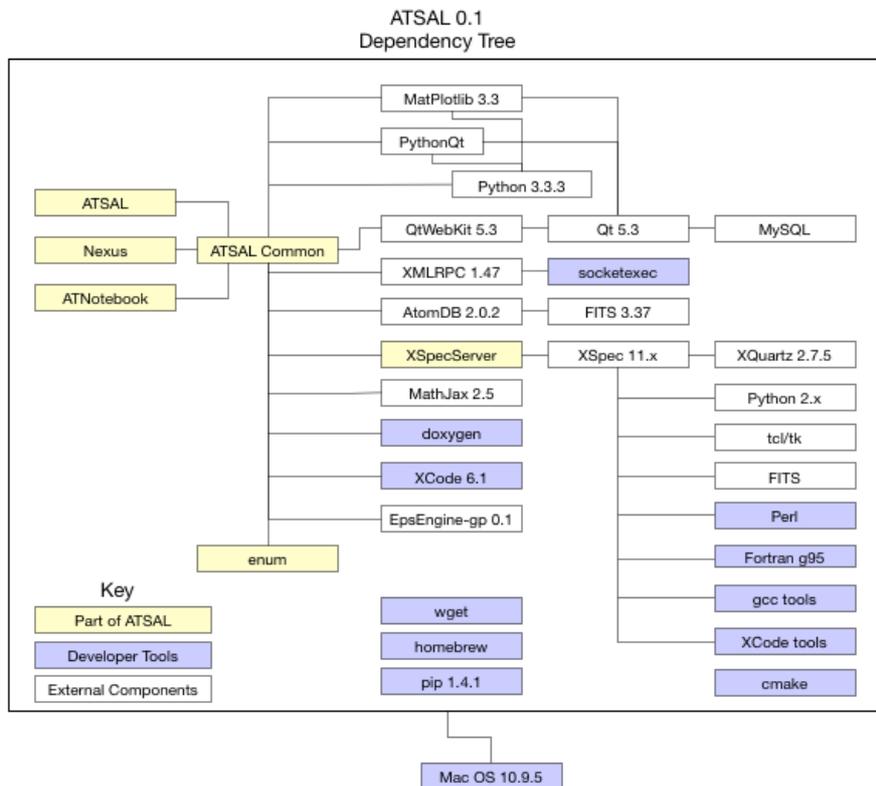
```
// in concept TotalOrder:
axiom Transitivity(Op op, T x, T y, T z)
{
    if (op(x, y) && op(y, z)) op(x, z) <=> true; // conditional equivalence
}
```

See also

- the C++ draft 14.9.1.4 Axioms
- ???.

## 3 AT SAL Classes

This section contains documentation on the classes comprising AT SAL, along with pointers to API documentation for the open source components that AT SAL uses. In the diagram below, yellow components are part of AT SAL itself, or, in the case of XSpecServer, a modification of XSpec currently maintained as part of the AT SAL project. Components in white boxes are external to the AT SAL (or XSpec) projects. Those in blue are software development tools that are typically needed to develop AT SAL.



### 3.1 Qt Cross-platform Class Library

AT SAL is implemented in Qt, a mature cross-platform class library. This section contains links to some commonly used Qt documentation. It should be kept up to date for the current version. Web searches often yield links to older versions.

#### Qt Main Page

[Qt Main Page](#)

(Remote URLs are not yet included inline in PDFs.)

#### Qt Complete Class List

[Qt Complete Class List](#)

(Remote URLs are not yet included inline in PDFs.)

##### 3.1.1 Qt Main Classes

These are the most frequently used Qt classes. For the complete list see [Qt's Classes](#).

<b>A</b> <a href="#">QAbstractItemDelegate</a>	<a href="#">QFrame</a>	<a href="#">QPen</a>	<a href="#">QStack</a>
<a href="#">QAbstractItemModel</a>	<a href="#">QFtp</a>	<a href="#">QPicture</a>	<a href="#">QStackedLayout</a>
<a href="#">QAbstractItemView</a>	<b>G</b> <a href="#">QGLWidget</a>	<a href="#">QPixmap</a>	<a href="#">QStackedWidget</a>
<a href="#">QAccessible</a>	<a href="#">QGraphicsScene</a>	<a href="#">QPlainTextEdit</a>	<a href="#">QStatusBar</a>
<a href="#">QAction</a>	<a href="#">QGraphicsView</a>	<a href="#">QPluginLoader</a>	<a href="#">QString</a>

QApplication	QGridLayout	QPointer	QStringList
<b>B</b> QButtonGroup	QGroupBox	QPrinter	QStringListModel
QByteArray	<b>H</b> QHash	QProcess	QStyledItemDelegate
<b>C</b> QCache	QHBoxLayout	QProgressBar	<b>T</b> QTabBar
QCalendarWidget	QHeaderView	QProgressDialog	QTableView
QCheckBox	QHttp	QPushButton	QTableWidget
QClipboard	<b>I</b> QIcon	<b>Q</b> QQueue	QTabWidget
QColor	QImage	<b>R</b> QRadioButton	QTemporaryFile
QColorDialog	QInputDialog	QRegExp	QTextCursor
QColumnView	QItemDelegate	QResource	QTextDocument
QComboBox	<b>K</b> QKeySequence	QRubberBand	QTextEdit
QCommandLinkButton	<b>L</b> QLabel	<b>S</b> QScriptable	QThread
QCoreApplication	QLCDNumber	QScriptClass	QThreadStorage
QCursor	QLibrary	QScriptContext	QTime
<b>D</b> QDataStream	QLibraryInfo	QScriptContextInfo	QTimeEdit
QDate	QLineEdit	QScriptEngine	QTimer
QDateEdit	QLinkedList	QScriptEngineAgent	QToolBar
QDateTime	QList	QScriptEngineDebugger	QToolBox
QDateTimeEdit	QListView	QScriptString	QToolButton
QDesktopWidget	QListWidget	QScriptSyntaxCheckResult	QToolTip
QDial	QLocale	QScriptValue	QTranslator
QDialog	<b>M</b> QMainWindow	QScriptValueIterator	QTreeView
QDialogButtonBox	QMap	QScrollArea	QTreeWidget
QDir	QMdiArea	QSet	<b>U</b> QUrl
QDomDocument	QMdiSubWindow	QSettings	<b>V</b> QValidator
QDomNode	QMenu	QSharedDataPointer	QVariant
QDoubleSpinBox	QMenuBar	QShortcut	QVBoxLayout
<b>E</b> QExplicitlySharedDataPointer	QMessageBox	QSignalMapper	QVector
<b>F</b> QFile	QModelIndex	QSlider	<b>W</b> QWhatsThis
QFileDialog	QMultiHash	QSound	QWidget
QFlags	QMultiMap	QSpinBox	QWidgetAction
QFocusFrame	QMutex	QSplashScreen	<b>X</b> QXmlSimpleReader
QFont	<b>O</b> QObject	QSplitter	QXmlStreamReader
QFontDialog	<b>P</b> QPainter	QSqlDatabase	QXmlStreamWriter
QFormLayout	QPalette	QSqlQuery	

### 3.1.2 Qt Classes Used Often by ATSAL

Use the Table... command to convert this list between a one-topic-per-line list and a table. Makes it a lot easier to update the list over time.

QAbstractItemDelegate	QLinkedListIterator	QSharedDataPointer	QTimeEdit
QAbstractItemModel	QList	QSharedMemory	QTimeLine
QAbstractItemView	QListIterator	QSharedPointer	QTimer
QAbstractListModel	QListView	QShortcut	QTimerEvent
QAbstractScrollArea	QListWidget	QSize	QTimeZone
QAbstractSlider	QListWidgetItem	QSizeF	QToolBar
QAction	QMacToolBar	QSizePolicy	QToolBox
QApplication	QMacToolBarItem	QSlider	QToolTip
QAtomicInt	QMainWindow	QSplashScreen	QTransform
QAtomicInteger	QMap	QSplitter	QTranslator
QAtomicPointer	QMapIterator	QSplitterHandle	QTreeView
QBitArray	QMenu	QSqlDatabase	QTreeWidget
QBitmap	QMenuBar	QSqlDriver	QTreeWidgetItem

QBoxLayout	QMessageBox	QSqlDriverCreator	QTreeWidgetItemIterator
QBrush	QMetaClassInfo	QSqlDriverCreatorBase	QUndoCommand
QBuffer	QMimeData	QSqlDriverPlugin	QUndoGroup
QButtonGroup	QMimeType	QSqlError	QUndoStack
QByteArray	QModelIndex	QSqlField	QUndoView
QChar	QMouseEvent	QSqlIndex	QUrl
QCheckBox	QMoveEvent	QSqlQuery	QUrlQuery
QClipboard	QMutex	QSqlQueryModel	QValidator
QCloseEvent	QMutexLocker	QSqlRecord	QVariant
QColor	QNetworkAccessManager	QSqlRelation	QVBoxLayout
QColorDialog	QNetworkAddressEntry	QSqlRelationalDelegate	QVector
QColumnView	QNetworkCacheMetaData	QSqlRelationalTableModel	QVector2D
QComboBox	QNetworkConfiguration	QSqlResult	QVector3D
QCoreApplication	QNetworkConfigurationManager	QSqlTableModel	QVector4D
QCursor	QNetworkCookie	QStack	QVectorIterator
D	QNetworkCookieJar	QStackedLayout	QWeakPointer
QDataStream	QNetworkDiskCache	QStackedWidget	QWebDatabase
QDate	QNetworkInterface	QState	QWebElement
QDateTime	QNetworkProxy	QStateMachine	QWebElementCollection
QDialog	QNetworkProxyFactory	QStatusBar	QWebFrame
QDialogButtonBox	QNetworkProxyQuery	QString	QWebHistory
QDir	QNetworkReply	QStringList	QWebHistoryInterface
QDirIterator	QNetworkRequest	QStringRef	QWebHistoryItem
QDockWidget	QNetworkSession	QSvgGenerator	QWebHitTestResult
QDomAttr	QObject	QSvgRenderer	QWebInspector
QDrag	QPageLayout	QSvgWidget	QWebPage
QDragEnterEvent	QPageSetupDialog	QTabBar	QWebPluginFactory
QDragLeaveEvent	QPaintDevice	QTableView	QWebSettings
QDragMoveEvent	QPainter	QTableWidget	QWebSocket
QDropEvent	QPainterPath	QTableWidgetItem	QWebSocketCorsAuthenticator
QEnterEvent	QPainterPathStroker	QTcpServer	QWebSocketServer
QErrorMessage	QPaintEvent	QTcpSocket	QWebView
QEvent	QPair	QTemporaryDir	QWidget
QException	QPalette	QTemporaryFile	QWidgetItem
QFile	QPdfWriter	QTextBlock	QWindow
QFileDevice	QPen	QTextBlockFormat	QXmlAttributes
QFileDialog	QPersistentModelIndex	QTextBlockGroup	QXmlContentHandler
QFileInfo	QPicture	QTextBlockUserData	QXmlDeclHandler
QFlag	QPixmap	QTextBrowser	QXmlDefaultHandler
QFlags	QPlainTextEdit	QTextCharFormat	QXmlDTDHandler
QFont	QPoint	QTextCodec	QXmlEntityResolver
QFontComboBox	QPointer	QTextCursor	QXmlErrorHandler
QFontDialog	QPointF	QTextDecoder	QXmlFormatter
QFontInfo	QPolygon	QTextDocument	QXmlInputSource
QFontMetrics	QPolygonF	QTextDocumentFragment	QXmlItem
QFontMetricsF	QProcess	QTextDocumentWriter	QXmlLexicalHandler

QFrame	QProgressBar	QTextEdit	QXmlLocator
QGLWidget	QProgressDialog	QTextEncoder	QXmlName
QGradient	QPushButton	QTextFormat	QXmlNamePool
QGraphicsWidget	QRadioButton	QTextFragment	QXmlNamespaceSupport
QGridLayout	QRect	QTextFrame	QXmlNodeModelIndex
QGroupBox	QRectF	QTextFrameFormat	QXmlParseException
QHBoxLayout	QRegExp	QTextImageFormat	QXmlQuery
QIcon	QRegExpValidator	QTextInlineObject	QXmlReader
QIconEngine	QRegion	QTextItem	QXmlResultItems
QImage	QRegularExpression	QTextLayout	QXmlSchema
QIODevice	QRegularExpressionMatch	QTextLength	QXmlSchemaValidator
QItemDelegate	QRegularExpressionMatchIterator	QTextLine	QXmlSerializer
QKeyEvent	QRegularExpressionValidator	QTextList	QXmlSimpleReader
QKeyEventTransition	QResizeEvent	QTextListFormat	QXmlStreamAttribute
QKeySequence	QResource	QTextObject	QXmlStreamAttributes
QKeySequenceEdit	QScreen	QTextObjectInterface	QXmlStreamEntityDeclaration
QLabel	QScrollArea	QTextOption	QXmlStreamEntityResolver
QLayout	QScrollBar	QTextStream	QXmlStreamNamespaceDeclaration
QLayoutItem	QScroller	QTextTable	QXmlStreamNotationDeclaration
QLCDNumber	QScrollEvent	QTextTableCell	QXmlStreamReader
QLine	QSessionManager	QTextTableCellFormat	QXmlStreamWriter
QLinearGradient	QSet	QTextTableFormat	
QLineEdit	QSetIterator	QThread	
QLineF	QSettings	QThreadPool	
QLinkedList	QSharedData	QTime	

## QMake Manual

[QMake Manual](#)

(Remote URLs are not yet included inline in PDFs.)

## Deploying on Mac OS X

[Deploying on Mac OS X](#)

(Remote URLs are not yet included inline in PDFs.)

### 3.1.3 Qt Add-ons

These are some open source Qt extensions of possible future use to ATSAL.

Reports	<a href="http://www.kdab.com/kd-reports/">http://www.kdab.com/kd-reports/</a>
Text Widgets	<a href="http://api.kde.org/frameworks-api/frameworks5-apidocs/ktextwidgets/html/index.html">http://api.kde.org/frameworks-api/frameworks5-apidocs/ktextwidgets/html/index.html</a>
XMLRPC	<a href="http://api.kde.org/frameworks-api/frameworks5-apidocs/kxmlrpcclient/html/index.html">http://api.kde.org/frameworks-api/frameworks5-apidocs/kxmlrpcclient/html/index.html</a>
Plots	<a href="http://www.qcustomplot.com/index.php/introduction">http://www.qcustomplot.com/index.php/introduction</a>
DropBox	<a href="http://include.org/libraries/qtdropbox.html">http://include.org/libraries/qtdropbox.html</a>
Reports	<a href="http://include.org/libraries/qtrpt.html">http://include.org/libraries/qtrpt.html</a>

## Matplotlib

Matplotlib

(Remote URLs are not yet included inline in PDFs.)

## 4 Matplotlib Notes

### 4.1.1.1 Matplotlib and PythonQt

Below is from the [PythonQt discussion on Sourceforge](#). A user posts this snippet of sample code for creating a plot:

```
import sys
sys.path.append("/usr/lib/pymodules/python2.6/matplotlib")
# maybe others ...
from pylab import *
t = arange(0.0, 2.0, 0.01)
s = sin(2*pi*t)
plot(t, s, linewidth=1.0)
xlabel('time (s)')
ylabel('voltage (mV)')
title('About as simple as it gets, folks')
grid(True)
show()
```

Florian Link, developer of PythonQt, comments:

Your test only works because PyQt or PySide are installed with your Python and the matplotlib backend uses those. To make this work in a standalone application you will need a PythonQt backend. MeVisLab 2.3 contains a matplotlib backend that was ported to PythonQt, by coincidence I did that port. It is not implemented as a full backend of its own, just a patch that replaces calls to PyQt/PySide with PythonQt in the Qt4 backend.

I did not have time to clean it up and to provide it back to the matplotlib community. It is fully functional except for the figure options dialog. You can get it by downloading MeVisLab 2.3 RC and going to

```
INSTALLDIR/Packages/MeVis/ThirdParty/Python/.../Libs/matplotlib/...
```

(or Libs/site-packages if you are downloading 2.3 Stable Release) and taking the matplotlib/backends/ directory. You can diff that against your matplotlib version to see the changes.

### 4.1.1.2 MeVis Matplotlib Files

```
./Packages/MeVis/ThirdParty/Configuration/Installers/Libraries/matplotlib.mli
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/backends/Matplotlib.nib
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/mpl-data/images/matplotlib.gif
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/mpl-data/images/matplotlib.png
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/mpl-data/images/matplotlib.svg
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/mpl-data/matplotlibrc
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib-1.4.3-py2.7.egg-info
./Packages/MeVis/ThirdParty/ThirdPartyInformation/matplotlib
./Packages/MeVis/ThirdParty/ThirdPartyInformation/matplotlib/matplotlib.license
./Packages/MeVis/ThirdParty/ThirdPartyInformation/matplotlib/matplotlib.mlinfo
./Packages/MeVisLab/Examples/Documentation/Publish/ModuleReference/MatplotlibMDLExample.html
./Packages/MeVisLab/Examples/Documentation/Publish/ModuleReference/MatplotlibMDLExampleBanner.ht

./Packages/MeVisLab/Examples/Modules/Macros/Matplotlib
./Packages/MeVisLab/Examples/Modules/Macros/Matplotlib/MatplotlibMDLExample.def
./Packages/MeVisLab/Examples/Modules/Macros/Matplotlib/MatplotlibMDLExample.py
./Packages/MeVisLab/Examples/Modules/Macros/Matplotlib/MatplotlibMDLExample.pyc
./Packages/MeVisLab/Examples/Modules/Macros/Matplotlib/MatplotlibMDLExample.script
./Packages/MeVisLab/Resources/Documentation/Publish/SDK/NewInMeVisLab2_3/figures/matplotlib.png
./Packages/MeVisLab/Standard/Modules/Controls/MatplotlibControls.def
./Packages/MeVisLab/Standard/Modules/Scripts/python/StandardControls/MatplotlibControls.py
```





```

./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/tests/test_backend_qt5.pyc
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/tests/test_backend_svg.py
./Packages/MeVis/ThirdParty/lib/MLAB_Python.framework/Versions/2.7/lib/python2.7/site-
packages/matplotlib/tests/test_backend_svg.pyc
./Packages/MeVis/ThirdParty/lib/phonon.framework/Versions/4/Headers/backendcapabilities.h
./Packages/MeVis/ThirdParty/lib/phonon.framework/Versions/4/Headers/backendinterface.h
./Packages/MeVis/ThirdParty/lib/qtplugins/phonon_backend
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/locale/localization_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/basic_sink_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/debug_output_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/event_log_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/syslog_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/text_file_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/text_multifile_backend.hpp
./Packages/MeVis/ThirdParty/Sources/misc/boost/boost/log/sinks/text_ostream_backend.hpp
./Packages/MeVis/ThirdParty/Sources/Qt4/qt/include/phonon/backendcapabilities.h
./Packages/MeVis/ThirdParty/Sources/Qt4/qt/include/phonon/backendinterface.h

```

## 4.2 FITS Files

This section points to docs on CFITSIO (C level interface to FITS files) and the OGIP Spectral FITS file format standard.

A few important keys/terms:

- PHA (Pulse Height Analyzer)
- Redistribution Matrix File, rsp, RESPCFILE
- Ancillary Response File, arf, ANCRFILE
- Background File, \*\_bg\*, e.g. pha2\_bg, pha2\_bg\_-1, BACKFILE
- Correction File, cor?, CORRFILE

### 4.2.1 OGIP Spectral FITS Format

The PDF file is here. At the moment, I don't support links directly to PDF files, so you'll have to paste this into a browser.

[http://heasarc.gsfc.nasa.gov/docs/heasarc/ofwg/docs/spectra/ogip\\_92\\_007.pdf](http://heasarc.gsfc.nasa.gov/docs/heasarc/ofwg/docs/spectra/ogip_92_007.pdf)

#### 4.2.1.1 Notes

A FITS file is a chain of HDUs. An HDU consists of a header and data unit. A header is a list of key/value pairs with comments. A data unit is an n-dimensional array of some basic integer or floating point type.

The first HDU is the primary. Its data may be of any type, or null.

Additional HDUs are called FITS extensions:

- An image extension is an n-dimensional array, usually of pixels (BITPIX=data type, NAXIS=arity, NAXISn=size of each dimension)
- An ASCII table extension is a 2D array of strings
- A BINTABLE is a 2D array of binary values

Tables and bintables: TFIELDS is number of columns (or fields, if you think of it as a flat file database); NAXIS2=# of records; TTYPEn=name of each column; TFORMn=data type of column; TUNITn=physical units.

A file with multiple spectra:

Note that any value in the vector for each spectrum that is common to all may be in the header instead.

SPEC\_NUM

num channel data stat.err sys.err qual grpg rowid exposure areasc bkgdfile ...

## FITS File I/O (cfitsio)

[FITS File I/O \(cfitsio\)](#)

(Remote URLs are not yet included inline in PDFs.)

## 4.2.2 PythonQt Usage Notes

### 4.2.2.1 Publishing Enums

At the time of this writing, `PythonQtWrapper_AT` defines a class that specifies the enumerations to be exposed to the python layer. An enumeration value such as `TRSIDle` appears in python as `at.TRSIDle`. A proposed enhancement to the enum generator utility is the addition of "pnums". Currently, the following declaration creates the `CellDataType` enum at the C++ level, along with classes that convert between the enum and its string name, and those that display popup menus of enumerations. This happens at the C++ level.

```
enum CellDataType
{
    CDTNumeric, "Numeric",
    CDTString,  "String",
    CDTDate,   "Date"
};
```

The extension, implemented in May 2017, looks like this:

```
python enum CellDataType AT
{
    CDTNumeric, "Numeric",
    CDTString,  "String",
    CDTDate,   "Date"
};
```

The optional keyword `python`, has the same effect as `enum`, but it is also published to python with the scope designator shown in this example as `"at"`. Thus the C++ enum `CDTNumeric` is accessed in python as `at.CDTNumeric`. The scope designator defaults to `at`.

### 4.2.2.2 Publishing Optional Arguments

I can't find any docs on publishing optional arguments to python, although python supports the concept.

## XMLRPC

[XMLRPC](#)

(Remote URLs are not yet included inline in PDFs.)

## 4.3 EPSEngine

Qt inexplicably dropped support for generation of PostScript files using `QPainters`. At least until a better solution becomes available, ATLAS currently uses `EpsEngine`, an open source drop-in replacement for the discontinued driver. There are only three source files, so I moved them directly into the ATLAS source tree (in utilities) rather than linking to a library.

There are a number of problems though. First, when a `QWidget` is imaged to the driver (see `ToolEditorPlot::exportPDF()`), the coordinate system translation for the subordinate widgets is lost—all the widgets appear bunched up in the corner of the origin. As nearly as I was able to tell, the driver is not at fault: it receives incorrect positioning information. Lacking a means of correcting this at the driver level, I instead implemented a Rude Hack. Plot-related base classes implement `hackForPostscript()`, which looks up the widget parent tree to find the `PlotMultizone` and asks it for the root widget that is being drawn. It uses this information to translate the origin as needed to generate the right output. Not a pretty sight, but this is easily removed when a better solution presents itself.

Rendering of widgets has other surprises too. Only those things actually drawn in `paintEvent()` appear in the PostScript output. Standard widgets are not imaged. This is mostly not a problem, since the graphs are drawn in `paintEvent()`, but if background drawing is enabled, Qt draws the background in the wrong place (since I cannot intercept the `paintEvent()`).

When a graph is drawn, a mystery rectangle appears, one whose source I cannot trace, and which obliterates part of the graph. Nothing `postProcessEPSToRemoveMysteryRectangle()` can't handle though. (If a rude hack contains a rude hack, is that an even ruder hack?)

Other problems include lost opacity, and substitution of incorrect fonts.

All these issues are easily corrected by adapting the plot classes to generate PostScript, *exactly the right PostScript*, in both EPS and PDF formats. As soon as this is done we can back out EpsEngine.

## EPSEngine Web Site

[EPSEngine Web Site](#)

(Remote URLs are not yet included inline in PDFs.)

## 4.4 Lossless File Compression

These are relatively recent lossless compression algorithms that might be suitable for storing compressed notebooks. They compress much more quickly and sometimes more compactly.

### 4.4.1 ZSTD

<https://github.com/Cyan4973/zstd> (Reached 1.0 2016-06-17). (See also: <http://fastcompression.blogspot.fr>.)

### 4.4.2 LZ4

<https://github.com/Cyan4973/lz4>

### 4.4.3 Apple LZFSE

<https://github.com/lzfse/lzfse> (Open-sourced 2016-06-08.)

## Libcurl

[Libcurl](#)

(Remote URLs are not yet included inline in PDFs.)

# 5 Configuring a Development Environment

ATSAL depends on a number of third party software components, and is organized into several different git archive. Following sections describe how to set up a Mac or Linux system for ATSAAL development, and subsequent sections discuss individual components.

See [ATSAL Class Hierarchy](#) for a diagram of the components that make up ATSAAL.

## 5.1 Setting Up a Mac for Development

**WARNING!** This is very preliminary. It will be easier to create a developer installer as part of an upgrade to newer releases of various software components, since this will allow pruning of non-essential dependencies. So this is just an outline of how it should work.

ATSAL stands on the shoulders of giants, and giants take up a lot of space. You need at least 15 GB of free space to do ATSAAL development.

### 5.1.1 XCode

Install XCode from Apple's App Store. The last few releases of XCode have offered command line tools as a separately installable option, but used a different way each time. You will need these too.

### 5.1.2 Text Editors

In addition to (or in place of) XCode's text editor, you will probably need one or two other editors. Install your favorites.

### 5.1.3 Qt

Qt is a large package. You can obtain pre-built libraries or download source and build it yourself. Most of Qt is very mature and it is rarely necessary to dive into the source code, but it is nice to have the option.

You can put Qt anywhere, but we recommend `~/Qt/Qtm.n`, because this allows you to keep multiple versions of Qt at the same time if needed. Wherever you install it, modify your PATH variable include a reference to its bin directory. This is discussed under *Setting up your startup file*.

### 5.1.4 Obtain ATSAAL

ATSAL's git archive is bulky. You can check out a subset of it if you have limited space. For details, see [TBD](#).

```
git clone TBD
```

It is essential to understand how the multiple git archives interact, so take a look at `atsal/private/docs/git/ATSAL` and `Git.docx`.

### 5.1.5 Setting up your Startup File

Following are suggested additions to your startup file, in Bourne shell derivative format. (The default Mac shell is bash.)

```
# ATSAALDEV is the root directory in the ATSAAL directory tree. This can be changed
# if you have more than one ATSAAL tree checked out at the same time.
ATSALDEV=${HOME}/atsal' ; export ATSAALDEV

# /usr/local/bin needs to be first. Supposedly. The Qt bin directory path is important.
PATH='./usr/local/bin:\
~/Qt/5.3/clang_64/bin:\
${ATSALDEV}/proj/bin:\
~/bin:\
/usr/bin:\
/bin:\
/usr/local/sbin:\
/usr/local:\
/usr/sbin:\
/sbin:\
```

```

'/usr/X11R6/bin:' ; export PATH

LD_LIBRARY_PATH='/usr/lib:/usr/local/lib' ; export LD_LIBRARY_PATH

QMAKESPEC='macx-xcode' ; export QMAKESPEC

# For Python
# ARCHFLAGS='-arch x86_64' ; export ARCHFLAGS

# -----
# For PyXspec
VERSIONER_PYTHON_PREFER_32_BIT=yes ; export VERSIONER_PYTHON_PREFER_32_BIT

# -----
# HEASARC/XSPEC

# Used to build and use XSPEC
HEADAS=${ATSALDEV}/proj/heasoft/i386-apple-darwin13.4.0' ; export HEADAS
alias heainit='source $HEADAS/headas-init.sh'
# The HEASOFT init script doesn't exist yet if a build hasn't been done
if [ -f $HEADAS/headas-init.sh ]
then
    source $HEADAS/headas-init.sh
fi
FC=${ATSALTOOLS}/g95-install/bin/i686-apple-darwin10.3.0-g95' ; export FC

# Logfiles, shared Python source, exported files, etc. go here
ATSAL_SHARED='~/atsal/proj/xdebug' ; export ATSA_SHARED

# Optional shortcuts for common build operations
alias cdxspec='cd "${ATSALDEV}"/xspec/tkent'
alias cdx='cd "${ATSALDEV}"/proj/heasoft/BUILD_DIR && pwd'
# alias cdr='cd "${ATSALDEV}"/xspec/xmlrpc-c && pwd'
alias cdr='cd "${ATSALDEV}"/proj/heasoft/Xspec/src/xmlrpc-c && pwd'
alias cda='cd "${ATSALDEV}"/proj/atsal && pwd'
alias cds='cd "${ATSALDEV}"/proj/atsal/supervisor && pwd'
alias cx='./configure'
alias cdf='cd ~/ATFiles && pwd'
alias openall='cd ~/atsal/proj/xdebug ; open ~/atsal/proj/xdebug/XDebug.xcodeproj/ ; cd
~/atsal/proj/atsal ; open ~/atsal/proj/atsal/XClient.xcodeproj/
~/atsal/proj/xserver.bbprojectd/'
alias opena='cd ~/atsal/proj/atsal ; open ~/atsal/proj/atsal/ATNotebook.xcodeproj/'
alias openn='open Nexus.xcodeproj/'
alias opens='open ATSA.xcodeproj/'
alias tailc='cd /Volumes/tkent/atsal/proj/xdebug ; tail -f xclient.log'
alias tails='cd /Volumes/tkent/atsal/proj/xdebug ; tail -f xserver.log'
alias xspec='/Users/tkent/atsal/proj/heasoft/Xspec/i386-apple-darwin13.4.0/bin/xspec'
alias ts='tail -f ~/atsal/proj/xdebug/xserver.log'
alias tc='tail -f ~/atsal/proj/xdebug/xclient.log'
alias ma='cd ~/atsal/proj/atsal ; sh ma.sh'
alias mn='cd ~/atsal/proj/atsal/nexus ; sh mn.sh'
alias ms='cd ~/atsal/proj/atsal/supervisor ; sh ms.sh'

# This force deletes xspec to make sure it rebuilds after rebuilding the XMLRPC library
alias mr='rm ~/atsal/xspec/heasoft-6.15/XSpec/i386-apple-darwin13.0.0/bin/xspec ; make clean ;
cd ~/atsal/proj/heasoft/Xspec/src/xmlrpc-c ; sh macbuild.sh'

# This rebuilds the XMLRPC libraries, then rebuilds XSPEC
alias mrx='cdr && mr && cdx && mx'
alias mx='rm "${ATSALDEV}"/proj/heasoft/i386-apple-darwin13.4.0/bin/xspec ; make && make
install ; cp "${ATSALDEV}"/proj/heasoft/i386-apple-darwin13.4.0/bin/xspec
"${ATSALDEV}"/proj/heasoft/i386-apple-darwin13.4.0/bin/xspecserver'
alias ix='make install'

# mtouch adds its argument, foo.cpp, to the current directory, and also adds foo.h to the
corresponding include
# directory.
alias mtouch='sh "${HOME}"/atsal/proj/atsal/mtouch.sh'

# Workaround for building XSPEC
LDFLAGS="-headerpad_max_install_names" ; export LDFLAGS

```

## 5.1.6 Run the Dev Installer

This script builds and installs some of the components needed by ATSal. It does not overwrite or interfere with components you have installed on other parts of your system—instead, components are installed locally in ATSal’s build tree.

```
cd ~/atsal
sh macdevinstall.sh
```

## 5.1.7 Mac Dev Installer Design

This is a list of things to do to create the installer.

### 5.1.7.1 The private Archive

There are a number of docs in the private archive that should be moved into the atsal archive so they are accessible.

### 5.1.7.2 Nexus

First, extend Nexus so it can be fired up from a shell command and pointed to a topic. Next, include a built version of Nexus in the developer distribution so it can be used to display docs prior to completing an install.

### 5.1.7.3 README

Modify README files at all levels to a URL that explains the developer install process in detail. This page can direct the user to ATSalDevHelp for further information.

### 5.1.7.4 macdevinstall

- Verify that all prerequisites are present.
- If Qt is not found, offer to install from archive.
- Verify that minimum necessary environment variables have been set.
- Install any other Unix tools needed.
- FITS: modify to install in ATSal dev area, link against it there too.
- Other components should already be locally installed.
- Generate any necessary .xcodeprojs.
- Create and populate ~/ATFiles.

## 5.1.8 Virtual Machines (Deprecated)

I encountered enough problems running virtual machines so that I eventually abandoned this approach. Although times may since have changed, the original rationale was to avoid skew problems among development tools for multiple projects developed on the same physical machine. This issue has been reduced, though not eliminated, by isolating many components inside the ATSal source tree. Nevertheless, I presently use a dedicated physical machine for ATSal development.

### 5.1.8.1 Virtual Environment Manager

If a dedicated physical system is not available for development of ATSal, a virtual machine is recommended. Both Parallels and VMWare Fusion may be used to create a virtual machine.

Note: I run the virtual machine on its own external disk so that it is easily moved between a desktop and laptop computer.

#### 5.1.8.1.1 Parallels Desktop (Mac)

WARNING: Regardless of how you create a Mac OS X virtual machine in Parallels 9.x, the virtual machine’s boot disk is

configured to auto-expand to a maximum size of about 65 GB, which is not sufficient. The following procedure seems rather overcomplicated, but it is the only workaround I could manage, and it has some other advantages.

1. Make sure you have enough disk space. I recommend a minimum of 330 GB, for reasons that will become apparent. (Even the initial configuration, with the fresh Git archive, is over 128 GB!)
2. Download the Mavericks updater, Install OS X Mavericks.app, from the app store. (You can also install from the emergency partition in newer Macs, but this option may install an older OS that you have to upgrade afterwards.)
3. Shut down the new virtual machine.
4. Set the virtual machine's parameters to use roughly half of the available processors on your system, and half the memory. You can change these at any time, provided that you shut down the VM first.
5. Create a second virtual disk using Parallels. Make it auto-expanding (default) and give it a size of 250 GB. This sets the upper bound on size, not the initial size.
6. Start the VM. Both the old and new disks will be called "Macintosh HD," not very useful. Rename the original (smaller) one "Recovery" and the larger one "ATSAL." The names don't matter, but "Recovery" will be needed later if you need to repair the filesystem or build a new, larger virtual disk.
7. Mount the host disk on the virtual machine and copy the installer, Install OS X Mavericks.app, to the local filesystem. (You can't run it over the network.)
8. Run the installer, and install Mavericks on the "ATSAL" disk.
9. Boot the ATSAAL disk.
10. Install the Parallels tools on the new system. (Parallels > Virtual Machine > Install Parallels Tools). A Parallels Tools disk appears on the desktop. Run the installer.
11. Restart the virtual machine afterwards.
12. Upgrade the OS to the current version via the app store.
13. Set the monitor size as preferred. Sometimes you can do this simply by resizing the window, which initially stretches the image, then replaces it with a correct image for the new size. Sometimes this doesn't work and you have to run the Displays option and pick a new screen resolution.
14. I have found that it is easiest to keep most personal stuff off the virtual machine, including e-mail—otherwise it is easy to lose track of what you are doing on different systems. Hence I hide icons for any applications that I don't want to use on the VM. (For the same reason, all ATSAAL-related project materials, except administrative ones, are removed from the host machine.)

### 5.1.8.1.2 VMWare Fusion (Mac)

If you have configured a Parallels virtual machine, or are using dedicated hardware, skip this section. I tried Fusion when Parallels refused to expand the OS X virtual machine beyond its default maximum size of about 65 GB. VMWare Fusion shows the same size limiting behavior with automatic disk expansion, but manual expansion to 128 GB was successful.

Execution speed within the virtual machine dropped dramatically (~10x or more) after awhile, but the effect was intermittent. An "internet herbal remedy" of turning off App Nap support for the Fusion app may have helped. Later, I added the following lines to the end of .vmx in the virtual machine folder, also in an attempt to improve performance.

```
MemTrimRate = "0"  
sched.mem.pshare.enable = "FALSE"  
prefvmx.useRecommendedLockedMemSize = "TRUE"
```

Later, I concluded that the speed problems were probably due to a developing hardware failure in the disk drive.

## 5.2 Setting Up for Linux Development

TBD.

## 5.3 Software Components

Open source components are developed independently. Each release of a given component is typically compatible with only one or two releases of other components. The multiple components that make up a large project such as ATSAAL can be thought of as axes in a multidimensional array. Each cell represents a combination of a release from each component. The number of workable combinations is sparse, becoming rapidly more so with increasing numbers of components.

Early experiments in creating a workable environment confirmed this. New releases of ATXSAL will be possible only when another workable combination of all the components is identified. Dependencies are reduced in magnitude by partitioning tasks into separate processes where possible, one reason why XSPEC remains an independent process. Nevertheless, we will probably need to perform some porting ourselves in order to alleviate this problem.

The following tables list software dependencies grouped by development environment. Later sections on XSPEC Server and ATXSAL list software component dependencies.

### 5.3.1 Software Component Summary

<i>Component</i>	<i>Version</i>	<i>Description</i>
Mac OS X	10.9.x	Apple store
Linux	?	TBD
XCode	5.0.2	Apple store
XCode tools	5.0.2	Got command line tools by logging into <a href="#">Apple developer downloads</a> .
XQuartz	2.7.5	Needed for XSPEC, probably not for server though, from <a href="http://xquartz.macosforge.org">xquartz.macosforge.org</a> .
BBEdit	10.5.8	Text editor and other tools, from <a href="http://www.barebones.com">www.barebones.com</a> .
BBEdit tools	10.5.8	Download from <a href="http://www.barebones.com">www.barebones.com</a> .
python3	3.3.3	Mac release bundled with python 2.x, need python 3 from <a href="http://www.python.org">www.python.org</a> . Install under separate name so as not to conflict with existing python
g95	?	Fortran compiler needed by XSPEC
gdb	7.6.2	Gnu debugger, from <a href="http://www.gnu.org/software/gdb/">http://www.gnu.org/software/gdb/</a>
cmake	2.8.12	Tool for building software projects, from <a href="http://www.cmake.org">www.cmake.org</a> .
homebrew	0.9.5	Tool for downloading software packages, downloaded via: <code>ruby -e "\$(curl -fsSL https://raw.githubusercontent.com/mxcl/homebrew/go/install)"</code>
wget*	1.14	Tool for downloading software packages, downloaded via: <code>brew install wget</code>
setuptools	?	Python package installer from <a href="https://pypi.python.org/pypi/setuptools">https://pypi.python.org/pypi/setuptools</a> , get via: <code>wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py -O -   sudo python</code>
ez_setup.py	?	Part of installation for virtualenv
pip	1.4.1	Python package installer, obtained by downloading and running <code>get-pip.py</code>
virtualenv	?	Restricts packages added to python to their own isolated environment. Obtain via: <code>sudo pip install virtualenv</code>
sip	4.15.4	Tool to create Python bindings to C++ code, from <a href="http://www.riverbankcomputing.com/software/sip/download">http://www.riverbankcomputing.com/software/sip/download</a>
PyQt	5.2	Bindings that make the Qt library available to Python programs. <a href="http://www.riverbankcomputing.com/software/pyqt/download5">http://www.riverbankcomputing.com/software/pyqt/download5</a>
socketexec	?	Tool for testing client/server connections, from: <a href="http://giraffe-data.com/software/about_socketexec.html">http://giraffe-data.com/software/about_socketexec.html</a> . Note: required a "giraffe library," I extracted the necessary components from this library and incorporated them into socketexec to remove this dependency

\*Note that wget, a tool for downloading software, must first be downloaded by homebrew, a tool for downloading software; homebrew in turn depends upon ruby, which, mercifully, is bundled with the operating system.

Note: Many of these tools install the same way under Mac OS X and Linux, but the Linux install procedure has not yet been debugged.

## 5.4 Development Environment

ATXSAL developers may choose between an integrated development environment or the traditional command line configure/make/make install approach. Qt's development approach is agnostic about this choice. You can use Qt

Creator, their own development environment, or Apple's XCode, or rely on command line tools and your favorite editor.

### 5.4.1 XCode (Mac)

Even if you won't use XCode for development, it is still necessary to download XCode from the app store in order to take advantage of associated tools. Next, download the command line tools package. These tools are available via a separate download from Apple. <https://developer.apple.com/downloads/index.action?name=for%20Xcode%20->

Or see `commandline_tools_os_x_mavericks_for_xcode__march_2014`, in the `atsal/dev/installers` directory.

### 5.4.2 TBD (Linux)

Qt Creator?

## 5.5 X Window System

This is to run XSPEC as a standalone user interface, but will not be needed (hopefully at least!) by XSpecServer. On the Mac, install XQuartz, in the installers folder. On Linux, this is bundled (?).

## 5.6 Editors

### 5.6.1 Macintosh Editors and Drawing Tools

#### 5.6.1.1 Microsoft Office

If you are editing docs, you may wish to install Microsoft Office. We are trying to reduce dependencies on this component by creating developer docs using Nexus, and will ensure that documents prepared with Office are also available in PDF format.

#### 5.6.1.2 OmniGraffle Pro

This diagramming tool is used for some of the docs. It is not needed if you are not creating diagrams. Presently diagrams are exported as PNGs, but when SVG support is most consistent among browsers, we will switch over to that. So diagram artwork is best done with a tool that can export both PNG and SVG.

#### 5.6.1.3 BBEdit

This is a popular, reasonably priced Macintosh text editor. If you don't have a preferred editor, I suggest it. Many other alternatives are available though, including emacs, vim, and several commercial products. BBEdit includes some command line tools which must be downloaded separately, due to Apple sandboxing restrictions. The tools are not necessary, but if you use BBEdit, download the tools too. Choose Install Command Line Tools... from the File menu. I especially like BBEdit's project windows, which are well-suited for browsing the files making up large projects such as XSpecServer.

### 5.6.2 Linux Tools

TBD...

## 5.7 Homebrew

Info is here:

<http://brew.sh>

Or download this package downloader as follows:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/mxcl/homebrew/go/install)"
```

If that doesn't work, use this instead:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/homebrew/go/install)"
```

```
brew update
```

verifies that brew is up to date.

```
brew doctor
```

looks for any problems.

If brew doctor reports a failure, it may be due to the fact that python3 is not yet installed. I suggest that you soldier on to the python3 install.

## 5.8 gdb

Procedure to install gdb on Mavericks, no longer installed by default, is in this article:

[wiki.freepascal.org/GDB\\_on\\_OS\\_X\\_Mavericks\\_and\\_Xcode\\_5](http://wiki.freepascal.org/GDB_on_OS_X_Mavericks_and_Xcode_5)

Install gdb locally:

```
brew install https://raw.githubusercontent.com/Homebrew/homebrew-dupes/master/gdb.rb
```

Article says gdb requires code signing, but it did not.

As far as I am presently aware, the clang debugger bundled with XCode may be used if you prefer.

## 5.9 Qt

Qt can be built in several ways—self-contained, or with build and install products separated, or publicly installed. The process described here separates the source, build products, and install products, to make it easier to discard build products and to build more than one configuration variant if needed. Products are arranged as follows:

- `~/tools/qt-everywhere-opensource-src-5.2.1` contains the raw source tree. At the time of this writing, the source tree is 909 MB.
- `~/tools/qt/5.2.1` contains build products for this version. These don't need to be retained after a successful build. Nearly 13 GB.
- `~/tools/Qt5.2.1` contains the installed products. `~/Qt5.2.1/bin` is added to the PATH variable. 489 MB.

To build the source tree, first make sure QMAKESPEC is not defined (normally it is set in the `.bash_profile` or equivalent file). Unzip the installer package into the `~/tools` directory, where support tools are placed:

```
cd ~/tools
tar -xvf ~/atsal/dev/installers/multiplatform/Qt/qt-everywhere-opensource-src-5.2.1.tar.gz
```

Create a separate build directory. This makes it easier to build with more than one set of configuration settings if necessary.

```
cd ~/tools # If not already in tools
mkdir qt # If it doesn't already exist
cd qt
mkdir 5.2.1 # If it doesn't exist
cd 5.2.1

# Configure the build. The prefix argument tells it where to
# install.
~/tools/qt-everywhere-opensource-src-5.2.1/configure -prefix $HOME/tools/Qt5.2.1 -debug-and-
release -no-pkg-config -opensource -confirm-license -optimized-qmake -nomake examples

# This configure failed at install time. I am recording it here,
# commented out, for posterity.
# ./configure -prefix $PWD/qtbase -platform macx-clang -opensource -developer-build -nomake
tests -confirm-license
```

Do the make. The “3” below is the number of CPUs to dedicate to the build. It should match the number you assigned to your virtual machine.

```
make -j3 >& make.log &

# And do the install
sudo make install >& install.log
```

Add `~/tools/Qt5.2.1/bin` to the `PATH` variable in your `.bash_profile` or equivalent. If working with XCode, restore the definition of `QMAKESPEC` to `macx-xcode`, also in the `.bash_profile`. This instructs qmake to generate a `.xcodeproj` file for XCode by default.

## 5.10 Python 3

Python 2 is bundled with Mac OS X, and is used by XSPEC is an extension language. ATSAAL uses Python 3, which is not fully backward-compatible. Hence Python 3 must be installed and must not replace Python 2. It is installed under the command name `python3`.

```
brew install python3 --with-brewed-openssl
```

`~/atsal/dev/installers/multiplatform/Python` contains the following:

- `Python-3.3.3.tgz` for Linux
- `python-3.3.3-macosx10.6.dmg` for Mac OS X
- An article on installation: [Python Development Environment on Mac OS X Mavericks 10.9 | Hacker Codex.pdf](#)

I used the brew install method above.

Python 3 is installed in `/usr/local/bin` by default, and invoked with `python3`. Python 2 remains available as `python`. Python 3 is “sandboxed” into a virtual environment using another package, `virtualenv`, later on. Don’t install any optional packages until the sandboxing is set up. (If the brew command doesn’t work, go back a bit to the section on `gdb` and install brew as shown there.)

### 5.10.1 Python on ATSAAL Development Machine

Python 2.7.5 is shipped with Mavericks (10.9), and invoked with `python`. Python 3.3.5 was downloaded via homebrew, and invoked with `python3`. 3.3 was downloaded pre-built from the Python web site. It is currently installed with a wrapper app and not directly invoked from the command line. The only reason 3.3 was installed was to try to debug a problem with PythonQt. Python 3.3.5 is the present version preferred for ATSAAL.

As of 2014-Oct-27:

<i>File</i>	<i>2.7.5</i>	<i>3.3.5</i>	<i>3.3</i>
Executable	<code>/usr/bin/python</code>	<code>/usr/local/bin/python3</code>	<code>/Applications/Python3.3</code>
Framework	<code>/System/Library/Frameworks</code>	<code>/usr/local/Cellar/python3/3.3.5/Frameworks</code>	<code>/Library/Frameworks</code>

On 2014-Oct-28, I removed the 3.3 installation and built Python from source, enabling framework generation. This was successful.

PythonBE (“bleeding edge”) is a test download of PythonQt from source current as of 2014-Oct-26.

### 5.10.2 Python and PythonQt Build Notes

For reasons I don’t understand, references to the Python and PythonQt dynamic libraries are not represented correctly in the generated libraries and executables. At the moment I am correcting them manually with a post processing shell script. For this to work, libraries and executables must be built with the `-headerpad_max_install_names` switch, which leaves extra space in the library pathnames so they can be overwritten using an external tool. Library references are listed with e.g.:

```
otool -L libPythonQt.dylib
```

Pathname references are changed with e.g.:

```
install_name_tool -change ./framework/Python.framework/Versions/3.3/Python  
/Users/tkent/atsal/dev/atsal/Python-3.3.3/framework/Python.framework/Versions/3.3/Python  
libPythonQt.dylib
```

The eventual goal is to fully embed the python and pythonqt libraries within the application bundle, to avoid skew problems with externally installed pythons, so these scripts will need to be modified accordingly.

Here is an article:

<http://www.cocoabuilder.com/archive/cocoa/210698-install-name-tool.html>

As a temporary workaround to these special build issues, I have introduced macbuild.sh scripts to Python-3.3.3 and pythonqt.

## 5.11 CMake

I forget which tools require cmake in order to build.

<http://www.cmake.org>

## 5.12 wget

Another package installer.

```
brew install wget
```

## 5.13 setuptools

A Python package installer:

```
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py -O - | sudo python
```

Download ez\_setup.py:

```
sudo python ez_setup.py
```

## 5.14 pip

Another Python package installer. Securely download get-pip.py.

<https://pypi.python.org/pypi/pip/1.2>

```
sudo python get-pip.py
```

## 5.15 virtualenv

This isn't in use at the moment, but this or something like it will be needed to fully isolate ATSA's python from the rest of the system.

This package sandboxes the installation of Python packages so they cannot interfere with other uses of Python on the same computer.

```
sudo pip install virtualenv
```

Create a directory for virtual Python environments in `~/atsal/tools`:

```
mkdir virtualenvs
cd virtualenvs
```

Create a virtual environment for Python 3:

```
virtualenv -p python3 py3
```

(Or use `virtualenv py` to create one for Python 2.)

To ensure that future Python installs apply to the Python 3 virtual environment, add the following to `.bash_profile` as follows:

```
# pip should only run if there is a virtualenv currently activated, in
# order to restrict the install to the virtual environment.
PIP_REQUIRE_VIRTUALENV=true ; export PIP_REQUIRE_VIRTUALENV
# cache pip-installed packages to avoid re-downloading
PIP_DOWNLOAD_CACHE=$HOME/.pip/cache ; export PIP_DOWNLOAD_CACHE

# Select the correct virtual environment for Python
cd ~/atsal/tools/virtualenvs/py3 ; . bin/activate ; cd ~
```

This activates the Python virtual environment and modifies the prompt string to indicate the selected environment. For example:

```
(py3)ATSAL:foo$
```

Indicates that the `py3` virtual environment is running on host `ATSAL`, in the current working directory `foo`.

**Install and configure this package before installing any Python packages. Otherwise the packages will be installed in the wrong version of Python. Do not proceed until you see the modified prompt that indicates the correct virtual environment is selected.**

The following subsections provide some background on virtual environments.

### 5.15.1.1 Restricting pip to virtual environments

What happens if we *think* we are working in an active virtual environment, but there actually is no virtual environment active, and we install something via `pip install foo`? Well, in that case the `foo` package gets installed into our global site-packages, defeating the purpose of our virtual environment isolation.

In an effort to avoid mistakenly pip-installing a project-specific package into my global site-packages, I previously used `easy_install` for global packages and the `virtualenv`-bundled `pip` for installing packages into virtual environments. That accomplished the isolation objective, since `pip` was only available from within virtual environments, making it impossible for me to `pip install foo` into my global site packages by mistake. But `easy_install` has some deficiencies, such as the inability to uninstall a package, and I found myself wanting to use `pip` for both global and `virtualenv` packages.

Thankfully, `pip` has a sparsely-documented setting that tells it to bail if there is no active virtual environment, which is exactly what I want. In fact, we've already set that above, via the `setenv PIP_REQUIRE_VIRTUALENV true` directive. For example, let's see what happens when we try to install a package in the absence of an activated virtual environment:

```
$ pip install markdown
Could not find an activated virtualenv (required).
```

Perfect! But once that option is set, how do we install or upgrade a global package? We can temporarily turn off this restriction by adding the following to your `~/.bashrc`:

```
syspip(){  
  PIP_REQUIRE_VIRTUALENV="" pip "$@"  
}
```

If in the future we want to upgrade our global packages, the above function enables us to do so via:

```
syspip install --upgrade pip setuptools virtualenv
```

You could, of course, do the same via

```
PIP_REQUIRE_VIRTUALENV=""  
pip install --upgrade foobar
```

but that's much more cumbersome to type.

### 5.15.1.2 Creating virtual environments

Let's create a virtual environment for Pelican, a Python-based static site generator:

```
cd ~/Virtualenvs  
virtualenv pelican
```

Change to the new environment and activate it via:

```
cd pelican  
. bin/activate
```

**Critical:** if you don't activate the virtual environment prior to installing new packages, they will be installed in the global environment.

My `.bash_profile` contains the above sequence. Once activated, the command line prompt is prepended with the name of the virtual environment. Instead of, say,

```
<host> <directory>$
```

you get

```
(py3) $
```

To install Pelican into the virtual environment, we'll use `pip`: [Don't actually install Pelican, it isn't needed.]

```
pip install pelican markdown
```

For more information about virtual environments, read the `virtualenv` docs.

## 5.16 socketexec

This is not presently in use.

`socketexec` is a utility that creates a socket, then forks and execs a server or client, passing the opened socket via `stdin`.

This utility depends upon a library, which I also downloaded. There were several incompatibilities with the Darwin build environment. I moved only the needed files from the library into the directory with the utility, and ported the files to get a clean build. The result will probably port to Linux, but has not been tested.

Some socket-related functions are differently implemented in Darwin. The mapping was not immediately obvious, so I disabled these options.

Eventually the function performed by `socketexec` will be performed by ATXSAL on the client side, and by the reverse proxy on the server side.

## ATOMDB: Atomic Data for Astrophysicists

[ATOMDB: Atomic Data for Astrophysicists](#)

(Remote URLs are not yet included inline in PDFs.)

## FITS File I/O (cfitsio)

[FITS File I/O \(cfitsio\)](#)

(Remote URLs are not yet included inline in PDFs.)

## FITS Utilities

[FITS Utilities](#)

(Remote URLs are not yet included inline in PDFs.)

### 5.16.1 FITS File Notes

#### 5.16.1.1 PHA Files

PHA files contain an HDU with an array of values, one per channel. In the example below, which contains just the first few channels, the row number is in the leftmost column. Note that the channel numbers are not entirely contiguous: some are skipped. In this example there are 125 PHA bins, but 128 channels, since three are ignored.

	CHANNE	RATE	STAT_ERR	QUALIT	GROUPI
1	4	0.00174014	0.00344162	5	0
2	5	0.0343295	0.0145093	5	0
3	6	0.0989650	0.0212513	0	1
4	7	0.131009	0.0201607	0	1
5	8	0.217443	0.0179256	0	1
6	9	0.279315	0.0165190	0	1
7	10	0.305150	0.0156953	0	1
8	11	0.258564	0.0149292	0	1
9	12	0.229426	0.0144044	0	1
10	13	0.250379	0.0142251	0	1

The QUALITY field is apparently non-zero for any channel whose data is considered “bad.” I don’t presently know whether higher values indicate more badness. If the user requests that bad channels be ignored, by default, ATXSAL produces an ignore selection set that consists of all channels with nonzero values.

#### 5.16.1.2 RMF Files

RMF files, sometimes with the extension “.rsp,” contain the response matrix for an observation (or group thereof). Some of the header information for this example (`s54405.rsp`) is shown below. HDU #3 contains the energy range associated with each valid channel, in keV (at least in this example).

```
liststruc s54405.rsp

HDU #1 Array: NAXIS = 0, BITPIX = -32

HDU #2 Binary Table: 6 columns x 128 rows
COL NAME          FORMAT
 1 ENERG_LO        E
 2 ENERG_HI        E
 3 N_GRP           I
 4 F_CHAN          3I
 5 N_CHAN          3I
 6 MATRIX          PE(125)

HDU #3 Binary Table: 3 columns x 128 rows
COL NAME          FORMAT
 1 CHANNEL         J
 2 E_MIN           E
 3 E_MAX           E
```

## 5.17 The .bash\_profile (or equivalent)

Some things you need or might want in the `.bash_profile` (translated as needed for your favorite shell).

```
umask 002
TZ=UTC ; export TZ

# Prompt is just <host>:$
PS1='\h:\W\$ ' ; export PS1

# Disables spelling corruption mode
set correct=none

# /usr/local/bin needs to be first. Supposedly.
PATH='./usr/local/bin:\
'/Users/tkent/Qt5.1.1/5.1.1/clang_64/bin:\
'/Users/tkent/bin:\
'/usr/bin:\
'/bin:\
'/usr/local/sbin:\
'/usr/local:\
'/usr/sbin:\
'/sbin:\
'/usr/X11R6/bin:\
'$FSLDIR/bin:\
'/opt/local/bin:\
'/opt/local/sbin:' ; export PATH

LD_LIBRARY_PATH='/usr/lib:/usr/local/lib' ; export LD_LIBRARY_PATH

# -----
# For Python
ARCHFLAGS='-arch x86_64' ; export ARCHFLAGS

# pip should only run if there is a virtualenv currently activated, in
# order to restrict the install to the virtual environment.
PIP_REQUIRE_VIRTUALENV=true ; export PIP_REQUIRE_VIRTUALENV
# cache pip-installed packages to avoid re-downloading
PIP_DOWNLOAD_CACHE=$HOME/.pip/cache ; export PIP_DOWNLOAD_CACHE

# Select the correct virtual environment for Python
cd atsal/atsal/virtualenvs/py3 ; . bin/activate ; cd ~

VERSIONER_PYTHON_PREFER_32_BIT=yes ; export VERSIONER_PYTHON_PREFER_32_BIT # For PyXspec

# -----
# HEASARC/XSPEC

# Used to build and use XSPEC
HEADAS='/Users/tkent/atsal/xspec/heasoft-6.15/i386-apple-darwin13.0.0' ; export HEADAS
alias heainit='source $HEADAS/headas-init.csh'
FC='/Users/tkent/atsal/xspec/g95-install/bin/i686-apple-darwin10.3.0-g95' ; export FC

# Optional shortcuts for common build operations
```

```

alias cdxspec='cd ~/atsal/xspec/tkent'
alias cdx='cd ~/atsal/xspec/heasoft-6.15/BUILD_DIR'
alias cx='./configure >& config.log & ; tail -f config.log'
alias mx='make >& build.log & ; tail -f build.log'
alias ix='make install >& install.log & ; tail -f install.log'
alias bc='bbedit config.log'
alias bm='bbedit build.log'
alias bi='bbedit install.log'

# Invokes server or client on local machine for testing
alias server='socketexec -listen -local_port=8080 /Users/tkent/xspec/heasoft-6.15/i386-apple-
darwin13.0.0/bin/xspecserver'
alias client='socketexec -connect -remote_host=localhost -remote_port=8080
/Users/tkent/xspec/xmlrpc-c/examples/cpp/pstream_client'

# For xmlrpc. Need to build 32-bit for compatibility with XSPEC, this was the only
# way I could find to do it. Note that this breaks the Qt build though.
#CFLAGS="-m32" ; export CFLAGS
#CXXFLAGS="-m32" ; export CXXFLAGS
#LDFLAGS="-m32" ; export LDFLAGS

# -----
# Qt

QTDIR='/Users/tkent/Qt5.1.1' ; export QTDIR
QMAKESPEC='/Users/tkent/Qt5.1.1/5.1.1/clang_64/mkspecs/macx-xcode' ; export QMAKESPEC

# Convenience

alias qft="qmake -spec macx-xcode -config debug *.pro"
alias bb=bbedit

```

## 5.18 XSPEC Server

The server presently consists of HEASARC's 6.15 release, coupled with XMLRPC. Changes to XSPEC are as minimal as possible, to facilitate building the server and the freestanding versions of XSPEC from the same source in the future.

### 5.18.1 XSPEC Extensions

heasoft/Xspec/src/main/xspec.cxx: add --server switch to invocation

heasoft/Xspec/src/XSUser/Global: Global.cxx, .h: add server accessors

heasoft/Xspec/src/XSUser/Server: add directory and RPC files

heasoft/Xspec/src/XSUser/Makefile: add many xmlrpc libs

### 5.18.2 XMLRPC

<http://xmlrpc-c.sourceforge.net/doc/xmlrpc-c-config.html>

[http://xmlrpc-c.sourceforge.net/doc/libxmlrpc\\_server\\_pstream++.html](http://xmlrpc-c.sourceforge.net/doc/libxmlrpc_server_pstream++.html)

I used the "stable" SVN build. XMLRPC-C needs to be built for a 32-bit environment for compatibility with the present version of XSPEC. On a 64-bit Mac, it defaults to 64-bit, and the makefiles are not designed to handle this case. Many attempts failed, including this one.

```

./configure --target=i386-apple-darwin --build=i386-apple-darwin "CFLAGS=-m32" "CXXFLAGS=-m32"
"LDFLAGS=-m32" --disable-wininet-client --disable-libwww-client

```

Eventually I discovered that defining CXXFLAGS and LDFLAGS to be -m32 produced the correct build. I made the following change as a workaround to a fatal compiler error. It may not be portable.

```

static __inline__ void
init_va_listx(va_listx * const argsxP,
              va_list   const args) {

```

```

#if 0
#if VA_LIST_IS_ARRAY
/* 'args' is NOT a va_list. It is a pointer to the first element of a
   'va_list', which is the same address as a pointer to the va_list
   itself. (That's what happens when you pass an array in C).
*/
    memcpy(&argsxP->v, args, sizeof(argsxP->v));
#else
    argsxP->v = args;
#endif
#else
// THK, 2013-11-27 See if this fixes the following compiler error:
// srcdir/lib/util/include/stdargx.h:63:15: error: array type 'va_list'
// (aka '__builtin_va_list') is not assignable
// argsxP->v = args;
va_copy(argsxP->v, args);
#endif
}

```

In examples/cpp, to get a 32-bit build, first

```
setenv CFLAGS_PERSONAL -m32
```

### 5.18.2.1 Mac Build

I added a shell script, `macbuild.sh`, to do a Mac build. It creates a set of universal (32- and 64-bit) libraries in `/usr/local/lib`. It does a forced rebuild of everything. Won't work with non-Mac systems.

XSpec presently requires a 32-bit library, linked against the Darwin standard C++ libraries. ATSAAL requires a 64-bit library, linked against a different set of standard C++ libraries located in the Mavericks SDK. The two libraries contain different implementations of `std::string` and perhaps other classes. The `macbuild.sh` script builds the correct library variants and creates universal (32- and 64-bit) libraries, which are installed in `/usr/local/lib`.

### 5.18.2.2 Debugging the Server with XCode

I used this article as a guide to using XCode to debug XServer:

[https://developer.mozilla.org/en-US/docs/Debugging\\_on\\_Mac\\_OS\\_X](https://developer.mozilla.org/en-US/docs/Debugging_on_Mac_OS_X)

### 5.18.2.3 Choosing XMLRPC Libraries

The XMLRPC libraries are factored into a lot of little pieces. Choosing the right subset for a particular use is nontrivial. They offer a utility to help, `xmlrpc-c-config`.

For a client:

```
xmlrpc-c-config c++2 client --cflags
```

Prints the include directories needed to compile.

```
xmlrpc-c-config c++2 client --libs
```

Prints the libraries needed to link. For a pstream server, the type used in ATSAAL:

```
xmlrpc-c-config c++2 pstream-server --cflags
```

Prints the include directories needed to compile.

```
xmlrpc-c-config c++2 pstream-server --libs
```

Prints the libraries needed to link.

### 5.18.2.4 Bidirectional IPC

ATSAL and XSPEC don't fit a pure client/server model very well, because it is necessary to return progress information during XSPEC command execution, not just after command completion. A single command cannot be atomic unless there is some independent means for conveying progress information.

One approach is to make both software components function as client and server simultaneously. There are many other symmetrical interprocess communication models that could be adapted for this purpose. In this case, we have chosen a model that is client/server, with polling to retrieve progress information. This allows us to use XMLRPC (which is unidirectional as presently designed) as a communications substrate.

We define three basic command types:

- **RPC.** In this implementation, a remote procedure call passes parameters from client to server, blocking the client thread until the server executes the call and returns results. Only one RPC can be outstanding at a time in this implementation. An example RPC might set a single parameter for a model, or retrieve all the current parameters for a model. A key feature of these RPC calls is relatively short execution time, never more than a second or so.
- **XSPEC command.** This is a special RPC that corresponds almost exactly to a single interactively issued XSPEC command. Since the command may take a long time to execute, this call always returns immediately. XSPEC's main thread blocks while the command executes. But from ATSAAL's perspective, the command execution is asynchronous, and no threads are blocked.
- **Status request.** This is another special RPC. It prepares an array of any lines of text written to stdout, stderr, and the logfile channel. It also includes an entry that indicates completion of an XSPEC command, if one completed. So each status request returns all the output from XSPEC that has accumulated since the last request. Like most RPCs, status requests are atomic, and execute quickly.

Both client and server have a communications thread which manages command interchange. (ATSAL can interact with multiple XSPEC instantiations, so it has one communications thread per instantiation.) ATSAAL's main thread operates the user interface, which does not block unless the users issues the Refresh command to bring it up to date. XSPEC's main thread blocks each time it executes a command.

On the client side, XSPEC commands are issued into a queue maintained by the communication thread. The thread's main loop waits for a delay period, the length of which will be discussed shortly. Then, if an XSPEC command is presently executing asynchronously, or no command is being executed but it is time for a new status update, it sends a status request. Otherwise, it issues a new XSPEC command if the command queue is not empty. (It is always true that at least one status request will occur between any two XSPEC commands, since this is how command completion is signaled.)

When status is received, it is demultiplexed into stdout, stderr, logfile output, and command completion. Data streams for each channel are displayed in window panes if requested by the user, and stored in files. Data directed to stdout is also attached to the current command, since it represents the results for the current command. When command completion is received, the current command is placed on a completed commands queue, where it becomes available to ATSAAL for further processing.

#### 5.18.2.4.1 File Transfers

In some cases file transfers are necessary. Two file transfer options are supported, and both use an independent mechanism to exchange the data:

- **Asynch file transfer** initiates a transfer whose progress may be monitored, but which happens completely independently of the command processing chain described above. This is useful, for example, when it is necessary to move a group of relatively large files to the server machine for processing. While the transfers are in progress, users can tend to other work within ATSAAL.
- **Synchronous file transfers** are synchronous insofar as they prevent other XSPEC commands from being sent until the transfer completes. ATSAAL's user interface remains available for use, but the Refresh command will block while a synchronous file transfer is in progress. Synchronous transfers are performed when the transfer must complete before XSPEC can do any additional work.

Note that the file transfer mechanism is used for spectra and RMF files. Data points for graphs, and other relatively small arrays, are transferred via the RPC mechanism.

### 5.18.2.4.2 Timing

Since we use a client/server model instead of symmetrical IPC, polling is necessary to retrieve status information from the server. To improve throughput without imposing excessive load, we use a simple predictive algorithm. The time between successive pollings,  $t$ , is bounded by  $t_{min}$ , the briefest interval, and  $t_{max}$ , the longest.  $t_{min}$  is a hedge against overloading. We assume a starting value here of 2 ms.  $t_{max}$  avoids choppy response; we'll choose 1 s for this discussion. Initially  $t = t_{min}$ . If a status reply indicates some sort of state change (new data received, command completion),  $t$  is again set to  $t_{min}$ . But if not,  $t$  is doubled, up to a maximum of  $t_{max}$ . The longer XSPEC is quiescent, the longer we wait between new status inquiries. Thus if XSPEC is busy on a long-running calculation,  $t = (2, 4, 8, 16 \dots 1,000)$  ms. Note that these times are in addition to network-imposed delays, or delays in processing synchronous RPC commands. Hence degraded response due to external factors will not result in a frenzy of increased polling. (However, a flood of logfile output will cause a lot of polling.)

### 5.18.3 Debugging XSPEC Server with XCode

This is adapted from:

[https://developer.mozilla.org/en-US/docs/Debugging\\_on\\_Mac\\_OS\\_X](https://developer.mozilla.org/en-US/docs/Debugging_on_Mac_OS_X)

It applies to XCode 5.

- Open XCode, and create a new Project with File > New Project. Under the Mac OS X list (not the iOS list, if present), select the "Other" template group and choose "Empty Project" as the project type, click Next, name the project and choose where to save the project, then click Create. Be sure to turn off "Create Git archive."
- In the Product menu, select "New Scheme" and name your scheme. After you click OK, XCode should open the settings window for the new scheme. If not, then open its settings from the Product > Edit Scheme menu.
- Select "Run" on the left-hand side of the settings window, then select the "Info" tab. Set the Executable by clicking on "None" and selecting "Other...". A new dialog titled "Choose an executable to launch" will pop up. If you are using the default layout, the executable is in something like `~/atsal/dev/heasoft/i386-apple-darwin13.1.0/bin/xspec`.
- Select the Arguments tab. Add `--server` as an argument to bring up XSPEC in server mode. (You can temporarily uncheck this box later if you want to run it in normal interactive mode.)
- Also in the "Arguments" panel, add an environment variable for the dynamic library load path. It should look something like this. It should look something like this:

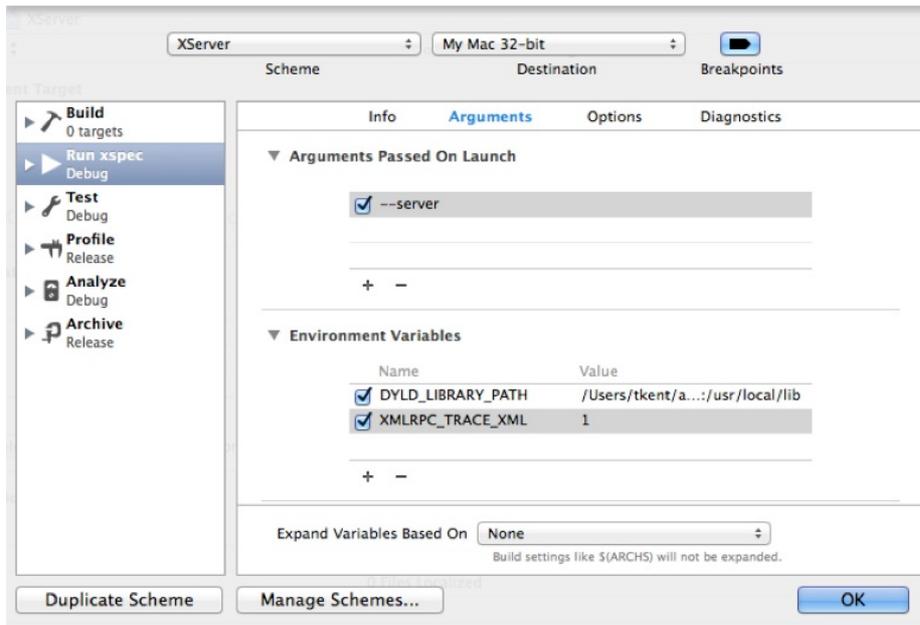
```
DYLD_LIBRARY_PATH ~/Users/tkent/atsal/dev/heasoft/i386-apple-darwin13.1.0/lib:/usr/local/lib
```

("~" didn't seem to work here as I recall, you need the full path.)

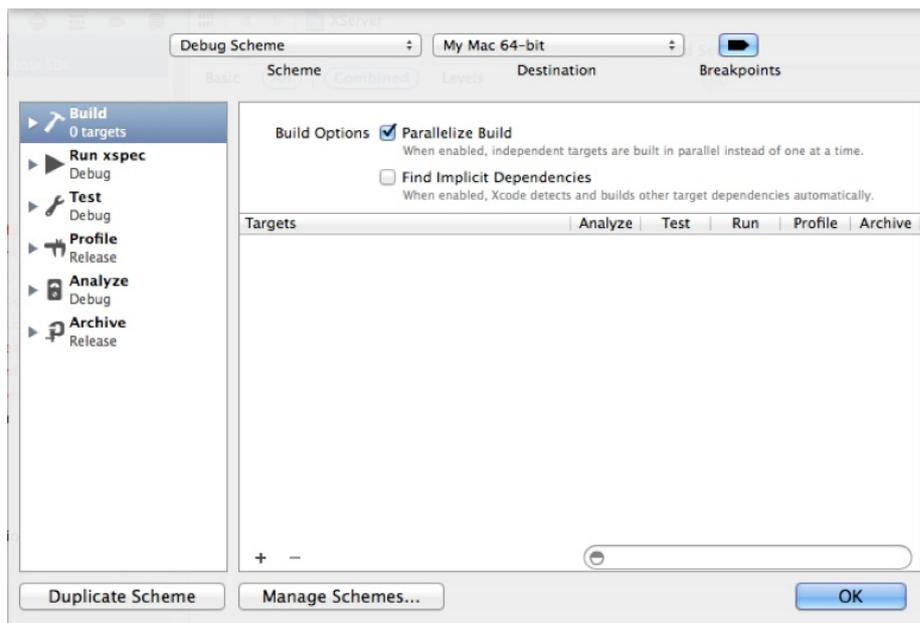
- If you want to see the XML packets exchanged between client and server, create a second environment variable:

```
XMLRPC_TRACE_XML 1
```

Just uncheck it to shut it off. It should look something like this:



- Select "Build" from the left of the scheme editor window, and check that there is nothing listed under Targets (otherwise it may cause problems when you try to run the executable for debugging since you will get build errors). Also shut off "Find implicit dependencies."



- Click OK to close the scheme editor.
- At this point you can run the application, and it should show you source files when you pause or hit breakpoints, but it doesn't automatically pick up your source tree.
- In order for Xcode to know what sources you want to debug, you need to tell it where to find them. In the File menu choose 'Add Files to "your-project"'. Browse to your source tree and either command-click the set of directories you care about, or just select the whole tree. Xcode will import everything, including .hg directories, so you will need to remove them from your project after the import. The amount of work to carefully select which source files to import doesn't seem to be much different than the effort to clean up after importing everything.

Starting a debug session:

Make sure breakpoints are active (which implies running under the debugger) by opening the Product menu and selecting "Debug/Activate Breakpoints" (also shown by the "Breakpoints" button in the top right section of the main window). Then click the "Run" button or select "Run" from the Product menu.

## 5.19 Defunct (But Interesting) Approaches

ATSAL originally incorporated PyQt, a system for merging a Qt application with a Python interpreter. PyQt interfaces to the Qt library. We abandoned this approach for three reasons. First, the licensing restrictions were not compatible with an open source project. Second, PyQt implements Python-to-Qt bindings, but not really the reverse as well. Third, PyQt runs ATSAL under Python, not the other way around.

### 5.19.1 PyQt and Sip

PyQt and Sip have been abandoned in favor of PythonQt, so this section is present in case it comes in handy later.

PyQt provides access to Qt via Python. Arguments to Qt methods are converted from Python native format to Qt/C++ native format and back again, so the environment behaves as expected to users of either language. PyQt's bindings are generated by sip, a Python-to-C++ bindings generator. Sip is described in the next section. Sip accepts descriptions similar to include files which specify the types of bindings, and emits C++ code that is compiled to produce the bindings. The resulting bindings are then compiled into a library for use with Python. Install instructions for Python and sip are upcoming shortly.

#### 5.19.1.1 SIP

Download SIP from here:

<http://www.riverbankcomputing.com/software/sip/download>

Build sip:

```
tar -zxf sip-4.15.4.tar.gz
cd sip-4.15.4
python configure.py
make
sudo make install
```

#### 5.19.1.2 PyQt

Download PyQt 5 from the above location. Build PyQt:

```
tar -zxf PyQt-gpl-5.2.tar.gz
cd PyQt-gpl-5.2
python configure.py
make
sudo make install
```

The article's authors suggest that you may have to ensure the QMAKESPEC, a Qt environment variable, is set for the above to work.

```
export QMAKESPEC=macx-g++
```

The configure step failed when I did this, with a message that qmake had failed because there was no cache file. I found that I had to copy a .pro file into the directory, type qmake foo.pro to generate an output file and a cache, then delete the .pro and generated files.

Also note that the QMAKESPEC setting does not match the default for XCode, so I had to set this manually to get the script to work.

#### 5.19.1.3 Python/Qt Superhybrids

##### 5.19.1.3.1 Software Components

<i>Component</i>	<i>Version</i>	<i>Description</i>
Qt	5.1.1	Cross-platform user interface library, from qt.digia.com
PyQt	5.2	Python-to-Qt bindings, from <a href="http://www.riverbankcomputing.com/software/pyqt">http://www.riverbankcomputing.com/software/pyqt</a>
SIP	4.15.4	<a href="http://pyqt.sourceforge.net/Docs/sip4/build_system.html">http://pyqt.sourceforge.net/Docs/sip4/build_system.html</a>

### 5.19.1.3.2 Python and Qt “Superhybrid”

ATSAL requires Python 3 as an extension language, so the Python interpreter needs to be an integral part of ATSAL. In addition to providing bits of glue code, Python programs should be able to access Qt to build custom user interfaces, or link in and use other C++ code. This article also assumes that Apple’s XCode development package is installed, along with the developer command line tools that are separately downloaded from Apple’s developer web site.

We use an approach described in an excellent tutorial on:

<http://lynxline.com/qt-python-superhybrids/>

Rapid changes in Qt, PyQt, and sip required many changes from the original article. The description below applies to Qt5.1.1, PyQt5, and SIP 4.15.3. The approach has been tested on the Macintosh platform, under OS X 10.9. The below assumes that Qt5.1.1 is already built and installed.

Next, the tutorial presents a small Qt application that serves as a base for the demo. They choose a Canvas-based application so they can prototype Canvas operations with PyQt at runtime. The application’s window consists of an area for typing Python code on the left and the canvas on the right.

In a fresh directory, create PyQtHybrid.pro:

```
TEMPLATE = app
CONFIG += qt
QT += core gui widgets
HEADERS += MainWindow.h
SOURCES += MainWindow.cpp
cache()
```

This file is used by QMake to generate a makefile. In this case we use Apple’s XCode environment to build the application. In Qt5.1.1, “widgets” have been factored out from “gui” as a separate module, so they need to be included separately. Finally, “cache()” is a recent addition that speeds up some aspect of the build process. It is present here to avoid a warning message.

To process this file, type:

```
qmake PyQtHybrid.pro
```

It will generate a makefile equivalent according to the setting of QMAKESPEC. If set for macx-xcode, it produces PyQtHybrid.xcodeproj.

Next, we create a small standalone Qt application that creates the main window. First, create a file called `main.cpp`:

```
#include <QtGui>
#include <QtWidgets/QtWidgets>
#include "MainWindow.h"

int main(int argc, char ** argv)
{
    QApplication app(argc, argv);
    MainWindow window;
    window.resize(1000,700);
    window.show();
    return app.exec();
}
```

Next, create MainWindow.h:

```
#ifndef MainWindow_H
#define MainWindow_H

#include <QtWidgets/QMainWindow>
class QPushButton;
class QGraphicsView;
class QGraphicsScene;
class QPlainTextEdit;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget * parent = 0L);
    virtual ~MainWindow();

signals:
    void runPythonCode(QString);

private slots:
    void runPythonCode();

public:
    QGraphicsView * viewer;
    QGraphicsScene * scene;
    QPlainTextEdit * editor;
    QPushButton * pb_commit;
};

#endif // MainWindow_H
```

And MainWindow.cpp:

```
#include <QtGui>
#include <QtWidgets/QtWidgets>
#include "MainWindow.h"

MainWindow::MainWindow(QWidget * parent):QMainWindow(parent)
{
    QSplitter * splitter = new QSplitter;
    setCentralWidget(splitter);

    QWidget * editorContent = new QWidget;
    splitter->addWidget(editorContent);

    QVBoxLayout * layout = new QVBoxLayout;
    editorContent->setLayout(layout);

    editor = new QPlainTextEdit;
    layout->addWidget(editor);

    pb_commit = new QPushButton(tr("Commit"));
    connect(pb_commit, SIGNAL(clicked()),
           this, SLOT(runPythonCode()));
    layout->addWidget(pb_commit);

    scene = new QGraphicsScene(this);
    viewer = new QGraphicsView;
    viewer->setScene(scene);
    splitter->addWidget(viewer);

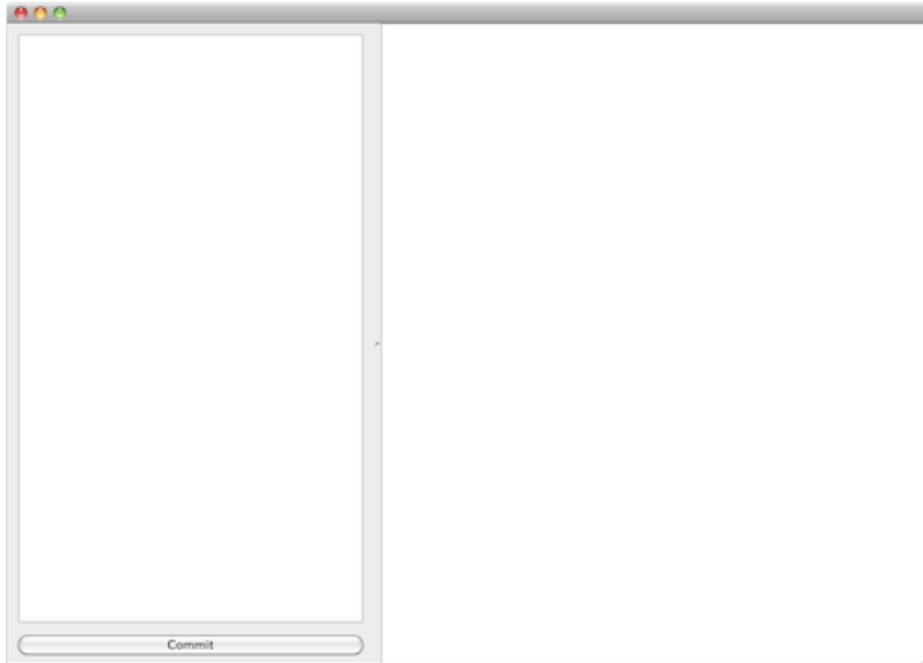
    splitter->setSizes(QList() << 400 << 600);
}

MainWindow::~MainWindow()
{
}

void MainWindow::runPythonCode()
{
}
```

```
emit runPythonCode(editor->toPlainText());  
}
```

Open the `PyQtHybrid.xcodeproj` file to launch XCode and build the application. It produces this window:



To use this in Python, several changes are needed:

- The application must be converted into a dynamic library.
- We must use SIP to generate bindings that allow this class to be called from Python.
- We must replace `Main.cpp` with a `Main.py` for Python.

First, alter `PyQtHybrid.pro` to create a dynamic library:

```
TEMPLATE = lib  
CONFIG += qt dylib  
QT += core gui widgets  
HEADERS += MainWindow.h  
SOURCES += MainWindow.cpp  
cache()
```

Next, create a folder, “sip,” in the current directory:

```
mkdir sip  
cd sip
```

Create a “sip” file that describes the Python wrapper for the `MainWindow` class, called `mainwindow.sip`. It looks very similar to the `MainWindow.h` include file.

```
%Module(name=PyQtHybrid, version=0)  
%Import QtGui/QtGui.mod.sip  
%Import QtWidgets/QtWidgets.mod.sip  
%If (Qt_5_0_0 -)  
  
class MainWindow : QMainWindow {  
%TypeHeaderCode  
#include "../MainWindow.h"  
%End  
public:  
    MainWindow();
```

```

    virtual ~MainWindow();
signals:
    void runPythonCode(QString);
private slots:
    void runPythonCode();
public:
    QGraphicsView * viewer;
    QGraphicsScene * scene;
    QPlainTextEdit * editor;
    QPushButton * pb_commit;
};
%End

```

To generate wrappings from sip we need a `configure.py` file:

```

import os
import sipconfig
from PyQt5 import QtCore

build_file = "PyQtHybrid.sbf"

# Get the SIP configuration information
config = sipconfig.Configuration()

pyqt_sip_flags = QtCore.PYQT_CONFIGURATION['sip_flags']

# Run SIP to generate the code. There must be a better way to get the SIP
# include files.
os.system(" ".join([ \
    config.sip_bin, \
    "-C", ".", \
    "-b", build_file, \
    "-I", config.sip_inc_dir, \
    "-I", "/Users/tkent/atsal/atsal/virtualenvs/py3/share/sip/PyQt5", \
    "-I", "/Users/tkent/atsal/atsal/atsal/Debug", \
    "-I", "/Users/tkent/Qt5.1.1/5.1.1/clang_64/include/QtCore", \
    pyqt_sip_flags, \
    "MainWindow.sip" \
]))

installs = []

installs.append(["MainWindow.sip", os.path.join(config.sip_inc_dir, "PyQtHybrid")])
installs.append(["PyQtHybridConfig.py", config.sip_mod_dir])

makefile = sipconfig.SIPModuleMakefile(
    configuration=config,
    build_file=build_file,
    installs=installs
)

makefile.extra_libs = ["PyQtHybrid"]
makefile.extra_lib_dirs = [".", "../Debug"]
# There must be a better way...
makefile.extra_include_dirs = ["/Users/tkent/Qt5.1.1/5.1.1/clang_64/include", \
"/Users/tkent/Qt5.1.1/5.1.1/clang_64/include/QtCore", \
"/Users/tkent/Qt5.1.1/5.1.1/clang_64/include/QtWidgets", \
"/Users/tkent/Qt5.1.1/5.1.1/clang_64/include/QtGui"]

makefile.generate()

content = {
    "PyQtHybrid_sip_dir": config.sip_inc_dir,
    "PyQtHybrid_sip_flags": pyqt_sip_flags
}
sipconfig.create_config_module("PyQtHybridConfig.py", "PyQtHybridConfig.py.in", content)

```

The configuration file reads a file called `PyQtHybridConfig.py.in` file, shown below:

```

from PyQt5 import QtCore
from PyQt5 import QtWidgets
# @SIP_CONFIGURATION@

class Configuration(pyqtconfig.Configuration):
    def __init__(self, sub_cfg=None):
        if sub_cfg: cfg = sub_cfg
        else: cfg = []
        cfg.append(_pkg_config)
        pyqtconfig.Configuration.__init__(self, cfg)

class PyQtHybridModuleMakefile(pyqtconfig.QtGuiModuleMakefile):
    def finalise(self):
        self.extra_libs.append("PyQtHybrid")
        pyqtconfig.QtGuiModuleMakefile.finalise(self)

```

Next we run the configuration file to generate the Makefile:

```

python configure.py
make

```

The output of this step is `PyQtHybrid.so`. Now all we need is a `Main.py` to invoke the library:

```

import sys
sys.path.append('sip')

from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQtHybrid import *

class RunScript(QObject):
    def __init__(self, mainWindow):
        QObject.__init__(self)
        self.mainWindow = mainWindow

    def runScript(self, script):
        mainWindow = self.mainWindow
        exec(str(script))

a = QApplication(sys.argv)
w = MainWindow()
r = RunScript(w)
w.setWindowTitle('ATSAL 1.0e-20')
w.resize(1000,800)
w.show()
w.runPythonCode.connect(r.runScript)
# Old way for signalling doesn't work any more
# a.connect(w, SIGNAL('runPythonCode(QString)'), r.runScript)
# a.connect(a, SIGNAL('lastWindowClosed()'), a, SLOT('quit()'))
a.exec_()

```

As the article says: “We have an application which can program itself—that’s cool.”

Let’s try small code snippets like:

```

mainWindow.statusBar().show()

```

A status bar appears. Let’s change the background of the canvas:

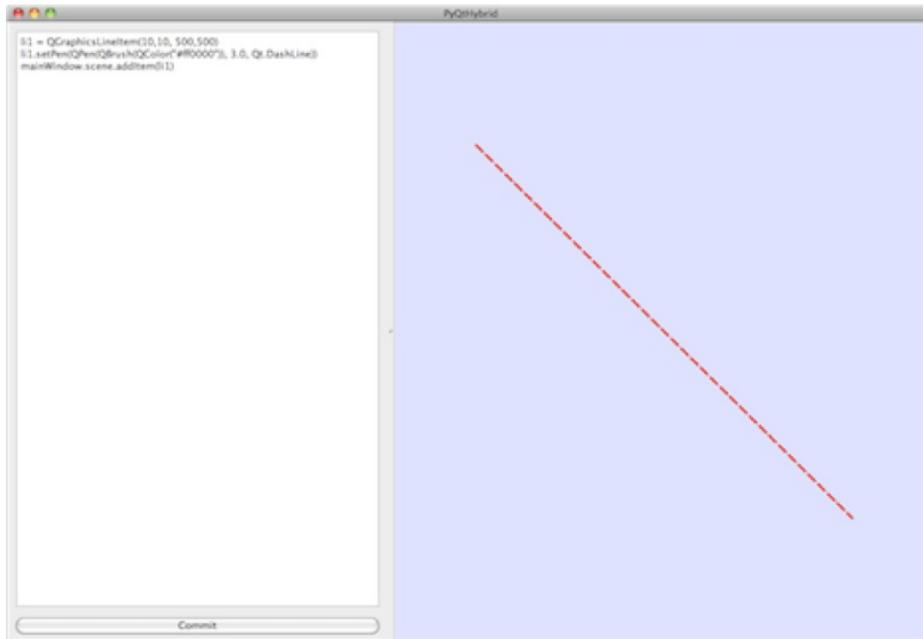
```

mainWindow.scene.setBackgroundBrush(QColor('#e0e0ff'))

```

Can we create new objects? Of course yes:

```
li1 = QGraphicsLineItem(10,10, 500,500)
li1.setPen(QPen(QBrush(QColor("#ff0000")), 3.0, Qt.DashLine))
mainWindow.scene.addItem(li1)
```



## 6 Update Issues

This log was started in Nov 2014 to note skew issues resulting from updates to software components.

### 6.1 XCode 6.1, 2014-11-14

<i>Issue</i>	<i>Resolution or Workaround</i>
qmake generates <code>.xcodeproj</code> file for Mac OS X 10.8 SDK, but only 10.9 and 10.10 bundled in XCode 6.1	As a workaround, created <code>ma.sh</code> script, which generates a new <code>.xcodeproj</code> , then edits the <code>project.pbxproj</code> file to use Mac OS X 10.9. Also created <code>mn.sh</code> for Nexus. Add to <code>.bash_profile</code> aliases <code>ma</code> and <code>mn</code> to produce the <code>ATSAL.xcodeproj</code> and <code>Nexus.xcodeproj</code> files, respectively.
The above fix was incomplete.	Changed <code>/Users/tkent/Qt/5.3/clang_64/mkspecs/macx-clang/qmake.conf</code> to contain <code>QMAKE_MACOSX_DEPLOYMENT_TARGET = 10.9</code>
XCode could not find any project include files because (apparently) it no longer looks for project include files in the project.	I added <code>USER_HEADER_SEARCH_PATHS = "include/**"</code> to <code>project.pbxproj</code> , also as part of the previously mentioned scripts.
ATSAL build fails at link time with references to the std C++ library	Starting with Mac OS X 10.9 (Mavericks), Apple changed the default standard C++ library from <code>libstdc++</code> to <code>libc++</code> . The Qt binary distribution is built for <code>-stdlib=libstdc++</code> . So for the moment, I changed <code>ATSAL.pro</code> to use <code>libstdc++</code> again.
XMLRPC built -m32 versions for both the 32- and 64-bit builds.	Previously, if the architecture was unspecified, it built for the default host, in this case, 64-bit. Instead it built for 32-bit. The fix involved explicitly specifying the desired build. (XMLRPC is built for both 32-bit, for compatibility with XSPEC server, and 64-bit, for ATSAL.)
Still trying to get qmake to use the 10.9 Mac OS X SDK.	Modified <code>qt-install-dir/5.3/clang_64/mkspecs/qdevice.pri</code> to include the line: <code>!host_build:QMAKE_MAC_SDK = macosx10.9</code>
PythonQt build.	Modified <code>pythonqt/build/common.prf</code> to build for 10.9.

### 6.2 Icons

The original reason for updating XCode was an attempt to resolve the problem with adding application icons. The new version's icon management doesn't work either. This didn't help:

<http://martiancraft.com/blog/2014/09/vector-images-xcode6/>

I tried both assets with each size supplied separately, and using a PDF file with vector data. The icon handling remains very buggy in XCode 6 and I could not get it to work.

## 7 Design Issues

This section discusses general design philosophies and considerations.

### 7.1 Sidebars

In this discussion I use “controls” and “widgets” interchangeably.

Tool editor controls are placed in sidebars because the number of controls varies widely from editor to editor, and sidebars are scrollable. Sidebars consist of hierarchies of collapsible panes so users can expose only the most frequently used controls.

A `SideBar` is a complete side panel of controls. There is one sidebar for each tool editor. A sidebar contains a set of `Bars` stacked atop each other. A `Bar` subclass is usually a single row of controls. A `Block` is a `Bar` subclass that may be several rows high and embodies a set of related controls.

A `CollapsiblePane` contains a disclosure triangle, a name, and some optional parameters. It also has an open pane, typically a `Block` subclass, that shows its contents when open. There are a few `CollapsiblePane` subclasses that specialize this behavior. A `CPFoo` subclass is a `Block` that contains the open state of a `CollapsiblePane`. (CP subclasses are not `CollapsiblePane` subclasses.)

There are several slightly specialized widgets, named `FooBar`, which are used inside `Bars`. All `Bars` are designed to be resizable in width over a limited range.

#### 7.1.1 Sidebars and Signals

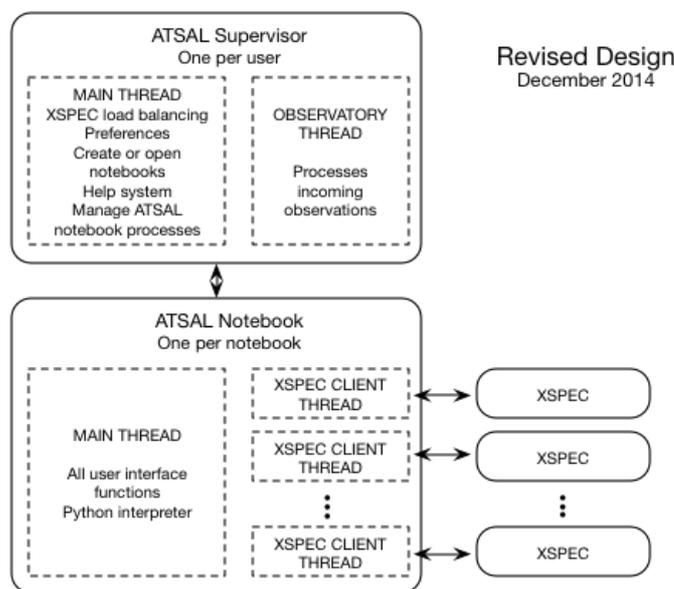
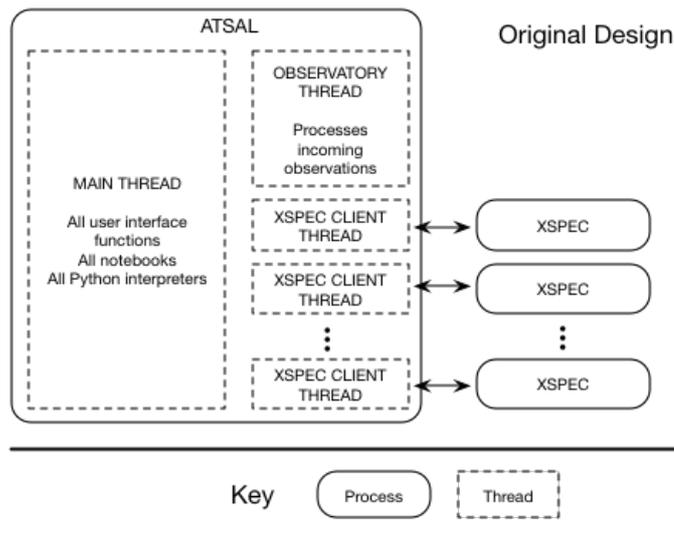
I have seen Qt applications that overuse signals and slots operate sluggishly as a result. Signals are reasonably efficient, but they can cause cascades of activity if not used carefully. Signals exist so that a control need not know anything about the thing it is controlling. For example, a `QPushButton` doesn't want to be in the business of calling a method in a `SpectrumPlot`. It doesn't even want to know about its immediate parent widget, which might be a dialog or a `Bar` or what-have-you. But this anonymity extends *up* the chain, not downward. Tool editors know about their controls, but the controls know nothing about the editors.

For this reason, the hierarchy of controls that make up sidebars generally uses signalling to send commands upward through the hierarchy, but normal function calls are used to convey state changes downward to update the controls. Subordinate widgets that contain several controls typically trap their individual signals, then consolidate the results and emit a new signal up the chain. As an example, `BarTextEmphasisAlign` contains several icons that accept state changes for text emphasis and alignment. Rather than wiring each of these signals to something higher up, all the icon signals are received in `BarTextEmphasisAlign`, and consolidated into a couple of new signals, one for emphasis and one for alignment, and these new signals are emitted. This means `BarTextEmphasisAlign` encapsulates the local user interface updates and sends out state changes in a more usable format. The resulting output can usually be fed directly to the target editor object, minimizing cascades of signals.

### 7.2 A Process for Each Notebook

It is not possible to run multiple instantiations of either PythonQt or Python in a single process, so in December 2014 I changed the ATSAAL architecture to accommodate this constraint. The new design splits ATSAAL into a supervisor component (in `atsal/supervisor`) and a notebook component (in `atsal`). The user thinks of the supervisor as ATSAAL, and double clicks it to run it. This is what shows up in the dock as well. The supervisor in turn runs a separate notebook process for each open notebook. These notebook processes currently show up in the dock—I haven't found out how to suppress them yet. Macs presently support two types of process: those with user interfaces and dock icons, and faceless background processes such as daemons. ATSAAL requires a third type: one with no presence in the dock but a complete user interface.

To the extent possible, the multiple processes look like a single application. For example, a common base class presents the same menu bar in the supervisor and each notebook.



## 7.2.1 Quitting in a Multiple Process Environment

A few things become quite a bit more complicated in the multiple process environment. Assume for example, that the user started ATSAAL (creating a supervisor process), then opened two notebooks, A and B. With B as the current process, the user quits. Any one of the processes might first need to offer to save information prior to quitting, and the offer includes the ability to abort the quit entirely. So the quit in B is not processed locally, but redirected to the supervisor. This involves serializing the request into a bit of XML and writing it to stdout. (Why XML? Right now all the commands are a single function call with arguments, but later serialized objects such as clipboard contents may need to be passed.)

The supervisor polls all the notebooks for input every 200 ms or so, so it receives the quit request fairly quickly. It must tell all the notebooks to quit, starting with notebook B, since this is the one the user believes to be current. So the first step is to tell B to quit, even though B just told the supervisor to quit. When B receives this request, it first prompts the user to save any changes. If the user cancels the save at that point, we have to tell the supervisor to abort the quit process. Otherwise, we don't have to tell the supervisor anything, because it receives a signal upon the death of the process. It can tell from this signal whether the death of B is a normal user-requested exit or a crash. In the former case, it quietly removes the process from its list of notebook processes. In the latter case, the crash is reported to the user.

If the quit process has not been aborted, the supervisor next relays the request to notebook A, with the same option to save and/or abort the quit.

I experimented with half a dozen ways to read from stdin without blocking, but I could not find any portable way to accomplish this without using threads. So each notebook process has a separate thread whose only function is to read a line from stdin. When each line is received, it is sent to the main thread via signal. Since signals are queued, they cross

the thread boundary safely.

This mechanism must be further extended to kill XSPEC processes, but this can probably be done abruptly, without a handshake.

## 7.2.2 Database Regeneration

Briefly stated, the user regenerates the observations database by clicking "Regenerate Database" in the sidebar of an observation tool in the notebook. The request is passed to the supervisor for processing, and when complete, each notebook is notified of the completion. Then each notebook needs to close and reopen its database, reload the observations, and refresh any open observation tools, tags, even plots. This also involves threads in both the supervisor and all the notebooks, so there is a lot of passing the buck. I am recording the process in detail here because I get a headache when I try to follow it.

ATFiles\_2015-10-10.tar.gz: original complete set.

ATFilesMaster.tar: abbreviated (but broken?) set.

Notebook	Notebook Thread	Supervisor	Supervisor Thread
Click "Regenerate Database"			
CPLocalSearch:: regenerateDatabase			
ATNotebookApp:: regenerateDatabase			
Supervisor:: regenerateDatabase			
Supervisor:: sendCommand ("regenerateDatabase")			
			ATDatabaseClient:: doWork, emits resultReady
			ATDatabaseController:: handleResults
		databaseOpCompleted (op)	
		ATSupervisorApp:: databaseOpCompleted	
		ATBaseApp:: databaseOpCompleted	
		ask controller for master list of obs records	
		Tell each notebook databaseOpCompleted	
		NotebookProcess:: databaseOpCompleted	
Supervisor:: processLine(line)			
Supervisor:: processCommand(cmd, args)			
NotebookWindow:: databaseOpCompleted			
ATDatabaseController:: reopenDatabase			
	ATDatabaseClient:: doWork		
	ATDatabase:: reopenDatabase		
	emit resultReady ("ReopenDatabase")		
ATBaseApp:: handleResults			

(presently ignores this)			
PrvNotebook:: databaseOpCompleted			
PrvNotebook:: updateInterestingFilesList			
Tell all tools databaseOpCompleted			
PrvToolObservatory:: databaseOpCompleted			
PrvToolObservatory:: search()			

## 7.2.3 Other Multiple Process Notes

### 7.2.3.1 LSUIElement

I added this setting to the root level of `QInfo.plist` for the ATSAAL notebook:

```
<key>LSUIElement</key>
<string>1</string>
```

This hides the notebook processes so they do not appear in the dock, nor do they appear as options in Force Quit. (There is also an `LSBackgroundOnly` key, but this does not appear to be necessary.)

**Update:** I backed out this change because it also prevents display of the menu bar. I was unable to find any way to suppress the display of icons in the dock for each notebook process, while preserving a menu bar. Experiments with `TransformProcessType` were also unsuccessful.

Some marginally helpful info:

[http://www.sheepsystems.com/developers\\_blog/transformprocesstype--bette.html](http://www.sheepsystems.com/developers_blog/transformprocesstype--bette.html)

<http://stackoverflow.com/questions/7596643/when-calling-transformprocesstype-the-app-menu-doesnt-show-up>

### 7.2.3.2 Switching Seamlessly Among Processes

When one process sends a message to another one, and the second process displays a window in response, the window remains below open windows in the current process because the first process is still active. To emulate the behavior of a single process requires that we push the new process' window to the front. I can't find a Qt function with this capability.

#### 7.2.3.2.1 Mac OS X Solution

The Mac native class `NSApplication` includes a method, `activateIgnoringOtherApps`, which makes the requesting application the active application. As nearly as I can tell from the web, Qt does not instantiate this class, so at startup time it is first necessary to do so:

Other notes on this problem:

[http://www.sheepsystems.com/developers\\_blog/transformprocesstype--bette.html](http://www.sheepsystems.com/developers_blog/transformprocesstype--bette.html)

<http://stackoverflow.com/questions/7596643/when-calling-transformprocesstype-the-app-menu-doesnt-show-up>

<http://www.cocoabuilder.com/archive/cocoa/303437-transformprocesstype-still-doesn-show-menu-bug-id-5905139.html>

```
gNSApp = [NSApplication sharedApplication];
```

Also, this brings all the application's windows forward (I think!), which isn't the desired effect, so a better solution is desirable.

```
ProcessSerialNumber psn;
pid_t pid = getpid();
OSStatus err;
err = GetProcessForPID(pid, &psn);
err = TransformProcessType(&psn, kProcessTransformToForegroundApplication);
err = ShowHideProcess(&psn, true);
[NSMenu setMenuBarVisible:NO];
SetFrontProcess(&psn);
[NSMenu setMenuBarVisible:YES];
```

Notes on transforming process types:

- [http://www.sheepsystems.com/developers\\_blog/transformprocesstype--bette.html](http://www.sheepsystems.com/developers_blog/transformprocesstype--bette.html)
- <http://stackoverflow.com/questions/7596643/when-calling-transformprocesstype-the-app-menu-doesnt-show-up>
- <http://www.cocoabuilder.com/archive/cocoa/303437-transformprocesstype-still-doesn-show-menu-bug-id-5905139.html>

### 7.2.3.2.2 Linux Solution

TBD.

### 7.2.3.3 Threads

#### 7.2.3.3.1 The Current Thread

Sometimes you need to know whether you are running in the main thread or another thread. This is the trick.

```
QApplication::instance()->thread() will give you main threadid.
```

## 7.3 Version Skew

Version skew is perhaps the single biggest impediment to using open source software components. We adopt a policy of rigid control to try to ameliorate this. ATLAS bundles all components needed for a release with each release, sandboxing them to be local to ATLAS. This guarantees a known environment for ATLAS (apart from the uncontrollable variable of which OS release the user is running). At the same time, it prevents ATLAS's component tools from interacting with different versions of the same tools that may be installed on the user's machine for other purposes.

This in turn dictates a slower update cycle, as many combinations of tools do not work and we must select those that do. Hence ATLAS's tentative update cycle is roughly 1 year.

## 7.4 Responsiveness and Parallelism

A critical ATLAS goal is to take the best possible advantage of parallelism—multiple CPU cores or remote computing resources—in order to provide a more responsive analysis system. But there are several constraints that complicate this. XSPEC itself, with a few notable exceptions, is not designed for parallel execution, and a retrofit would probably be prohibitively expensive. Python, ATLAS's user extension language, supports threads, but they are not truly parallel. Qt supports multithreaded operation, with the constraint (shared by all user interface systems) that the user interface code runs in the main thread. Finally, an ATLAS notebook contains a series of tools that are, at least conceptually, sequential in their execution. So how do we get parallel operation with all these constraints? And what does “parallel” really mean?

Before discussing this, we define a *refresh cycle* as the sequence of events between when the user initiates a refresh and the last tool completes execution. A refresh cycle consists of a preparatory phase, an arbitrary number of *refresh iterations*, and a post-completion phase. It will shortly become clear why the refresh cycle design profoundly influences parallelism.

The XSPEC server lies at the bottom of the hierarchy, and ATLAS achieves parallel operation by running multiple XSPEC server processes, one (or perhaps more) per CPU core on multicore machines. XSPEC memory and I/O

demands are relatively modest, so CPU time is the primary limiting factor in XSPEC performance. ATSSAL allocates a thread to each server, so that its own execution is not held up waiting for servers. This achieves **XSPEC parallelism**. Future versions of ATSSAL may support XSPEC servers running on remote computers as well, allowing a much greater degree of parallelism. (However, moving all the necessary files between systems may limit the effectiveness of this.)

When the user hits the refresh button, many user interface operations must be suspended during the refresh cycle, to avoid a possible crash. The user cannot delete a model, for example, while a fit is being calculated for it. So the notebook is mostly off limits during a refresh. Because of the way python is designed, this also prevents operation of other notebooks: no work can be performed during a refresh. To address this, ATSSAL was restructured so that there is a single ATSSAL supervisor and a separate process for each notebook. Since each notebook has its own python interpreter, they all run in parallel. Hence this achieves true **notebook parallelism**. The supervisor doles out XSPEC servers to various notebooks, imposing an overall quota and migrating servers to the notebooks that need them.

A refresh cycle brings all tools up to date, by executing a python-based refresh loop. The loop is implemented in python to allow advanced users to implement more complex actions. Many tools refresh instantly, but each plot tool contains a hierarchy of spectra; each spectrum a hierarchy of data plots and models; and each model another hierarchy of data plots. Corresponding to each model is a separate XProxy, which acts as an intermediary between the model and the XSPEC server assigned to it.

What happens if there are multiple spectra within a plot tool? Or multiple models within a spectrum? Or both? Since each model gets its own XSPEC server, these run fully in parallel, achieving **spectrum and model parallelism**. This means a user can configure multiple models and run them all at once.

If a user writes custom compute-intensive python code, the refresh loop and user interface will lock up, since python lacks true **python parallelism**. This is especially likely if the user implements custom models. However, such models are implemented using XSPEC's python interpreter, not ATSSAL's. Hence python models will not interfere with ATSSAL responsiveness. **Note however that LBA algorithms implemented in ATSSAL's python will lock up the user interface if they are compute-intensive.**

Finally, what if a notebook contains multiple plot tools? Do they all run at the same time? No, at least not yet. The reason is that some tools depend upon the output from prior tools: a notebook is inherently sequential, like most programs. So the default rule is that a single tool runs at a time, until all have completed. However, if a tool is known to have no dependencies on one or more prior tools, it *could* run at the same time. For example, each of a series of three consecutive plot tools do not normally depend upon the output of prior plot tools, so they could all be executed simultaneously. But since many tools can call a python function or use a python variable as input, if such calls exist, it isn't practical to trace their dependencies. So in the future, **tool parallelism** will be permitted in a few special cases, but not in general. This means that in a notebook containing multiple plot tools, each plot tool must complete execution before the next is started.

During ATSSAL's refresh cycle, each refresh iteration checks on the status of tools in the process of executing, and moves along to new tools when appropriate, then returns to the main event loop. This means that ATSSAL's user interface remains fully operational during a refresh cycle. Thus the design also achieves **user interface parallelism**.

## 7.4.1 Responsiveness

Both the XSPEC server and ATSSAL have threads that manage I/O between the processes. These threads are essentially polling loops. Continuous polling would be fastest, but would impose too high a compute load. Slower polling is more efficient, but limits throughput. It will take some tinkering to fine tune these parameters. See `kStatusPollingMs` in ATSSAL's `Config.h`, and `kServerPollingRateMs` in XSPEC's `XServer.h`.

## 7.4.2 Large-scale Parallelism

ATSSAL can run up to about 40 fits simultaneously on a local 8-core machine. But what about doing 100 fits? 1000? This is a very different design problem. Here are some preliminary thoughts on how this might work.

### 7.4.2.1 Use Cases

**Apply the same fit to multiple spectra.** For example, one might begin with a list of candidate sources that are suspected to share some trait, and apply the same fit to all of them allow the truly common sources to be identified.

**Apply a series of fits with different parameters, models, or various constants to the same source.** Here

the goal is to zero in on the models best suited to analyze a particular source.

### 7.4.2.2 Setup

Initially, users create a list of host computer resources, such as NASA's Hera facility or Amazon's virtual machines, and set up login information, billing info if appropriate, and tuning hints that help the load leveler work. After initial setup, the distribution of user work across specific compute resources is transparent.

Next, the user configures a block of work. Large scale operations of this sort are well-suited for python programs. A program might read a list of spectrum files and configure the same fit for each one, creating a batch of operations to be performed. The result is a list of tasks to be performed, and necessary input files for each task.

### 7.4.2.3 Deployment

Next, ATXSAL iterates through the available remote hosts, requesting a process slot from each in turn until one signals availability. If no process slots are available, and the user setup okays it, another remote host is allocated. If no more hosts are available, ATXSAL waits a few seconds and iterates again. If a slot is available, ATXSAL uploads the work packet, which includes input files. ATXSAL also downloads available results packets to the originating machine.

### 7.4.2.4 Remote Processing

The daemon running on each remote system is responsible for the following:

- Login handling.
- Load handling. This is done (tentatively at least) by accepting a new work packet only if the existing load factor is below a threshold.
- Instantiating and running an XSpec session. The daemon creates a thread for each XSpec instance, passing commands to the instance and receiving and processing results. Results are packaged for return to the originating system. Upon completion, the XSpec process and its thread are terminated, and the results packet is transferred to the originator.

Does each ATXSAL user get an entire virtual machine, or can virtual machines be shared among multiple users?

### 7.4.2.5 Results Processing

As results roll in, they are processed locally by ATXSAL, which parses them and organizes the results for viewing and/or python post-processing. Output from this step is a series of XML files containing structured results and plot data. A user can write a python program to sift through the structured results. Alternatively, they can use a results browser. This is a modified plot tool for viewing the plots, combined with a spreadsheet-style list of the results. This tool presents a list of all the fits performed, updating as new results become available. Users can select subsets of this list to view overplotted results, or view the numerical results in spreadsheet form.

## 7.5 User Interface

ATXSAL needs a modern, productive user interface that supports densely packed controls, because some tools have a large number of options. We settled on control panel sidebars as an approach, because scrolling permits any number of controls. We implemented progressive disclosure triangles to permit users to expose different control subsets during different phases of analysis. We redesigned these controls to be a bit smaller, and selected system fonts for each platform with similar metrics. The main purpose of this choice is to reduce instances of text that look correct on one platform but truncated on another. Sidebar controls also resize themselves when possible to accommodate the actual text, another attempt to improve platform independence.

### 7.5.1 ATXSAL User Interface Goals

- Implement compact, progressive disclosure controls.
- Reduce effort needed to make control panels look right on different platforms, by designing low level widgets with this in mind.
- Lay the groundwork for translation to different languages later.
- Give ATXSAL a distinct, self-consistent user interface style, rather than writing extra code to make it feel more

native on each platform.

- Use color to convey specific information only. Examples include color coding of selections to show membership in selection sets; color coding of elements; enabled controls in color, disabled in gray; colored text to convey hyperlinks and important notes; and coloring to distinguish among multiple overlaid plots. Color blind users should be able to use the software without serious inconvenience. See also NASA's [Color Usage Site Home Page](#).

### 7.5.1.1 ATSA User Interface Non-Goals

- No “gratuitous” use of animation. This is a popular technique among mobile applications, but labor intensive to implement. (ATSAL will employ animation where appropriate for data display though.)
- No tracking of the latest graphic arts trends.
- No vector artwork for icons. (Eventually vector artwork will probably be a better choice, because it gets around problems with displays of different pixel densities, but Qt's vector icon tools are not yet reliable.)
- No platform-specific code unless it meets a specific functional goal.

## 7.6 Python Design Issues

This covers various issues that surfaced while integrating Python into ATSA.

### 7.6.1 Python Debugger Notes

This covers several issues related to debugging of Python.

#### Pdb Debugger

[Pdb Debugger](#)

(Remote URLs are not yet included inline in PDFs.)

#### Bdb Low-level Debugger

[Bdb Low-level Debugger](#)

(Remote URLs are not yet included inline in PDFs.)

#### Inspect

[Inspect](#)

(Remote URLs are not yet included inline in PDFs.)

### Basic Types (incl. Frame and Traceback)

[Basic Types \(incl. Frame and Traceback\)](#)

(Remote URLs are not yet included inline in PDFs.)

### IO Module

[IO Module](#)

(Remote URLs are not yet included inline in PDFs.)

### 7.6.2 Python Editor

Modify so that return always goes to the same indent level as the preceding line. (Don't bother adjusting this rule based on syntax of previous line though.)

Menu functions (Linux bindings tentative):

Mac	Linux	Function
⌘]	^]	Indent block
⌘[	^[	De-indent block
⌘F	^F	Find
⌘G	^G	Find next
⌘J	^J	Replace
⌘⇧J	^⇧J	Replace and find next
⇧Tab	⇧Tab	Back tab

"J" is used for replace instead of "R" because "R" means Refresh. What about emacs-style bindings?

### 7.6.2.1 Tabs vs. Spaces

Many others have railed *ad nauseum* on the strengths and weaknesses of using tabs vs. spaces in editing files. I was shocked to find that some Python files use leading spaces rather than tabs, further shocked to see that Python 3 prohibits a mixture of tabs and spaces, and mortified to find that the Python PEP-8 Style Guide strongly suggests the use of spaces “in all new Python files.”

There, I've stated my prejudices unambiguously. So what should ATSAAL do about it? First there is the issue of why. Python is unusual in that leading whitespace conveys nesting information: it is syntactically significant. Combine this with the fact that editors allow different amounts of indentation to be shown for each tab level, and it becomes immediately evident that mixtures of tabs and spaces can appear to have different nesting than they actually *do* have. Hence the prohibition on mixing.

Okay, so let's everybody use tabs, I suggest cheerfully. Well that isn't the way people have done it. Allegedly at least, most existing Python uses spaces. So if ATSAAL abandons this convention, cutting and pasting code will yield the mixed use case. I considered translating spaces to tabs on file open and converting back on save, but some diff tools may register spurious changes from this, and guessing the user's preferred tab stops is not reliable.

So that leaves using spaces, *always*. But tab and backspace simulate the presence of tabs, by deleting or inserting the necessary groups of spaces. This preserves the more efficient behavior of tabs. Four-space tabs are hard-coded, per the PEP-8 spec, so if you edit a file that employs 2-space tabs the tab key won't behave as expected. I don't like it either, but there is no general solution.

Since the admixture of spaces and tabs doesn't work, ATSAAL's Python editor highlights tabs with a dotted line. This makes it easy to alter them to conform with the standard.

### 7.6.3 Python Reloading

Python's import command does not re-import a file that has already been imported, so changes to the source files that are part of a notebook aren't automatically incorporated. Currently we use [reloader 0.6](#) to perform the necessary reloads.

### 7.6.4 Surfacing ATSAAL to Python

This article is reproduced from the Digia (previously TrollTech) web site (to preserve it if it disappears). Here is the original link:

<http://doc.qt.digia.com/qq/qq23-pythonqt.html>

#### 7.6.4.1 Embedding Python into Qt Applications

by Florian Link

Embedding scripting languages into C++ applications has become very common. Alongside many mainstream products, such as Microsoft Office and Macromedia Director, there is a growing trend with smaller, more specialized applications to offer integrated scripting to their users.

- [The Benefits of Scripting](#)
- [About PythonQt](#)
- [Getting started](#)
- [Creating an Application Scripting API](#)
- [GUI Scripting](#)
- [The PythonQt Module](#)
- [Decorators and C++ Wrappers](#)
- [Other Features](#)
- [Future Directions](#)

For several years, there have been only two mainstream solutions for embedding scripting languages into commercial Qt applications: QSA (JavaScript 2.0) from Trolltech and PyQt (Python) from Riverbank Computing. The [Scripting Qt](#) article in *Qt Quarterly* issue 10 gives a good overview of QSA, PyQt and some other solutions in action.

There have been some developments since that article was written, and, as of today, there are two new script bindings to look at:

- QtScript, an ECMAScript interpreter with Qt bindings, is shipped as part of Qt 4.3.
- PythonQt, used by MeVisLab, is a dynamic script binding to the Python interpreter.

While both QtScript and PythonQt make it very easy to embed scripting into your existing Qt Application, this article will focus on the PythonQt binding, leaving an in-depth look at QtScript for a later article to cover.

#### 7.6.4.1.1 The Benefits of Scripting

Making a C++ application scriptable has several benefits:

- A well designed application interface can provide an easy access point for both novice and power users.
- Applications can be easily extended without requiring users to have deep C++ knowledge.
- Scripting makes it easy to create macros and do batch processing.
- Automated testing can be realized with scripting.
- Scripting is cross-platform, so the scripts will work on all platforms the application runs on.
- Scripting can speed up the prototyping stage a lot; e.g., your support team can add a feature/workaround by scripting much faster than it would take to develop it in C++ and redeploy the application.

Scripting APIs can range from a simple interface that allows activities such as batch processing of common application tasks to a fully-fledged interface that allows the user to customize/extend the menus and dialogs, and even to access the core functionality of the application (e.g., JavaScript in Web Browsers).

When considering scripting solutions for Qt applications, the following features are considered to be beneficial:

- Easy integration into an existing Qt application.
- Based on a well-known scripting language, so that new users do not need to learn a new language.
- Good integration with the Qt framework; e.g., it should know about signals, slots and properties.
- Support for marshalling data types between the scripting language and Qt, ideally supporting all [QVariant](#) types.
- Support for debugging---when scripts get bigger, debugging becomes a crucial feature.

#### 7.6.4.1.2 About PythonQt

Unlike PyQt and Qt Jambi, PythonQt is not designed to provide support for developers writing standalone applications. Instead, it provides facilities to embed a Python interpreter and focuses on making it easy to expose parts of the application to Python.

PythonQt makes extensive use of the Qt 4 meta-object system. Thus, PythonQt can dynamically script any [QObject](#) without any prior knowledge of it, using only the information supplied by [QMetaObject](#) (defined using Qt's `moc` tool in C++ applications). This dynamic approach allows several different script bindings to be embedded in the same application, each of which can use the same scripting API; e.g., JavaScript (QSA or QtScript) and Python.

The following sections will highlight some core features of PythonQt. For a detailed description of PythonQt's features and more sophisticated examples, visit the project's website.

### 7.6.4.1.3 Getting started

The following example shows the steps that are needed to integrate PythonQt with your Qt Application.

```
#include "PythonQt.h"
#include <QApplication>

int main(int argc, char **argv)
{
    QApplication qapp(argc, argv);

    PythonQt::init();
    PythonQtObjectPtr mainModule =
        PythonQt::self()->getMainModule();
    QVariant result = mainModule.evalScript(
        mainModule, "19*2+4", Py_eval_input);
    return 0;
}
```

We first initialize a PythonQt singleton, which in turn initializes the Python interpreter itself. We then obtain a smart pointer (PythonQtObjectPtr) to Python's `__main__` module (this is where the scripts will be run) and evaluate some Python code in this module.

The `result` variable in this case will contain 42, evaluated by Python.

### 7.6.4.1.4 Creating an Application Scripting API

The art of application scripting is to find the level of detail for the API of your application that best suits your users, developers and support staff. Basically, you invent a domain-specific language for your application that makes it easy for your users to access exactly the features they want, without needing to have a C++ compiler to hand.

A typical use case of PythonQt is to expose a single application object to Python and then let the users, developers or support staff create small scripts to change aspects of your applications via scripting.

Typically, you will create a new `QObject`-derived API class and use it as an adapter to various other classes in your application. You may also expose any existing `QObject`s of your application directly, but normally this exposes too many details to the script users, and it forces you to keep the interfaces of all exposed classes stable—otherwise your user's scripts will break or behave in unexpected ways.

Creating specialized API objects is often the preferred solution, making it easier to keep a stable interface and to document what parts of the application a script can access. PythonQt supports all `QVariant` types, so you can create rich application APIs that return simple types such as `QDateTime` and `QPixmap`, or even hierarchical `QVariantMap` objects containing arbitrary `QVariant` values.

### 7.6.4.1.5 About Python

Python (<http://www.python.org>) is an object-oriented programming language with a growing user community and a huge set of standard modules.

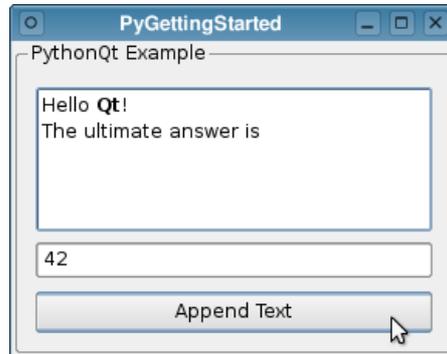
Python was designed to be “extended and embedded” easily. Libraries written in C and C++ can be wrapped for use by Python programs, and the interpreter can be embedded into applications to provide scripting services.

There are several well known applications which support Python scripting:

- Blender (<http://www.blender.org>)
- Autodesk Maya (<http://www.autodesk.com>)
- OpenOffice.org (<http://www.openoffice.org>)
- MeVisLab (<http://www.mevislab.de>)
- Scribus (<http://www.scribus.net>)
- EVE Online (<http://www.eve-online.com>)

### 7.6.4.1.6 GUI Scripting

Let's consider a simple example in which we create a small Qt user interface containing a group box, which we expose to Python under the name "box".



The C++ code to define the user interface looks like this:

```
QGroupBox *box = new QGroupBox;
QTextBrowser *browser = new QTextBrowser(box);
QLineEdit *edit = new QLineEdit(box);
QPushButton *button = new QPushButton(box);

button->setObjectName("button1");
edit->setObjectName("edit");
browser->setObjectName("browser");

mainModule.addObject("box", box);
```

Now let us create a Python script that uses PythonQt to access the GUI. Firstly, we can see how easy it is to access Qt properties and child objects:

```
# Set the title of the group box via the title property.
box.title = 'PythonQt Example'

# Set the HTML content of the QTextBrowser.
box.browser.html = 'Hello <b>Qt</b>!'

# Set the title of the button.
box.button1.text = 'Append Text'

# Set the line edit's text.
box.edit.text = '42'
```

Note that each Python string is automatically converted to a `QString` when it is assigned to a `QString` property.

Signals from C++ objects can be connected to Python functions. We define a normal function, and we connect the button's `clicked()` signal and the line edit's `returnPressed()` signal to it:

```
def appendLine():
    box.browser.append(box.edit.text)

box.button1.connect('clicked()', appendLine)
box.edit.connect('returnPressed()', appendLine)
```

The group box is shown as a top-level widget in the usual way:

```
box.show()
```

To evaluate the above script, we need to call a special function in the main module. Here, we have included the script as a file in Qt's resource system, so we specify it with the usual ":" prefix:

```
mainModule.evalFile(":/GettingStarted.py");
```

Now, whenever you press return in the line edit or click the button, the text from the line edit is appended to the text in the browser using the Python `appendLine()` function.

### 7.6.4.1.7 The PythonQt Module

Scripts often need to do more than just process data, make connections, and call functions. For example, it is usually necessary for scripts to be able to create new objects of certain types to supply to the application.

To meet this need, PythonQt contains a Python module named `PythonQt` which you can use to access constructors and static members of all known objects. This includes the `QVariant` types and the `Qt` namespace.

Here are some example uses of the `PythonQt` module:

```
from PythonQt import *  
  
# Access enum values of the Qt namespace.  
print Qt.AlignLeft  
  
# Access a static QDate method.  
print QDate.currentDate()  
  
# Construct a QSize object  
a = QSize(1,2)
```

### 7.6.4.1.8 Decorators and C++ Wrappers

A major problem that comes with the dynamic approach of PythonQt is that only slots are callable from Python. There is no way to do dynamic scripting on C++ methods because Qt's meta-object compiler (moc) does not generate run-time information for them.

PythonQt introduces the concept of the "decorator slot", which reuses the mechanism used to dynamically invoke Qt slots to support constructors, destructors, static methods and non-static methods in a very straightforward way. The basic idea is to introduce new `QObject`-derived "decorator objects" (not to be confused with Python's own decorators) whose slots follow the decorator naming convention and are used by PythonQt to make, for example, normal constructors callable.

This allows any C++ or `QObject`-derived class to be extended with additional members anywhere in the existing class hierarchy.

The following class definition shows some example decorators:

```
class PyExampleDecorators : public QObject  
{  
    Q_OBJECT  
  
    public slots:  
        QVariant new_QSize(const QPoint &p)  
            { return QSize(p.x(), p.y()); }  
  
        QPushButton *new_QPushButton(const QString &text,  
                                       QWidget *parent = 0)  
            { return new QPushButton(text, parent); }  
  
        QWidget *static_QWidget_mouseGrabber()  
            { return QWidget::mouseGrabber(); }  
};
```

```

void move(QWidget *w, const QPoint &p) { w->move(p); }
void move(QWidget *w, int x, int y) { w->move(x, y); }
};

```

After registering the above example decorator with PythonQt (via `PythonQt::addDecorators()`), PythonQt now supplies:

- A `QSize` variant constructor that takes a `QPoint`.
- A constructor for `QPushButton` that takes a string and a parent widget.
- A new static `mouseGrabber()` method for `QWidget`.
- An additional slot for `QWidget`, making `move()` callable. (`move()` is not a slot in `QWidget`.)
- An additional slot for `QWidget`, overloading the above `move()` method.

The decorator approach is very powerful, since it allows new functionality to be added anywhere in your class hierarchy, without the need to handle argument conversion manually (e.g., mapping constructor arguments from Python to Qt). Making a non-slot method available to PythonQt becomes a one-line statement which can just be a case of forwarding the call to C++.

### 7.6.4.1.9 Other Features

PythonQt also provides a number of other "advanced" features that we don't have space to cover here. Some of the most interesting are:

- Wrapping of non-`QObject`-derived C++ objects.
- Support for custom `QVariant` types.
- An interface for creating your own import replacement so that, for example, Python scripts can be signed/verified before they are executed.

The examples supplied with PythonQt cover many of the additional features we have not addressed in this article.

### 7.6.4.1.10 Future Directions

PythonQt was written to make MeVisLab scriptable, and has now reached a satisfactory level of stability. It makes it very easy to embed Python scripting support into existing Qt applications that do not require the extensive coverage of the Qt API that PyQt provides.

I would like to thank my company [MeVis Research](#), who made it possible for me to release PythonQt as an open source project on SourceForge, licensed under the GNU Lesser General Public License (LGPL). See the project's home page at [pythonqt.sourceforge.net](http://pythonqt.sourceforge.net) for more information.

I am looking for more developers to join the project. Please contact me at `florian (at) mevis.de` if you would like to contribute!

**The full source code for the examples shown in this article is available as part of the PythonQt package, available from SourceForge.**

Copyright © 2007 [Trolltech Trademarks](#)

## 7.6.5 Making ATSA Classes and Variables Available to Python

Python is not meant to be front-and-center in ATSA operation, but rather lurking in the background for those who need it. There are three ways to use it, ordered by learning curve:

- To substitute variables into tables and text blocks. The variables are re-evaluated on each refresh cycle. This provides a way to integrate analysis results.
- The python tool is a fragment of python code that performs some function. It might print some summary information to the python console, or solicit some user input before continuing analysis.
- The notebook morphs into a python program development environment. This allows users to write more complex code, such as application extensions or new analysis algorithms. This also allows for inclusion of externally

developed libraries and packages.

See the [Python ATSA Library](#).

Python can also assist greatly for development and testing, as discussed below.

### 7.6.5.1 Python for Development and Testing

During development, it became clear that I needed a way to accelerate creation of a consistent starting state. While making a series of changes, reproducing the same starting state may be required dozens or even hundreds of times in succession. So I developed a very minimal scripting language in support of this goal. Then it dawned on me that this language should be python. Seems obvious in retrospect, but it hit me like a ton of bricks. This use is a sort of “meta-ATSA,” a way of creating and manipulating notebooks, not just extending their analytical capabilities. There are good reasons for avoiding this kind of python access. The more I surface to python, the more I constrain ATSA’s growth in the name of backward compatibility. But what if the extended capabilities are *only for testing*? No immediate need for backward compatibility, and a great opportunity to dry run and refine python interfaces before making them available to users.

So I reimplemented the little scripting language in python, and extended the python libraries to contain a test directory with scripts for each of several configuration functions on the developer menu. This worked nicely, and immediately exploded into a larger issue. Surfacing ATSA’s internals isn’t a mechanical process. There are a number of design considerations:

- **Which subset of the profusion of internal classes makes sense to surface to python?**  
Currently, only a few major classes are surfaced. The mechanism used does not demand a one-to-one mapping between python classes and their C++ analogs, so a python class may access a hierarchy of C++ classes.
- **How do I make the interfaces “pythonic,” feeling natural and native to python users?**  
There are many ways in which mappings must be adapted. As a trivial example, a C++ function that returns cosmology parameters might be defined in C++ as `cosmologyParameters(double& h0, double& q0, double& lambda0)`. The “`double&`” type means “call by reference,” so the called routine can modify the three values to return them. There is no equivalent in python—parameters are passed by value (conceptually at least)—so the parameters must be returned as a dynamic array. A more interesting example of transliteration follows shortly.
- **Which objects do I name and provide automatically? Which ones should a python user obtain by asking ATSA to create one? Which ones should a python user be able to instantiate directly?**  
At the moment, very few objects are automatically surfaced as variable names in the python environment: `app` (the application object), `notebook`, and each tool (e.g. `plot1`). These objects may then be queried to obtain others.
- **Who owns all these objects, in the sense of having responsibility to delete them when they are no longer needed?**  
As a rule, ATSA-created objects are owned by ATSA (even if a python program requests their creation), and python-created objects are owned by python. But there are exceptions. `TableRanges`, for example, are created by ATSA but used only by python, so ownership of these objects is transferred to python, using `PythonQtPassOwnershipToPython<T>`. **These functions are not present in the current version of PythonQt, so these objects are presently leaked.**
- **What about enumerated types?**  
Enumerated types don’t have the type safety they do in C++, but they are still preferable to string names because misspellings are caught. The `enum` utility, created as part of this project, is used to generate code for popup menus and save/restore of enumerated types. I extended this utility to include an optional python keyword:

```
python enum TrafficLight
{
  TLRed, "Red",
  TLYellow, "Yellow",
  TLGreen, "Green"
};
```

ensures that the enumerated values are surfaced to python as `at.TLRed`, `at.TLYellow`, and `at.TLGreen`.

- **How do I document the python interface?** I cannot use Doxygen, which is solely concerned with the C++ level interface. I need to document the interface manually, and, of course, pythonically. (Note that “pythonesque” is reserved for those familiar with the Ministry of Silly Walks, or whom, perhaps, have a parrot who is pining for the fjords.) Python doesn’t declare types, yet this information is still useful, so I emulate the python library docs as

faithfully as possible. Here is one example of the transliteration problem. This block of four overloaded C++ functions looks like this in the “Doxygenated” view:

<p><b>TableRange * createRange (TableRange *range, int row, int column, int rows=1, int columns=1, int access=TATopTable)</b></p> <p>Creates a range from a table or another <b>TableRange</b>. <code>row</code> and <code>column</code> specify the origin of the range. <code>rows</code> and <code>columns</code> specify the height and width, respectively. If either is 0, the count extends to the end of the entire table. If either is -1, the count is terminated by the first blank row or column, respectively.</p>
<p><b>TableRange * createRange (TableRange *range, const QString &amp;row, int column, int rows=1, int columns=1, int access=TATopTable)</b></p> <p>Creates a range from a table or another <b>TableRange</b>. <code>row</code> and <code>column</code> specify the origin of the range. <code>row</code> is specified as a string, which matches the first label cell containing the string. <code>rows</code> and <code>columns</code> specify the height and width, respectively. If either is 0, the count extends to the end of the entire table. If either is -1, the count is terminated by the first blank row or column, respectively.</p>
<p><b>TableRange * createRange (TableRange *range, int row, const QString &amp;column, int rows=1, int columns=1, int access=TATopTable)</b></p> <p>Creates a range from a table or another <b>TableRange</b>. <code>row</code> and <code>column</code> specify the origin of the range. <code>count</code> is specified as a string, which matches the first label cell containing the string. <code>rows</code> and <code>columns</code> specify the height and width, respectively. If either is 0, the count extends to the end of the entire table. If either is -1, the count is terminated by the first blank row or column, respectively.</p>
<p><b>TableRange * createRange (TableRange *range, const QString &amp;row, const QString &amp;column, int rows=1, int columns=1, int access=TATopTable)</b></p> <p>Creates a range from a table or another <b>TableRange</b>. <code>row</code> and <code>column</code> specify the origin of the range. <code>row</code> and <code>column</code> are specified as strings, which match the first label cell containing each string. <code>rows</code> and <code>columns</code> specify the height and width, respectively. If either is 0, the count extends to the end of the entire table. If either is -1, the count is terminated by the first blank row or column, respectively.</p>

The four functions map to a single function, described quite differently, in python:

<pre>createRange(row, column, rows = 1, columns = 1, access = at.TATopTable)</pre>	<p>Creates and returns a <code>TableRange</code> whose origin at top left is (<code>row</code>, <code>column</code>), with the specified number of rows and columns. Examples:</p> <pre># A single cell at (10, 15) table.createRange(10, 15)  # A single cell whose column is the label "Energy" table.createRange(10, 'Energy', 1, 1)  # range is 5 rows by 1 column table.createRange('Alpha Centauri', 15, 5, 1)  # a block bounded by the first blank row and column table.createRange('Alpha Centauri', 'Energy', -1, -1)  # a "Distance" column selected from another range range.createRange(1, 'Distance', -1, 1)  # A cell selected from the main table instead of the uppermost subtable table.createRange(10, 10, 1, 1, at.TABaseTable)</pre> <p>If <code>row</code> or <code>column</code> is a text string, it is the first label cell that matches the string. Label cells must be specially marked, and appear with a light gray background. If the <code>rows</code> and <code>columns</code> arguments are omitted, they default to a single cell. If supplied:</p> <ul style="list-style-type: none"> <li>◦ A value of -1 extends to the first blank row or column. This provides a way to pick a block out of a larger table without knowing its size in advance. If both <code>rows</code> and <code>columns</code> are -1, the block’s row count is terminated by the first blank row all the way across, then the first blank column within that range of rows.</li> <li>◦ A value of 0 extends to the edge of the table or subtable. This is most often used</li> </ul>
------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

to specify the rest of a row.

- A value of `>0` specifies a fixed number of rows or columns.

The `access` argument:

- `TABaseTable`: the base table, i.e., cells underneath any overlaid arrays
- `TATopTable`: the uppermost table cell. If this argument is chosen (default), and the range's origin lies within a subtable, the range is also restricted to the subtable.

For efficiency reasons, a **range does not adapt to changes in table size or content**. It is calculated once at creation time. If you need to recompute the range, use `autosize()`, or create a new `TableRange`.

- **Which of three possible PythonQt mechanisms should be used to surface access to these functions?**

PythonQt uses “python decorator” classes to achieve precise control over the classes and functions that are surfaced to python. For each class to be surfaced, e.g. `MyClass`, there is a corresponding class `MyClassDecorator`. The original class has no special constraints: it may be a base class or a descendent of `QObject`, there are no argument constraints, and it may use Qt signals and slots. The decorator class must be a `QObject`, and it uses signals and slots in a clever way. Any signal or slot in this class is available to python. Each decorator class is instantiated once and registered with PythonQt; any variable of type `MyClass` is automatically recognized by python. Since the decorator class completely controls the python-visible functions, any necessary mappings may be done to achieve a python-like feel. So this mechanism is used consistently in ATSSAL.

- **How do I preserve the illusion of a monolithic application, when in fact ATSSAL consists of a lot of interacting processes?**

This is presently done in part by hiding any direct access to `XSpec`. But each notebook runs as a separate process, under a controlling supervisor. The supervisor process does not support python. The new test framework extends the supervisor so it can run a test suite, a list of python programs. It creates a new notebook for each test in the suite and runs them sequentially. Thus notebooks remain partitioned from each other, known only to the supervisor.

- **How do I update the user interface in response to changes made by python programs?**

It isn't efficient to update after every function call. So each tool and the notebook as a whole offer an `updateUI()` function which brings the user interface in sync with python changes.

- **How do I experiment with new python extensions without making them available to users?**

At the moment, experimental functions are marked . But they may be omitted from user documentation, or even disabled entirely, except in developer mode.

- **Can these extensions assist with automated testing?**

These extensions add an important new level of support for automated testing. Lower level tests, such as unit tests and round-tripping, help to validate the data classes, but the new extensions also exercise the user interface. Normally users manipulate the user interface to cause state changes in the data, but these functions do the converse: change the data and expect the user interface to reflect the change. This tests a lot of user interface code, not just data classes. (There is a higher level yet, the “player piano” style of testing. This mimics actual user actions and verifies that the user interface is pixel-for-pixel correct. But this level of testing relies on an expensive third party tool, *Squish*, not yet available to us.) See [ATSSAL Testing](#).

- **What about tutorial production?**

Since PythonQt supports full access to the Qt class library, these new extensions could be used to create interactive, step-by-step tutorials. Probably not justifiable in terms of level of effort required, but an interesting idea.

- **Do these extensions have any implications for large-scale parallelism?**

They sure do! Imagine for a moment that ATSSAL supports arbitrary levels of parallelism, shipping out analysis tasks to remote computers as needed. These python extensions make it possible to write a program in a few lines of code that applies the same analysis to a thousand observation files, all at once. One could also imagine development of programs that apply many different analytical techniques to the same observation, all in parallel.

## 7.6.5.2 Registering Classes and Variables

Register classes at global scope within Python via:

```
gApp->py()->registerClass(...)
```

You only have to register classes that you allow Python users to construct. Otherwise registering an instance of the class implicitly registers the class.

Register variables at global scope within Python via:

```
gApp->py()->registerVariable(...)
```

For example:

```
ToolTable* toolTable = new ToolTable(this, "table1", args...);  
gApp->py()->registerVariable(toolTable);
```

The `PythonPublic` class contains methods to manage variables, including:

- `renameVariable()`, which renames a variable by unregistering and re-registering it.
- `unregisterVariable()`, which unregisters a variable prior to deletion or perhaps cutting and pasting.
- `registerVariable()`, which registers a variable under a supplied name. This triggers an assertion if the variable is already registered.

## 7.7 XSPEC Allocator

In the short term, XSPEC instances are allocated on the user's machine, typically up to the number of available cores, or to a user-settable threshold. Longer term, the allocator will also allocate XSPEC instances on remote computers. Since observation files are fairly bulky, and they must be transferred to other machines, this is a different type of optimization problem, considered briefly later on this page.

An XSPEC instance is simply a unique process running the XSPEC server. It is described by an `xProcessInfo`, or `xpi`. An `xpi` consists of a host computer, an XSPEC process ID, an owning notebook process ID, a unique sequence number, and the model last computed by the instance—just enough information to pass the serialized class between the supervisor, or a remote XSPEC server, and notebooks. An `xpi` represents an XSPEC server compute resource, but not necessarily a specific process. If a notebook finishes using an `xpi`, it returns the `xpi` to the supervisor for reallocation. If a different notebook receives the `xpi` next, the XSPEC process is shut down and a new one started in its place. On the other hand, if an `xpi` is reallocated to the same notebook, it is not restarted, but simply reused.

Class `XSPECAllocator` is a singleton that operates in the supervisor. When a notebook needs an `xpi`, it requests one from the supervisor. The allocator checks its list of free `xpis` and returns a free one, if any; otherwise the request goes on a pending `xpi` request queue. The requesting process receives no indication if an `xpi` is not available, since it will proceed on that assumption anyway. When a notebook finishes with an `xpi`, it tells the supervisor, which adds the `xpi` to the free list or hands it out to the next requesting notebook. Thus `xpis` are global to the ATSA process, rather than managed on a per-notebook basis. XSPEC instances can therefore be transferred from notebook to notebook as needed.

When a notebook executes a spectrum plot tool, it requests as many `xpis` as the tool uses. A single plot might specify three models, for example. Then it simply suspends refresh of the notebook. When the supervisor makes a new `xpi` available, ATSA starts it running a model. ATSA does not move onto the next tool unless all the models have completed execution. If resources are constrained, the models will be processed in succession, but if there are enough `xpis`, all of them will execute in parallel. As each execution completes, the notebook returns the `xpi` to the supervisor's free list.

When an `xpi` is allocated out to a different notebook than the last one to use it, the last notebook is notified, and it shuts down the previous process. One reason for this is that passing an XSPEC session that is *in progress* between notebooks seems risky at best.

The allocator uses a couple of simple optimizations to improve throughput. First, whenever possible, it returns an `xpi` to the same process that last used it. This avoids the need to shut down the XSPEC process and start a new one. Second, the allocator knows what model was last computed. If the model is a simple model, there is little speed advantage to reusing it to calculate a new model or a changed parameter. But if the model is a compound model, XSPEC keeps enough partially computed results around so that a changed parameter will produce a new fit much more quickly than starting from scratch. So the allocator will reassign a compound model to the same process if the same model is in use,

when possible.

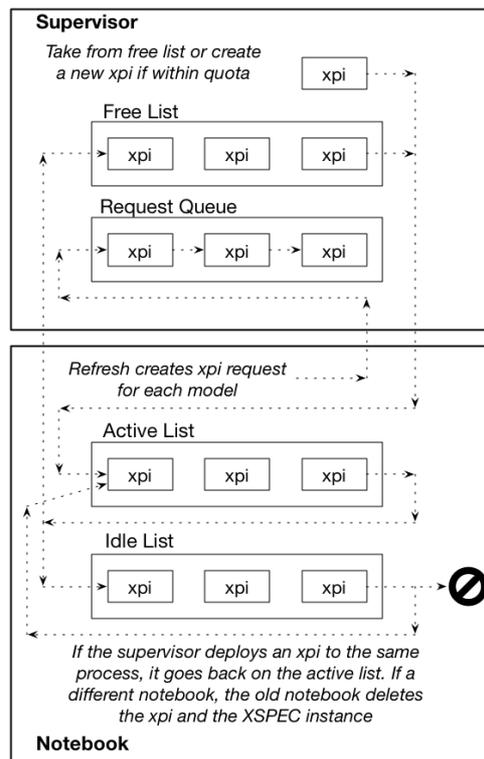
Xpis are allocated on a first-come first-served basis. If the system-wide quota is exhausted, a notebook refresh locks up until resources are available. Such a hung refresh can be aborted easily by the user if desired. Computations in progress can also be aborted.

Each model has its own XSPEC proxy and logfile "channel." The XSPEC proxy may in turn connect to multiple XSPEC instances over time, but the proxy makes it look like a single instance. The user can view the combined progress of all XSPECs associated with a notebook. Busy "LEDs" beside each model indicate active XSPECs for each model. Clicking on a busy indicator displays the log for this XSPEC alone.

### 7.7.1 Pictorial Summary

When the user performs a refresh, and the refresh cycle reaches the spectrum plot tool, ATSAL creates a request xpi for each model in the current tool. The requests are passed to the supervisor, where they are entered on the request queue. Notebook execution is suspended, awaiting assignment of xpis. At regular intervals, the supervisor takes entries from this queue, and satisfies them using either a free list xpi or, quota permitting, a new xpi. The request queue entry is discarded, and the xpi that fulfills the request is passed to the notebook, where it enters the active list. The notebook instantiates the process if it is not already running, and sets it to work performing the fit. When a fit completes, the xpi is moved to the idle list, and also sent back to the supervisor, where it is added to the free list. This cycling process allows the xpi to be allocated to a different notebook when necessary, but whenever possible, keeps the same XSPEC instance associated with the same model.

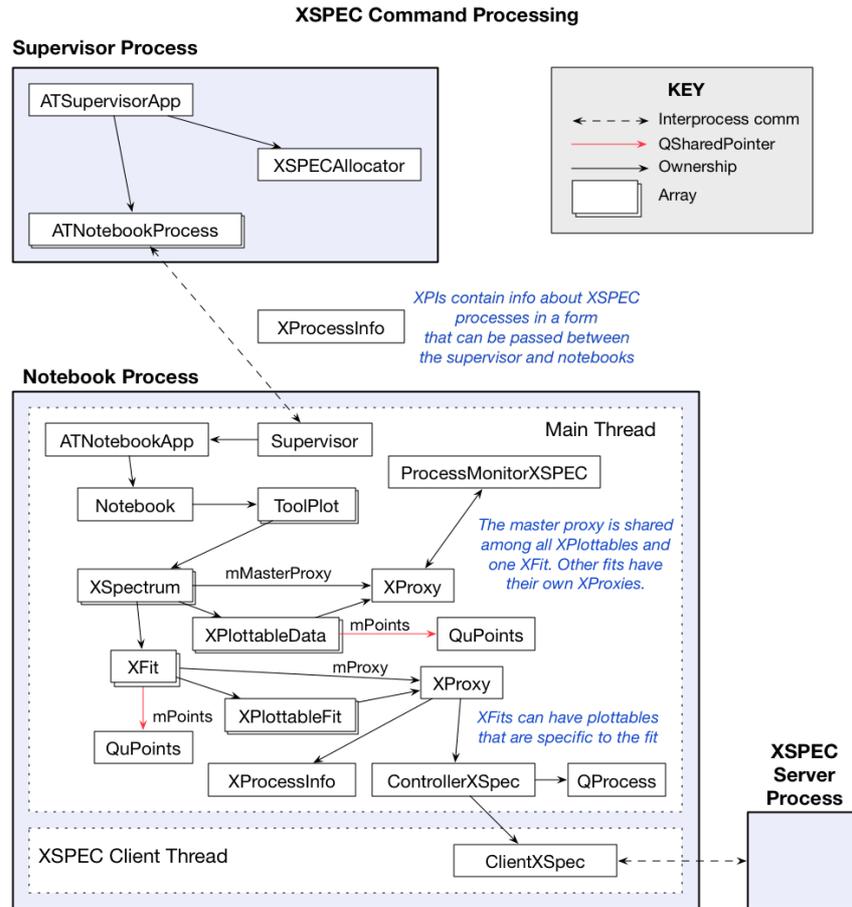
**Update:** in the notebook process, xpis are not placed on queues; they remain with their owning XProxies. Their state changes instead. The diagram below is still conceptually correct.



### 7.7.2 XProxies

Whenever possible, the same XSPEC instance is used to recompute a fit as changes are made to a model, but this is not always possible if demand for processes is in excess of the quota. For this reason, the actual XSPEC instance associated with a model can change. The XProxy class acts as a proxy between the analysis (XFit) and the actual XSPEC instance, accessed via ControllerXSPEC. When a new process is assigned by the supervisor, the XProxy instance is connected to the new process, so that I/O is redirected. This makes the log of activity for each XFit appear to represent an interaction with a single XSPEC process. The XFit is notified of a change in process, reissuing any startup commands when this happens.

One consequence of this organization is that in the event of an XSPEC crash, a new process is assigned and the analysis picks up again, with only a time delay. All user settings are retained.



### 7.7.3 XSPEC Versions

Each XSPEC process is started with a parser matched to the XSPEC version. ATSal enforces a minimum version of XSPEC and will not run XSPEC on a remote server that does not support at least this version. However, it is possible for a single ATSal session to communicate with XSPEC servers of more than one version. Hypothetically at least...

### 7.7.4 Projections

A preferences setting determines whether projections are to run automatically after each fit. This sets a default which may be overridden on a per-model-instance basis. When XSPEC finishes computing a fit, the tool displays results immediately. The projection runs asynchronously. The user interface remains locked while projections are running, but there is a clear status indication so that projections are easily aborted if no longer desired.

### 7.7.5 User Interface Locking

During a refresh cycle, most of a notebook's user interface locks up, because changes to the notebook could have unexpected results. A "locked" notebook user interface means that all commands that alter the notebook and its tools are prohibited. However, some view-only commands work, such as opening and closing tool editors, opening new notebooks, scrolling, viewing different logs, etc. Because new notebooks can be created or cloned, users can continue to set up new problems while existing ones are in progress.

### 7.7.6 Running XSPEC on Multiple Computers

In the future, the allocator will be extended to permit use of XSPEC running on remote systems. Since the remote computer must deal with access by multiple ATSal programs, it will manage the assignment of resources to each requesting ATSal. ATSal's local allocator always starts with the local computer in search of xpis, then makes requests

of remote resources. Each remote system has its own user-assigned maximum quota which may be further limited by the remote resource. Remote xpis are requested in the order listed. When such an xpi is made available, the supervisor instructs the requesting notebook as to how to connect to it.

Files required by the remote computer are transferred from the user's local system prior to calculating a model. A per-user disk quota determines how long the files are buffered. Least recently used files are discarded first. Because file transfer times influence throughput, the allocator tries to reassign xpis to the same notebook, or to a notebook using the same input files.

### 7.7.7 Orphaned Notebooks

Debugging multiple processes simultaneously is inconvenient at best. Although it is possible to run multiple debugger sessions and connect to other processes to assist with debugging, the other processes must be instantiated first, and then a debugger connected to them. This entirely misses initialization.

**Update: this isn't true if there is only one process to be run with a given name. You can connect to it from the XCode debugger in advance. But since the connection is based on process name, I assume that this doesn't work with multiple identically named processes.**

To minimize this problem, an ATSal notebook can run without a host supervisor to allocate XSPEC instances. This mode is designed solely for debugging. It creates a simplified XSPEC allocator that runs within the notebook, implicitly assuming that only a single notebook is running, so there is no need for load balancing.

At startup, a notebook assumes it is an orphan (`ATNotebookApp::isOrphan()` returns `true`). Each time a message is received from the supervisor, this flag is set to `false`. When a request is made for an `XProcessInfo`, it is serviced by the notebook itself instead of the supervisor in orphan mode. The logic to service these local requests is much simpler, since the allocator need only assign new processes until a quota is reached.

To run in orphan mode, the developer simply runs the notebook application directly, rather than starting the supervisor.

### 7.7.8 Abolishing the Load-leveler

The load-leveling I am addressing here is that which allocates XSPEC instances on the user's own computer, not larger-scale parallel operation.

I still have a tendency to view XSPEC as a large resource consumer and personal computers as small, limited resource providers. Of course this makes sense given the history of computing. In the past I have done a considerable amount of performance measurement and program optimization in order to allow large programs to run more efficiently on much more expensive computers. I have also written some load-leveling programs.

But of course times have changed, and XSPEC is no ordinary large program. XSPEC's memory demand is modest compared to the amount of physical memory on today's computers—about 40-100 MB. As I get more of a handle on the XSPEC's operation, I can see that the I/O is modest too: a flurry of reads at startup, or when a new input file is selected, and that's about it. I have written a fairly elaborate load-leveling algorithm for ATSal. The supervisor allocates XSPEC instances out to various notebooks, or moves them among notebooks (by killing a process for one notebook and creating a new one for another), until a quota is reached. I've been thinking that this quota is roughly equal to the number of cores.

But what if it is unlimited? What if there isn't *any* load-leveling? XSPEC is mostly compute-bound, so on an 8-core machine, 8 1-minute fit computations should complete in a minute. And 16 should complete in about two minutes. In other words, the performance degradation is linear, not exponential as it would be if memory were exhausted and "thrashing" commenced. Two minutes is also the *best case* time for the load-leveler. The reason this is an upper bound is because the process-sharing algorithm will often be forced to move an XSPEC from one fit to another, requiring input files to be read in again, or a new XSPEC session to be started.

So for the special case of a mostly compute-bound task, ample memory, and high bandwidth SSD-based I/O, a load-leveling algorithm probably won't improve over simply permanently allocating an XSPEC server to each model, even if 100 such instances are in existence.

There is another reason for this: the odds are that only a few XSPECs will be active at once, because of the way the user thinks while performing analyses. If there are a lot of fits, most of them were computed previously and are now quiescent, really just containers for results.

So it may turn out that I don't need the fancy and rather finicky load-leveler at all. This doesn't make any difference to the end user, but it would make the ATSSAL code less complex and easier to maintain. I am going to proceed for the time being with the quota set to unlimited and see how it goes. If the strategy works, I will back out the load leveler later.

### 7.7.8.1 Addendum 2017-04-25

On an 8-core machine, I get these results:

<i>XSPEC Processes</i>	<i>Total Elapsed Time</i>
2	00:32
21	00:58
41	02:10

Even with the 41-process case, memory impact is negligible, and degradation is very reasonable. So ATSSAL no longer employs load leveling. It simply allocates as many XSPECs as the user requests.

Future versions of ATSSAL will probably support the ability to spill over to remote computers for very large numbers of XSPECs. Users will be able to tune the offloading process between cheaper, more responsive local computes and more expensive, scalable remote computes.

## 7.8 Client/Server Architecture

`xspecserver` is an extension of XSPEC designed to accept commands and return results over a socket connection. This allows one or more instantiations of XSPEC, running on the local or remote systems, to be used by ATSSAL or by other programs, with a general goal of more efficient use of compute resources.

At this stage, server activation and use is a manual process, but the long term goal is to make this process automatic, allowing a user to configure secure access to one or more copies of the server running on a given machine. The server uses an XML protocol for executing remote procedure calls (RPC). XML is chosen to make it easier to debug client/server transactions and to avoid problems in passing binary data between dissimilar computer architectures. This protocol has an escape for binary data blobs, encoding them in base 64. Large data files are transferred separately, so the relative inefficiency of XML is avoided.

We use a mature open source package called XMLRPC to implement this communication. The XMLRPC source tree includes variants for several types of interconnect. We use "pstream protocol," an informal protocol defined by the XMLRPC project, for the specific reason that it establishes a session—a long-lived connection—rather than establishing a new connection for each transaction. This is more natural and efficient for this application, since `xspecserver` can address only one client at a time, and XSPEC is inherently session-oriented.

XSPEC has an internal C++ API that is brought out to its built-in Python interpreter. Initially we considered replicating this part of the API on the client, but this approach was abandoned because of likely severe performance problems. Instead, we adopted a more structured version of the command line itself, together with a series of proxy classes.

During the exploratory phase of ATSSAL development, the goal is to prove the viability of the client/server architecture, rather than to refine it by resolving all the security issues. But we start here with the long term goals before returning to the short term goals.

### 7.8.1 Long Term Architecture

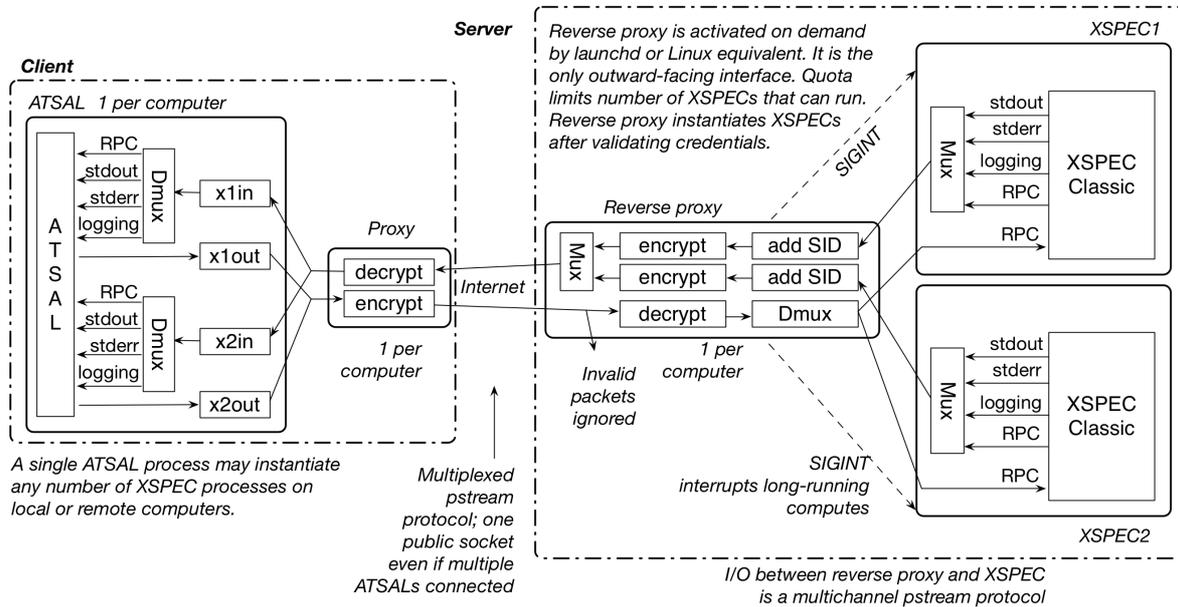
A user will be able to designate any supported system (initially Mac OS X and Linux) as a server, by installing a server package and supplying some simple parameters, including:

- Usernames and passwords of users authorized to access the server
- A quota controlling the total number of XSPEC instantiations permissible
- A quota restricting the total that may be used by an individual user
- A port, in the event that the default port is in use

At boot time, `launchd`, or its equivalent on Linux systems, will launch an XSPEC listener daemon that also acts as a reverse proxy. There is one listener per system, and its purpose is to receive and validate connection requests, instantiate XSPEC servers, and manage their shutdown or restart. The listener also processes all network-facing traffic, acting as a first line of defense in order to reject invalid packets before passing them to XSPEC.

On each client, a similar proxy process isolates ATSA L from the network connection. Since there is a maximum of one ATSA L process and one proxy per system, ATSA L itself instantiates the proxy when it starts up.

Encryption of traffic is not very important for an application of this sort, except as a hedge against attempts to use the server as an attack vector. Shown below as part of the proxies, it will instead be implemented by optional use of tunneling through ssh.



XSPEC performs many long-running fits that the user may elect to interrupt. This is implemented by sending SIGINT to the XSPEC process. The interprocess communication link shown here handles basic RPC—commands to XSPEC and responses. This traffic level is usually quite low, at the most, arrays of moderate size. Large files such as response matrix files are not sent through this connection, but are sent separately via HTTP.

In addition to the pure RPC connection, stdout, stderr, and logging outputs are shown. Data sent to stdout is parsed to determine command completion status. Data sent to the other channels is available in optional panes for monitoring or debugging.

ATSA L's preference settings will include a list of candidate servers, accessed in the order listed as XSPEC instantiations are needed.

See this article on [creating Unix daemons](#).

## 7.8.2 Short Term Architecture

During the prototyping phase, an XSPEC server is instantiated manually, using a public domain utility called `socketexec` to create a socket on a port and pass it via stdin to XSPEC server. ATSA L implements the client side of the connection by opening a socket, using a preference setting for the port. Client and server talk directly rather than via proxy. For debugging and development, they typically run on the same machine.

## 7.8.3 Socket Connection Refused Errors

ATSA L instantiates an XSPEC server process corresponding to each fit requested in a notebook, so all fits can proceed in parallel. A thread in the notebook process communicates via socket with each XSPEC. Normally this runs reliably. But if a hierarchy of processes are terminated, and ATSA L is rerun, it receives a socket connection refused error (ECONNREFUSED). The new run attempts to reuse the same port(s) as the preceding run. Once a connection refused error occurs, retries once per second, up to a 30 s timeout, always fail.

Chris Kent tells me I can use `SO_REUSEPORT`, issued on the listener (XSPEC) side, to circumvent this problem. But Qt's abstraction for TCP sockets made it unclear as to how to achieve this setting (I was looking in the wrong place), so I examined the source code. I found that the enum `QNativeSocketEngine::AddressReusable` activates this setting,

and noticed this:

```
    case QNativeSocketEngine::AddressReusable:
#ifdef SO_REUSEPORT
    // on OS X, SO_REUSEADDR isn't sufficient to allow multiple binds to the
    // same port (which is useful for multicast UDP). SO_REUSEPORT is, but
    // we most definitely do not want to use this for TCP. See QTBUG-6305.
    if (socketType == QAbstractSocket::UdpSocket)
        n = SO_REUSEPORT;
    else
        n = SO_REUSEADDR;
#else
    n = SO_REUSEADDR;
#endif
    break;
```

Following is the bug report referenced above:

### 7.8.3.1 QTBUG-6305

#### **QTcpServer::listen() returns true when someone is already listening on that same port - Mac OS X**

If I start the server two times, both servers report they have successfully bound the ports. When I send requests to the ports the following happens:

Requests to port 1501 are processed by the first instance

Requests to port 1502 are processed by the second instance

If I stop the second instance, requests to port 1502 are also processed by the first instance.

#### **7.8.3.1.1 02/Dec/09 4:39 PM**

Peter Hartmann (closed Nokia identity) (Inactive) added a comment - on Linux, when running the second instance, it cannot bind, as it should be. On Mac OS X, I observe the same behaviour as reported.

#### **7.8.3.1.2 15/Apr/10 8:52 PM - edited**

Jay Barra added a comment - I am observing this as well on Mac OS X Mac OS X 10.6.2 QMake - 4.6.2 GCC - 4. 2.1

```
// START EXAMPLE
// main.cpp
#include
#include
#include
#include
int main(int argc, char* argv[])
{
    QCoreApplication app(argc, argv);
    QTcpServer* a = new QTcpServer;
    QTcpServer* b = new QTcpServer;
    qDebug() << a->listen(QHostAddress::Any, 9000);
    qDebug() << a->listen(QHostAddress::Any, 9000);
    // Timestamp added to provide something verification program is still running
    while (true)
    { qDebug() << "Server A:" << a->isListening() << "\tServer B:" << b->isListening() << "\t" <<
      QTime::currentTime(); app.processEvents(); }
    return app.exec();
}
// END EXAMPLE
```

On the Mac, both servers will start and return true that they are listening, on Windows and Linux the server B will not start and will not listen. If I attempt to connect to the either server with a QTcpSocket, the second server started will

accept the connection.

### 7.8.3.1.3 04/Jun/11 5:36 AM

Nam T. Nguyen added a comment - Qt treats `SO_REUSEADDR` and `SO_REUSEPORT` the same.

```
    case QNativeSocketEngine::AddressReusable:  
#if defined(SO_REUSEPORT) && !defined(Q_OS_SYMBIAN)  
    n = SO_REUSEPORT;  
#else  
    n = SO_REUSEADDR;  
#endif
```

(This is prior to the bug fix shown at the top.)

They might be the same on some OSes, but might also be differing on others. Mac OS X, for instance, treats them differently,

[http://developer.apple.com/library/ios/#documentation/system/conceptual/manpages\\_iphoneos/man2/getsockopt.2](http://developer.apple.com/library/ios/#documentation/system/conceptual/manpages_iphoneos/man2/getsockopt.2).

```
///apple\_ref/doc/man/2/getsockopt  
SO_REUSEADDR    enables local address reuse  
SO_REUSEPORT    enables duplicate address and port bindings
```

So, in `qtcpserver.cpp`, before the `bind()`, Qt calls

```
d->socketEngine->setOption(QAbstractSocketEngine::AddressReusable, 1);
```

and this sets `SO_REUSEPORT` on. The best fix is for Qt to treat them differently, as they are meant to be. If duplication on port binding is indeed required (which is quite rare), let it be handled specifically by the application developers. Fair?

### 7.8.3.2 What exactly does `SO_REUSEADDR` do?

(This is from another source and may not apply to MacOS.)

This socket option tells the kernel that even if this port is busy (in the `TIME_WAIT` state), go ahead and reuse it anyway. If it is busy, but with another state, you will still get an address already in use error. It is useful if your server has been shut down, and then restarted right away while sockets are still active on its port. You should be aware that if any unexpected data comes in, it may confuse your server, but while this is possible, it is not likely.

It has been pointed out that "A socket is a 5 tuple (proto, local addr, local port, remote addr, remote port). `SO_REUSEADDR` just says that you can reuse local addresses. The 5 tuple still must be unique!" by Michael Hunter ([mphunter@qnx.com](mailto:mphunter@qnx.com)). This is true, and this is why it is very unlikely that unexpected data will ever be seen by your server. The danger is that such a 5 tuple is still floating around on the net, and while it is bouncing around, a new connection from the same client, on the same system, happens to get the same remote port. This is explained by Richard Stevens in "2.7 Please explain the `TIME_WAIT` state."

### 7.8.3.3 ATSA Solution

At the moment, the XSpec code now includes this fragment between socket instantiation and a listen.

```
// THK, 2017-04-22: Add SO_REUSEADDR option to allow reconnects to the port  
// after process terminaton. This is Mac-specific at the moment.  
int value = 1;  
if (setsockopt(listenSocket, SOL_SOCKET, SO_REUSEADDR, (char *) &value, sizeof(value)) != 0)  
{  
    srvLogErr << "SO_REUSEADDR assign failed: " << strerror(errno) << std::endl;  
    return(-1);  
}
```

### 7.8.3.4 Additional Comments

Additional comments from Chris Kent:

Sample code below. It's also common to set `SO_NOSIGPIPE`:

```
long on = 1;
int rc;

/* Prevent SIGPIPE signals- return EPIPE instead */
rc = setsockopt(listenSocket, SOL_SOCKET, SO_NOSIGPIPE, &on, sizeof(on));
if(rc == -1) return nil;

/* Since we have a fixed port option, need the reuse port flag too */
rc = setsockopt(listenSocket, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
if(rc == -1) return NO;
```

When debugging any process that makes use of sockets using XCode, you will inevitably run into un-maskable `SIGPIPE` errors that drop you into the debugger. The act of debugging makes the signals unmask-able, even if you've explicitly ignored them using `sigmask` and/or `NO_SIGPIPE` on each individual socket. Just a heads up... there's nothing wrong with your code.

Also, those processes should be exiting automatically when the parent exits. To force the issue you can have the children watch the parent and exit when the parent does:

```
// Exit when the parent exits
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC, getpid(),
DISPATCH_PROC_EXIT, queue);
if (source) {
    dispatch_source_set_event_handler(source, ^{
        // The parent exited
        exit(1);
    });
    dispatch_resume(source);
}
```

## 7.9 Observatory Tool

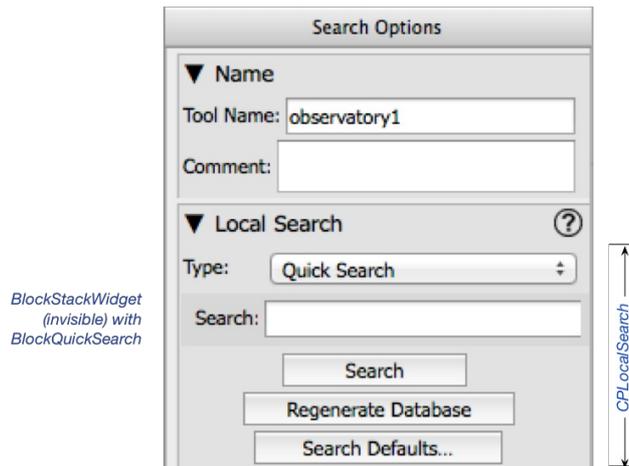
In its present implementation, the observatory tool displays observations previously downloaded to the local machine. [A long term goal is to apply searches directly to remote observatory web sites.](#)

### 7.9.1 Search Queries

The observatory tool offers two search modes. Quick Search mode, the default, is targeted at relatively small collections of observations. Detailed search mode offers additional search constraints needed to manage large collections.

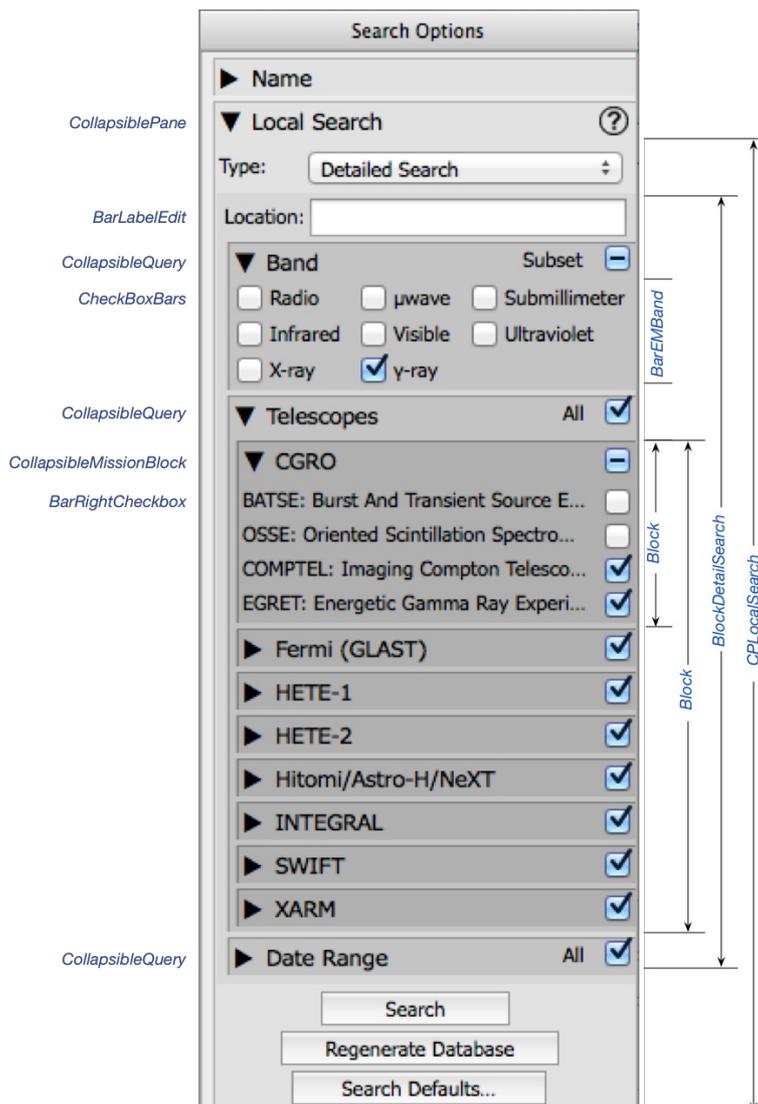
#### 7.9.1.1 Quick Search

In this mode, there is a single search box. If the box is empty, the entire collection is displayed. If the box contains a search string, a file matches if its filename, telescope, instrument, detector, astronomical object, or parent directory contains the string. The match is case insensitive and matches substrings as well. "Astro" matches "ASTRO-H" and "Astro-1".



### 7.9.1.2 Detailed Search

In this screen shot, the Band and Telescopes subpanels are open. If the user enters a location, a case-insensitive match against the FITS OBJECT field is performed. **“Location” is used rather than “Object” in anticipation of allowing coordinates to be entered here as well.** As the user selects different bands of interest, the missions and instruments list reconfigures to show only instruments that operate in the selected bands.

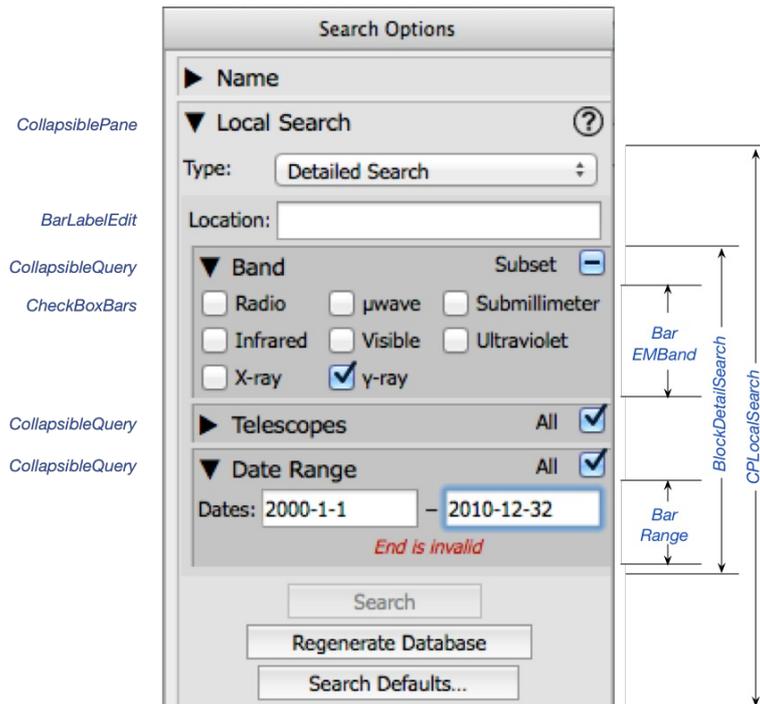


In the Band subpanel, the checkbox on the right cycles between subset (partially clicked), and on. (Off doesn't make sense here, since that would always produce an empty search.) In the on state, all the bands are searched and the individual band checkboxes are disabled. In the subset state, the band checkboxes are enabled.

The Telescopes subpanel works the same way. The toplevel checkbox cycles between partial and on. In the partial state, all the individual mission/instrument settings are disabled.

At the Mission level, the checkbox behaves differently. We could cycle between all, subset, or none. But because there are so few instruments per mission, we cycle between all and none, and a click on the mission checkbox actually sets or clears all the subordinate instrument checkboxes.

Searches may also be constrained to a range of dates:



Dates must be entered in ISO date format (i.e. YYYY-MM-DD). Syntax errors are reported underneath the range, and the search button is disabled if there is an error. A blank date means the search is unconstrained.

This was more complex to implement than anticipated. The observation date is taken from the "DATE OBS" FITS field. It may be entirely absent, or present but empty. When this field is present, its format is inconsistent. Currently I convert any of MM/DD/YY, MM/DD/YYYY, YYYY/MM/DD, YYYY-MM-DD, YYYY-MM-DDTHH:MM:SS.FFFF, or YYYY-MM-DD HH:MM:SS.FFFF into YYYY-MM-DD HH.MM.SS.FFFF, where FFFF is fractional seconds. The supported date range is from 1000-01-01 to 9999-12-31, a limit imposed by MySQL. If you happen to have observations of the Crab Nebula from AD 1054, you're in luck.

A date-constrained search matches any FITS file with an unspecified date. This is necessary because many of the supporting files, such as RMF files, lack observation dates.

## 7.9.2 Missions, Instruments, and Proxies

Searches for observations can be restricted to particular missions and instruments. In support of this is a simple database, stored in Missions.xml, that contains a few properties for each mission. A `Mission` contains names and a list of the bands covered by it. It also contains a list of `Instruments`, each of which includes a name and EM range.

Each `Mission` has several names:

- An ID. This string must be unique among all missions, invariant, and case sensitive. It is used like a UUID, as a tag to reconstruct pointers from a saved file. Users do not normally see this string, it is just for pointer reconstruction.
- A name. This string is a short name or names. Examples are "Planck," "Spitzer (SIRTF)," or "ORFEUS-SPAS II."

There are no special format conventions other than keeping the name relatively short. This is one of the two names displayed by the user interface.

- A full name. This is usually an expanded acronym, e.g. “Orbiting Retrievable Far and Extreme Ultraviolet Spectrometer.” Again, there are no special formatting conventions, and this is displayed by the user interface.
- FITS keys. This is a list that usually contains only a single element. It contains the mission name as it appears in a FITS file, and it is the name that is actually searched for. If the `fitskey` option is not supplied, it defaults to the ID. If there is more than one name in this list, the names are separated by commas.

A `Missions` contains a list of all the missions, loaded at startup from resource `Missions.xml`. This list cannot be modified by the user.

The user interface requires objects of class `MissionProxy` and `InstrumentProxy`, which tack on some additional properties. The `search` Boolean determines whether a mission or instrument is searched for, and the `open` Boolean in `MissionProxy` determines whether the instrument list is displayed for a given mission.

A `MissionProxies` is a list of all the `MissionProxys` for a given use. There are several `MissionProxies`:

- A “factory default” list of settings is loaded at startup from resource `MissionProxies.xml`.
- A user default list, not yet implemented, adjusted via Preferences, will be added at some point. This will allow users to configure a default search list to constrain searches to missions of interest.
- Each observatory tool has its own `MissionProxies` so that each tool’s search options can be configured independently. Each of these instances are saved as part of the notebook.

### 7.9.2.1 SearchQuery

A `SearchQuery` contains a list of search parameters: a `MissionProxies`, `EMBands`, a date range, and some other options.

### 7.9.2.2 Defaults

There are two levels of defaults for the `SearchQuery`. “Factory defaults” are loaded from the resource file `FactorySearchQuery.xml` at startup. User defaults are stored in `UserSearchQuery.xml` in the user’s `~/ATFiles` directory. When ATSAAL starts running, it first loads the factory defaults, then overwrites them with user defaults, if any. This approach means that new missions included with newer ATSAAL releases will find their way into the user defaults as well. During a run, any current search query may be saved as new user defaults. The current user defaults are saved as soon as the user updates them, so they will be available to other notebooks. The user can also revert an observatory tool to either user or factory defaults. These operations are performed in the observatory tool, using Search Defaults....

## 7.10 Results Tool

A results tool is a readonly view of the results of any fit:

Untitled-294.cnbk: results1

	0	1	2	3	4	5
0	<b>Results for plot1, Spectrum 0, Model 0</b>					
1	/Users/tkent/ATFiles/Obs/Observations/W49b-sxs.pha					
2						
3	Parameters					
4	$\chi^2$	beta /N	Level	1:nH	2:PhoIndex	3:norm
5	3.3201×10 <sup>5</sup>	19837	-1	1.4495	1.4271	0.81813
6	3.0202×10 <sup>5</sup>	21205	-1	2.0618	1.8587	1.9052
7	2.577×10 <sup>5</sup>	24064	-1	2.5262	2.1093	4.0671
8	2.3551×10 <sup>5</sup>	18498	-1	2.7881	2.2445	6.3732
9	2.3308×10 <sup>5</sup>	7787.2	-2	3.5945	2.6425	11.739
10	2.2212×10 <sup>5</sup>	19214	-3	4.014	2.7281	16.881
11	2.2149×10 <sup>5</sup>	5472.9	-4	4.1278	2.7439	18.361
12	2.2148×10 <sup>5</sup>	380.07	-5	4.1781	2.7531	18.78
13	2.2147×10 <sup>5</sup>	86.231	-6	4.199	2.7563	18.925
14	2.2147×10 <sup>5</sup>	26.337	-7	4.2087	2.7579	18.996
15	2.2147×10 <sup>5</sup>	12.084	-8	4.213	2.7586	19.026
16	2.2147×10 <sup>5</sup>	4.9436	-9	4.2149	2.7589	19.04
17	2.2147×10 <sup>5</sup>	2.2458	-10	4.2157	2.759	19.046
18						
19	Variances and principal axes					
20		1	2	3		
21	1.8326×10 <sup>-6</sup>	-0.0728	-0.9968	0.0345		
22	8.9562×10 <sup>-5</sup>	0.9942	-0.0752	-0.0764		
23	0.043924	-0.0788	-0.0287	-0.9965		
24						
25	Covariance matrix					
26		1	2	3		
26		0.0003612	9.278×10 <sup>-5</sup>	0.003442		
27		9.278×10 <sup>-5</sup>	3.852×10 <sup>-5</sup>	0.001257		
28		0.003442	0.001257	0.04362		
29						
31	Model phabs<1>*powerlaw<2>					
32	Model par	Model comp	Component	Parameter	Unit	Value
33	1	1	phabs	nH	10 <sup>^22</sup>	4.2157
34	2	2	powerlaw	PhoIndex		2.759
35	3	2	powerlaw	norm		19.046
36						
37	Summary	$\chi^2$	PHA bins	Degrees of free...		
38	Fit statistic	2.2147×10 <sup>5</sup>	16384			
39	Test statistic	2.2147×10 <sup>5</sup>	16384			
40	Reduced $\chi^2$	13.52		16381		
41	Null hypothesis	0				

Results Options

▼ Name

Tool Name: results1

Comment:

▼ Fit Results Settings

Plot: plot1

Spectrum: Spectrum 0

Model: Model 0

Export table...

Users may select any fit for display, and insert multiple results tools into the notebook to display multiple results at once. Results may be exported in several spreadsheet exchange formats, or as HTML.

### 7.10.1 Results Processing

During the refresh cycle, ATXSAL prepares a series of xcmds that instruct XSPEC to carry out the operations requested by the user. As each xcmd completes, its output is parsed and values extracted for later use. At the end of a chain of related commands, each xcmd has its own set of results.

When the last command completes, `commandSequenceCompleted` is emitted. This is handled by the object that issued the commands, for example, an `xFit` or `xPlottable`. The issuing object cherry-picks through the data, extracting everything of potential value and rearranging it into more logical form. Then, having copied what it needs, it discards the rest. And it forwards the `commandSequenceCompleted` signal to the notebook.

The notebook keeps a cache of results tables for fits. The tables are used primarily by results tools, but they are stored with the notebook because they may also be needed by python programs, and because this allows tables to be shared among multiple results tools. When the completion signal arrives, if there is a table for the fit in the cache, it is updated to contain the revised results.

Next, the notebook checks to see if any results tools are present, and, if so, passes the completion signal to them. Each results tool may have table widgets that display the updated table. If so, the results tool updates the widgets in the notebook pane and in the results tool editor.

## 7.10.2 Python Access

One way to access results (the *only way at the time of this writing*), is to access the table cells in a results tool. This is described in more detail in *Parsers, Datasets, Tables, and Python: A Refactoring*.

## 7.10.3 Multiple TableWidgets

The current implementation of results tools keeps a table widget for each set of fit results the user displays. The widgets are kept in a `ResultsStack`. The original reason for keeping multiple table widgets was because tables were accessed via a results tool, and the widgets had to be present even if they weren't currently visible to allow for python access. This has since been revamped so that the tables, which have the underlying data, are stored with the notebook, not with the results tools. This permits sharing of the tables as well as python access even if the results tools are not present.

For this reason, there is no longer any need to keep multiple `TableWidgets` around; only the currently visible `TableWidget` is needed.

## 7.10.4 Saving and Restoring Results Tools

A single 64-bit integer is the only datum stored with a results tool: the fit ID of the presently displayed fit. This is because the data for the table is stored with the plot tool.

## 7.11 Table Tool

The table tool shows a `Table`, which can be thought of as a scientifically oriented spreadsheet. The `TableCells` that populate a table are composed of quantities (class `Quantity`) and formatted for display by `TableCellFormats`. A `Quantity` optionally carries extra properties, at a minimum the value itself and an associated error value. `Quantity` subclasses such as `QuPhoton` and `QuDistance` embody a particular type of physical quantity, and include logic to perform units conversions and display units and error information. When the units are set to auto, values are displayed in the units preferred by the user.

The python code fragments may have syntax errors—I don't think in python yet.

### 7.11.1 Tables and Python

There are several ways to get data between Python and a table tool. A special type of table cell is a `TableCellExpr`, a Python expression. As with typical spreadsheets, starting a string with an equals sign identifies a cell as an expression; the rest of the line is any valid Python expression that returns a scalar value:

```
=math.sin(theta)
=f1.fullPath()
```

Consider this table, a fragment of results from a fit:

	0	1	2	3	4	5
0	<b>Results for plot1, Spectrum 0, Model 0</b>					
1	/Users/tkent/ATFiles/Obs/Observations/W49b-sxs.pha					
2						
3	Parameters					
4	$\chi^2$	beta /N	Level	1:nH	2:PhoIndex	3:norm
5	3.3201×10 <sup>5</sup>	19837	-1	1.4495	1.4271	0.81813
6	3.0202×10 <sup>5</sup>	21205	-1	2.0618	1.8587	1.9052

From a Python program, you can access a table by name and row/column:

```
chiSquared = table1.cellAt(5, 0).value()
```

The first two rows, the title and the filename, are set to span multiple columns. Access a spanned datum by accessing its leftmost column, e.g.:

```
path = table1.cellAt(1, 0).stringValue()
```

Labels, shown in light gray, may be used to specify a row or a column. These are equivalent:

```
betaN = table1.cellAt(5, 1).value()
betaN = table1.cellAt("|beta|/N", 1).value()
```

Lookups by name aren't blindingly efficient, so if you need to use a particular coordinate repeatedly, call `findCell` to obtain its coordinates.

```
cell = table1.findCell("|beta|/N", row, column, access)
```

### 7.11.1.1 TableRanges

Many tables, like those containing fit results, are comprised of blocks of variable size. Isolating those blocks requires iterating through the entire table, search for anchor points and size boundaries, before the blocks can be accessed. `TableRanges` address this by making it easy to express *ranges*, or rectangular subsets of a table. A `TableRange` origin may be expressed with absolute coordinates, and/or with the names of label cells. The number of rows and columns may be fixed to specific values, expressed as 0 (the rest of the rows or columns in the table), or as -1, which terminates the range with the first blank row or column.

**Below is deprecated.**

Some tables contain variably sized blocks of information, and are most easily accessed as 2D arrays even if their underlying data is not organized as an array. A *block*, expressed by a `TableRange`, is a rectangular portion of a table. A `Table` has a current block. (I thought about creating a `Block` class to contain this information instead, but I couldn't think of a reason to have multiple blocks at once.) To set the current block, call:

```
setBlockOrigin(int row, int column, TableRangeType type = TRTopRange)
```

This call creates a block whose origin is (row, column). Its `rowCount` extends down to the first row that is entirely blank, and its `columnCount` extends to the right to the first entirely blank column. If `type` is `TRTopRange`, the range is constrained to the boundaries of the topmost array, if any, within which the origin lies. If `type` is `TRBaseRange`, the range is defined for the base table, ignoring any overlaid arrays. Alternatively,

```
setBlockRange(const TableRange& range, TableRangeType type = TRTopRange)
```

sets the range explicitly. In both cases, the range size is clipped to remain within the table. If the origin is out of range, it is set to (0, 0) and the row and column counts are also set to 0.

Once created, the current block *does not change size or shape in response to changes in the shape or content of the table*, mostly for reasons of efficiency. If you make changes to the table, define a new block.

The block is a convenience for accessing portions of a `Table`. For example, here is a larger portion of the fit results:

	0	1	2	3	4	5
0	<b>Results for plot1, Spectrum 0, Model 0</b>					
1	/Users/tkent/ATFiles/Obs/Observations/W49b-sxs.pha					
2						
3	Parameters					
4	$\chi^2$	beta /N	Level	1:nH	2:PhoIndex	3:norm
5	3.3201×10 <sup>5</sup>	19837	-1	1.4495	1.4271	0.81813
6	3.0202×10 <sup>5</sup>	21205	-1	2.0618	1.8587	1.9052
7	2.577×10 <sup>5</sup>	24064	-1	2.5262	2.1093	4.0671
8	2.3551×10 <sup>5</sup>	18498	-1	2.7881	2.2445	6.3732
9	2.3308×10 <sup>5</sup>	7787.2	-2	3.5945	2.6425	11.739
10	2.2212×10 <sup>5</sup>	19214	-3	4.014	2.7281	16.881
11	2.2149×10 <sup>5</sup>	5472.9	-4	4.1278	2.7439	18.361
12	2.2148×10 <sup>5</sup>	380.07	-5	4.1781	2.7531	18.78
13	2.2147×10 <sup>5</sup>	86.231	-6	4.199	2.7563	18.925
14	2.2147×10 <sup>5</sup>	26.337	-7	4.2087	2.7579	18.996
15	2.2147×10 <sup>5</sup>	12.084	-8	4.213	2.7586	19.026
16	2.2147×10 <sup>5</sup>	4.9436	-9	4.2149	2.7589	19.04
17	2.2147×10 <sup>5</sup>	2.2458	-10	4.2157	2.759	19.046
18						
19	Variances and principal axes					
20		1	2	3		
21	1.8326×10 <sup>-6</sup>	-0.0728	-0.9968	0.0345		
22	8.9562×10 <sup>-5</sup>	0.9942	-0.0752	-0.0764		
23	0.043924	-0.0788	-0.0287	-0.9965		

The array of fit results and their parameters has a variable number of rows and columns, so to access any of the first block of information requires some means of dealing with this. First, we establish a block with an origin, a reference position from which the rows and columns of an array may be accessed:

```
table1.setBlockRange(5, 0)
```

Access cells in the current block using the access argument `TABlock`:

```
rows = table1.rowCount(TABlock) // rows = 13
cols = table1.columnCount(TABlock)
print("array is", rows, " x ", cols)
print("The first |beta|/N value is ", table1.cellAt(0, "|beta|/N", TABlock))
```

prints:

```
array is 13 x 6
The first |beta|/N value is 19837
```

Now suppose we wish to access the values in the variances and principal axes block. Since we don't know how many rows are in the fit results, we need some way to set the origin for the following block. We do this by using a label as an *anchor*—a position that may change, but is fixed in relation to data of interest. (Most labels are chosen to be unique, so they can serve as anchors; while data values are unreliable as anchors and their use is not advised.)

```
if (!findCell("Variances and principal axes", row, column))
  print("Cell not found!")
setBlockRange(row+2, column)
```

Here, the anchor is a couple of rows above the cell we want to designate as the origin, so we use the code above to establish the desired origin. Now it is easy to iterate through the values in the variances and principle axes block.

```

for (row = 0; row < table1.rowCount(TABlock); row++)
  for (col = 0; col < table1.colCount(TABlock); col++)
    print("row = ", row, ", col = ", col, ": ", table1.cellAt(row, col, TABlock)

```

When any coordinate is specified by cell value, it is resolved *within the implied scope of its block*. This can sometimes be used to resolve name ambiguities.

19	Variances and principal axes			
20		1	2	3
21	1.8326×10 <sup>-5</sup>	-0.0728	-0.9968	0.0345
22	8.9562×10 <sup>-5</sup>	0.9942	-0.0752	-0.0764
23	0.043924	-0.0788	-0.0287	-0.9965
24				
25	Covariance matrix	1	2	3
26		0.0003612	9.278×10 <sup>-5</sup>	0.003442
27		9.278×10 <sup>-5</sup>	3.852×10 <sup>-5</sup>	0.001257
28		0.003442	0.001257	0.04362

In this example, the origin is set to row 20, column 0, and the scope of the block is highlighted in red.

```

table1.findCell("Variances and principal axes", row, col)
covarianceOrigin = table1.cellAt("Covariance matrix") // Succeeds, relative to table
covarianceOrigin = table1.cellAt("Covariance matrix", TABlock) // Fails, not in the block

```

As stated earlier, this technique for treating a block like an array does not depend upon whether the underlying data structure used to build the block is itself an array. This summary block, for example, brings together several related bits of data, but is not internally an array. Note also that the scope of the array is the largest number of rows and columns, despite the presence of some blank columns.

37	Summary	$\chi^2$	PHA bins	Degrees of free...
38	Fit statistic	2.2147×10 <sup>5</sup>	16384	
39	Test statistic	2.2147×10 <sup>5</sup>	16384	
40	Reduced $\chi^2$	13.52		16381
41	Null hypothesis	0		

Below are some access examples. The first is table-relative (because there is no access argument), using row/column labels to specify coordinates. The next two lines establish a block, and the remaining lines make references from there.

```

reducedChiSquared = table1.cellAt("Reduced  $\chi^2$ ", " $\chi^2$ ")
table1.findCell("Fit statistic", row, col)
table1.setSubtableRange(row, col)
fitStatBins = table1.cellAt(0, 2, TABlock)
fitStatChiSquared = table1.cellAt(0, " $\chi^2$ ", TABlock)
p = table1.cellAt(3, " $\chi^2$ ", TABlock)

```

The table below, an example used for debugging, includes a partial periodic table beginning several rows into the table. Python code to print the periodic table, and its output, is shown.

The screenshot shows a Jupyter Notebook window titled "Untitled-167.cnbk". It contains two main components: a table and a Python code block.

**Table 1:**

		182	1.13 Å	23.993	
		344	1.14 Å	29.113	
Hydrogen	H	1	1.0079	1s1	
Helium	He	2	4.0026	1s2	
Lithium	Li	3	6.941	1s2 2s1	

**python1:**

```

1 # Find the start of the partial list of elements
2 row = table1.findCellRow("Hydrogen", at.TATopTable)
3 column = table1.findCellColumn("Hydrogen", at.TATopTable)
4
5 # Set the subtable origin to this anchor position
6 table1.setSubtableOrigin(row, column, at.TRTTopRange)
7
8 # Iterating through the subtable is bounded by a blank row and column
9 for r in range(0, table1.rowCount(at.TASubTable)):
10     for c in range(0, table1.columnCount(at.TASubTable)):
11         print (table1.cellString(r, c, at.TASubTable), "\t", end='')
12     print()

```

**Python Console:**

```

Hydrogen H 1 1.0079
Helium He 2 4.0026
Lithium Li 3 6.941
Beryllium Be 4 9.0122
Boron B 5 10.811
Carbon C 6 12.011
Nitrogen N 7 14.007
Oxygen O 8 15.999
Fluorine F 9 18.998
Neon Ne 10 20.18
Sodium Na 11 22.99
Magnesium Mg 12 24.305
Aluminum Al 13 26.982
Silicon Si 14 28.085
Phosphorus P 15 30.974
Sulfur S 16 32.065
Chlorine Cl 17 35.453
Argon Ar 18 39.948
Potassium K 19 39.098

```

## 7.11.2 Arrays

If a Python expression returns a one- or two-dimensional array, this is displayed in the table with its top left corner over the cell with the expression. The table extends down and to the right. You can think of the array as if its upper left corner has been taped to the underlying table. If the array changes in size, it may hide other cells on the spreadsheet. The hidden cells remain intact, but they are temporarily inaccessible. Multiple arrays are stacked, if they overlap, with higher rows covering lower; and higher columns within a row covering lower. You can avoid overlap altogether by placing scalar cells above or to the left of arrays, or simply by assigning arrays to separate table tools.

A pure Python array contains unadorned values and is displayed accordingly. But if an array contains Quantities, the extra information they carry (units, error) may also be displayed. If the array is a `QuantityArray`, it also carries optional row and column headings.

## 7.11.3 TableCellFormats

A `TableCellFormat` contains information used to format quantities for display. Since many table cells share identical formatting, `TableCellFormats` are shared by multiple cells. `Notebook::allocateTableCellFormat()` obtains a pointer to a `TableCellFormat` with a given set of properties. The notebook maintains a list with an entry for each unique combination of properties. This list grows during a session and is culled to remove unused entries only at notebook save time. For typical tables this list should remain relatively short. Multiple tables all share the same pool of `TableCellFormats`.

## 7.11.4 Selections

Currently, selections involving mixed data types or mixed physical types suppress most operations, so the user is forced to select objects of uniform type before changing their state. This is overly restrictive, making it hard to do operations such as select all followed by a format change. So a planned design change is to show all display options for all data and physical types all the time. Controls are enabled if they apply to at least one cell in the selection, and the changes they make are applied only to candidate cells. This makes it easy to do operations such as selecting a column of entered numbers and assigning them a physical type.

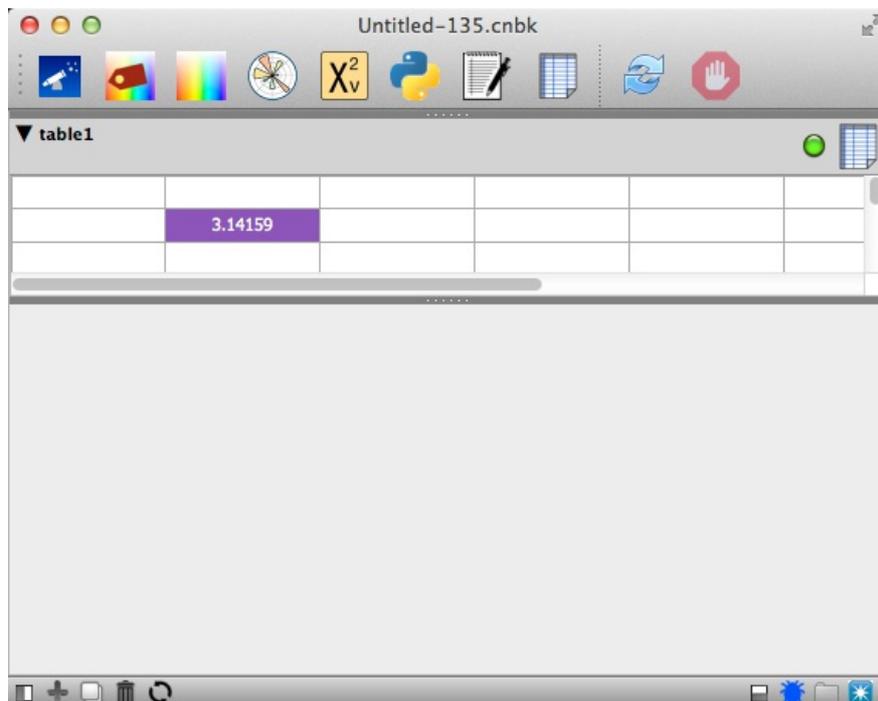
We will also add a Set physical type... button. This puts up a dialog that allows the physical type and units to be changed for `QuDoubles` in the selection. The reason for the dialog is that a type change must be accompanied by units assignment to be meaningful. Set physical type... can also be used to strip units from a selection, converting `QuDouble` subclasses into `QuDoubles`.

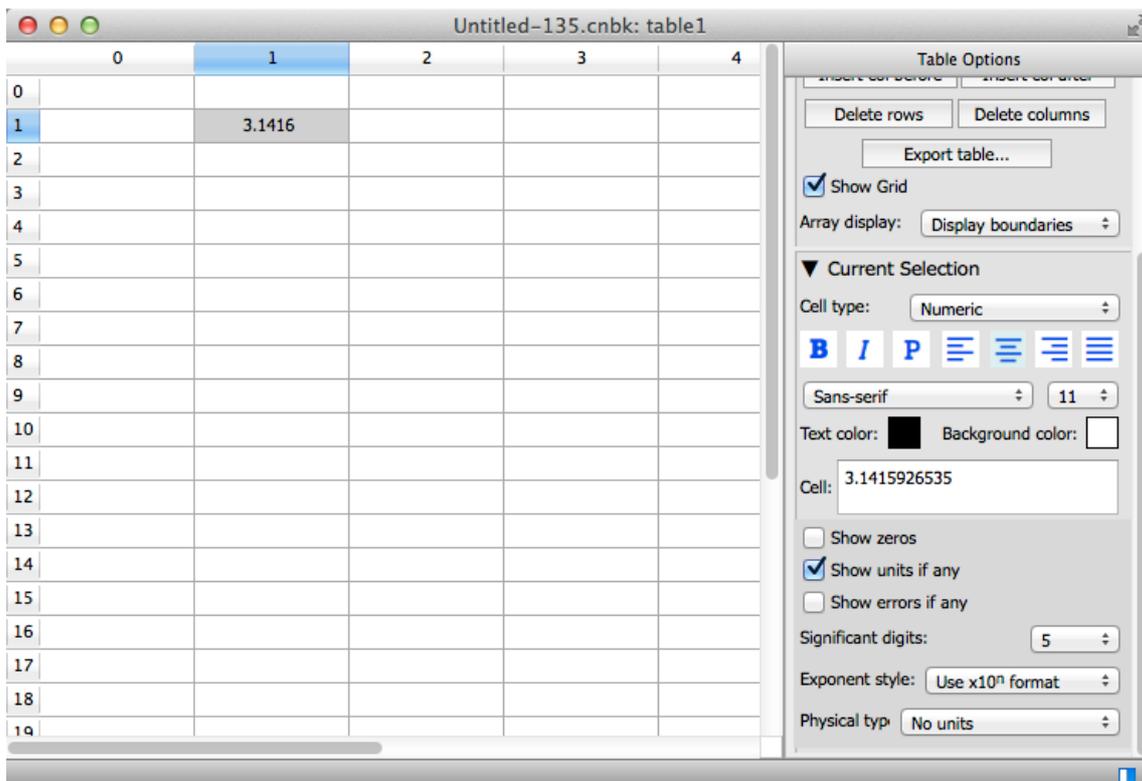
Many units designators apply to more than one physical type. For example, “eV” can be a temperature or an energy, and “m” could be a distance or an energy. I presently define a default interpretation for all ambiguous units. “20 eV” is always interpreted as an energy by default. But if a selection contains members of a uniform type, e.g. `QuPhoton`, and that type can be interpreted as more than one physical type, an Interpret as popup menu lists the possible types. In this example, the menu contains “Energy” and “Temperature.”

## 7.11.5 Table Tool Editor

### 7.11.5.1 Table Cell Edit vs. View Properties

Editing cells imposes some interesting challenges. A table tool cell has up to four representations, three of which are evident in the screen images below:





- The notebook has a read-only view of the table.
- The table tool editor has a table view that may be double-clicked for editing.
- The tool editor’s sidebar has a box for editing the current cell, if a single cell is selected for editing.
- When a cell is double-clicked for editing, a `QLineEdit` the same size as the cell is overlaid on the spreadsheet. This overrides Qt’s normal mechanism in order to customize the behavior.

The view of a cell not selected for editing, in both the notebook and the editor, is called a readonly view. This view may differ significantly from the cell representation during editing. And even the cell edit view and the sidebar views are not always identical, as shown in this table.

<i>Operation</i>	<i>Readonly Views</i>	<i>Cell Edit View</i>	<i>Sidebar View</i>
High precision	View as user-set precision, e.g. 3.142 for 4-digit precision	View full entered precision, e.g. 3.1415926535, to avoid precision loss during edits	Same as cell edit view
Errors	If error display enabled, show in user-set precision, e.g. 50.23 ±2.123 keV or 50.23 +2.123 -2.05 keV	Full precision, e.g. 50.2278 ±2.1229 keV or 50.2278 +2.1229 -2.05 keV	Same as cell edit view
Plain/bold/italic	Display as set	Display as set	Display plain text only
Font style	Display as set	Display as set	Display fixed sans-serif font
Font size	Display as set	Display fixed (small) size to allow more space for data entry	Display fixed (small) size
Text alignment	Display as set	Display as set	Left alignment
Show/hide zeros	Display as set	Always show zeros	Always show zeros
Exponent style	Display as set	“Computerese” E notation	E notation

Physical units	Display as set	Always show units	Always show units
Python expressions	Display computed result, “??” if not refreshed, “(Undefined)” if undefined, $\emptyset$ for null	Display python expression, e.g. <code>=sqrt(2)</code>	Same as cell edit view

Implementing this disparate behavior required overriding Qt’s cell editing behavior and replacing it with my own.

### 7.11.5.2 Navigator Keys

Cell navigation works as follows. (Needs a design review.)

<i>Key</i>	<i>Action</i>
←	Left one char in cell
→	Right one char in cell
⌘←	To left edge of cell
⌘→	To right edge of cell
Home	To first column of this row
End	To last column of this row
⌘↑	Top of table in same column
⌘↓	Bottom of table in same column
Tab	Next cell, if last cell in row, to first cell of next line
Backtab (⇧-tab)	Previous cell, if first cell in row, to last cell of previous line
Return	Next row, same column

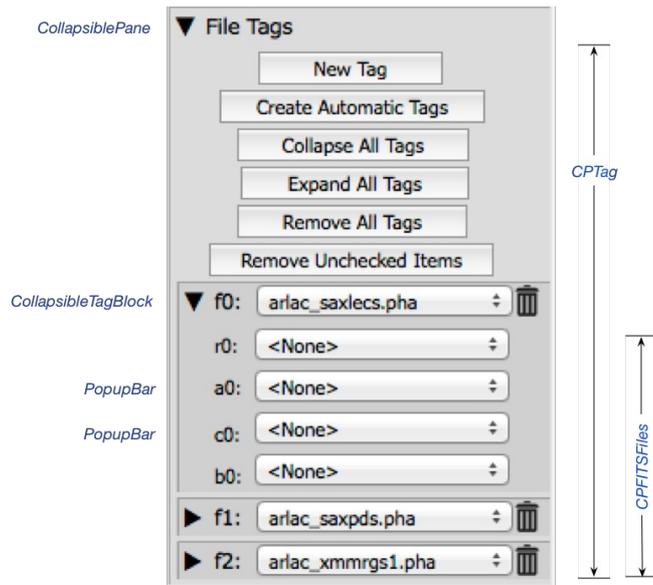
## 7.12 Tag Tool

### 7.12.1 Tag Sidebar

The tag tool creates “tags” which represent a set of files that together serve as input to a fit. A tag serves as a shorthand way of referring to the set of files, and also makes it easy to analyze a different set of files simply by adjusting the tag, rather than all the models that may be applied to it. At a minimum, a tag consists of a primary file. Both tag and primary file are referred to as  $f_n$ , where  $n$  is an automatic sequence number. A tag also has an optional RMF file,  $r_n$ ; ARF file,  $a_n$ ; COR file,  $c_n$ ; and background file,  $b_n$ . Create Automatic Tags creates tags for each primary file selected in the observatory tool. It uses information from the primary file’s metadata to make default selections for the related files. Users can fine tune these selections or create tags manually.

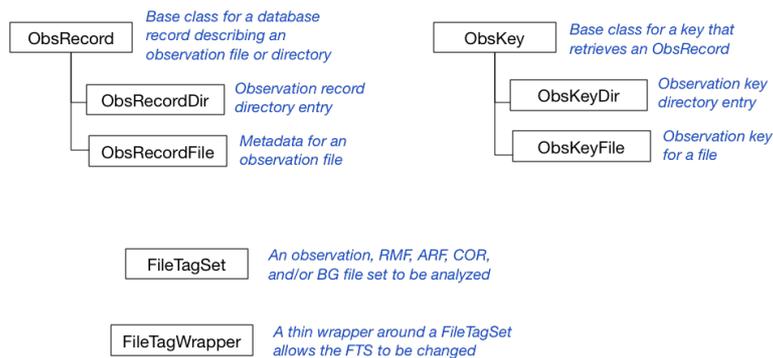
It is sometimes necessary to apply fits to groups of tags. Usually the same model parameters apply to all members of the group, but sometimes a few parameter settings are overridden on a per-member basis. ATLAS supports grouping via Tag Groups, which are simply a set of previously defined tags, identified as  $g_n$ .

There is only one tag tool per notebook. Unlike other tools, its position in the notebook doesn’t really matter, but the intuitive sequence is to create one or more observatory tools, a tag tool, then everything else. The relevant portion of the tag tool sidebar is shown below.

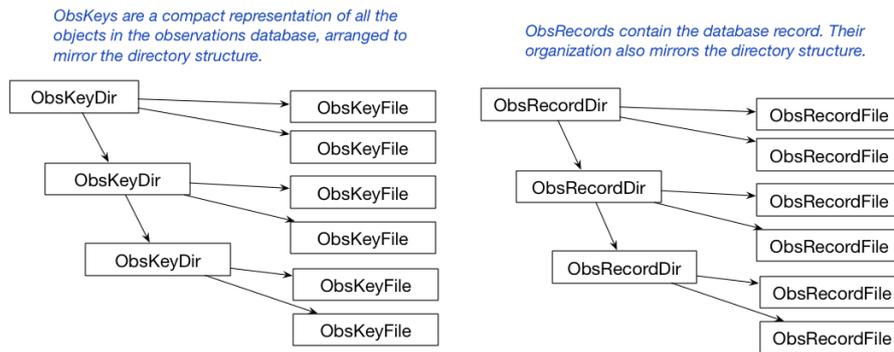


### 7.12.2 Tag Classes

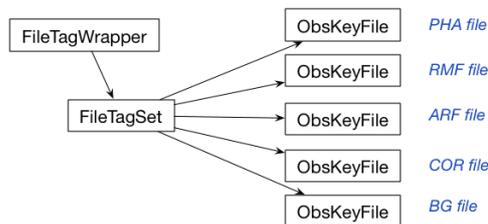
ObsKeys are compact keys used to retrieve a database entry. ObsRecords contain a database record. A FileTagSet aggregates the files needed to produce an analysis. A FileTagWrapper is essentially a named pointer that may be redirected to another FileTagSet.



### 7.12.3 Tag Ownership Diagram



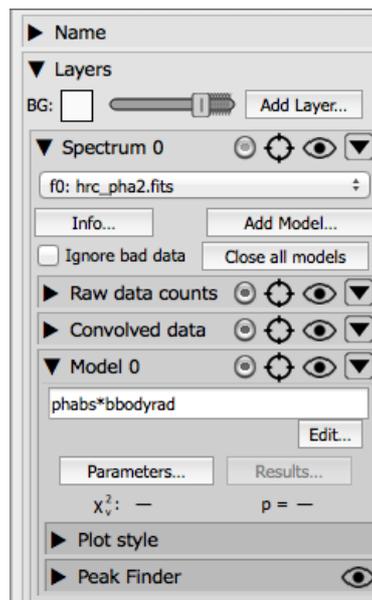
*Wrapper handles an empty tag or can be pointed to a different tag set.*



## 7.12.4 Tag Tool Redesign

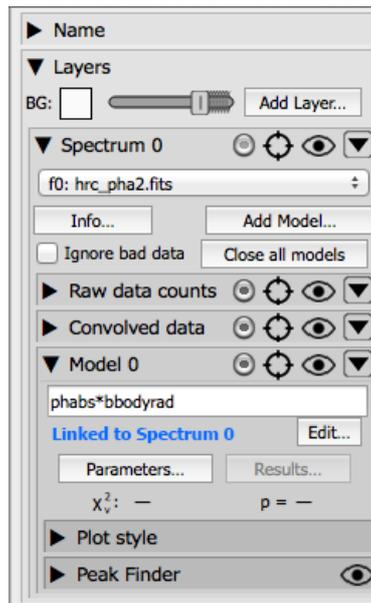
Below is an e-mail to Randall from July 2016, since slightly edited as a result of design refinements, describing a redesign for handling file tag groups. We agreed on the approach. At the time of this writing, it is not yet implemented, but the file groups option in the Tag Tool is turned off in anticipation of implementing this solution.

I was thinking last night about the data grouping. I introduced file tag groups as a means of handling this some time back, but now that I better understand its operation, that won't work well. Here is a block with a single spectrum created. The spectrum has one or multiple models, an ignore region, raw and convolved data, and the fit. Each plot in turn has independently settable styles and visibility as a means of making them distinguishable. Each model even has its own peak finder (though of course only one set of line labels is visible at a time).



So what happens when we add a second spectrum to the data group? *Every single option shown here is replicated*, at least based on my present understanding. For this example you have a total of three new plots, potentially a new set of line labels, and a new set of results.

So I could add a menu item to a spectrum's popup menu at top right, "Duplicate and Share Spectrum". This would insert a new spectrum block identical to the original (pretend this one is labelled Spectrum 1):



The only thing that changes is that each model reminds the user that the model settings are shared, and I cram an icon in somewhere else to indicate that the spectrum is shared. Once two or more spectra are shared, both of their model panels indicate this, and either can be used to edit the shared model information. Any spectrum may secede from the sharing arrangement via the settings menu. **Two existing spectra cannot be shared with each other. While this is possible in principle, it requires that one spectrum's models and parameters be completely replaced with the other's. Since this operation is equivalent to deleting one of the spectra and using Duplicate and Share Spectrum to insert another one, I don't presently support this kind of sharing. Among other things, this means that if a spectrum is unshared, it cannot then be re-shared.**

This provides independent control over all the purely stylistic options, and is highly compatible with the existing logic. Each linked spectrum has its own results, its own labels, etc.

Another interesting aspect of this is that I don't have to make any changes to the code that goes out to XSpec. I can use the existing logic to independently fit the two spectra and superimpose their results. (`setplot group 1-2` excepted of course.) And I can drop the file tag groups entirely.

So to summarize, to create what you have here:

```
data 1 hrc_pha2.fits{1}
data 2 hrc_pha2.fits{2}
back 1 hrc_bkg_pha2.fits{1}
back 2 hrc_bkg_pha2.fits{1}
resp 1 leg_m1.rmf
arf 1 leg_m1.arf
resp 2 leg_p1.rmf
arf 2 leg_p1.arf
```

the user takes the following steps:

1. Mark the six participating files in the observation tool as interesting files.
2. Generate autotags. This produces a single tag, fo, with spectrum 1 selected. It has a background, but no RMF or ARF as discussed. Select these manually.
3. Generate a second manual tag, f1. Select the same PHA file, but select the second spectrum. Select the appropriate ARF, RMF, and background files. **(In principle, the autotagger could generate a separate tag for each spectrum in a file containing multiple spectra, but this might generate superfluous tags. Maybe autotagging should occur only for files containing fewer than  $n$  spectra, where  $n$  is, say, 5.)**
4. Insert a plot tool and insert a spectrum. Select fo and configure with a model.
5. Select Duplicate and Share Spectrum to create a second, shared spectrum, then select tag f1 for the duplicated spectrum.

6. Subsequent edits to the model expressions or parameters are shared between the spectra.
7. Hit refresh. ATsAL actually treats them exactly the same as if they were not shared. So, for example, this means that two different XSpecs can work on both fits in parallel.

If this seems right to you, it is fairly clean to implement.

[End of email.]

## 7.12.4.1 Design and Implementation

This describes the design in more detail.

### 7.12.4.1.1 Micro-glossary

- A parameter has a hardwired “*factory default*,” which may be restored by the user. The factory default is assigned by XSpec for standard models, or by the definer of a user model for those.
- When a refresh cycle updates a parameter value, the most recently entered *user default* is retained so it can be restored if desired.
- Two or more spectra are *shared* if their models and parameters are synchronized and modified together.
- Two or more shared parameters are *linked* if a change to the value also updates the shared value (hence other spectra sharing them), or a change to the shared value updates this parameter value. When spectra are first shared, all their parameters are linked by default, but users can unlink some of them. So sharing happens between spectra, while linking is used to override sharing at the parameter level.
- When a user edits a parameter value, **all spectra** are notified of the change. A given spectrum is a *target* of a parameter change if the parameter is part of this spectrum or a shared spectrum. The spectrum is a *primary target* if its own parameter was edited. It is a *shared target* if it is shared with the primary target but is not itself the primary.
- A *broadcast* is a specialized operation, used only with the `features` model. This model creates a set of Gaussians, one per peak as located by the peak finder. The user may copy, or broadcast, a given Gaussian subparameter to other peak finder Gaussians.

### 7.12.4.1.2 User Interface

Spectrum sharing menu options are:

- Duplicate and Share Spectrum duplicates a spectrum and its models, with sharing enabled between them. The user then selects a different tag for the duplicated spectrum.
- Unshare Spectrum secedes from the share, giving the spectrum its own copy of the models and parameters that were previously shared. Any other spectra that were previously shared remain so. That is, if A, B and C are shared, and B unshares, A and C remain shared.

Each `xSpectrum` has a list of other spectra, `mSharedSpectra`, with which it is shared. These pointers are saved as UUIDs and restored after all spectra have been read in.

Add `xSpectrum::duplicateAndShareSpectrum(xSpectrum* target)`. This creates a spectrum whose models and parameters match those of `target`. After duplication, the user selects a different tag for the duplicate.

Add `xSpectrum::unshareSpectrum()`. This secedes from the sharing. This function makes local copies of any shared data first. It notifies all the shared spectra so they can purge the spectrum from their respective `mSharedSpectra` lists.

When deleting a spectrum, call `unshareSpectrum()` first so that other linked spectra are not effected.

When a model expression is changed in a panel, tell the notebook to update everyone.

When a model expression is changed in a model editor, tell the notebook to update all models if the dialog is accepted.

Model param changes don't need to be broadcast to other user interface elements, since they are only visible in a modal editor. However, parameter changes are processed by the `Notebook`, which forwards the change to all target spectra.

Style changes are all local, so there is no need to propagate them.

Tag changes are also local.

Changes to the peak finder are local to each spectrum and do not require propagation to other targets. Hence broadcasts are also local.

When a spectrum is shared, and its model panel's error message region is empty, display "Shared with *n* spectra" in blue.

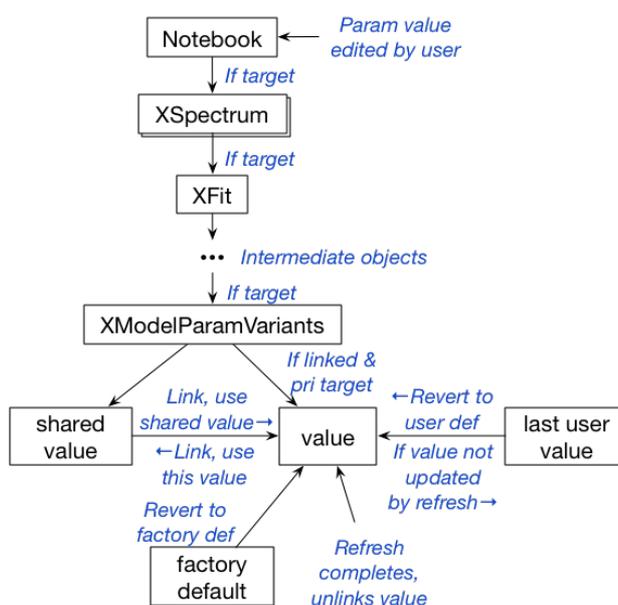
### 7.12.4.1.3 Handling Parameter Updates

Originally I intended to keep a single set of shared parameters for each shared `XSpectrum`. But this turned out to be much more complex than the current design. Instead, each parameter is represented by an `XModelParamVariants`, which in turn keeps four variants of a parameter: (1) the factory default, or hardwired original default value (actually stored in the `XModelParamInstance`, not the `XModelParamVariants`); (2) the last edited user value (`mUserDefault`); (3) the current value (`mValue`); and (4) the shared value (`mShared`). So if two spectra are shared, each parameter of each model *has its own copy of the shared value*, and they must be kept synchronized. This sounds a little risky, but it solves problems of sharing/unsharing, linking/unlinking, and deletion more cleanly.

The table below describes the action taken when each parameter receives a signal that parameter text has changed.

Primary Target	Shared Target	Linked	Action
N	N	Y	None
N	Y	N	Update <code>mShared</code>
N	Y	Y	Update <code>mShared</code> , <code>mValue</code>
Y	N	N	Update <code>mValue</code>
Y	N	Y	Update <code>mShared</code> , <code>mValue</code>
Y	Y	N	Update <code>mValue</code>
Y	Y	Y	Assertion error: shouldn't happen

This diagram shows how various actions transfer values among the variant copies stored in an `XModelParamVariants`. When a parameter is edited by the user, the signal is passed to the notebook, and then to those spectra that are targets (i.e. an actual target or a shared spectrum). When a refresh completes, any values that are changed as a result of the fit are updated, and changed parameters are automatically unlinked, since these computed values are specific to each spectrum.



When a shared spectrum is created via Duplicate and Share Spectrum, all shared values are set to match the originating spectrum, and all parameters are originally set to linked. In unshared spectra, the shared values are ignored and undefined.

## 7.13 Parsers, Datasets, Tables, and Python: A Refactoring

This is a little essay on a significant refactoring within ATXSAL, recently completed. In the world before the Great Refactoring, the following steps took place:

- The parser extracted data from XSPEC output, representing it as a series of scalar values.
- Upon completion of a command sequence, the extracted data was then cherry-picked to extract a series of key-value pairs, associated with each Dataset. (A Dataset is the output from a command series, typically a plot such as a fit, along with associated data, such as fit results.
- For fits, some of this data is then formatted into a small table of fit results.

Let's take a closer look at the output from XSPEC's fit command:

```

Parameters
Chi-Squared |beta|/N   Lvl      1:nH      2:PhoIndex      3:norm
332012      19836.9   -1        1.44948        1.42713        0.818131
302025      21204.9   -1        2.06177        1.85874        1.90518
257698      24063.7   -1        2.52621        2.10928        4.06707
235514      18498.4   -1        2.78808        2.24452        6.37316
233080      7787.2    -2        3.59446        2.64245        11.7389
222122      19213.7   -3        4.01402        2.72815        16.8814
221490      5472.92   -4        4.12783        2.74391        18.3615
221476      380.067   -5        4.17810        2.75313        18.7801
221474      86.2314   -6        4.19903        2.75625        18.9251
221474      26.3372   -7        4.20874        2.75790        18.9964
221473      12.0835   -8        4.21297        2.75856        19.0260
221473      4.94361   -9        4.21488        2.75887        19.0397
221473      2.24578   -10       4.21572        2.75901        19.0457
=====
Variances and Principal Axes
              1          2          3
1.8326E-06 | -0.0728  -0.9968  0.0345
8.9562E-05 |  0.9942  -0.0752 -0.0764
4.3924E-02 | -0.0788  -0.0287 -0.9965
-----
=====
Covariance Matrix
              1          2          3
3.612e-04   9.278e-05   3.442e-03
9.278e-05   3.852e-05   1.257e-03
3.442e-03   1.257e-03   4.362e-02
-----
=====
Model phabs*powerlaw Source No.: 1  Active/On
Model Model Component  Parameter  Unit      Value
par  comp
  1    1    phabs      nH        10^22    4.21572   +/-  1.90059E-02
  2    2    powerlaw   PhoIndex  2.75901   +/-  6.20650E-03
  3    2    powerlaw   norm      19.0457   +/-  0.208843
-----
Fit statistic : Chi-Squared =          221473.5 using 16384 PHA bins.

***Warning: Chi-square may not be valid due to bins with zero variance
in spectrum number(s): 1

Test statistic : Chi-Squared =          221473.5 using 16384 PHA bins.
Reduced chi-squared =          13.52014 for 16381 degrees of freedom
Null hypothesis probability =          0.000000e+00

***Warning: Chi-square may not be valid due to bins with zero variance
in spectrum number(s): 1

```

### 7.13.1 Parser Extensions

The data are organized into a series of blocks. The parameters at the top are of interest, because their best values become the input in most cases for the next round of fitting. Looking at the block nearer the bottom, the one that begins with the model expression, the table of best fit values has a few special properties beyond the capabilities of the existing parser. First, there are blanks in the table, in the units column. ATXSAL's parser is token-based, not position-based, and it cannot deal with blanks. Second, there are errors associated with the values that we wish to capture. Third, the units column itself should be preserved so that downstream use will not lose track of the physical units.

ATXSAL's parser now supports this, but doing so required that the existing data representation be generalized. Instead of handling just scalars (strings, ints, doubles, etc.), the parser now represents data as quantities—`Quantity` subclasses. This means that error values are parsed and preserved, and also means that physical units are preserved where appropriate.

But quantities cannot have names because the overhead of including even an optional name and description is significant in some contexts. For instance, a `QuPoint` is an ordered pair of quantities, and the inclusion of an optional name would double the size of a `QuPoint`. Since these points become part of large arrays, and there may be multiple copies of the arrays, the overhead accumulates. So instead, `QuVariables` were added. A `QuVariable` is name and optional description bound to a `Quantity`. `QuVariables` include scalars, structures, and arrays, and may be nested arbitrarily. This allows us to fully represent the fairly elaborate structure of the output from the fit command, or that from groups of commands.

`QuVariables` also bring a general mechanism for nested variable naming.

`plot1.spectrum[0].fit[0].results.fit.fit_statistic. $\chi^2$`  is one way of accessing a particular result value.

### 7.13.2 Datasets

The data structures created by the parser are somewhat haphazard, a side effect of the organization of the data in the output rather than an organization best suited for downstream use. While unavoidable, this is not an interface we want to surface to python. In addition, most logical operations in XSPEC involve a whole series of XSPEC commands, and each of those commands has its own block of parsed results. When the last command completes in a sequence, ATXSAL processes the chain of commands, cherry-picking the relevant results and reorganizing them into more useful form. Once this completes, the commands and their results are discarded in favor of this cleaner representation.

*At the time of this writing, the Dataset representation is simply the entire set of fit results, because I don't have enough domain-specific knowledge yet to settle on a more suitable arrangement.*

The original `Dataset` representation used key/value pairs to represent results, but this is simplistic: we need fully nested structures and arrays, as well as physical types. In fact, we need exactly what the parser needed—a completely general way to represent that data. Hence I simply appropriated the parser's new mechanism for the metadata. This consolidates three distinct mechanisms for representing data into a single uniform mechanism.

### 7.13.3 The Results Tool

The `Dataset` representation is more rational, but its detailed organization is still in flux and it is premature to surface it to python. It could be dumped directly into a spreadsheet (or `Table`, in ATXSAL parlance), but the result would still be a bit ungainly. Instead, the newly implemented results tool produces a neater representation of the data:

Results for plot1, Spectrum 0, Model 0						
/Users/tkent/ATFiles/Obs/Observations/W49b-sxs.pha						
Parameters						
$\chi^2$	beta /N	Level	1:nH	2:PhoIndex	3:norm	
3.3201×10 <sup>5</sup>	19837	-1	1.4495	1.4271	0.81813	
3.0202×10 <sup>5</sup>	21205	-1	2.0618	1.8587	1.9052	
2.577×10 <sup>5</sup>	24064	-1	2.5262	2.1093	4.0671	
2.3551×10 <sup>5</sup>	18498	-1	2.7881	2.2445	6.3732	
2.3308×10 <sup>5</sup>	7787.2	-2	3.5945	2.6425	11.739	
2.2212×10 <sup>5</sup>	19214	-3	4.014	2.7281	16.881	
2.2149×10 <sup>5</sup>	5472.9	-4	4.1278	2.7439	18.361	
2.2148×10 <sup>5</sup>	380.07	-5	4.1781	2.7531	18.78	
2.2147×10 <sup>5</sup>	86.231	-6	4.199	2.7563	18.925	
2.2147×10 <sup>5</sup>	26.337	-7	4.2087	2.7579	18.996	
2.2147×10 <sup>5</sup>	12.084	-8	4.213	2.7586	19.026	
2.2147×10 <sup>5</sup>	4.9436	-9	4.2149	2.7589	19.04	
2.2147×10 <sup>5</sup>	2.2458	-10	4.2157	2.759	19.046	
Variances and principal axes						
	1	2	3			
1.8326×10 <sup>-5</sup>	-0.0728	-0.9968	0.0345			
8.9562×10 <sup>-5</sup>	0.9942	-0.0752	-0.0764			
0.043924	-0.0788	-0.0287	-0.9965			
Covariance matrix						
	1	2	3			
	0.0003612	9.278×10 <sup>-5</sup>	0.003442			
	9.278×10 <sup>-5</sup>	3.852×10 <sup>-5</sup>	0.001257			
	0.003442	0.001257	0.04362			
Model						
phabs<1>*powerlaw<2>						
Model par	Model comp	Component	Parameter	Unit	Value	
1	1	phabs	nH	10 <sup>22</sup>	4.2157	
2	2	powerlaw	PhoIndex		2.759	
3	2	powerlaw	norm		19.046	
Summary						
	$\chi^2$	PHA bins	Degrees of free...			
Fit statistic	2.2147×10 <sup>5</sup>	16384				
Test statistic	2.2147×10 <sup>5</sup>	16384				
Reduced $\chi^2$	13.52		16381			
Null hypothesis	0					

So we don't want to surface the parser level organization to python programmers because its organization is awkward. And although the Dataset representation is more natural, it hasn't settled down yet (though eventually this will be surfaced as well). But the representation as shown by the results tool is cleaner, and less likely to change. This makes it the best candidate for python access, using accessors already in place for accessing other Tables. **I need feedback on a more suitable arrangement and content of data for this table.**

But that leads smack-dab into another challenge: how to write portable python code that accesses the fairly complex table.

### 7.13.4 Blocks

Fit results fall naturally into a series of blocks. The blocks are visually obvious, but variable in size, so if row/column notation is used, the python code may not work for a different set of results, or even for the same set once run with a different model expression. We handle this by formalizing the notion of a *block*. A block is simply a rectangular subset of a table. The user determines the position of some *anchor*, typically one of the light gray labels (since these are invariant), and expresses the *origin* of a block (top left corner) in relation to the anchor. ATLAS determines the boundaries of the block by finding the first blank row and column. Now the block may be accessed as a small table.

In the example below, the results tool is scrolled to the block containing variances and principal axes. This is followed by a python tool with code to locate and print the block to the python console. Output produced by a refresh is shown in the console pane at the bottom.

The screenshot shows a software window titled "Untitled-297.cnbk". It contains several panels: "observatory1", "tag1", "plot1", "results1", and "python1". The "results1" panel displays a table with the following data:

Variances and principal axes	1	2	3
1.8326×10 <sup>-6</sup>	-0.0728	-0.9968	0.0345
8.9562×10 <sup>-5</sup>	0.9942	-0.0752	-0.0764
0.043924	-0.0788	-0.0287	-0.9965

The "python1" panel contains the following code:

```

1 # Find the fit ID for the fit
2 fitID = plot1.fitIDFromName("Spectrum 0", "Model 0")
3
4 # Get the associated table from the notebook. The true argument creates
5 # the table if it doesn't exist already
6 table = notebook.resultsTable(fitID, True)
7
8 # Find an anchor cell to use to set an origin for the variances and principal axes
9 row = table.findCellRow("Variances and principal axes", at.TATopTable)
10 column = table.findCellColumn("Variances and principal axes", at.TATopTable)
11
12 # Set the block's origin a couple of rows past the anchor position
13 table.setBlockOrigin(row+2, column, at.TRTTopRange)
14
15 # Now print out the block
16 for r in range(0, table.rowCount(at.TABlock)):
17     for c in range(0, table.columnCount(at.TABlock)):
18         print (table.cellDouble(r, c, at.TABlock), "\t", end='')
19     print()
20

```

The "Python Console" panel shows the output of the code:

```

1.8326e-06    -0.0728    -0.9968    0.0345
8.9562e-05    0.9942    -0.0752    -0.0764
0.043924    -0.0788    -0.0287    -0.9965
py>

```

The first line asks the plot tool for the Spectrum 0, Model 0 fit ID:

```
fitID = plot1.fitIDFromName("Spectrum 0", "Model 0")
```

The fit ID is a magic number that uniquely identifies the fit, even—crucially—across ATLAS sessions. Next, we ask the notebook for the results table associated with the fit:

```
# Get the associated table from the notebook. The true argument creates
# the table if it doesn't exist already
table = notebook.resultsTable(fitID, True)
```

Note that we ask the *notebook* for the table, not the results tool. This is because we want to write the python fragment to run regardless of whether results tools are present, and which sets of results they are presently displaying. If we wanted to make the python code operate on the results from the currently displayed table shown in the results tool, we would use `results1.table()` instead. Next, we need to determine the origin of the block of interest:

```
# Find an anchor cell to use to set an origin for the variances and principal axes
row = table.findCellRow("Variances and principal axes", at.TATopTable)
column = table.findCellColumn("Variances and principal axes", at.TATopTable)
```

We use the label "Variances and principal axes" as the anchor position, since this remains the same for any set of

results, though its actual position in the table changes. Using this position, we set the block's origin:

```
# Set the block's origin a couple of rows past the anchor position
table.setBlockOrigin(row+2, column, at.TRRTopRange)
```

This sets both the origin and the extent of the block, using a blank row and column (or table edge) as a boundary. Hence `rowCount()` and `columnCount()` return the block boundaries, in block mode. To print out the table:

```
# Now print out the block
for r in range(0, table.rowCount(at.TABlock)):
    for c in range(0, table.columnCount(at.TABlock)):
        print (table.cellDouble(r, c, at.TABlock), "\t", end="")
    print()
```

Using a table access mode of `at.TABlock`, we retrieve values from the block instead of the table as a whole. The output looks like this:

```
1.8326e-06 -0.0728 -0.9968 0.0345
8.9562e-05 0.9942 -0.0752 -0.0764
4.3924e-02 -0.0788 -0.0287 -0.9965
```

Tables have two other previously implemented convenience mechanisms. A row or column can be expressed as a string with the cell's content. Labels are often used instead of explicit coordinates. And subtables permit arrays to be displayed on top of the underlying table. These are not further described here.

## 7.13.5 Summary

The extensions described here consolidate three data representations into a single richer one, and give more thought to the issue of how to represent XSPEC output at various stages of processing. By temporarily restricting python access to table output, the internal formats are free to evolve as needed. The block mechanism makes it easier to write python code that adapts to output from various models, or other tables. Results tools themselves can display results from any fit, and they share their tables for efficiency. Finally, although results tools implicitly document the output data structure, they aren't necessary for writing python programs that use the output. Results tools may be exported in a variety of generic spreadsheet formats, as well as in HTML format, providing for access by other programs.

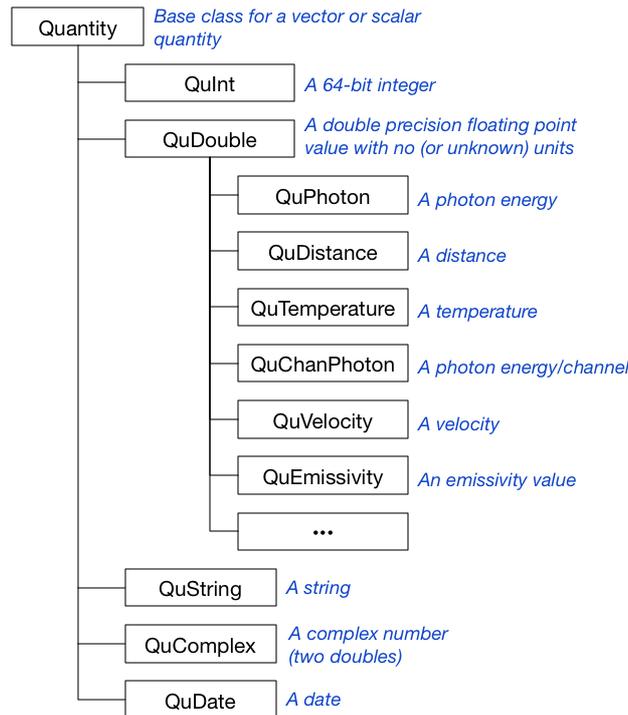
## 7.14 Physical Data Types

Basic data types are represented in ATLAS as `Quantity` subclasses. `QuString`, `QuInt`, `QuDouble`, `QuComplex`, and `QuDate` are the basic quantities (`Quantities.h`). `QuDouble` includes optional error values. A `QuComplex` contains a complex number and associated errors. `QuDouble` subclasses embody basic physical data types such as `QuDistance` and `QuPhoton`. These classes are constructed and accessed by supplying specific units, performing conversion as needed.

`QuVariables` associate a name and optional description with quantities. `QuVarScalar` represents a single scalar value. `QuVarStructs` contain multiple `QuVariables`, and may be nested as needed and accessed with field names like C structs or with array indices. `QuVarVectors` contain quantities rather than variables, so their space overhead is lower; `QuVarVectors` cannot be nested. Finally, `QuPoints` is a special type of vector that contains a vector of `QuPoints`, each of which has  $x/y$  values that are quantities.

### 7.14.1 Quantities

Quantities—subclasses of `Quantity`—represent a single quasi-scalar quantity and the logic necessary to accept it as input, convert between various units, and format it for output. By “quasi-scalar,” `Quantity` subclasses include `QuComplex`, which, while scalar, can be viewed in some contexts as an ordered pair of doubles. A `Quantity` does not have an associated name.



### 7.14.1.1 Physical Units

The physical units classes have gone through several design iterations. Here is a summary of some of the design issues:

- Current units. When a quantity is constructed or set, the units used to set it are preserved as `mCurrentUnits`. The value can be retrieved in these original units via, e.g., `EUCurrent`, or its equivalent for another type of physical unit. Thus by default, the creating format of units are preserved unless specifically retrieved in another form. (Note: this means that the user's preferred units can be retained, but it does not mean that the original value is retained. Each `Quantity` has a native units, to which the value is always converted.)
- When doing calculations on pairs of quantities, the choice of units may be arbitrary. If there is no reason to convert the quantities to preferred units, it is best to use the native units, since the internal representation is the most accurate available for the quantity. The native units is available via `QuPhysicalType::nativeUnits()`, or use `nativeValue()` to retrieve this value.)
- Each physical unit also has a system-wide default type. This default is applied when requested by e.g. `EUAuto`.
- Forced specification of units. Physical quantities must be constructed, and values retrieved, with explicitly specified units. This reduces errors introduced by mistaking the units of values. It also means that a quantity can be used in expressions and functions without any attention to units. For example, given a `QuVelocity` `v` and a `QuElapsedTime` `t`, `QuDistance d = v * t` produces a valid result regardless of the units with which `v` and `t` were created.
- Preservation of accuracy. When a physical quantity such as a `QuPhoton` is constructed, it is converted to internal format in standardized units once, then converted to the desired display format when it is retrieved. This reduces the number of intermediate conversions to some degree, limiting loss of precision.
- Error tracking. Physical quantities include a positive and negative error, which are also converted as needed to other units. It isn't possible to convert the value without also converting the errors.
- Generation of appropriate metric multipliers. Often this amounts to calculating a multiplier such as 'k' and applying it. The classes must apply a consistent multiplier to the error values. That is, 720 GHz ± 320 kHz doesn't work.
- Multipliers must not be used for non-metric quantities (e.g. no gigamiles).
- Multipliers are also inappropriate for metric quantities that are not "fundamental" quantities. "4.35 kÅ" and "19.7 Mmicrons" are not workable. Instead, the former value is displayed as 4350 Å and the latter as 19.7 m. Note that the latter actually converts the *units*, not just the multiplier.
- Each quantity has an editable value, retrieved with `editString()`, which may look very different from the displayed value. The editable string (1) preserves as much precision as possible (within reason); (2) shows the error, or both errors in the case of asymmetric errors; (3) shows the exponent in standard "e" notation. This allows the value to be adjusted without losing information or precision. For example, a value displayed as

$6.023 \times 10^{23}$  might appear as `6.0225232342e23 +3.24234353434e21 -3.22123243439e21` for editing. This is a pathological example, but it illustrates that information is preserved.

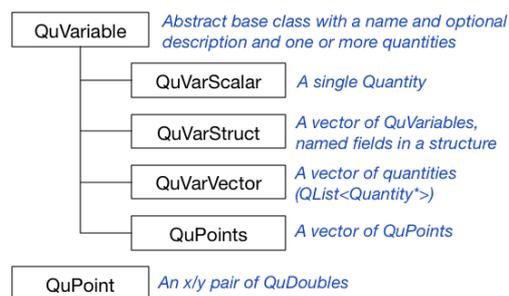
- In a few cases, input units are accepted but converted to another form; they cannot be represented with the same units. The only case I know of so far is microns. Microns were formally replaced with  $\mu\text{m}$  as the preferred designation, so they are accepted and converted on input.
- I am careful about the symbol representing "micro," per this information from Wikipedia. "The symbol for the SI prefix micro- is a Greek lowercase mu. In Unicode, it has the codepoint U+00B5 ( $\mu$ ), distinct from the codepoint U+03BC ( $\mu$ ) of the Greek letter lowercase mu, so that machines can recognize it as the SI prefix symbol rather than as a letter. Most fonts use the same glyph for the two characters."

### 7.14.1.2 Reading and Writing Quantity Values

Since `Quantity` instances are not named, and they need some sort of name for the XML file, `Quantity::write` writes only the `Quantity` type and values, not any opening lead-in that identifies the `Quantity`. The converse of this is the global function `readQuantity`, defined in `PhysicalUnits.cpp`.

### 7.14.2 QuVariables

`QuVariables` contain hierarchies of data picked up, typically, by parsing `XSpec` output. `XSpec` commands that contain data of use to ATLAS each have an `xCmdResults`, which contains the `QuVariables`. At the end of a command sequence, ATLAS cherry-picks the subset of data it needs to retain, consolidating it into a new blob of metadata that is stored in a `Dataset` or subclass.



A `QuVariable` binds a name and an optional description to a set of `Quantity` instances. A `QuVariable` is not itself a `Quantity`. It would be easier to give each `Quantity` an optional name, but this adds a fair amount of avoidable space overhead, and sometimes there are, er, large quantities of `Quantity`s. So instead we separate the variable name from the quantity itself. Each `QuVariable` takes ownership of its `Quantity` instance(s).

`QuVariable` is the base class of named variables, containing a name and optional description. The description is sometimes displayed in table views. `QuVariable` subclasses are as follows.

- A `QuVarScalar` binds a name to a single `Quantity`.
- A `QuVarStruct` is analogous to a structure—a possibly heterogeneous set of `QuVariables`. The members of the structure, or *fields*, may be accessed by index or by field name. Since the fields are themselves `QuVariables`, variables may be nested to any degree.
- A `QuVarVector` is analogous to a dynamic array, but there are some differences. First, the members of the array are `Quantity` subclasses, **not** `QuVariables`, which means they cannot be nested the way `QuVarStructs` are. There is a reason for this: a `QuVarVector` avoids some of the size overhead in its elements. Second, while many dynamic arrays must contain elements of the same type, `QuVarVectors` can contain mixed types. ATLAS typically uses them with homogenous types though.
- A `QuPoints` is specialized to handle vectors of points (`QuPoint`), which in turn are an  $x/y$  pair of `QuDoubles`. A `QuPoints` always contains a vector of `QuPoints`, so instead of using the generic `quantity()` accessor, it is simpler to use array-style referencing. However, for a `QuPoints` named "foo", `quantityNamed("foo[12].x")` returns a `QuDouble*` with the 12th element's  $x$  value. (Well okay, technically the 13th element.)

How do you nest structures and arrays? Since `QuVarVectors` don't contain `QuVariables`, they don't work for this purpose. However, `QuVarStructs` can be accessed as arrays as well, so they can serve as nestable vectors or nestable structures.

### 7.14.2.1 Variable Names

Variable names are patterned after C-like languages. They can contain Unicode characters. “`results.χ²`”, “`var[21][12][3].x`”, and “`foo.bar[10]`” are acceptable names. For a `QuVarStruct`, `foo`, whose third element, `greeting`, is the string “`Hello, world.`”:

```
myStruct->quantityNamed("foo.greeting")
myStruct->quantityNamed("foo[2]")
(*myStruct->quantity)[2]
```

all produce the same result: “`Hello, world.`” Alternatively, `variableNamed()` may be used to retrieve the variable containing the quantity instead of the `Quantity` itself. If the specified variable or quantity does not exist, these functions return `NULL`.

### 7.14.2.2 Cloning Variables

Use `cloneVariable()` to clone a `QuVariable` of unknown type. Similarly, `cloneQuantity()` clones a `Quantity`.

### 7.14.3 Tables and Table Cells

A `Table` is a 2D array data class containing `TableCells` (`Table.h`). Each `TableCell` contains a `Quantity` and a `TableCellFormat` that contains information needed to display the quantity. A `TableCellExpr` subclass contains a Python expression, and may return a scalar value or an array.

A `TableWidget` (`TableWidget.h`) is a subclass of `QTableWidget` that knows how to display and modify a `Table`. It is used by `ToolTable` (data class for `Table` tool) and by `ToolResults` (data class for `Results` tool).

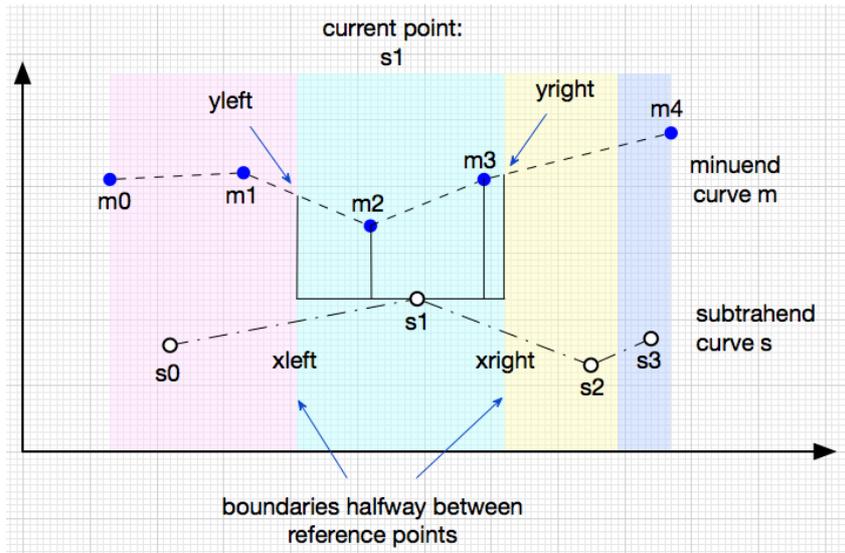
A `TableCellFormat` (`Table.h`) represents the formatting for *all possible* types of table cells, not just the current type of the cell. This is done so that a format change applied to a wide range of cells will take effect later if the data type of a cell changes. For example, setting the display mode for exponents for the entire spreadsheet is meaningless for strings or dates, but cells that later contain numeric values take on this latent property.

### 7.14.4 Subtracting Curves

This documents the algorithm used to implement the `QuPoints` “difference constructor,” which constructs a curve representing the difference between two curves, *m* (the minuend) and *s* (subtrahend). The curves aren’t mathematical, they are collections of points, which complicates the subtraction. The result of the subtraction is a curve constrained to the common ranges of the two operands, and contains a point for each point in the subtrahend (or, optionally, minuend) that lies within this range. We designate one curve as the *reference* and one as the *comparand*. The result curve has the same number of points as the reference curve, at the same *x* locations. The default, `QuPoints::MatchSubtrahendPoints`, uses the subtrahend as the reference. Or use `QuPoints::MatchMinuendPoints`.

#### 7.14.4.1 MatchSubtrahendPoints

In this example, the subtrahend is the reference.

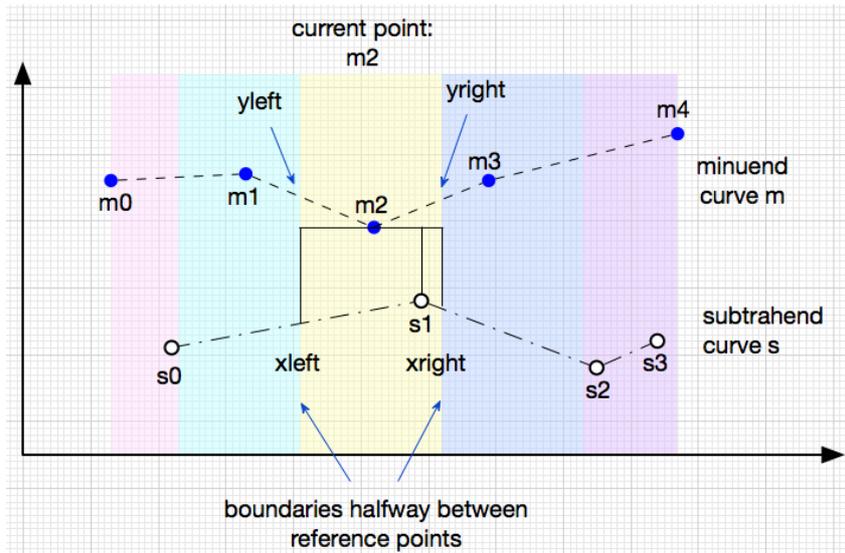


Steps:

1. Points outside of the common range are ignored.
2. For the sample point highlighted in the middle above,  $s_1$ , we wish to create a point  $d_1$  whose x location matches  $s_1$ , and whose y value is the difference between the curves. Locations  $x_{left}$ , halfway between the x locations of  $s_1$  and  $s_2$ ; and  $x_{right}$ , between  $s_2$  and  $s_3$ , are used as the range of interest (light blue above) within the curve  $m$ .
3. Zero or more points of  $m$  fall within this range, shown here by  $m_2$  and  $m_3$ . We calculate  $y_{left}$  as the intersection of  $x_{left}$  and the line segment connecting  $m_1$  and  $m_2$ ; and, analogously,  $y_{right}$ . Now we can subdivide the range of interest into a series of one or more trapezoids similar to the three shown here.
4. The area of the three trapezoids are summed and divided by the x range to get an average height at the x position of  $s_2$ , producing the y value for point  $d_2$ .
5. Repeat for all points in  $s$ .

### 7.14.4.2 MatchMinuendPoints

If you supply `QuPoints::MatchMinuendPoints` instead, the algorithm looks like this:



Here, the minuend serves as the reference curve and the subtrahend is the comparand. The vertical distance between  $m_2$  and  $s_1$  is approximated as the sum of the areas of the two trapezoids shown, divided by the width ( $x_{right} - x_{left}$ ). This produces a result with five points, one for each point in  $m$ ; instead of four, one for each point in  $s$ .

### 7.14.4.3 Residuals

There is another reason for designating a specific curve as the reference: the difference inherits any error values unchanged from the reference curve. This is needed, for example, to produce a residuals plot.

## 7.15 XML Files

This is a list of conventions followed for XML files such as saved notebooks. All files begin with this header and are in UTF-8 format:

```
<?xml version= "1.0" encoding="UTF-8"?>
```

Some common data types are stored as follows:

<i>Type</i>	<i>Example</i>	<i>Description</i>
Boolean	<code>open="false"</code>	Stored as "true" or "false" unless stored with context-specific names such as "enabled" or "disabled"
Color	<code>bgColor="#FFF00000"</code>	Stored as #hhhhhhhh, where h is a hex digit, and a missing alpha channel is assumed to be opaque
UU_ID	<code>spectrumID="UUSpectrumID6452f91c0079c42"</code>	Stored as a pair with data type and a hex UUID. A zero value indicates a null pointer
Enum	<code>refreshState="Idle"</code>	Stored by their enumerated name as assigned in <code>atsal.enum</code>
Date	<code>startDate="2017-02-10"</code>	Stored in ISO date format: YYYY-MM-DD
Timestamp	<code>modifyDate="2015-11-01T17:54:20Z"</code>	Stored in ISO date/time format: YYYY-MM-DDTHH:MM:SS[.FFFF]Z and are in UTC format. The 'T' is a literal, the 'F's are optional fractions of seconds, and the 'Z' is a literal implying UTC

Hexadecimal values are not case sensitive.

`QuDouble` classes include the positive and negative error values only if they are non-zero.

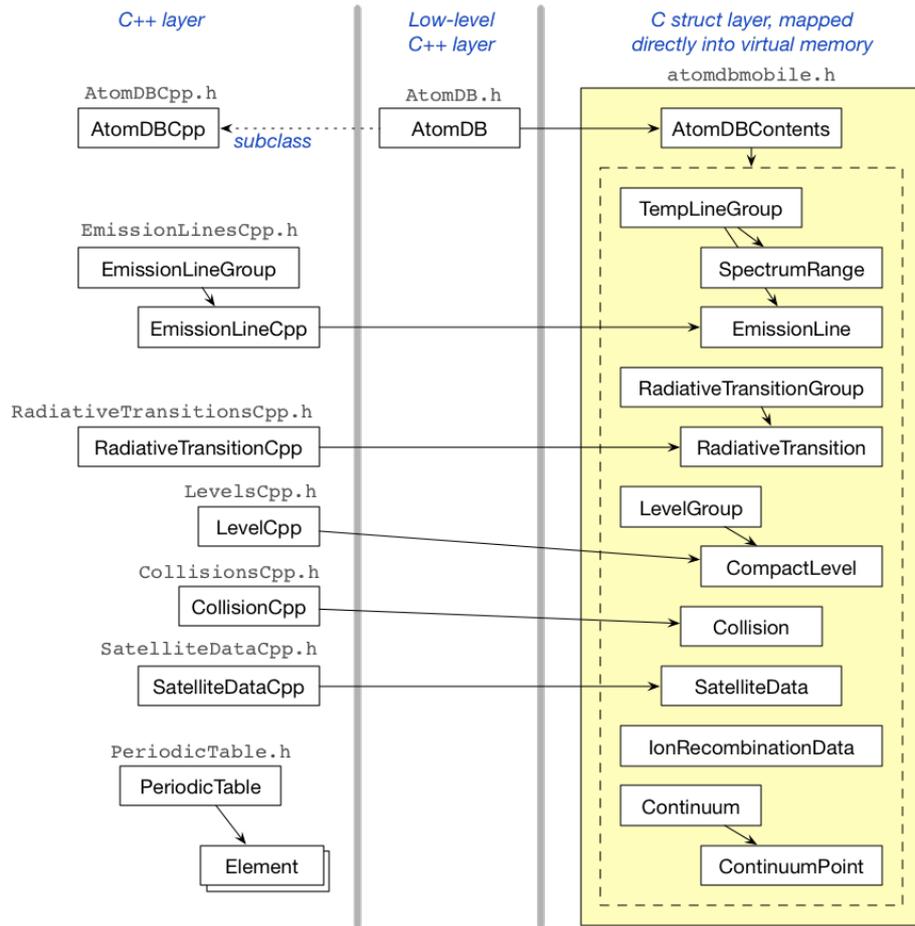
See `XMLUtilities.h`.

## 7.16 AtomDB

The iPad version of AtomDB underwent several rewrites in search of improved performance. The iPad application displays all the emission lines in a given temperature group, and I (Tom Kent) wanted to be able to adjust the temperature control and have the emission line display update nearly in realtime. To achieve this I abandoned Objective C, and also abandoned use of a MySQL database. Instead, I wrote a converter that produces a reasonably compact (~40 MB for this version of the database) amorphous blob of binary data, and I map the data directly into virtual memory. After adjusting offsets to be pointers, the strictly C-level interface, accessed via `AtomDBContents` in `atomdbmobile.h`, is available. This entry level, shown at right below, is as efficient as possible. It is a good starting point for accessing AtomDB from other languages as well.

In the iPad application, a higher level Objective-C interface, `NSAtomDB`, is used. This isn't shown here because ATSA is largely written in C++. Within ATSA, the next level interface is `AtomDB`, a thin C++ wrapper that provides some basic accessor functions.

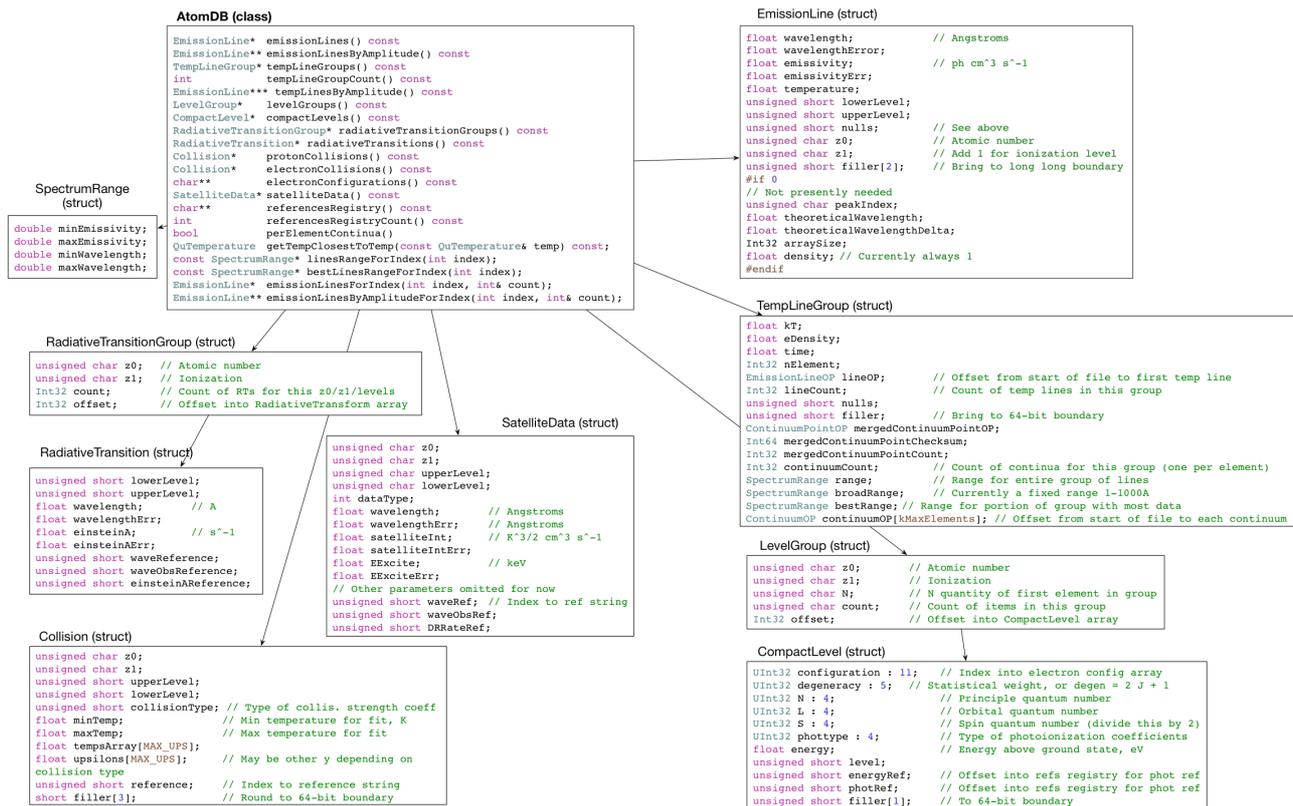
One level up, `AtomDBCpp` provides additional accessors which are written to provide a Qt-flavored interface to AtomDB. In addition to using familiar Qt container classes (well, familiar if you use Qt anyway), this level also ties in some of ATSA's physics classes. `AtomDBCpp` is a subclass of `AtomDB`, so it also provides access to the lower level functions. `AtomDBCpp` is not currently a full-featured C++ interface, though it could be extended into one quite easily.



Also at the C++ level are C++ classes that are wrapped around many of the lower level structures. These wrappers provide some convenience accessors. When arrays of these data items are returned, they are returned as a `QList<Foo>`, not `QList<Foo*>`, so they do not need to be deallocated. Since the arrays are constructed on the fly, performance-sensitive code is probably better off using the low-level interface.

**A warning about `LevelC`.** This class provides information about a level. Many emission lines have a level that is represented numerically as a value  $\geq 10000$ , and such levels (electrons thrown clear of the atom, if my less-than-perfect understanding is correct) do not have corresponding `CompactLevel` entries. But it is legal to construct a `LevelC` with one of these pseudo-levels. Such an object has a valid `z0()`, `z1()`, and `level()`. Many other calls return `-1`. `isValid()` returns `false` for these.

### 7.16.1 Low-level AtomDB Classes



## 7.17 The Refresh Cycle

This is a brief description of what happens when the user hits the refresh button in ATXSAL. A *refresh cycle* begins when the user executes the refresh command and ends when all tools report that they have finished execution. The cycle begins with a call to `prefresh()`, which decides which objects in the object hierarchy need to be recomputed to be brought up to date. This is followed by a call to `refreshLoop()`, which takes one pass through each tool's `refresh()` function (a *refresh iteration*). In the special case of the plot tool, results may not yet be available from XSPEC, so `refreshLoop()` sets a timer and is invoked repeatedly until all tools report they are finished. Only a single tool may be active at a time, because a tool may depend in some way upon the completion of a previous tool, so each refresh iteration passes control to the next tool in succession until all the tools complete.

Because a single shot `QTimer` is used to schedule the next refresh iteration in a refresh cycle, the main event loop receives control between each iteration. This allows the user interface to be updated, and allows full function of the user interface. (This is a mixed blessing, since many user interface functions need to be disabled during execution, to avoid potential crashes, but it allows the abort function to work, and a few other functions may be enabled.) Since the refresh iteration never blocks waiting for completion, the user interface remains fully responsive. (Exception: if user-written python code takes a long time to execute, it cannot be interrupted, so the user interface will lock up. But this should rarely be the case, since custom models are not implemented using ATXSAL's python, but are instead implemented at the XSPEC level.)

At the `NotebookWindow` level, `refresh()` kicks off the refresh cycle. It readies the python environment, then calls `prefresh()`. This function checks the modify state of all objects in the hierarchy associated with each tool, and moves any modified objects from the finished state to the idle state. Only idle objects are refreshed. Next, the toplevel refresh function calls `refreshLoop()`, which issues a timer to call it again every `kRefreshIntervalMs` milliseconds. Then `refreshLoop()` executes a python function, the only bit of python code that is built into ATXSAL, to perform the refresh. The python program passes the command on to each tool in the notebook. The default loop is trivial, but implemented in python so that advanced users can adjust it to their liking.

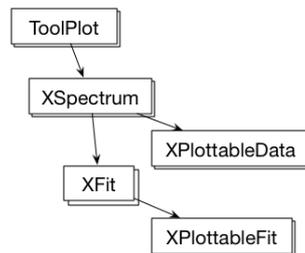
```

1 # By default, ATSAAL executes a notebook by calling each tool's
2 # refresh function until one returns false or they are all called.
3 # You can change this by altering this function as needed.
4 from PythonQt.private import PdbIO
5 import atsal
6
7 def refresh():
8     for i in range(notebook.toolCount()-1):
9         tool = notebook.toolFromIndex(i)
10        print(tool.toolName())
11        result = tool.refresh()
12        if result == at.RSFailed:
13            return
14        if result == at.RSActive: # tool busy, don't try any more
15            return
16        # Else move on to the next tool
17
18 # This ensures that ATSAAL's variables are available to the imported
19 # modules.
20 from __main__ import *
21

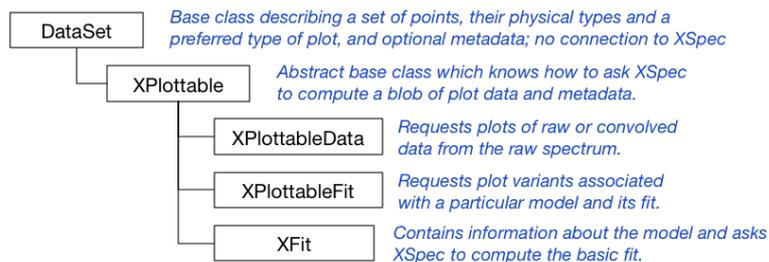
```

Most tools perform their refresh function immediately. For example, the table tool may substitute the values of some python variables into cells in the spreadsheet. Upon completion, they enter a state of success (or failure). The python refresh loop then moves onto the next tool in succession, or aborts if there was a failure.

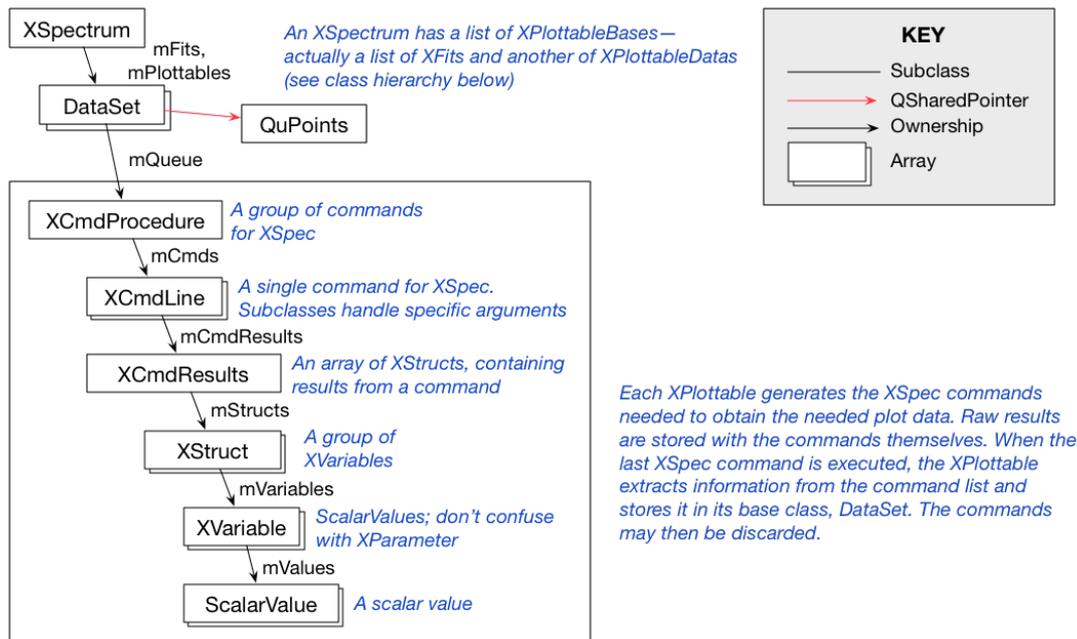
The plot tool behaves differently though. Each plot tool consists of some number of spectra (data files or data groups). Each spectrum consists of several `XPlottableData`s and several `XFit`s. Each `XFit` in turn may have multiple `XPlottableFit`s. Like this:



(Arrows above indicate ownership, and the doubled boxes indicate arrays of instances.) `XPlottableData`s, `XFit`s, and `XPlottableFit`s are all subclasses of `DataSet`. A `DataSet` is a set of points with some physical units information, and some optional associated metadata. While `DataSets` have no direct association with the means by which they are generated, `XPlottables` (subclass) do: they can generate the set of commands needed for `XSpec` or some other agent to produce results for them, and do something constructive with the results. The class hierarchy for this is shown below.

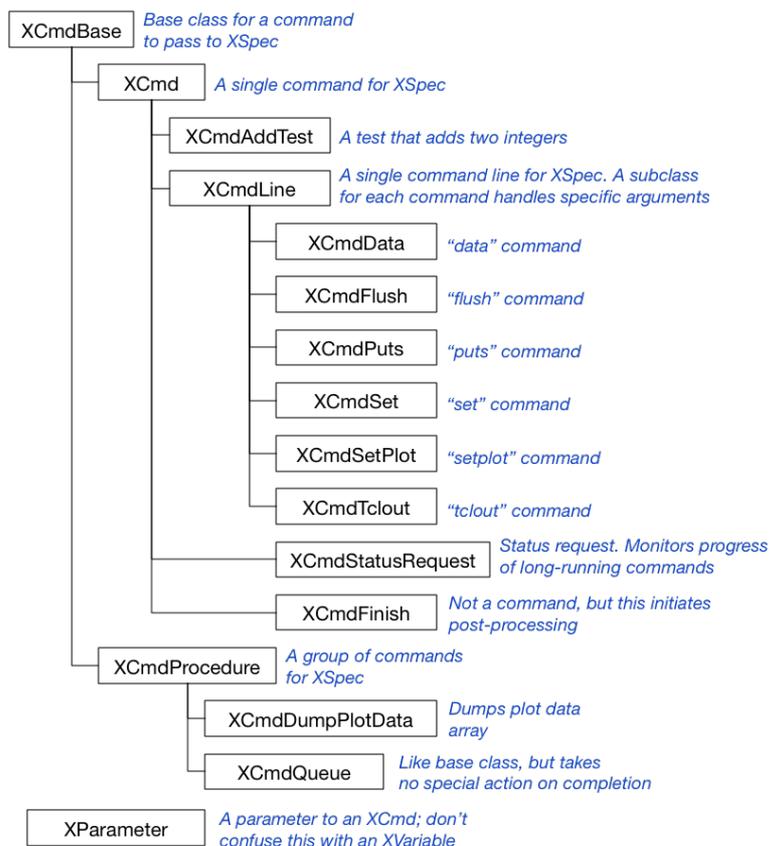


As the user works with the plot tool, they end up with a set of objects like this. Two `XPlottableData`s are created automatically, one each for the counts and the convolved data. The user adds additional `XFit`s and additional plots for those fits as needed. During refresh, each of these `DataSet`s generates sequences of commands and results as shown in the box.



When a refresh command arrives for the plot tool, it is passed to each X Spectrum in succession. The X Spectrum passes the message along to subordinates. For example, consider an XPlottableData that produces a plot of the counts data. Commands have corresponding classes. Most commands are specializations of XCmdLine, which generates a single XSpec command.

### XSpec Commands Classes



Commands issued from ATSAAL are sometimes a little more involved than their command line equivalents. For example, instead of plotting the result, we need to convince Tcl/Tk to give us the raw data.

```

data /Users/tkent/ATFiles/Obs/Observations/s54405.pha
setplot energy
Pr: dumpPlotData
  Pr: dumpPlotVector
    tclout plot counts x 1
    set atsal_temp $xspec_tclout
    puts server1 $atsal_temp
  Pr: dumpPlotVector
    tclout plot counts xerr 1
    set atsal_temp $xspec_tclout
    puts server1 $atsal_temp
  Pr: dumpPlotVector
    tclout plot counts y 1
    set atsal_temp $xspec_tclout
    puts server1 $atsal_temp
  Pr: dumpPlotVector
    tclout plot counts yerr 1
    set atsal_temp $xspec_tclout
    puts server1 $atsal_temp
# Finish

```

Lines beginning with “Pr:” represent procedures, analogous to their programming language counterparts but not quite the same: they are command sequences, not programs *per se*, at least not yet. To get a plot back from XSpec it is necessary to ask for each component (x, y, x error, and y error) as a separate vector of floating point numbers.

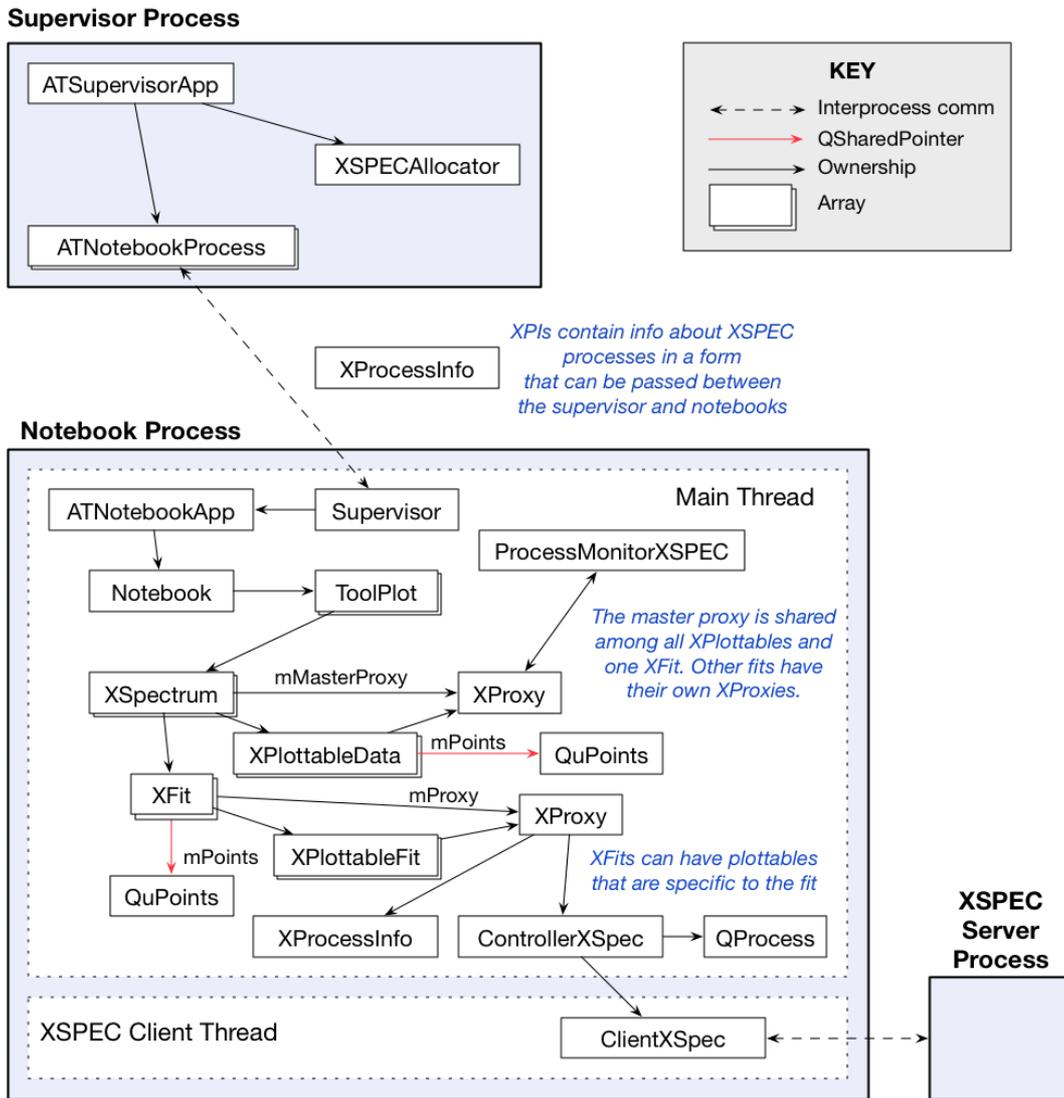
So the `XPlottableData` knows how to generate this sequence of commands. Results from each command are then parsed into generic form, and the `XPlottableData` cherry-picks the returned information of interest, then discarding the command sequence that generated it.

But there are some complicating factors. The first is that each fit gets its own XSpec server. It may not get the same server each time, if the pool of servers is smaller than the pool of clients. So it needs a way to know whether the server has changed. Second, it needs some way of dealing with a single “virtual XSpec server” for continuity. Hence, an `XProxy` class acts as a mediator between an XServer user and a particular XServer. One of the `XProxy`’s functions is to redirect log output so that a given fit appears to be in communication with a single XServer.

Another complication is that although there is an XServer per fit, there are other functions that need to be performed by a single XServer per spectrum, such as generating plots of the data. Who performs this work when no fits have yet been created? To resolve this, each `XFit` has an `XProxy`, but the `XSpectrum` also has an `XProxy`. Spectrum-level functions and the first fit share the spectrum’s “master proxy.”

Some of the major players in the refresh process are shown below:

## XSPEC Command Processing



The refresh request causes each spectrum and plottable to generate its own series of commands, which are passed to the appropriate `XProxy` in preparation for execution. The `XProxy` has been assigned an `XServer` that is guaranteed to stick around *only for the duration of this refresh cycle*. (The supervisor process assigned it, so that it can dole out available servers at a level global to all the notebooks. This means that `XServers` gradually migrate to whichever notebooks need them.)

Each spectrum component then returns active status if it has work to do, or success or failure if it is finished or something is wrong. If at least one component is active, `ATSAL` then starts the `ControllerXSpec` actually executing the command sequence. Commands are actually sent to the server from a client thread so that other operations may occur in parallel.

As results come back from the server, the `ControllerXSpec` passes them to the `XParser` subsystem (not shown above) for parsing. This system consults XML parse tables, essentially patterns matched against the output, using them to pick out information of interest. This example is the parse table entry that parses XSpec's data command:

```
<command name="data ...">
<block>
***Warning: Unrecognized grouping for channel(s). It/they will be reset to 1.
<bool name="unrecChannelGroup" value="true" />
</block>
<block>
```

```

HOWEVER NOTE: Resets only occurred for channels of bad quality
</block>

<block>
***Warning: POISSERR keyword is missing or of wrong format, <words />
<bool name="POISSERRMissing" value="true" />
</block>

<block name="tlmin">
***Warning: No TLMIN keyword value for response matrix <word name="matrix" /> column.\n<w
/>Will assume TLMIN = <int name="tlmin" />.
<bool name="TLMINMissing" value="true" />
</block>

<block>
Note: CHANTYPE keyword not found in the rmf file
<bool name="CHANTYPEMissing" value="true" />
</block>

<block>
<int name="spectrumCount" /> spectrum<w />in use
</block>

<block name="specInfo1">
Spectral Data File: <filename name="file" pattern="*.pha" /><w />Spectrum <int name="specnum"
/>
</block>

<block name="specInfo2">
Net count rate (cts/s) for Spectrum:<int name="spectrum" /><w /><double name="countRate" /> +/-
<double name="variance" />
</block>

<!-- Example of regexp
<block name="foobar">
Hello this is a <regexp name="myPattern" pattern="[A-Za-z]*[0-9]*<Q" /> the end
</block>
-->

<block name="dataGroup">
Assigned to Data Group <int name="dataGroup" /> and Plot Group <int name="plotGroup" />\n<w
/>Noticed Channels:<w /><int name="startChannel" />-<posint name="endChannel" />
</block>

<block name="telescope">
Telescope: <word name="telescope" /> Instrument: <word name="instrument" /><w />Channel Type:
<word name="channelType" />\n<w />Exposure Time: <double name="exposureTime" /> sec
</block>

<block name="fit">
Using fit statistic: <word name="fitStatistic" />\n
<w />Using test statistic: <word name="testStatistic" />\n
<w />Using Response (RMF) File<w /><filename name="responseFile" pattern="*.rsp" /> for Source
<int name="source" />
</block>

</command>

```

This info is stored in generic (i.e. not-especially-efficient) form in the commands themselves. An `XCmdFinish` is a pseudo command that doesn't tell the `XServer` to do anything, but signals completion of a command chain. When an `XCmdFinish` is received, the controller delivers a signal to the `DataSet` subclass, in this case the one that deals with acquiring counts data to plot.

`XCmdFinish` is special in another sense: it defines a precise moment when the data for a command block has been received and parsed, and moved into the `DataSet`. Since this is an atomic operation, if the user issues an abort, there is no ambiguity as to whether a given subtask is complete. Prior to the finish command, `ATSAL` is blissfully ignorant of the results data, which remains localized in the commands themselves. After the finish, `ATSAL` has the results. This means that an abort can be issued at any time without leaving the system in an undefined state. Abort simply interrupts the `XServer`, and discards all commands. Any `DataSets` that are completed have reliable data.

The `XPlottableData` class simply pulls out a pointer to the `QuPoints`, the object that contains the actual data, and keeps it for itself. (The pointer is shared via `QSharedPointer`, so `QuPoints`, which are fairly bulky, are not normally

copied.) All the rest of the commands and their results are of no interest in this case, so we can discard the completed command chain. Oh, and signal the plot tool editor that this plot's data changed, so it will get (re)drawn.

We only want to ask XServer to do work it hasn't done already. So each `DataSet` subclass contains logic to compare its current state to its previous state. For the counts example, the commands to retrieve the plot data are issued only if this is the first time, or the input file changed. For a fit, the logic needs to take all the parameters and the model expression into account. This logic is complicated by the fact that the server may change and the master proxy is shared by several components.

There is another significant complication here. The refresh process cannot wait around for the commands to complete, because this locks the entire user interface. Although most of the user interface *should* be locked (in this notebook, that is), we can't lock the whole thing, or the user wouldn't even be able to hit the abort button. So when a spectrum completes execution of its refresh function, it returns an indication as to whether it is still active.

If we move up a couple of levels into the python refresh loop, we get back an active state from a tool if any component is active, and pass this back out of python to the notebook. Now we're back in ATsAL's main event loop: we aren't locked. If the status is active, we set a timer, and check again in a few dozens of milliseconds. This means everything else works: we can handle signals if, say, an XServer crashes, or the user aborts, or whatever is necessary.

This is mostly good, except that it means that most of the user interface will have to be artificially disabled in order to prevent changes to the system that would put things in an inconsistent state.

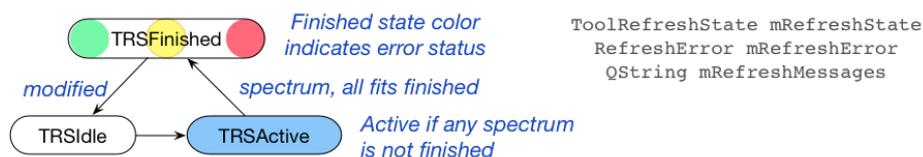
### 7.17.1 Refresh State Tables

In several tools, such as the tag tool, refresh is a no-op. In others, such as the table tool, refresh may involve evaluating some python expressions, a process that is usually instantaneous from the user's perspective. In the plot tool, though, refresh is more complex, involving a hierarchy of objects and one or more XSpec servers.

Refresh is handled via several nested *refresh state tables*. Each state table has its own states, despite the large degree of overlap. (I tried using a single universal set of refresh states, ignoring the irrelevant ones in particular state tables, but this caused more problems than it resolved. By defining a separate set of states for each state table, the code's operation is easier to follow, and the total number of possible state transitions is reduced.) Each state table operates any subordinate state tables. Each state table is represented by an "LED" in the user interface—a compact way to represent state by color, report errors at each level via tooltip, and in some cases, support extra options, such as aborting a particular XSpec operation without disturbing others in progress at the same time.

A refresh state is represented by a state, an error status, and an error message which appears as a tooltip. The error status, `RefreshError`, is one of `RESuccess`, `REWarn`, or `REFail`, representing the most severe error to occur at this or subordinate levels.

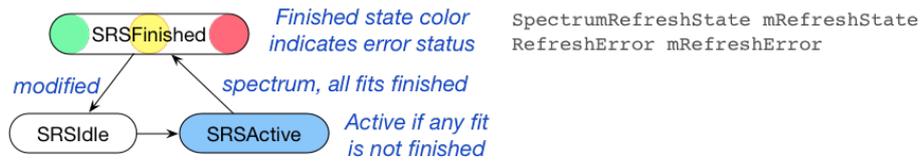
At the tool level, the state table (`ToolRefreshState`) is very simple. Each tool's state table looks like this:



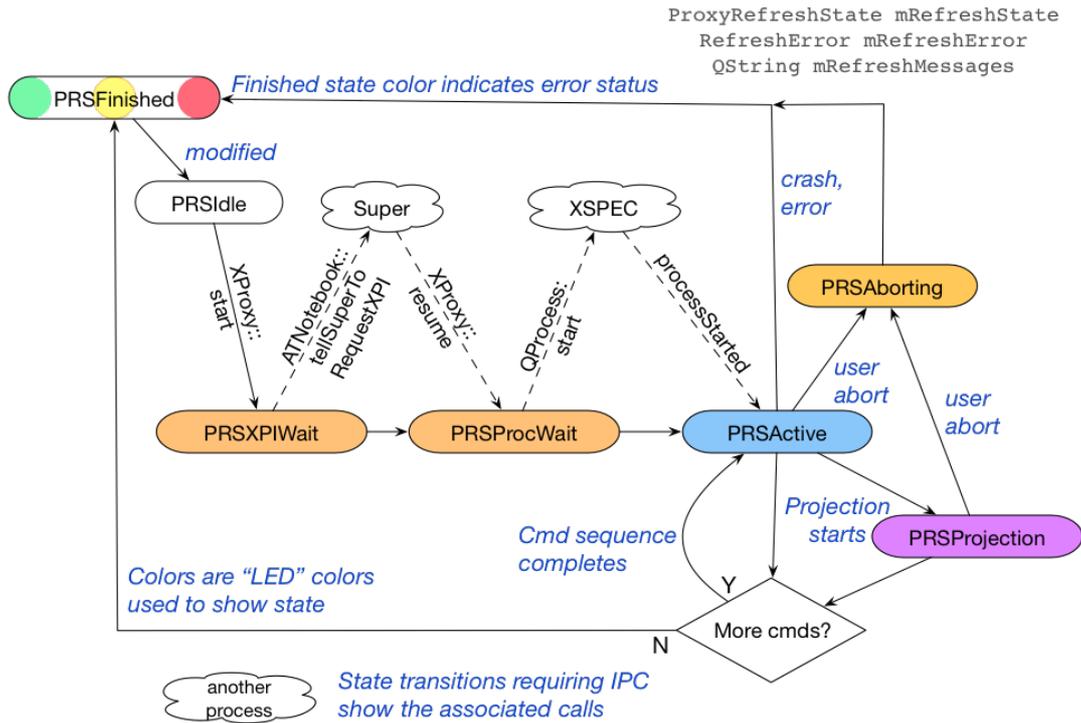
The initial state is `TRSIIdle`, which appears as an uncolored LED. When the user issues a refresh command, the state becomes `TRSAActive`, appearing blue. When the tool refresh completes, the state becomes `TRSFinished`; only in this state does the LED color reflect the `RefreshError` status. The tool remains in state `TRSFinished` until the user modifies some aspect of the tool's settings via the user interface. The modification means that any displayed output from the tool may be out of date, so it provides subtle feedback that another refresh command is needed to bring things up to date.

#### 7.17.1.1 Plot Tool

This is where the plot thickens. (Sorry.) Plots have multiple spectra. Each spectrum has its own state table, `SpectrumRefreshState`:



This is virtually identical to the tool state table, but may be extended later, so it is kept separate. Each spectrum in turn has plots of the data as well as plots of one or more fits. The first fit's `xProxy` is shared with the plots of the spectrum data, so there is a state table corresponding to each `xProxy` rather than each `xPlottable`. (The `xProxy` mediates between consumers of XSpec data and a particular XSpec server; the server may change on each refresh cycle.) The `proxyRefreshState` table looks like this:



This table has a few extra states. `PRSPiWait` is entered when waiting for the supervisor to assign an Xspec process. `PRSProcWait` is entered when waiting for the XSpec server to start up. All the ancillary wait states—that is, wait states other than that for XSpec command completion—are denoted with orange. These states are typically short in duration. The only wait condition with a potentially long duration is when the pool of servers has been exhausted, so orange means “out of XSpec servers” to the user. `PRSProjection` is entered when a fit is calculated but a projection is being calculated. The projection calculation is started automatically, regardless of whether the user requested it, so signalling this state back to the user provides an option to abandon the projection if it isn't wanted. Finally, if the user aborts a refresh cycle, the proxy enters state `PRSAborting` while waiting for XSpec to be ready for new commands. An XSpec crash also enters this state: a crash is similar to a user abort, and the supervisor will assign a new server instance on the next refresh.

Here is the proxy refresh process:

- `PRSPiWait` is the starting, or idle state. `XProxy::refresh()` requests the supervisor to assign an `XProcessInfo` (XPI), and advances the state to `PRSPiWait`. (The dashed lines and “clouds” indicate asynchronous communication with other processes, not state transitions.) The `xProxy` remains in this state until the supervisor has a free XSPEC instance to assign, or the user aborts the refresh cycle.
- The supervisor (eventually) responds with a call to `resume()`. This call checks to see if it has been assigned an XSPEC process that is already running. If so, it calls `resumeProcess()`, which enters state `PRSActive` and begins processing the commands. If not, it creates a `QProcess` and instantiates a new XSPEC server. When the process instantiates and finishes initialization, `QProcess` delivers a `processStarted` signal, and the state becomes `PRSActive`.

- The `ControllerXSpec` processes the already-queued commands that make up a refresh unit, remaining in state `PRSActive` for as long as the queue contains commands. When the command queue is exhausted, the `XProxy` transitions to `PRSFinished`. `PRSFinished` is almost identical to `PRSIIdle`, except that it indicates that this part of a refresh cycle is completed, so `ATSAL` can move on to the next portion. An error status is set to a `RefreshError` (currently `RESuccess`, `REFail`, or `REWarn`), and a text description is saved. The state indicator “LED” color is set to green, red, or yellow, and the user can hover over the LED to find the cause of the error. Command processing *errors* interrupt processing and also enter the `PRSFinished` state with an error indication, while command processing *warnings* continue processing commands, but signal a warning status upon final completion.
- If a projection was requested, and `XSPEC` begins computing it, the state advances from `PRSActive` to `PRSProjection`. This state is treated as a special case because the user often does not need a projection, so it is initiated but signaled with a different color so the user knows it can be aborted. From the refresh cycle’s perspective, the command is finished at this point, and the user interface becomes available for other purposes. This is treated as a special subcase because it continues (almost) transparent to the user—except for an activity indicator, and the fact that the `XSPEC` process is still tied up, the user can perform other actions. For example, if a new refresh cycle is performed, the projection is automatically cancelled. (I think... In order to free up the user interface while a projection is underway, `ATSAL` must prevent user actions that would lead to inconsistent state, or quietly cancel the projection. The former case is fairly complicated to implement, requiring that a good-sized number of specific actions be prevented, such as deleting the model being calculated. If the projection is automatically cancelled, a long-running calculation could be aborted by accident.)
- If the user aborts, or a serious error occurs (e.g. an `XSPEC` crash), `XProxy` enters `PRSAborted`. In the event of a user abort, this sends the equivalent of a Ctrl-C to `XSpec`. In the event of a crash, it informs the supervisor that the process is gone, so it can assign a new one on the next refresh cycle.
- The fit remains in the `PRSFinished` state until the user changes some input to the fit, such as a model expression or parameter. This modification means that user interface is no longer up-to-date, so the state transitions to `PRSIIdle`. The LED goes off, indicating that a new refresh is needed.

### 7.17.1.2 Aborting a Refresh

Aborting a refresh causes clean-up that differs depending upon the current state. Although not shown in the state table to reduce clutter, an abort can occur at any point in the state table.

- `PRSIIdle`, `PRSFinished`: no effect (the command is disabled in these states).
- `RSXPiWait`: notify supervisor that we are canceling request for XPI, then enter state `PRSAborting`. There are two cases: (1) supervisor receives request prior to allocating an XPI, in which case the supervisor cancels the request and sends back a request cancelled indication. (2) Supervisor has already allocated an XPI, and the `XProxy` receives this allocation while in `PRSAborting`. In this case, the `XProxy` returns the XPI to the supervisor. In both cases, the supervisor acknowledges, and the `XProxy` then transitions to state `RSFFinished`. Since the abort was requested, the status is `RESuccess`.
- `PRSProcWait`: enter `PRSAborting`. When the process finishes initializing, move to state `PRSFinished`. The process remains available for subsequent use.
- `PRSActive`, `PRSProjection`: this clears the pending commands queue, then sends a signal to the `XSPEC` process to interrupt it, the equivalent of a Ctrl-C. The `XProxy` enters state `PRXPSAborting` and sets a timer to give the `XSPEC` process some time to recover its composure. When the timer expires, we enter state `PRSFinished`.
- `PRSAborting`: An impatient user! This is ignored.

### 7.17.1.3 Normal Process Exit

If the supervisor assigns an `XSPEC` server to a different notebook, the original notebook receives a normal termination message and shuts down the process to free resources. Such a notification occurs only if a refresh is not in progress, so it does not effect the proxy state table. It simply marks the proxy as not having a process assigned.

### 7.17.1.4 XSPEC Crashes

If an `XSPEC` process terminates abnormally, `ATSAL`’s `QProcess` emits a `processError` signal that is wired to the `ControllerXSpec`. This is re-emitted as `XSPECError`, and passed to the `XProxy`, trapped by `slot processError`. This process calls `ControllerXSpec::shutdown`, which calls the thread’s `exit` function to shut down the client thread. Next, the `XProxy` emits `serverTerminated`, which notifies the owning `PrvToolPlot` of the disappearance. The plot

tool passes the message to each of the spectra, which in turn pass it to all the plottables. Each plottable bumps its `mSomethingChanged` counter if it is already modified, to preserve the modify state for the next refresh cycle.

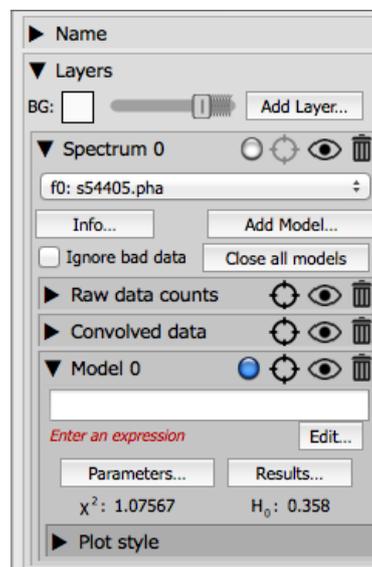
Now that everyone has been notified of the crash, the plot tool calls `proxy->deleteProcess`. The process itself is already gone, but this deallocates the `ControllerXSpec`, which will cause a new server to be allocated on the next refresh cycle.

Recovering from a crash is a little like stopping a tennis game by removing one player at a time, with the proviso that if the ball is not returned, the universe explodes. Oh, and you can't just stop the ball first! There is probably a simpler way to do this...

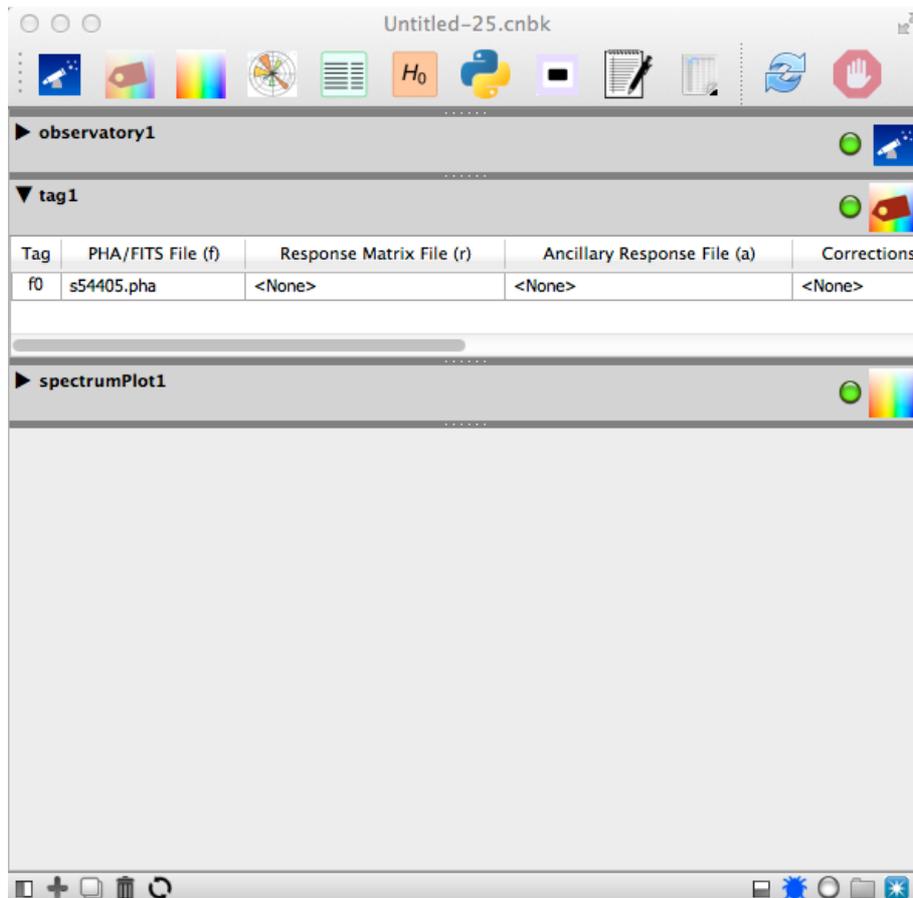
### 7.17.1.5 Error Propagation

Each state table accumulates its own error messages via `mergeRefreshError()`, which appends any new error message to an existing list and promotes the severity level to the worst of the current and previous levels. These error messages become the tooltip for the level. At the next level above, any errors are propagated upward by setting the `RefreshError`, but the messages are not; instead, the tooltip indicates the presence of lower level errors. Since errors propagate upward, users can see an indication of any failures even if the notebook is the only visible window.

For example, LEDs are shown below for a spectrum and a model.



The model's LED is blue, indicating an active state. The spectrum's should be blue as well, but the code isn't implemented yet. Even if the plot tool editor is closed, its status is summarized in the notebook, where it should also be blue:



### 7.17.1.6 MulticolorLEDs

MulticolorLEDs are used, among other things, to convey information about the states described above to the user. (These are defined in `ProcessLEDs.h`.) Each of the states described above, and their corresponding error status values, are mapped to `MonitoredProcessStates`, whose associated colors are shown in the state diagrams. Tooltips show any error messages. The proxy LEDs are also active controls. Clicking one opens its XSPEC console in the notebook window. Control-clicking may display a menu with other options, such as aborting one XSPEC server without interrupting other busy servers.

Since there are a lot of process monitoring LEDs, and their state may change at high frequency, it is not efficient to emit a signal on each state change in order to update the user interface. It doesn't interact well with the blinking, either. So instead, a single timer emits a signal to the notebook at regular intervals (about 2 Hz), and this signal is passed to all the LEDs. Each LED queries and updates its state if needed. This is much more efficient. A blink counter is passed to each LED, so that if the LED is in blink mode, it chooses its state based on whether the count is odd or even. This causes all LEDs to blink in unison.

### 7.17.1.7 Modify States

Any plottable, spectrum, or tool has a modify state (`ModifyState`), which may be `MSNotModified`, `MSModified`, or sometimes `MSChildModified`. `MSChildModified` is set when a given level in a hierarchy is not modified, but one or more subordinates are modified. An object is considered modified if a refresh command is needed to bring the user interface up to date. So, for example, if the python program contained in a python tool is changed, a refresh is needed to make it run. If a spectrum's input file(s) changes, or a model expression or parameters change, a refresh is needed. On the other hand, if the user changes the units for the horizontal axis of a plot, or selects a thicker line style, this change takes effect immediately, so it is not considered to be a modification.

As a rule at least, modification of a particular object is implemented with a `modifyState()` function. The `modifyState()` function compares the current state of the portion of an object that would require refresh against a copy of the state as it existed as of the last baseline. `resetModified()` is implemented to update this copy. ATLAS calls

`resetModified()` only after completing a refresh cycle. `isRefreshEquivalent()` is the function that compares the current object against its last saved state, named to emphasize that this is specific to changes requiring refresh.

If the modify state is either `MSModified` or `MSChildModified`, a dark dot appears in the middle of the LED, mimicking the Mac OS X convention for showing a modified file.

The text tool and table tool may include python expressions; if so, they are re-evaluated only at refresh time. However, since there is no practical way to determine whether they need re-evaluation, these tools are never shown as modified, but they are always re-evaluated at refresh time. Changes to python programs are also not represented by the refresh mechanism, because ATSA does not currently implement any kind of a dependency mechanism to determine which changes would require a refresh. So python code changes are mostly exempt from the modification checking process. This means that if the user is performing a time-intensive python task, they may need to implement their own modify checking to avoid repeated evaluations on each refresh.

## 7.17.2 Refresh Phases

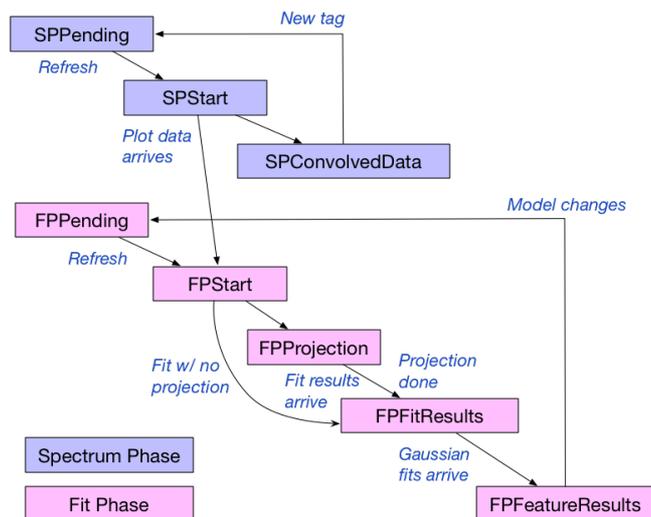
We use the term *refresh states* to describe the clockwork that manages sending commands to XSpec and processing the results. In the active refresh state (`SRSActive` for spectra, `PRSActive` for fits), we refer as well to *refresh phases*. The latter are a state table within a state table—we use “phases” to distinguish them. While refresh states have meaning to the user and are reflected back in the form of process LEDs, refresh phases are internal transitions important for kicking off new activity of some kind. By way of analogy, if states are ticks of a clock, phases are activities, such as advancing the date to the next day or sounding an alarm, that occur infrequently on certain ticks.

The need for phases evolved initially for line-based analysis—originally they were called `LBASStates`. They have now been generalized to spectrum phases (`SpectrumPhase`) and fit phases (`FitPhase`). At first, they were considered to be a simple serial progression. But several newer issues demonstrated the need for a more general approach:

- For LBA, a convolved spectrum is needed before peak finding can take place. Hence no fits should be performed prior to the arrival of this data. The prior design kicked off all fits at the same time, without waiting for the convolved data.
- Also for LBA, after the non-feature portion of the model has been fit, the fit must be re-run, using locked parameters from the first fit in conjunction with the Gaussians needed for emission lines. This adds another step.
- Since fits can run for days, it is important to lock out changes that could crash a fit in progress, as well as to be able to add and configure new spectra and models during a refresh cycle. Hence some means is needed to keep these new spectra and models apart from the refresh cycle until the user finishes configuring them.
- Other plots derived from existing data, e.g. residuals plots, cannot be calculated until the prerequisite data arrives.

These constraints imply that the refresh phases form a dependency tree (i.e. branching structure), not just a linear set of serial events. Branching may mirror the hierarchy of objects involved in an XSpec analysis: spectra contain “plottables” and fits, and fits contain “plottable fits.” Branching could also occur when more than one independent chain of events result from a configuration change, although I haven’t seen any cases of this yet. Each time a phase completes, we check to see if there are any other phases that need to be performed.

As with refresh states, refresh phases are arranged to minimize the amount of new computation needed for each refresh cycle. If the user changes a temperature line group for LBA, clearly we don’t need to recompute the convolved data, or the model. But if XSpec crashes, we must recalculate the associated model in order to return to the pre-crash state.



Phases are as follows:

1. `SPPending`→`SPStart`. A newly created spectrum begins in state `SPPending`. In this state, it is ignored by any refresh loop that is already in progress. This allows the user to configure the spectrum and model(s) during a refresh cycle, then hit refresh again to include the new spectrum in the refresh loop. So the refresh command moves to `SPStart`. This phase prepares the command sequence needed to obtain counts and convolved data, and begins processing them. Any subordinate fits in state `FPPending` are also moved to `FPStart`.
2. `SPStart`→`SPConvolvedData`. Occurs when convolved data is successfully received. If an error occurs, the state transitions to `SPPending`. At this point, any subordinate fits in the `FPStart` state are fired up: commands are generated for them and their proxies are activated. The reason we wait for the convolved data is that if LBA is enabled, we need convolved data before we can request Gaussians.
3. `FPPending`→`FPStart`. This transition occurs when a new refresh cycle is started. The newly configured fit is added to the list of current activities for the refresh cycle. Commands are generated for the fit (but not for the features portion, if any), and the proxy is started.
4. `FPStart`→`FPProjection`. Commencing a projection calculation is classified as a state, not a phase, because it is a user-relevant change (it turns the LED purple). But its position in the phases is shown here. If the user aborts a projection, the phase changes to `FPFitResults`, not `FPPending`. If the model includes features, this computation begins.
5. `FPProjection`→`FPModelResults`. When the model command executes partway through the fit refresh cycle, this phase is entered, and the results are copied into the fit. If a projection was cancelled, this phase is also entered. Upon entry to this phase, additional commands are issued to retrieve the fit plot data. If an error occurs, the phase returns to `FPPending`.
6. `FPModelResults`→`FPFitResults`. This phase is entered when the fit data become available. They are copied into the fit and displayed. If an error occurs, the state reverts to `FPPending`. If the model includes features, generate commands to perform the fit again, this time with locked parameters from before, and a set of Gaussians. Changes to the model or any fit parameters revert to state `FPPending`.
7. `FPFitResults`→`FPFeatureResults`. Occurs when results for fit and Gaussians become available. Emission line labels are computed and displayed for the results. Changes to the model or any fit parameters revert to state `FPPending`.
8. `FPFeatureResults`→`FPFitResults`. Occurs when the user modifies any LBA selection parameter that requires recalculation of the features Gaussians (e.g. change in peak sensitivity). This transition ensures that the next refresh command will recompute the features Gaussians.

A system-wide abort simply leaves everything in its current state, discarding incomplete results but preserving previously complete results. Aborting a specific fit does the same thing for that fit only. **Probably a good idea to add a Reset option to each spectrum and model menu. This reverts the state to SP/FPPending, forcing a recalculation. This would also send a ^C to the XSPEC.**

### 7.17.3 Refresh Loop (Old)

This is superseded by a newer write-up, but still of some interest.

The Refresh button simply passes a refresh call to each tool in succession. This is actually performed by a short Python program, as a starting point for more advanced programs. Before enabling the refresh button, a call is first made to `refreshState()`. The button is enabled only if all tools report quiescent (`RSQuiescent`) state.

### 7.17.3.1 Concurrency

ATSAL uses several techniques to preserve responsiveness while it is busy with a computation. First, ATSal monitors XSPEC with a thread for each XSPEC server, so the progress monitoring does not cause ATSal to block. Second, each notebook runs as a separate process, so that Python execution can occur in parallel in multiple notebooks. Third, the refresh loop does not block at a busy XSPEC; instead, it sets the status of the tool to `RSActive` and returns to the event loop. The refresh loop is said to be active if any subtask in a tool is active. When a refresh loop returns active status, ATSal sets a timer which will cause its status to be rechecked repeatedly, until the loop either completes execution or stops due to an error. When the loop is in completed status (either `RSSucceeded` or `RSFailed`), it remains there until ATSal explicitly resets it back to `RSQuiescent` via `resetRefreshStatus()`.

ATSAL can still block, but only if user-written Python code is compute-intensive. As long as Python is busy, the user interface is locked out, except for handling an abort. In most cases this isn't a problem, because any compute-intensive Python, such as user-written models, probably executes under XSPEC's Python implementation, not ATSal's. When this is the case, ATSal does not block. Most notebook operations are prohibited during these long computations, but some view-only operations are allowed. For example, a user might open different plot tools during execution to view progress.

ATSAL provides execution feedback by showing an activity icon in the notebook's currently executing tool. In addition, there is an activity LED for each model. The LED indicates inactive, active, calculating a projection, or error.

#### 7.17.3.1.1 Plot Tool Command Completion

An `XSpectrum` or an `XFit` generate a series of `xcmds` to carry out various operations. Results from each command are buffered in the commands themselves until an `xcmdDone` is encountered. When the `ControllerXSpec` sees an `xcmdDone`, it calls its owning `XProxy`'s `procedureHasCompleted` function, and this emits a `procedureCompleted` signal. This signal cannot be connected directly to the `XSpectrum` or `XFit`, because these are not `QObject` subclasses (and there is a reason for this, having to do with a prohibition on copying `QObject`s). So instead the `ToolPlot` receives the `procedureCompleted` signal.

Note that most signals are delivered to the `ToolEditorPlot`, not the `ToolPlot`. This is because the only place most user interface signals can originate from is the plot tool editor, so it is known to be open at the time the signal is received. But a completed command might have been configured by an earlier ATSal session, and this signal might arrive when a plot editor has not yet been opened. So the signal goes to the tool, which forwards it as needed.

### 7.17.3.2 Parallelism, One Step Finer

Suppose a notebook contains two spectrum plot tools. The first such tool contains two models and the second another two models. What degree of parallelism occurs? As currently designed, an ATSal notebook is also a program: the sequence of steps performed in the notebook must actually execute in the same order, to avoid issues like using a result that has not yet been calculated. So although there are a total of four models to fit, and, we will assume, enough XSPEC servers to work on all of them at the same time, only two will be active at once. *ATSAL does not execute a tool until all the subtasks in previous tools have completed execution.*

#### 7.17.3.2.1 Projections

As currently planned, projections will be an exception to this rule. If a projection is requested, the tool is considered to have executed successfully after fit results become available, and the projection continues asynchronously. When its results become available later, they are integrated into the tool status. This means the user interface is available for other uses during projections, so that, for example, the model being computed could be deleted from the notebook. Hence there is special case logic to shut down the projection if the user interface state changes in a way that renders it useless.

### 7.17.3.3 Keeping ATSal Busy

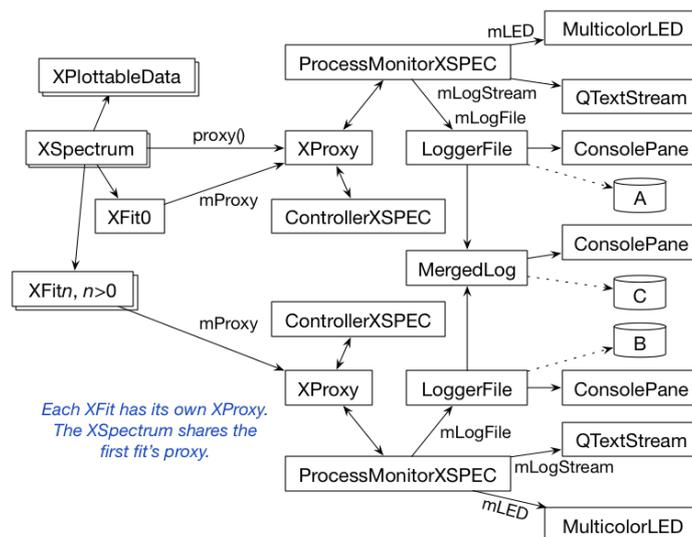
One mode of operation that might prove very useful is to start a notebook executing, then duplicate it. Set the duplicated notebook's settings differently and hit refresh. Repeat as needed. In this way, many variations of an analysis may be tried in parallel. As a more advanced use, the Python main loop program might be modified to activate multiple plot tools at once, computing a dozen or more models simultaneously (provided that there are enough XSPEC servers available). In any situation where the demand for servers exceeds the supply, ATSal automatically reassigns servers to new models ASAP.

### 7.17.4 Process Monitoring and Logging

Each `XFit` has an associated `XProxy`, which acts as an intermediary between the fit and the particular instance of XSPEC that calculates it. At the `Xspectrum` level, there are also tasks to perform. Some spectrum-level commands need be issued only by one `XProxy`, such as those that obtain plots of the spectrum data. Since this information can be obtained quickly, there is no reason to assign an entire XSPEC instance for this purpose. Instead, we share the first fit's `XProxy` with the spectrum. This works fine unless there aren't any fits yet, in which case there is no proxy either. Working around this led to the deprecated design described shortly.

To simplify matters, I added the constraint that each spectrum *must have a minimum of one XFit*, with which it shares a proxy. The model expression for the fit may be empty, but the `XFit` has to exist. This solves several problems. It provides a logical place for the LED that represents XSPEC status and doubles as a button for displaying the console log or aborting XSPEC execution. It avoids awkward boundary cases like dormant `XProxies` and reassigned status LEDs. And since the user will need to create a fit anyway, it saves the user a step. By implication, the last fit cannot be deleted.

The simplified approach looks like this:



This leads to the following cases:

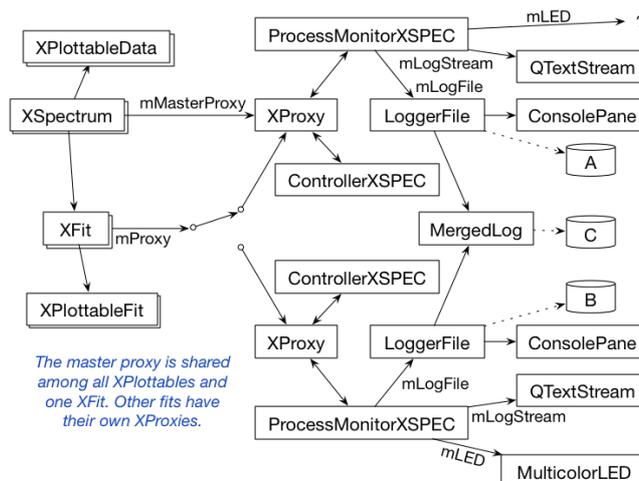
- Add a new spectrum. This creates a spectrum and a fit. The spectrum shares its proxy with the fit.
- Add another fit. This appends a fit with its own proxy.
- Delete the last fit. This is not allowed.
- Delete a fit when there are two or more fits remaining. If the deleted fit is the first, the spectrum's proxy and the first remaining fit's proxy are shared. If the deleted fit is not the first, there is no change.

Console windows are named using sequence-numbering. For example, the console window for plot tool plot3, spectrum 0, fit 4 is "XSPEC plot3 0.4 (0E29)". The parenthesized value is the low four hexits of the owning object's UUID, mostly a debugging aid. These names change as spectra and fits are added and removed, except that the partial UUID remains invariant. The logfiles are named using only the low four hexits of the owning object's UUID, so the names can remain invariant during a session. So the corresponding file would be named "XSPEC 0E29.10g". The merged log is simply `XSPeCs.10g`. It contains merged output from XSPEC sessions for all tools and fits.

### 7.17.4.1 Deprecated (Overly Complicated) Approach

This diagram shows classes associated with spectra and fits that are involved in process monitoring and logging. This would be straightforward if were not for the issue of sharing XProxys among consumers. An XSpectrum has a *master proxy* which is used to obtain information about the raw spectrum, such as counts plots and convolved data plots. The master proxy has its own XSPEC instance (ControllerXSPEC), and its own ProcessMonitorXSPEC. The monitor directs logging information to a console pane, a merged log, and a logfile. The proxy's job is to make an XSPEC session look continuous to the user (from a status and logging point of view) even if the associated ControllerXSPEC changes. It also keeps enough state information for the spectrum and fit to calculate the minimum set of XSPEC commands needed to bring the system up to date.

The spectrum's master proxy is shared with the first fit. This is done because spectrum-level plots take almost no time to generate, so there is no sense in allocating an entire XSPEC instance to their calculation.



The “switch” above symbolizes the sharing. A `ProcessMonitorXSPEC` usually has an associated `MulticolorLED`, a simple “LED” that is displayed in the sidebar for each fit. The LED changes state to reflect the `XProxy`'s associated XSPEC instance (represented here by `ControllerXSPEC`). A click on the LED displays the console log for the associated XSPEC. An option click aborts processing for that XSPEC only. (It is also possible to abort processing for *all* active XSPEC instances via the Abort icon at the top of the notebook window.) But the LED is associated with a fit, not a spectrum—a spectrum with no fits has no LED, hence no process monitoring.

Here are the cases:

1. Add a new spectrum. This creates a master proxy, but initially the proxy has no associated LED, making it “faceless”—there is no way in the present design to view its log or interrupt execution. (The log is maintained, but not visible.) However, before any fits share this interface, execution is very fast, so there is no need for status or abort controls.
2. Add the first fit. The fit's proxy is that of the owning spectrum. An LED is created for the fit with its `ProcessMonitorXSPEC`. For as long as the spectrum has at least one fit, the first fit's LED displays the master proxy's status and the console log, or aborts execution.
3. Add second and subsequent fits. Each fit creates its own proxy, and its own log and LED.
4. Remove the only fit. The master `ProcessMonitorXSPEC`'s LED is set back to `NULL`, so the associated status is once again invisible.
5. Remove the first fit, when there is at least one more remaining. In this case, the second fit becomes the first, so its `XProxy` is deleted and its XSPEC instance, if any, returned to the pool. The LED associated with the master proxy is reassigned to that of the new first fit.

### 7.17.5 Fits of Pique

Sometimes fits run for days and never converge, and sometimes they eventually *do* converge. ATLAS must support these long-running fits while allowing other work to proceed. (Don't confuse fits of pique—long-running fits—with fits of peaks, AKA [line-based analysis](#).) In the current (Mar 2017) design, once a refresh cycle begins, the user can configure other spectra and models for execution. It is also possible to abort the entire refresh cycle, or to abort any specific fit in

progress. But until all fits complete execution or are manually aborted, a new refresh cycle cannot be started.

A planned design modification accommodates long-running fits with an adjustment to the current user interface. Once a refresh cycle begins, the user can duplicate existing spectra or models, or create new ones. The refresh icon will remain enabled, and if a new refresh command is issued, it **adds** any newly configured spectra/fits to the pending list of active fits.

From the user's perspective, this means that the configuration panel for a long-running model can simply be collapsed, where it is out of the way, with only its flashing activity light as an indicator that it is still active. This means it has almost no impact on continued use of ATSAAL. In a typical use case, a user might duplicate the long-running model and edit the model expression or its parameters in hopes of finding a better match. Hitting refresh will start the new model running alongside the old.

### 7.17.5.1 Performance Impact

How much do long-running fits impact system performance? Probably very little...

**Memory.** XSPEC consumes about 20 MB of virtual memory during a typical fit, although this number could vary substantially depending upon input files and other specific parameters. This is negligible relative to today's typical memory configurations.

**CPU Time and System Responsiveness.** On a typical multicore machine, multiple fits won't have any appreciable effect on system responsiveness until the number of concurrent fits exceeds the number of cores. ATSAAL could automatically reduce the priority of long-running fits to provide more graceful degradation in the event that too many fits are running at the same time.

**I/O Bandwidth.** After a flurry of loading overhead when a fit begins, long-running fits are not anticipated to have a significant impact on I/O bandwidth. Fits tend to be almost entirely compute-bound.

**Logfiles.** Another impact is logfile size, both in memory and on disk. Limits will need to be enforced to prevent excessive memory consumption. The in-memory console log for each XSPEC will be truncated to e.g. 100KB. On-disk XSPEC logs, currently stored as part of the notebook, are not truncated, but will be omitted from the compressed notebook at save time. The uncompressed notebook directories will be deleted upon normal exit from ATSAAL, so these logs will not normally accumulate. An override will probably be needed for cases when the logs are needed to assist debugging.

### 7.17.5.2 Fit Progress

If it is possible to develop a meaningful progress indicator, some compact data visualization that even guesses at likelihood of fit convergence, this would be a helpful addition.

## 7.18 XFit and XSPEC Models

An `XFit` is simply a representation of a model expression, along with its parameters, input files, and anything else needed to ask an XSPEC session to compute a fit. The tree completely specifies the problem: it will produce the same output regardless of whether it asks the current XSPEC session or a new one.

The model and parameter editors are used to select models, create model expressions, save user models for later use, and modify parameters and defaults. Following are some of the properties needed:

- A model instance must produce the same analytic result regardless of which version of XSPEC it is run against
- A model instance must produce the same result even if it references other user-defined models that have changed in definition or are not present.
- A notebook containing model expressions must run the same on another user's computer, even it refers to user models that do not exist on the other computer. (This is really equivalent to the previous requirement.
- Model expressions are parsed during entry, and must display errors in syntax and model names.
- Model expressions are also evaluated for correctness of model types and the operators used to combine them.
- Individual models that comprise a model expression must retain any associated parameter changes unless the user deletes a model name.
- If a model name in an expression is a user-defined model, or a new user-defined model is entered, the model is expanded in place, so that further changes to the user model do not effect the expression.

- Most user-defined models are transient: they are not named, and do not appear in the models list. They are saved with the notebook for which they are defined. But any such model may be named and saved, with its current set of parameters, as a permanent user model.
- Permanent user models are never exchanged with other users (though this could be added). This avoids accidentally cluttering other users' models lists when exchanging notebooks. However, another recipient can take models defined in the notebook and save any of them as permanent models.

### 7.18.1 Model Editor

Models with problems (probably XSPEC 11 vs. 12):

- disk: arg list doesn't match
- gnei: crazy parameter name with angle brackets
- pileup ... ?
- compTT: spherical/disk radio buttons right?
- plcabs: \*nmax confusion
- smaug is example of model that could use popup menu params
- zedge appears in additive model list in docs, but marked mul in models.dat
- acisabs: parameter mismatch between model.dat and docs

### 7.18.2 ToolEditorModelParameter

The model parameter editor is designed for three slightly different uses:

- Invoked by double-clicking a model in the model editor, it permits edits to the *default* parameters for user-created models.
- Invoked by clicking Parameters... in a model instance in a plot editor, it permits edits to the *current* parameters for this model instance.
- If the user inserts a Parameter tool into the notebook, this editor appears when the tool is double-clicked for editing. It can be used to edit the current parameters for any selected model instance.

In all of these cases, this dialog is modal, in part to reduce ambiguity that might otherwise result if several such windows were open at the same time. In the first case, which edits user models, any parameters that vary from the baked-in XSPEC defaults are highlighted. In the other two cases, deviations are shown between the model instance and the user-assigned defaults. The Restore all defaults button therefore restores XSPEC defaults in the former case and user defaults in the latter.

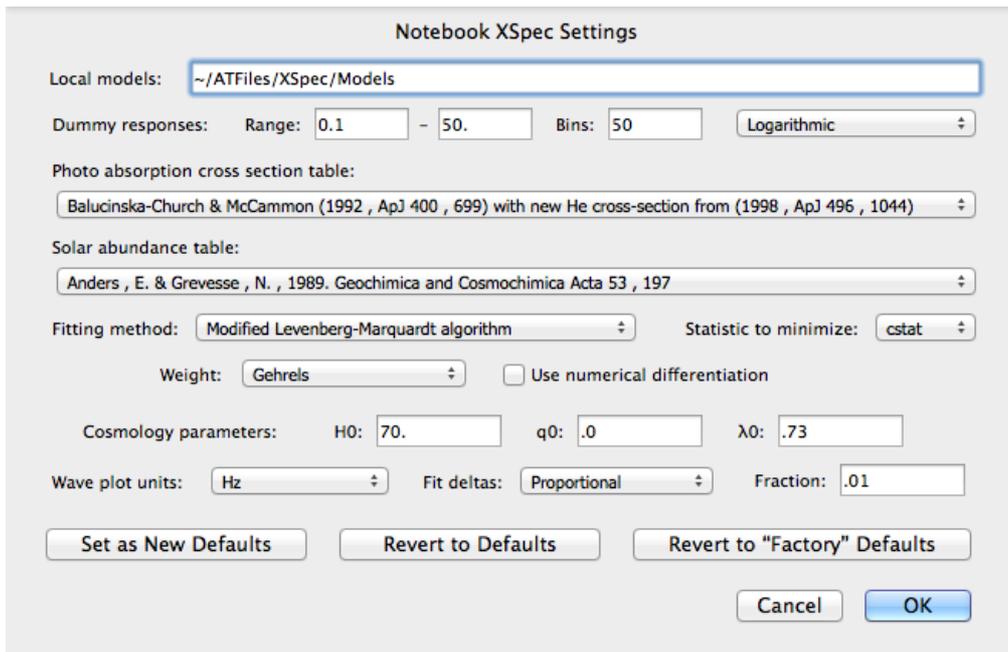
In the third case, the parameter tool displays a popup-menu that allows the user to select and edit the parameters for any model instance in the notebook. Any number of parameter blocks may be added to a notebook, but only one may be opened for edit at a time.

Parameters may include Python variables or expressions in place of hard values. The Python expressions are evaluated for a given model instance as part of the refresh cycle for the plot tool with which they are associated. Both user models and model instances may contain Python expressions. Expressions that are undefined are reported as errors.

In each case, the `ToolEditorModelParameter` is instantiated, and a model instance provided via `setModelInstance`. When the user hits Cancel, the editor emits `modelParamsCanceled()` to allow the caller to delete the editor. When the user hits OK, the editor emits `modelParamsChanged()`, which is caught to retrieve the changed parameters prior to deleting the editor. Call `updateModelInstance(XModelInstance* instance)` to copy the changed parameters into the supplied instance.

### 7.18.3 XSPEC Initialization Parameters

This is the XSpec Settings Dialog. Some of these choices are mutually exclusive, or clearly inappropriate for some tasks. [What can be done to reduce the incidence of incompatible choices?](#)



See this discussion on [XSPEC and low count spectra](#).

These values can be set via the Notebook object using the python interface.

## 7.18.4 Model Classes

An `XFit` is the root of an analytic sequence to be performed by XSPEC. It contains information on any necessary setup, input files, the model expression, current and default parameters, and documentation. It can generate the sequence of XSPEC commands and parse the results. The tree contains an `XModelInstance` which contains the model expression and its parameters.

An `XSimpleModelInfo` is a class that contains invariant information about a single XSPEC model. (“Simple” here means “not compound”—not a group of models.) A model *instance*, like an instance of a C++ class, contains information about a particular model expression. The outward-facing class representing a model instance is `XModelInstance`, which represents the expression and the models and parameters specified in the expression. Internally, an `XModelInstance` is composed of zero or more `XSubModelInstances` (one per term in the expression); each `XSubModelInstance` is composed of `XSimpleModelInstances`.

An `XModelInstance` may represent a single XSPEC model, a combination of simple models, or a combination of user-defined models. The last of these cases presents some special considerations. What happens if the user models referenced in an expression are later changed by the user? What happens when the user gives another user a notebook, for which these models are not defined? ATLAS must guarantee that a notebook will produce the same result for any user. To accomplish this, user models are expanded to their component expressions, and are not effected when other models from which they were created are changed. See *Parsing Model Expressions*.

Similarly, a model instance inherits its parameter settings from the models of which it is composed at the moment it is created—that is, the moment when the user creates or edits a model expression. This means that all parameters for a given model instance are not effected by subsequent changes to the default parameters for its constituent models. This ensures that notebooks will run identically when run later even if component user models have changed.

Most model instances are defined once as part of a specific plot tool. There is not a need to name these instances. The only time such instances must be named is when selecting them from a popup in the parameter or results tools, and they are assigned temporary names formed from the tool name, sequence number, and current model expression in such lists. For example, **plot3.2: bextrav+bbbody** selects the second model from spectrum plot `plot3`, which presently refers to the model expression `bextrav+bbbody`.

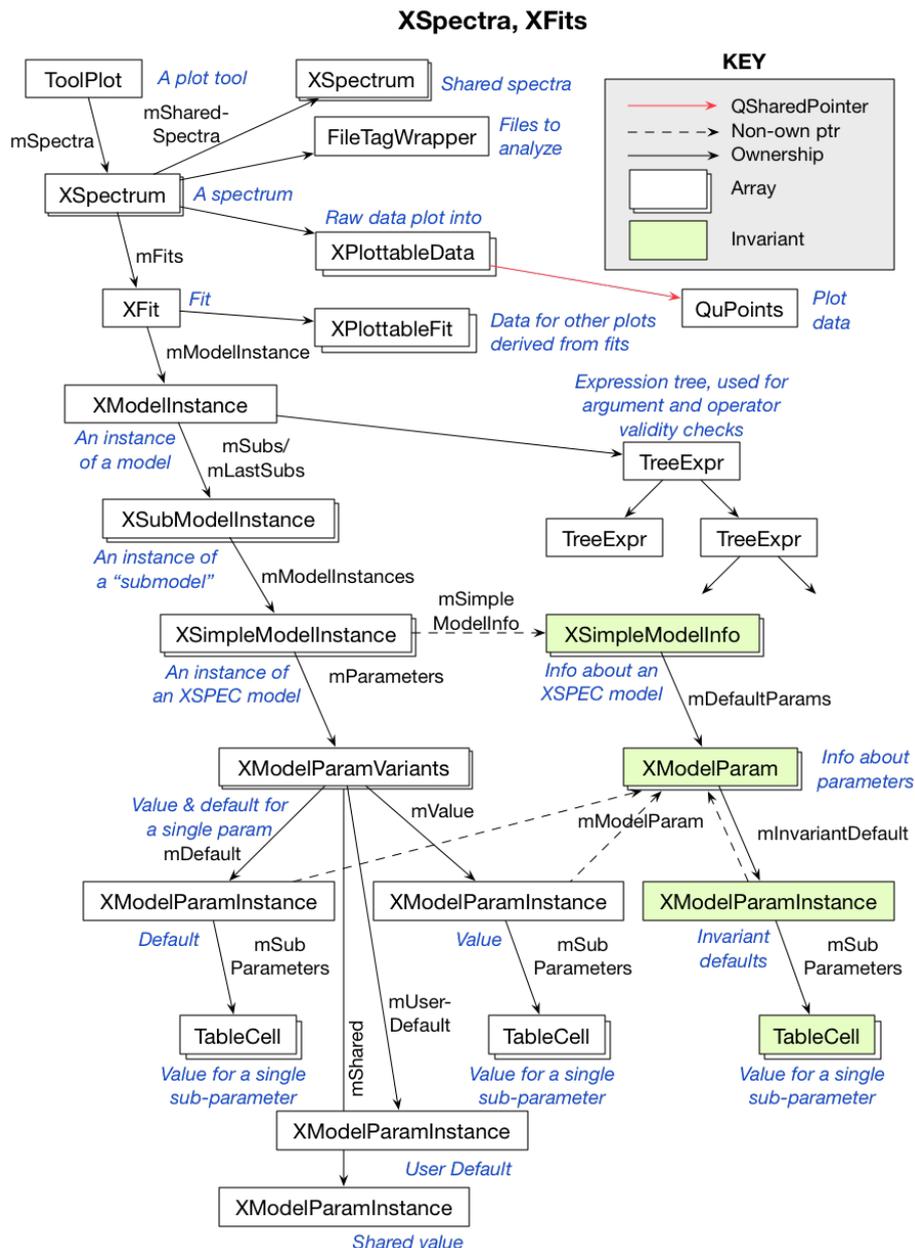
Any model instance defined in the model editor may be named and stored as a permanent model. Permanent models act as templates: the user can copy the permanent model and use it as is or further modify it; or they can update the settings and copy them back to the permanent list. Permanent models are shared among all notebooks and

remembered across ATLAS sessions.

A more complex issue arises while the user is editing the model expression of a given model instance. Imagine an expression such as `bexriv+tbody`. The user edits the parameters for the `bexriv` instance, then decides to get rid of the `tbody` term. `XModelInstance` was heavily revamped to retain parameters associated with models during model editing. Parameters are retained as long as a model in an expression remains in place. So when, e.g., the user hits the delete key, creating the expression "`bexriv+bbod`", the `tbody` parameters are discarded, but the `bexriv` parameters are retained.

### 7.18.4.1 Model Pointers

This diagram shows how `XFit`s are represented within a plot tool. This diagram shows only some major data classes.



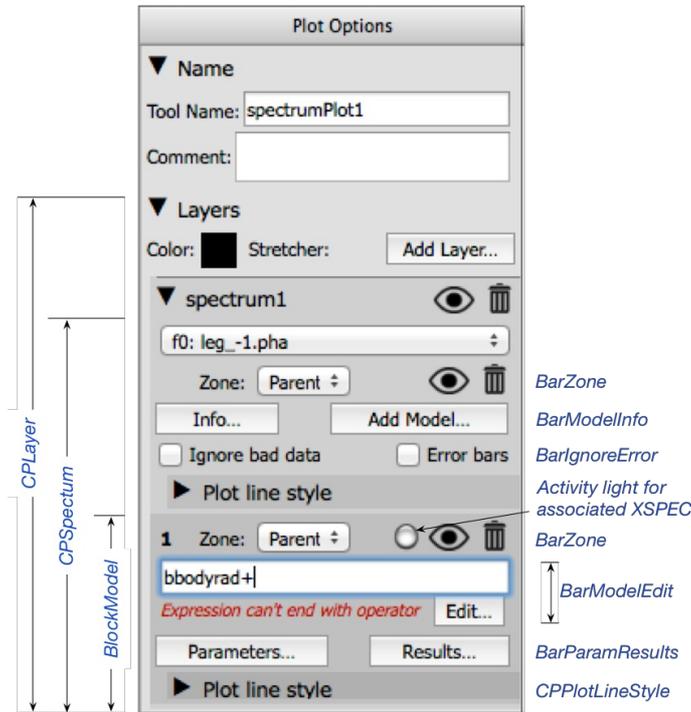
### 7.18.4.2 Saving/Restoring

Simple model infos are not saved, since they are created by loading the `model.dat` file at startup and they are invariant at runtime. User model infos are saved because they are created and edited by users. A user model info is saved as sub model infos; each sub model info is saved as a model expression, an expanded expression, and a sub model instance

containing parameters and defaults.

User model infos are saved and restored before restoring any user model instances, since the latter need the former.

### 7.18.5 Spectrum Plot Sidebar Classes

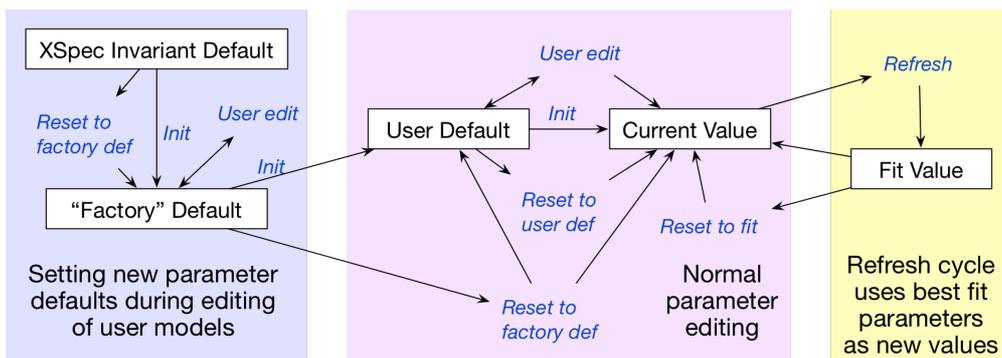


### 7.18.6 Default Parameters

Each model has one or more parameters. Each parameter has six subparameters. And each subparameter has as many as five versions:

1. The XSpec invariant default is the subparameter’s initial value, as assigned by XSpec, the “factory default.”
2. User models, composed of one or more XSpec models, may override the XSpec defaults, using the model editor’s default parameter editor. Collectively, these two layers of defaults are considered the “factory defaults.”
3. In the plot tool, users add spectra, then models to be fit to the spectra. Each model’s parameters are edited via Parameters.... This shows each parameter’s current value.
4. When a fit is performed, XSpec computes new values for the value subparameter. By default, the fit values become the new current values.
5. Finally, the last entered user default is stored.

The user can revert all the parameters to factory defaults, to the last manually entered user values, or to the most recent fit values. They can also restore the factory or user defaults on a per-parameter basis. In the diagram below, values that comprise the factory defaults are shown in blue. Values set in the plot tool for a model are shown in the pink region, and the value computed during a fit is shown in yellow. The actions that result in copying of values are shown in blue.



### 7.18.6.1 Parameter Editors and Defaults

These diagrams show how the two types of parameter editors deal with defaults. The top diagram is the editor used to adjust parameters prior to running a fit. The bottom editor is used to adjust the factory parameters for a user-defined model.

#### Model Parameter Editor

*Resets to the last user default (vs. most recent fit default)*

*Resets to Xspec default or user-assigned default for a user model*

*Sets the parameters prior to running a fit. Fit results set the new current values, but may be adjusted.*

*Shows most recent user (blue) or factory (purple) default*

*Resets all parameters to their "factory" defaults*

*Resets all parameters to their last user-entered defaults (or factory if there aren't any)*

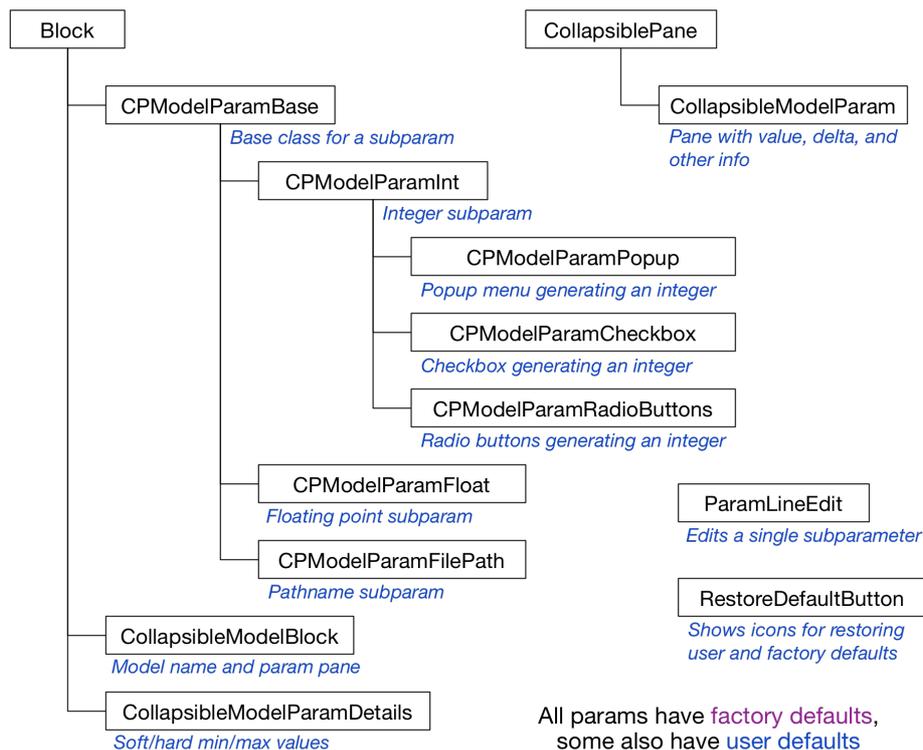
*Resets all parameters to the values calculated by the most recent fit*

#### User Model Defaults Editor

*Resets to Xspec default*

*Editor looks closely similar, but lacks last user values and fit results. This editor set the "factory" results for user-defined models.*

Some of the user interface classes associated with parameters are shown below.



## 7.18.7 Parsing Model Expressions

Model expressions are generally simple, but their parsing nevertheless involves several stages. There are two major parse phases:

1. Error-checking. This phase checks syntax, model name validity, and correctness of operators. It also expands references to user models.
2. Preservation of parameters. All expressions, whether syntactically correct or not, are parsed in such a way as to preserve any parameters the user has entered for models in the expression.

### 7.18.7.1 Phase 1: Parsing Model Expressions

Model expressions are generally simple, but their parsing nevertheless involves several stages. On each keystroke during model editing:

- First, a preparse is performed by `TreeExpr::preparse`. This parse checks for valid characters, paren balancing, and valid model names. If any of these simple tests fail, an error is reported and deeper parsing is not performed. This makes the next parse phase a bit simpler. Even if there are syntax errors, though, the phase two parse is done.
- Next, instances of `TreeExpr` are created to represent the expression as a parse tree. Even though ATSal will not actually evaluate the expression, the parse tree is needed for error-checking. The main goal of the parse tree is to verify that operands to each operator are of the appropriate type. This parse also calculates the expression's `TreeExprType` (`TETAdditive`, `TETMultiplicative`, or for malformed expressions, `TETBogus`). Once again, further parsing is only performed if this parse passes all error checks.

### 7.18.7.2 Phase 2: Preserving Parameters

When a user modifies a parse expression, ATSal attempts to keep track of changed model parameters as the expression is edited. This parse is complicated by the fact that it must keep parameters even when the expression is syntactically invalid.

For example, the user might create a model expression:

```
bextrav+bbbody
```

At this point, they have a valid expression. They can click the Parameters... button to edit the parameters for both of these models. Now suppose they add further characters to produce:

```
bextrav+bbbodyrad
```

At this point they have another syntactically valid expression. But because the expression has changed, it replaces the previous expression representation and discards any parameters associated with the previous expression. Ideally, in this example, since `bextrav` hasn't changed, any parameters for it shouldn't either. This isn't as easy as it sounds though.

### 7.18.7.2.1 Some Examples

A few pathological examples are shown below.

#### 7.18.7.2.1.1 Changing an Operator

Consider:

```
pileup+bbbody
```

This would generate an error, because `pileup` is a multiplicative model. So the next step might be to place the cursor after the '+', and press delete:

```
pileupbbbody
```

The expression now appears to be a single invalid model, so any parameters for both of the original constituent models are discarded. Since this is a fairly common case, though, ATSSAL handles it, but to do so it must keep track not only of a previous and current expression, but also the position of the text cursor after each keystroke. Armed with this knowledge the expression is really:

```
pileup|bbbody
```

(using a vertical bar to represent the cursor). ATSSAL can keep track of the split and retain the parameters on both sides. If this is followed by a new delimiter:

```
pileup*|bbbody
```

ATSSAL maintains the association between the parameters.

#### 7.18.7.2.1.2 Repeated Models

Another pathological (and perhaps astrophysically ludicrous) example is:

```
bbbody+bbbody+|bbbody
```

If the user has entered different parameters for all three of these models, then deletes the middle model, ATSSAL needs to maintain the first and last parameter sets. To do this, ATSSAL must parse backward from the end to the cursor position, then forward from the beginning.

#### 7.18.7.2.1.3 User Models Containing User Models

Another issue that complicates this process is the expansion of user models. Once a model expression is parsed, it was previously represented internally as a set of simple models. So, for example, if:

```
model_1 = bextrav+bbbodyrad  
model_2 = bextriv+bbbody
```

then:

```
model_1+model_2
```

is represented internally as `bexrav+bbodyrad+bexriv+bbody`. This is because any further changes to the component definitions should not influence the current definition of a user model; otherwise changes to user model definitions would cause older notebooks to produce different results when run again. Hence the only time a user model is re-evaluated is when its definition is changed.

So let's say that the user redefines `model_2` to be `bexriv` only. The expression `model_1+model_2` is not effected. But if the user *edits* the expression, it is re-evaluated. So if they change it to:

```
model_3+model_2
```

It looks like `model_2` is unchanged, so its parameters can be preserved, but since its definition has changed, they cannot. Even though both the old and the new definition of `model_2` contain the simple model `bexriv`, they can't be considered to be equivalent. So for the particular example all parameter settings are lost.

**In summary, ATSAAL attempts to preserve parameters when it seems intuitively natural to do so.**

### 7.18.7.2.2 The Parameter Preservation Parse

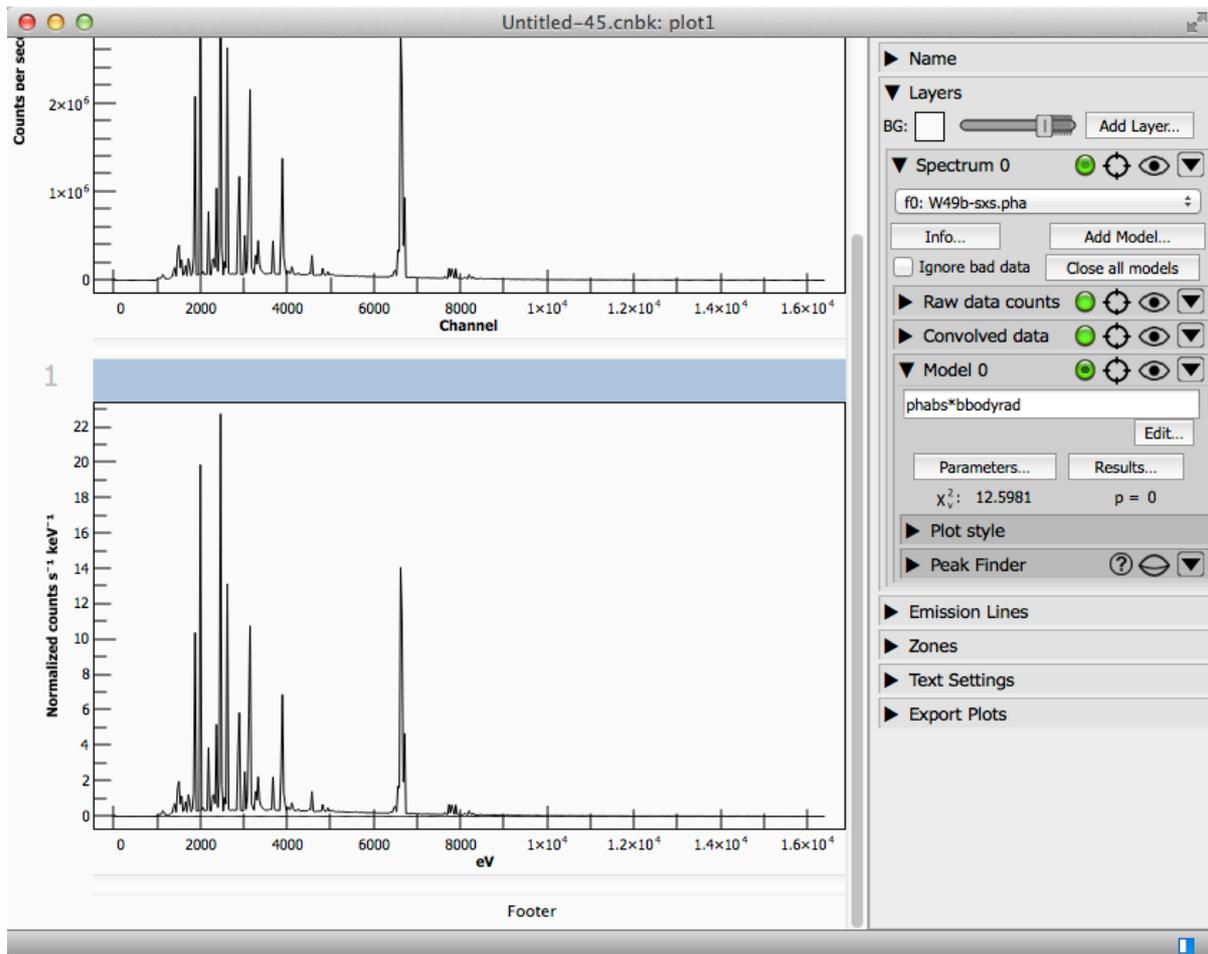
`XModelInfo::merge()` accepts a new expression and an ending cursor position. This parse is very shallow: it only distinguishes between model characters and delimiters. It compares the current string and cursor position against the previous expression. First it scans backwards through the expression to the cursor position, handling trailing parameters; then it moves forward from the start to the first point where the expressions diverge. It creates a series of new `XSubModelInstances (mSubs)`, copying parameter data across from the previous set of `XSubModelInstances (mLastSubs)` where appropriate.

## 7.18.8 Line-based Analysis Design

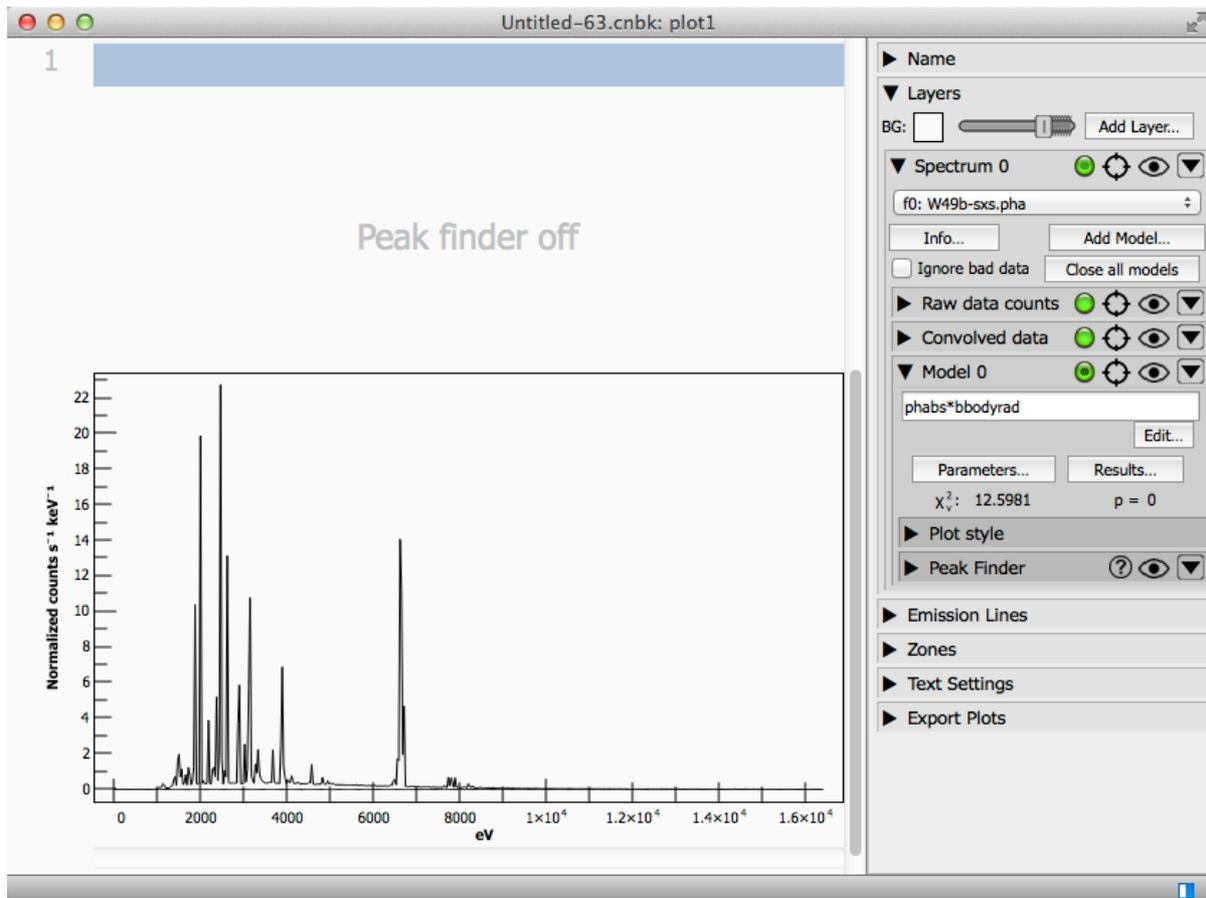
Line-based analysis extracts and labels emission line peaks in a spectrum, by creating Gaussian models for each major peak and performing a fit.

### 7.18.8.1 Using Line-based Analysis

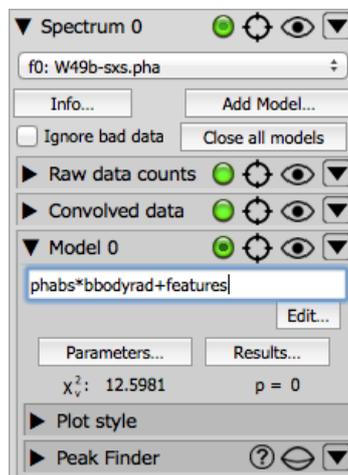
Line-based analysis may be separately enabled for any model. In this example we assume that a spectrum has been created, and a model added, `phabs*bbodyrad`. A convolved spectrum is needed for this operation, so hit refresh if necessary to obtain one. It will appear in the second zone of the plot window as shown:



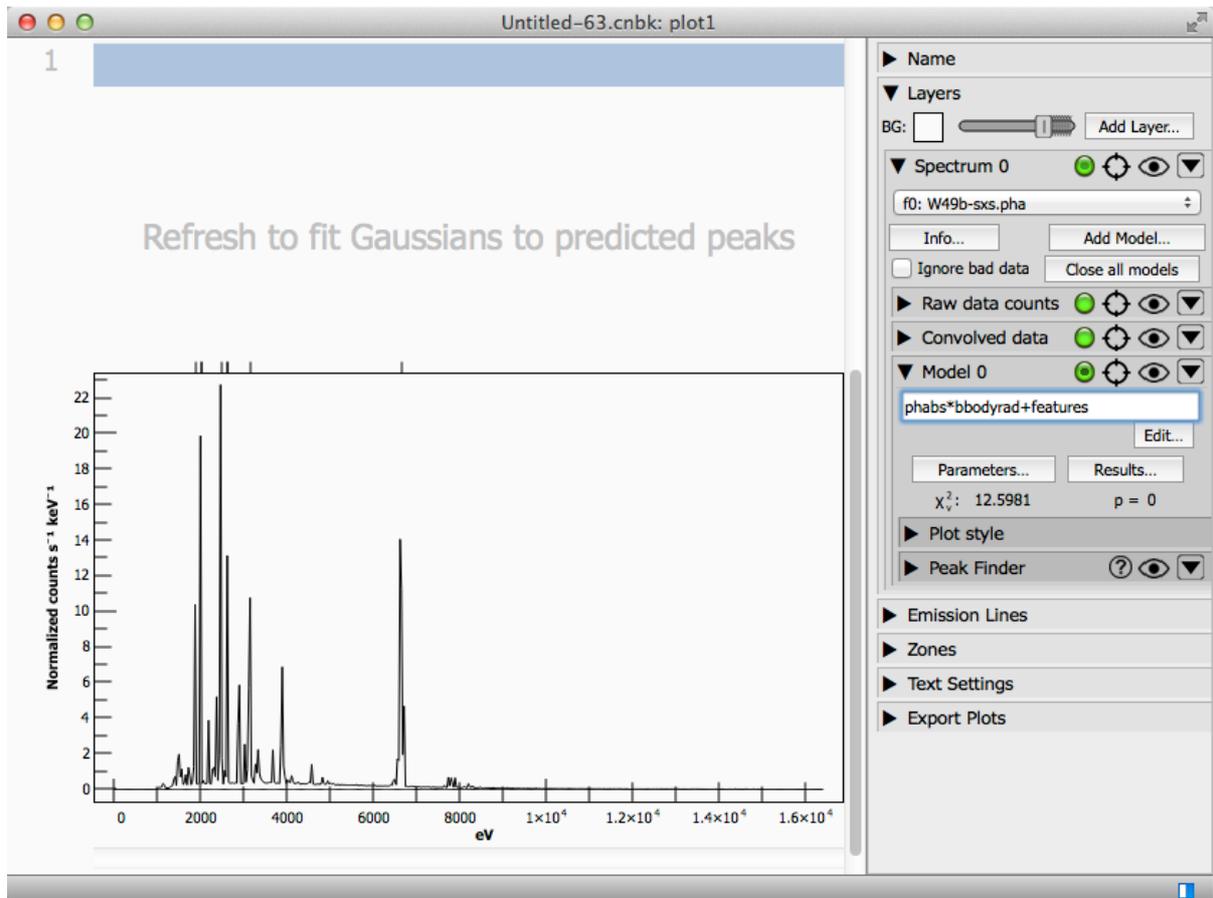
In the Peak Finder section of the model panel, click the eye icon to display the label area for the plot.



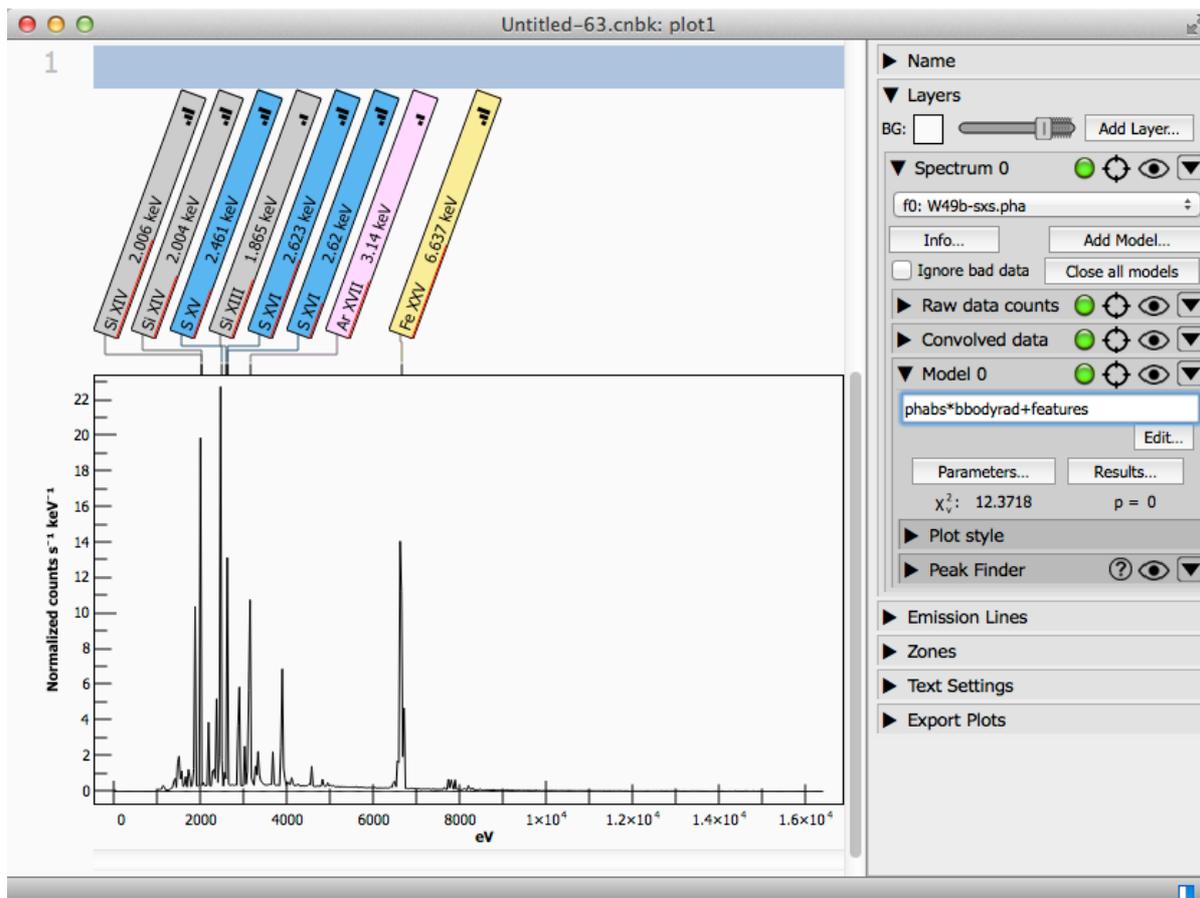
This displays the label region, used for emission line labels as well as user annotations. It does *not* enable peak finding, however. Initially it simply indicates that peak finding is disabled. Next, append the `features` pseudo-model to the model expression:



The peak finder evaluates the convolved data, marking a subset of the features for Gaussian analysis. The peaks are denoted by stubs along the bottom of the label area. Labels cannot be shown until the Gaussian analyses are performed. (The peak sensitivity is adjustable. More on this soon.)



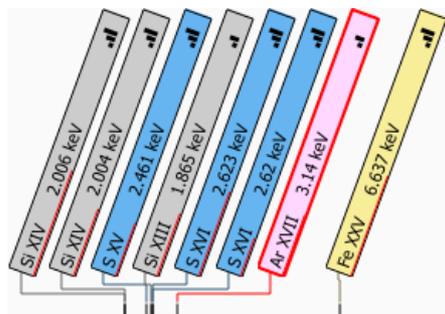
Refresh the plot once again. This time, Gaussians are computed for the peaks and the results shown. Up to two lines are shown for each peak, in order to help identify two nearly overlapping peaks.



If you take a closer look at each label, it shows, depending upon available space, one or more of the following:

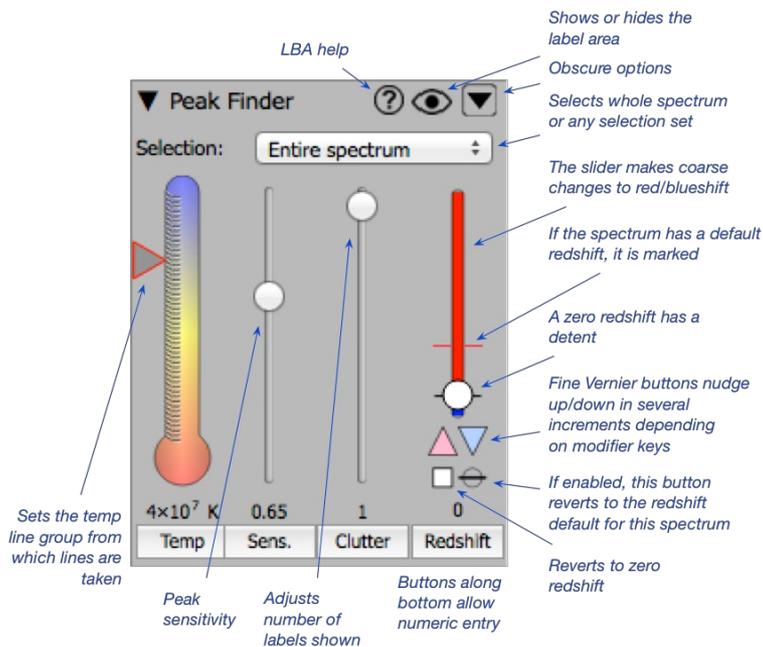
- The ion, color coded by element according to an inscrutable process.
- The line energy, which appears in red or blue if it is red- or blueshifted.
- The approximate normalized line emissivity at this temperature, shown as cell tower bars.
- The starting and ending levels (*not shown*).
- A red bar across the bottom indicates the relative match quality of this emission line to the calculated peak. **This is mostly for debugging and may be removed later.**

Hovering over a label highlights it and its connecting line, making it a little easier to see the associated peak, as shown with the Ar XVII peak. Double-clicking a label displays details.



### 7.18.8.1.1 Peak Finder Controls

Open the peak finder to fine tune its operation:



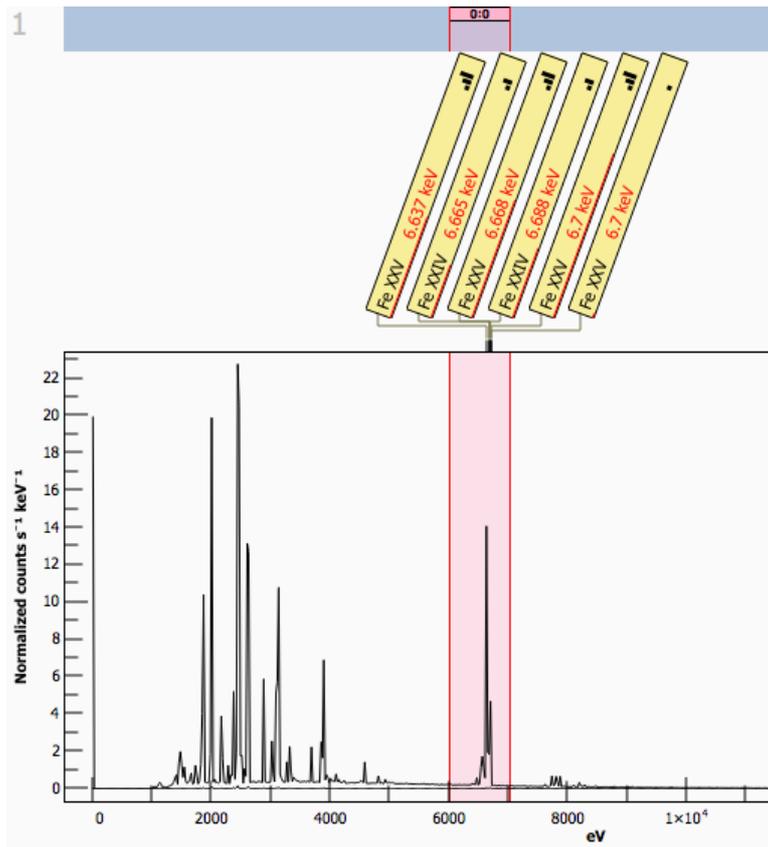
The temperature control selects an approximate temperature for the source, which may alter the emission lines that match peaks. Each gradation on the “thermometer” corresponds to a different temperature line group in the AtomDB database.

The sensitivity control determines which subset of peaks are checked for emission lines. Lower values select fewer peaks. As the control is adjusted, the small stubs representing the predicted peak values change to reflect the setting.

The redshift slider covers a span from a blueshift of  $-1$  to a redshift of  $11$ . Adjusting the control causes different emission lines to match. The black line marks the zero position. **If there is a way to estimate the source redshift, a red line shows this value. (This is not implemented yet.)** The control covers far too great a range to adjust by dragging, so there are two Vernier-style fine tuning buttons below it. These auto-repeating buttons adjust the  $z$  value  $\pm 0.0001$  (or  $0.001$  if option is pressed, and  $0.01$  if command is pressed). The white box resets the control to zero, and the other button resets it to the FITS default, if any.

The number of emission lines may fill the available space, especially when zoomed out. The clutter control removes labels that fall below a match quality threshold, so that remaining labels are positioned closer to their peaks.

The selection popup menu is used to restrict the peak matching to a particular selection set. Create a selection set first, then select it using the popup. By selecting only a subset of the range, additional lower quality match candidates may appear. By default, the entire spectrum is used. In the example below, matching is restricted to selection set 0, which has been set to roughly the range of 6-7 keV. (Selection sets can contain multiple discontinuous selections too.)



Of these five controls, *only the peak sensitivity and selection set controls determine which peaks are fitted with Gaussians*. The remaining controls have no effect on the fit performed by XSPEC.

### 7.18.8.1.2 Emission Line Details

Double-click an emission line for detailed information:

26 55.845  
Fe XXV  
 Iron  
 $[Ar]3d^64s^2$

Peak Info:

Peak Energy	Å	keV
Predicted	1.86822	6.6365
Gaussian fit	1.86822	6.6365
Emission line	1.86819	6.63658

Level Info:

Level	2 → 1				
Electron configuration	5g <sup>1</sup> → 1s <sup>1</sup>				
Energy above ground	6646.6 eV → 0.0000 eV				
Quantum state	<i>n</i>	<i>L</i>	<i>S</i>	<i>degeneracy</i>	<i>parity</i>
	2 → 1	0	1 → ½	3 → 2	even
Energy level reference	Whiteford ICFT → 2012ApJ...756..128F				
Photoionization reference	1986ADNDT..34..415C → NIST				

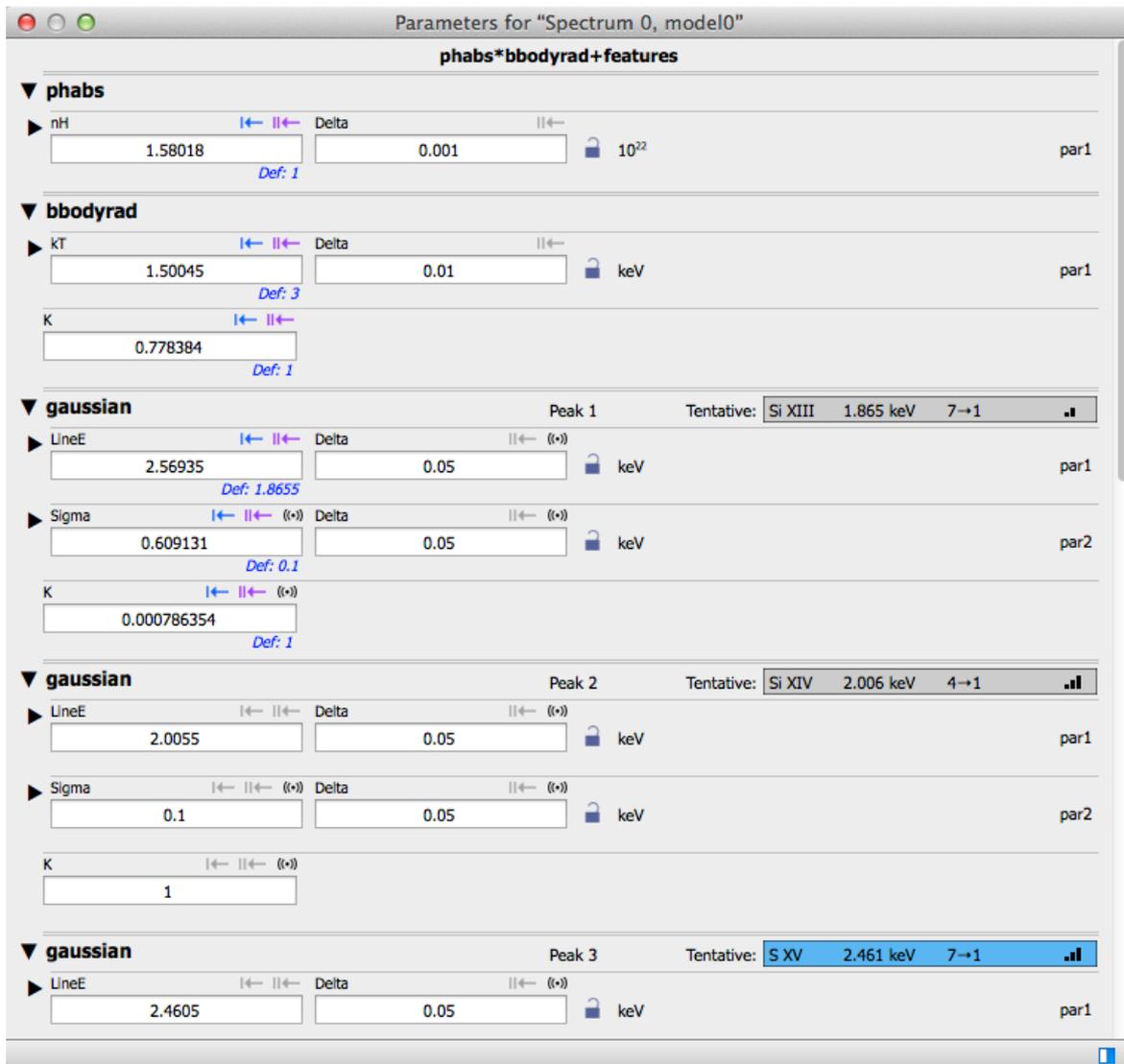
Radiative transition info:

Electron collision rate	nonzero
Collision rate reference	TBD
Wavelength (theory)	1.86819 Å
Transition rate/Einstein A	1.93×10 <sup>8</sup> s <sup>-1</sup>
Transition type	M1
Oscillator strength $f_{2→1}^e$	1.51479×10 <sup>-7</sup>
Wavelength (theory) reference	Whiteford ICFT
Transition rate reference	Whiteford ICFT

Temperature vs. emissivity for Fe XXV 2 → 1:

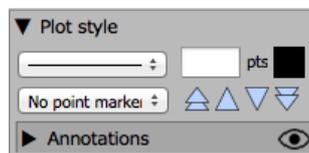
### 7.18.8.1.3 Fine-tuning Parameters

You can adjust Gaussian fit parameters for the peak matching the same way you adjust other model parameters—use the Parameters button in the model pane. Peak-fitting Gaussians are denoted by their respective emission line labels. Most controls work the same way, but there is one new control, the little dot surrounded with waves. This control copies, or broadcasts, the setting for one peak Gaussian subparameter to the other peak Gaussians, providing a way to synchronize the settings of all of them at once. `LINEE` parameter changes beyond very small tweaks will produce undefined results.

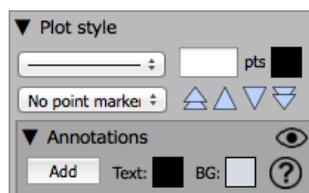


### 7.18.8.1.4 User Annotations

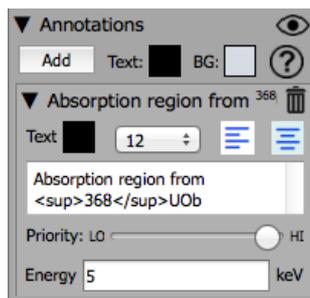
You can also add *anchored annotations* to the graph. They are anchored in the sense that they are linked to a particular photon energy, like emission line labels, so you can mark a particular energy of interest. To add a user annotation to a fit, open the model's plot style panel:



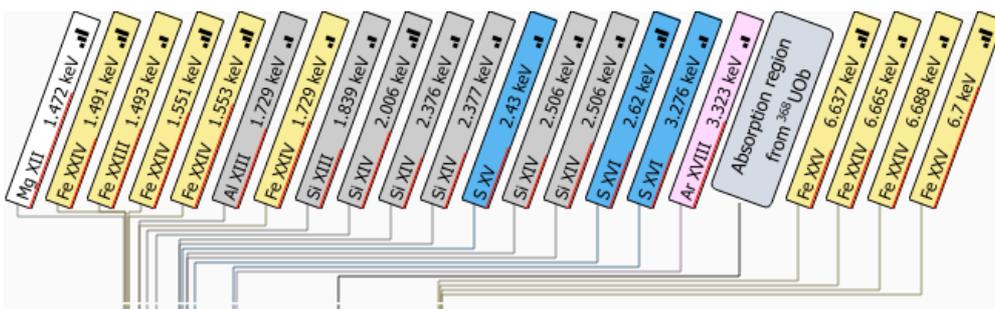
The show/hide control applies to all user annotations. Open the annotations panel to add annotations:



This panel controls default text and background color. Add an annotation:



The text is shown at the energy you assign. The priority is used by the clutter control. At the default maximum priority, the annotation takes precedence over all emission lines and always appears. Lower settings cause it to be squeezed out if space is limited or the clutter control is lowered. Long annotations may appear truncated.



Annotations support a miniscule subset of HTML, as with the unobtainium isotope above. Use `<sub>` and `</sub>` for subscripts, `<sup>` and `</sup>` for superscripts, or `<b>` or `<i>` for bold and italic.

### 7.18.8.2 Overall LBA Design

This is a complete rethink of how Line-based Analysis should work, based on a meeting with Randall on 2016-04-24 and subsequently refined. As before, LBA performs a peak-finding step and creates a set of gaussians (and maybe gabs at some future time), but the details are quite different.

First, some ground rules:

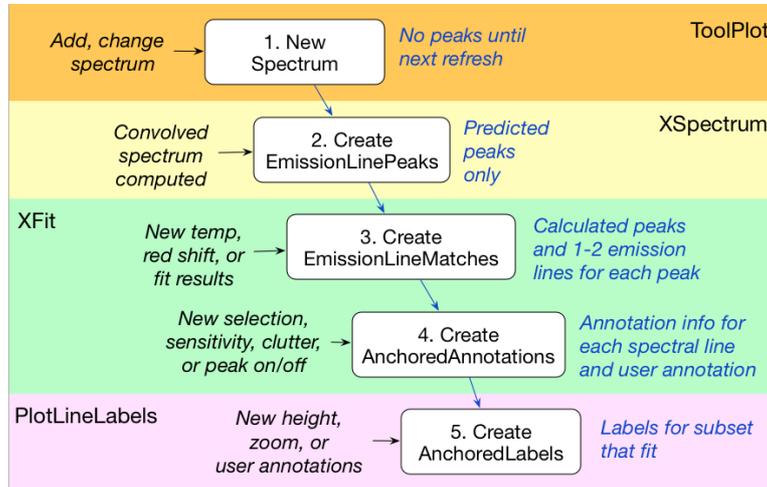
- A set of line labels is associated with a *model*. Not with a spectrum (though of course it uses the model's spectrum), and not with a plot pane. In this sense it differs from the Emission Lines panel of the plot tool, which shows lines associated with a cursor position in a plot pane. The two types of display have independent temperature and red shift controls. Similarly, peak finding settings for two different models are independent. One reason for this is that we have to tack the Gaussian commands onto a specific model.
- After experimenting with several approaches, a pseudo-model, *features*, has been added. If *features* is part of the model expression, ATSAAL performs a two-stage analysis. Stage one runs with the expression *except* for *features*, to fit the rest of the model. Stage two runs with the resulting values for the rest of the model frozen, as well as a series of Gaussians corresponding to the peaks.
- Since it is possible to overplot models in a plot pane, and since each such model may have its own line labels, only labels for the topmost visible model's plot are shown.
- **Continuum subtraction is not currently implemented because it requires that we know the approximate temperature of the spectrum, and that may not be available. I am also unhappy with the present algorithm.**
- Selection sets are still used to establish subsets of a spectrum for peak-finding. But they work very differently now. The selection set optionally restricts the range of values over which peak extraction is performed.

Line-based analysis is performed as a series of phases, each of which depends upon the previous phase. The separate phases minimize the recalculation needed each time something changes state in the system.

At the highest level, the user first opens a spectrum and does a refresh that produces a convolved spectrum. ATSAAL computes predicted peaks for the convolved spectrum. Next, the user adds a model and enables peak fitting for the model. When the next refresh is performed, the predicted peaks are each assigned a gaussian model, and the model is tacked onto the end of the user-defined model and passed to XSPEC. The fit results are used to set calculated peaks, and from these new photon energy values, matching emission lines are assigned and displayed.

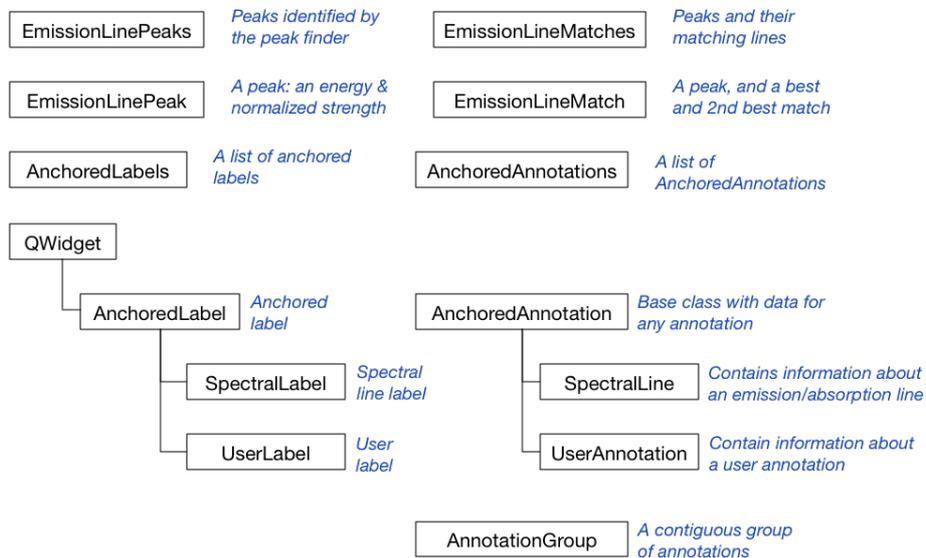
### 7.18.8.2.1 LBA Dependency Tree

This diagram shows the phases of activity that occur in computing and displaying emission lines. Each phase of computation depends upon the previous ones, but many state changes require only part of the dependency tree to be recomputed. Except for the first, each phase of activity is represented by a separate class, because this makes the internal logic easier to follow.



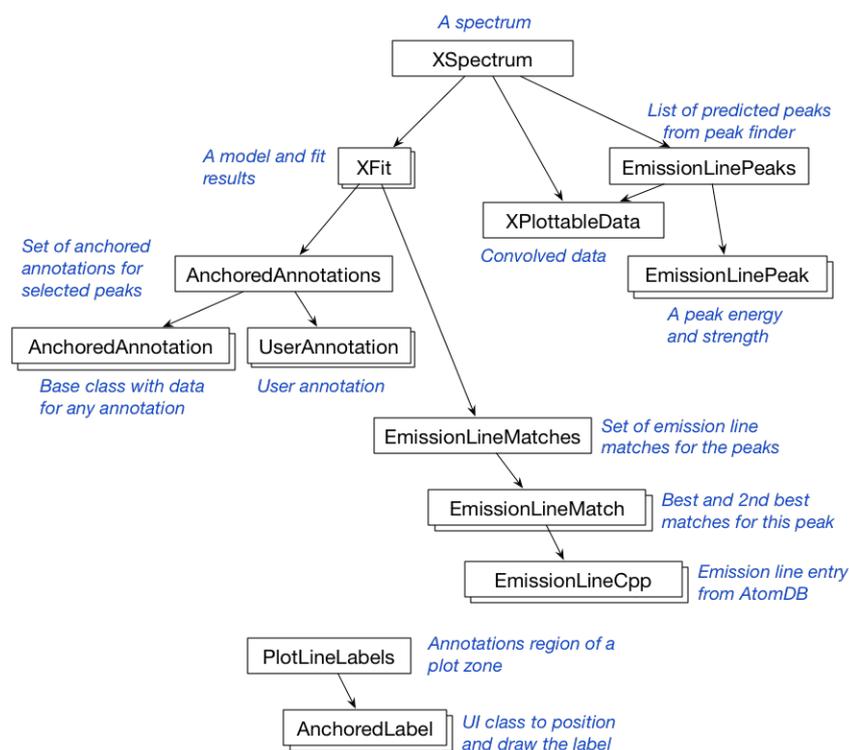
#### 7.18.8.2.1.1 Line-based Analysis Classes

Classes associated with line-based analysis are shown below. `AnchoredLabels` are user interface classes, but the others are data classes that serve primarily to partition various steps in the calculation.



#### 7.18.8.2.1.2 Line-based Analysis Ownership Graph

`EmissionLinePeaks` are associated with a spectrum. The list of peaks is shared by all the models (`XFit`s) for that spectrum. But `EmissionLineMatches` and the `AnchoredAnnotations` are stored with each `XFit`.



### 7.18.8.2.1.3 Phase 1: EmissionLinePeaks

The peak extractor, in class `EmissionLinePeaks`, is updated whenever a new convolved spectrum is computed. This happens when a new spectrum is created, or an existing spectrum's file tag is changed, and a refresh is performed.

The original proposed algorithm attempted to subtract the continuum prior to finding peaks. I wasn't happy with how it worked and have disabled it.

The current algorithm:

1. Performs a rolling average filtration on the data, to slightly soften the peaks, in an attempt to reduce false peaks resulting from noise. (This could have the undesired effect of merging distinct peaks, but the matcher assigns up to two labels per peak, so that is acceptable.) **The amount of filtration will need further tuning.**
2. Scans across the filtered plot for each reversal in slope. The topmost point serves as a predicted peak, and its distance relative to the baseline as a strength. This creates an `EmissionLinePeaks`, a list of all the peaks over a minimum threshold (currently 5% of the highest peak). Each `EmissionLinePeak` consists of a predicted photon energy, a strength, and an ID that is used later to correlate peaks with their gaussians.
3. When all the peaks have been extracted, their strengths are normalized to the range of 0-1.

Most sets of data points are represented with `QuPoints`, which contain information about the type of the data. The filtration operation produces a `MiniPoints` instead, a more primitive and efficient representation, since this is discarded after peak finding.

### 7.18.8.2.1.4 Phase 2: EmissionLineMatches

The next phase involves assigning `EmissionLineMatches`. Note the difference between a *predicted peak* and a *calculated peak*. The former has a photon energy that is guessed from the convolved data, while the latter is the line energy as computed by the Gaussian fitted to the peak. As a rule, we don't want to fit labels to the predicted peaks, because they aren't close enough to the correct values. But we fit them anyway, because displaying the tentative labels in the parameter editor is helpful in clarifying which peaks are associated with which Gaussians. Labels associated with predicted peaks are clearly marked as tentative.

The `EmissionLineMatches` for a spectrum is recomputed:

- When a convolved spectrum is updated, for the reason describes above.
- When fit results for the Gaussian peaks become available.
- When the user adjusts a temperature or redshift control.

The emission line matcher begins by finding the 20 lines closest to each predicted peak. Then it assesses a match quality for each of the 20 emission lines, using this algorithm:

$\epsilon_{line}$	emissivity of candidate emission line from AtomDB, normalized to 0-1 against the highest value for this temperature line group
$\epsilon_{peak}$	emissivity of this peak, normalized to 0-1 against the highest peak recorded for this spectrum
$\omega_{line}$	photon energy of candidate emission line from AtomDB, in eV
$\omega_{peak}$	photon energy peak center, computed by XSPEC via Gaussian fit
$c_{\omega}$	mystery constant that adjusts the energy delta
$c_{\epsilon}$	mystery constant that adjusts the emissivity delta
$\Delta_{\epsilon}$	emissivity delta
$\Delta_{\omega}$	energy delta
$d$	the “distance” between a given peak and a candidate emission line. Lower numbers indicate a better match. Numbers greater than 1 are very bad
$q$	the match quality, a normalization that produces a value from 0 (bad) to 1 (perfect)

After a fair amount of tinkering, I am currently using this algorithm:

$$c_{\omega} = 0.3, c_{\epsilon} = 0.3$$

$$\Delta_{\epsilon} = c_{\epsilon}(\log \epsilon_{line} - \log \epsilon_{peak})$$

$$\Delta_{\omega} = c_{\omega}(\omega_{peak} - \omega_{line})$$

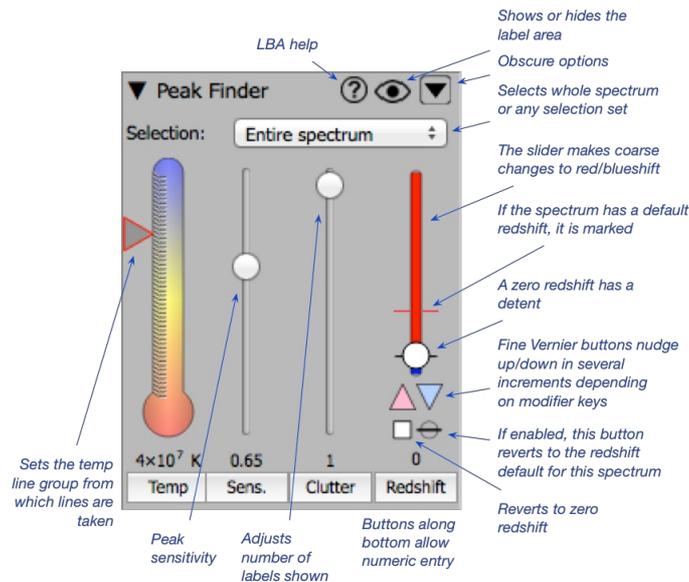
*[Math Processing Error]*

$$q = (d > 1 ? 1 : 1 - d)$$

This needs further fine-tuning. Small changes to the multipliers yield fairly significant changes to the matches. Randall suggested improvements to this.

The best two emission line matches are stored in the `EmissionLineMatch`. One or both may be null if no good candidates were found. Since up to two lines are selected, the algorithm has a reasonably good chance of picking out cases where a single peak is composed of two nearby peaks. Note that when two candidate emission lines are found for a single peak, a single Gaussian is still fitted to the peak, so its calculated peak,  $\sigma$ , and emissivity are “shared” among the two lines.

These controls operate the peak finder and emission line matcher:



The temperature control selects an approximate temperature for the source, which determines the temperature line group from which peaks are selected. If possible, it defaults to a value obtained from the FITS file; otherwise a mid-range value is used. (Is there a good default for this?) Each gradation on the “thermometer” corresponds to a different temperature line group.

The sensitivity control determines which subset of peaks are checked for emission lines. Lower values select fewer peaks. As the control is adjusted, the small stubs representing the predicted peak values change to reflect the setting.

The redshift slider covers a span from a blueshift of  $-1$  to a redshift of  $11$ . Adjusting the control causes different emission lines to match. If redshift information is present in the FITS file, it sets the default, shown with the red line. The black line marks the zero position. The control is far too coarse to adjust precisely, so there are two Vernier-style fine tuning buttons below it. These auto-repeating buttons adjust the  $z$  value  $\pm 0.0001$  (or  $0.001$  if option is pressed, and  $0.01$  if command is pressed). The white box resets the control to zero, and the other button resets it to the FITS default, if any.

The number of emission lines may fill the available space, especially when zoomed out. The clutter control removes labels that fall below a match quality threshold, so that remaining labels are positioned closer to their peaks.

Of these four controls, *only the peak sensitivity control determines which peaks are fitted with Gaussians*. The remaining controls have no effect on the fit performed by XSPEC.

### 7.18.8.2.1.5 Phase 3: AnchoredAnnotations

An `AnchoredAnnotations` is created for each `xFit`. It contains the data needed to represent a set of annotations in the label area of the graph. Each annotation is anchored in the sense that it is associated with a specific photon energy, and when displayed, it is placed as close as possible to that energy.

A `SpectralLine` is the primary type of `AnchoredAnnotation`. It represents a particular emission line, and has a pointer to an `EmissionLineCxx` with information about the line. Each `EmissionLineMatch` can spawn up to two `AnchoredAnnotations`, one each for the best and second best matches.

Users can add their own `UserAnnotations`. These are sandwiched between surrounding `SpectralLines`. By default, `UserAnnotations` take precedence over `SpectralLines`, but their priority may be adjusted individually.

The `AnchoredAnnotations` is recomputed when:

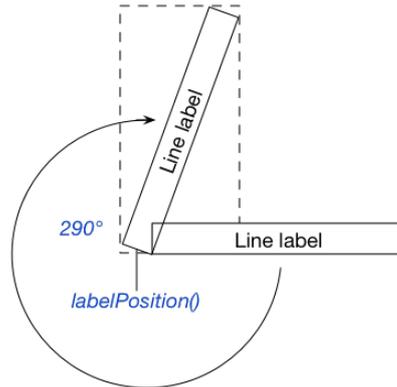
- A new selection set is chosen, or an existing set is altered.
- The peak sensitivity control is altered.
- Peak finding is toggled on/off.
- A dependent datum changes state.

### 7.18.8.2.1.6 Phase 4: AnchoredLabels

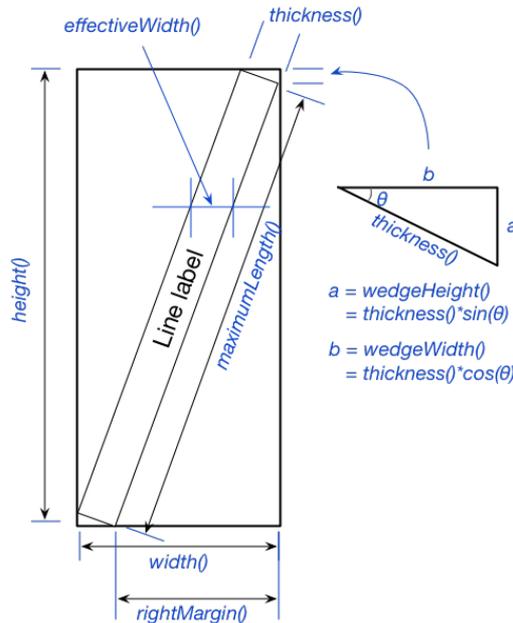
An `AnchoredLabels` contains the labels as they appear in the `PlotLineLabels` pane. The `AnchoredLabel` base class contains information about label geometry and other common elements, including the rotation logic. The `SpectralLabel` subclass displays an emission line, and the `UserLabel` subclass displays a `UserAnnotation`.

### 7.18.8.2.1.7 Label Geometry

`AnchoredLabels` are rotated in an attempt to allow room for as many labels as possible while preserving readability.

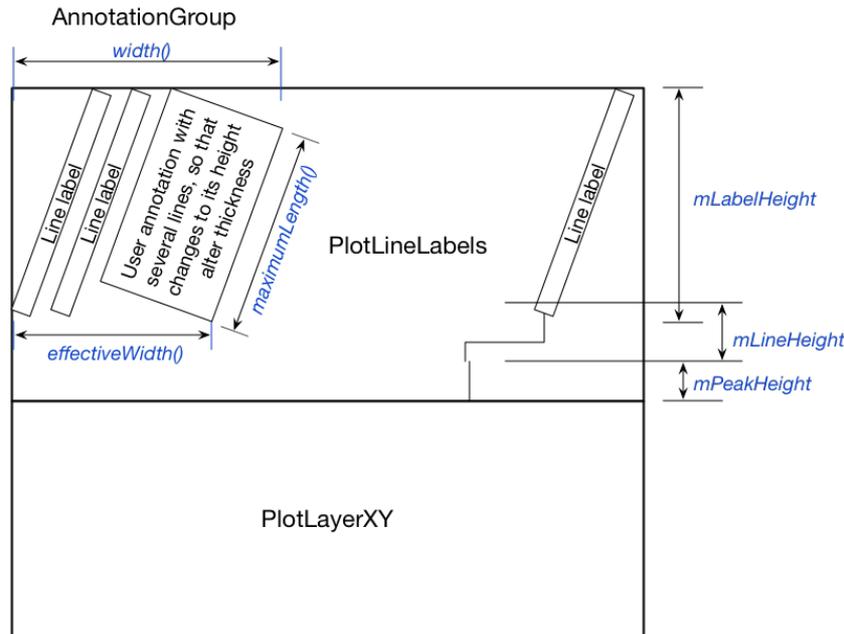


The rotation makes concepts like width and height ambiguous, so we add a few new measures:



In a pure Macintosh application, the *widget* is rotated, so, e.g., a mouse click is rotated as well and the widget's boundaries are that of the rotated label. In Qt, one transforms the coordinate system at paint time, and this may be done any number of times while painting. This is more flexible, but has some vexing implications. First, the widget's boundaries remain rectilinear (*outer boundary*), so a mouse hit inside the widget but outside the label is still trapped by the widget. And since there is no defined transform for the widget as a whole, we have to apply the inverse of the desired transform to the mouse coordinates to figure out whether the mouse hit is actually inside the label.

This diagram shows how space is allocated within the `PlotLineLabels` area.



`mPeakHeight` is a fixed height region used to show the predicted or calculated peaks. The `mLineHeight` region encloses all the connected lines. Lines are organized into lanes, and the number of lanes needed determines the `mLineHeight`. The `mLabelHeight` is whatever is left over.

The layout algorithm works by first creating an `AnchoredLabel` for each `UserAnnotation` and `SpectralLine` until the available horizontal space is exhausted or there are no more labels. The clutter control limits the number of available labels, and at higher zoom levels there is room for more labels. If the number of labels exceeds the available space, there is nothing to do but line them up and connect them with lines.

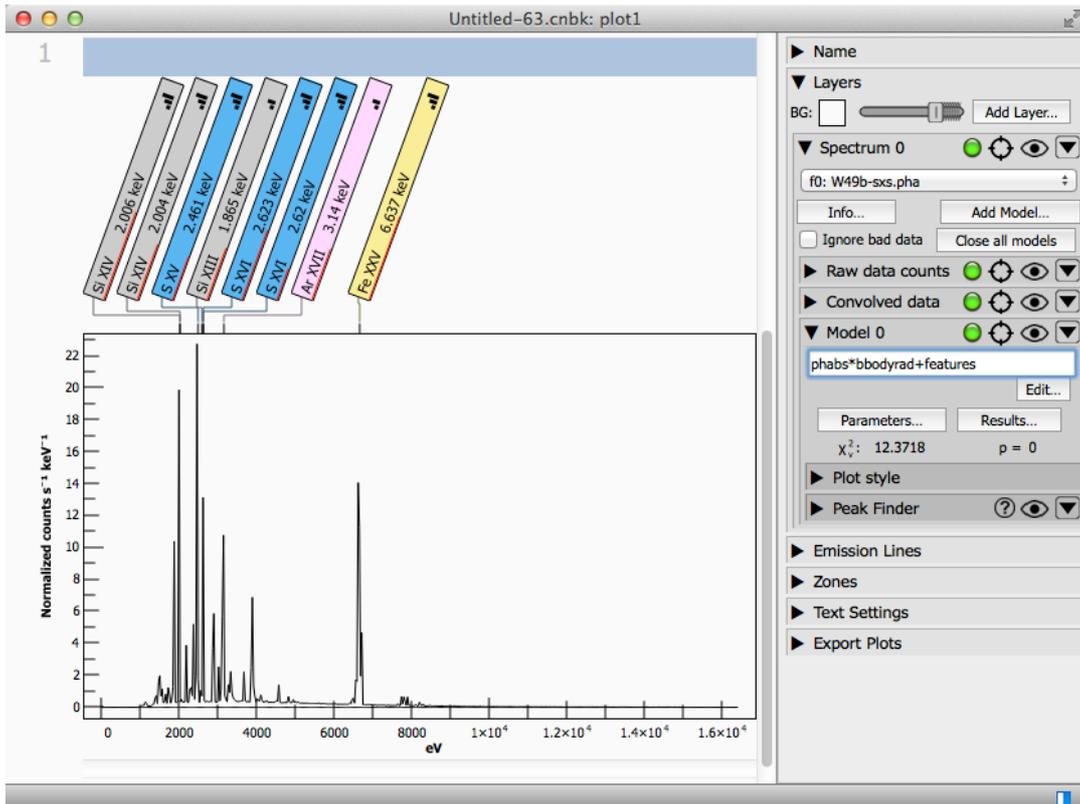
If there are fewer labels, the layout process tries to be a bit more intelligent. It organizes the labels into a set of `AnnotationGroups`. Each such group contains all the contiguous labels that partially overlap their immediate neighbors if displayed at their ideal locations. Each `AnnotationGroup` is then centered over its geometric center (though “center of gravity” might be a better choice).

This centering may cause the group to stick out past the edge of the pane. Hence we make iterative passes forward and backward, pushing annotation groups in from the edges as needed. This in turn may cause collisions among groups, and such collisions result in the merging of groups. When the iteration ends, there are one or more non-overlapping groups of labels, and all the labels are positioned as closely as possible to their respective peaks.

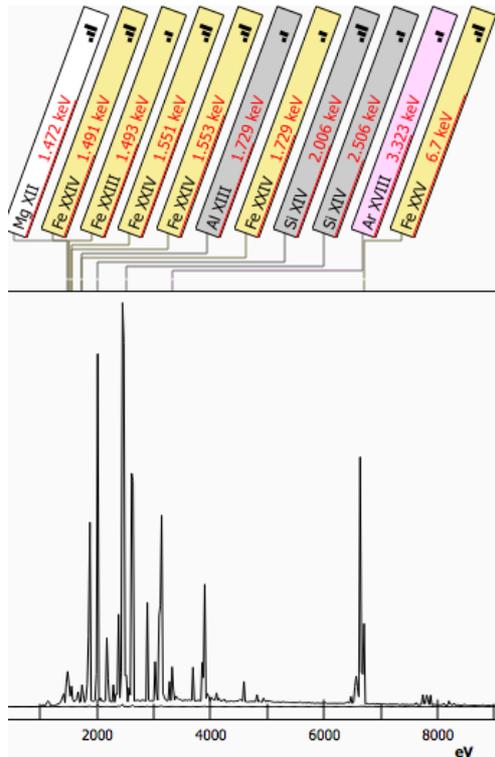
Next, the number of lanes needed to display the connecting lines is calculated. This may increase or decrease `mLineHeight`, in turn changing `mLabelHeight`. When the label height of a `SpectralLabel` is changed, we simply display more or less data according to its recomputed length. But `UserLabels` are more complex. If the height is restricted, a `UserLabel`'s thickness may change because its margins are reduced and the text is re-wrapped. This increases its `effectiveWidth` and may thus reduce the room for `SpectralLabels`.

...Which in turn reduces the number of lanes and increases the space for labels. So the entire label layout process iterates on more than one level and is fairly compute intensive. This is the main reason why I try to optimize the computes that are performed.

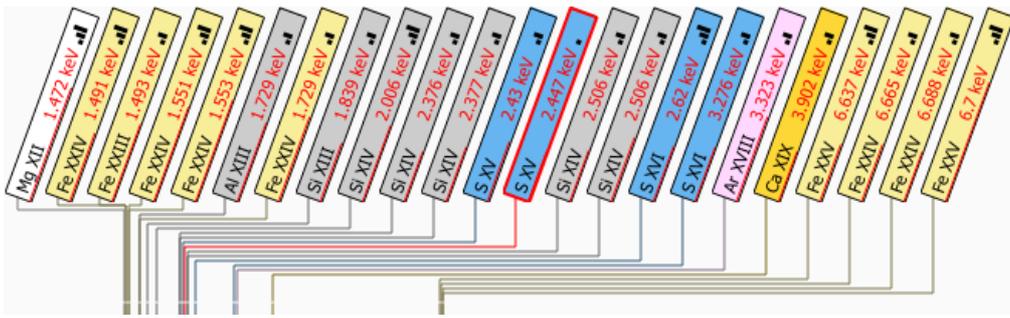
With the clutter set on maximum, the result looks like this. The predicted or calculated peak is about 1/8" in height, with a small break over it. The associated emission lines are shown at their actual position, so that at higher zoom levels any offset between the label and the peak should be clearly visible. Lines are color coded to make it easier to follow them, but they are still very crowded. The line label area can be resized to display more peak detail. There is a small red line at the lower right of each label. It shows the relative match quality of each peak with its emission line. It is mostly for debugging, and will probably be removed.



The clutter control omits poorer matches:



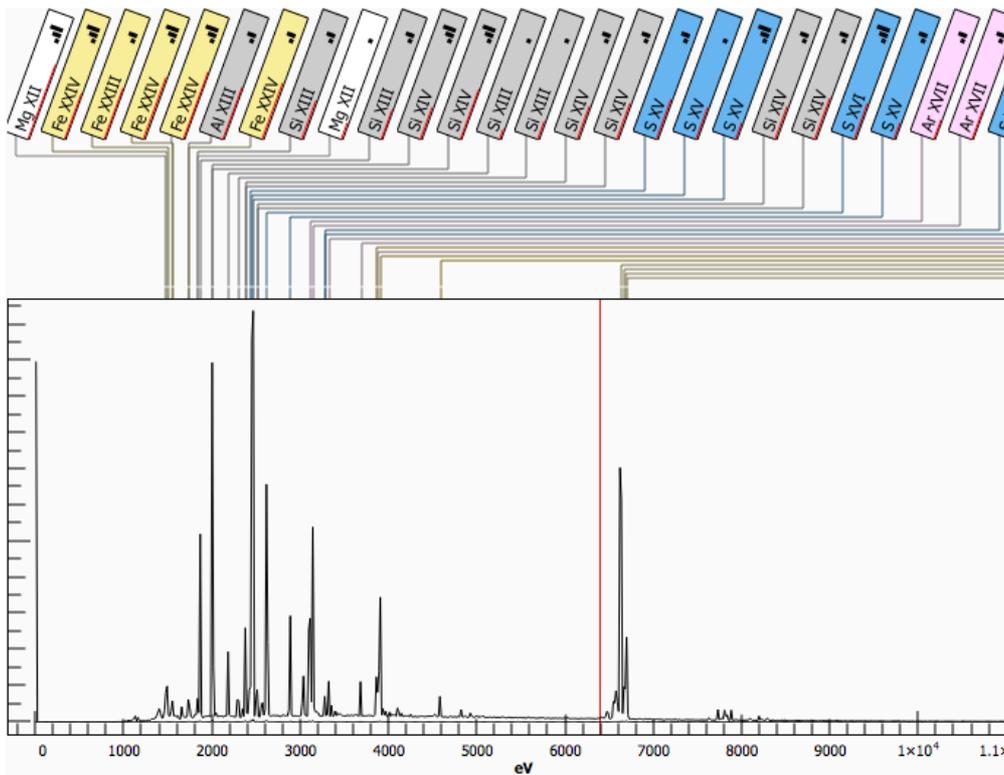
Another way to clarify the association between a peak and its label is to hover the mouse over its label, as with the S XV line below:



Double-clicking a label displays more details about it.

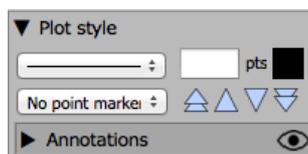
### 7.18.8.2.1.8 Panning and Zooming

Zooming in allows room for more labels. Unfortunately, the added labels require too many lanes for routing the connecting lines. Hence the maximum number of labels is limited regardless of zoom level. The sample below shows excessive labels, prior to the implementation of this limit.

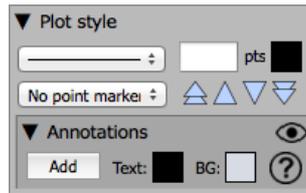


### 7.18.8.2.1.9 User Annotations

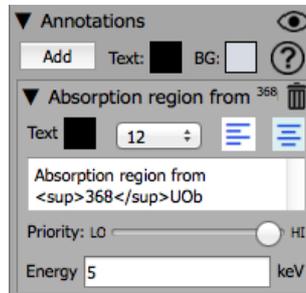
Like SpectralLines, UserAnnotations are anchored to a specific photon energy to mark an energy or region of interest. Adding a user annotation is a fairly uncommon operation, so it is hidden in the model's plot style panel:



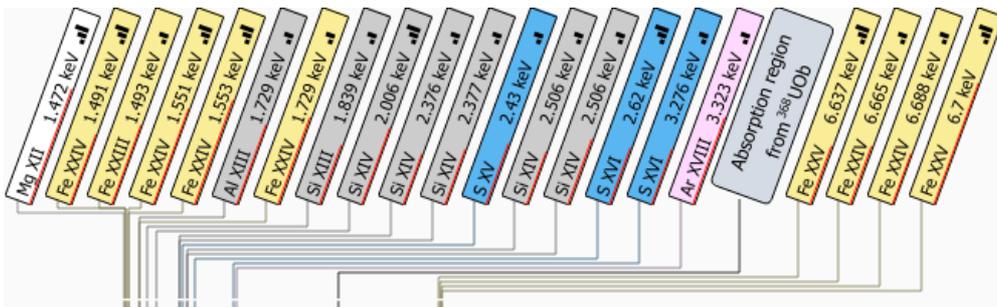
The show/hide icon applies to all user annotations. Open the annotations panel to add annotations:



This panel controls default text and background color. Add an annotation:



The text is shown at the assigned energy. The priority is used by the clutter control. At the default priority of 1, the annotation takes precedence over all emission lines and always appears. Lower settings cause it to be squeezed out if space is limited or the clutter control is lowered. Long annotations may appear truncated.



Annotations support a miniscule subset of HTML, as with the unobtainium isotope above. Use `<sub>` and `</sub>` for subscripts, `<sup>` and `</sup>` for superscripts, or `<b>` or `<i>` for bold and italic.

### 7.18.8.2.1.10 Mouse Input

`AnchoredLabels` are rotated at paint time, but the widget as a whole is not. The rotated label lies within the widget's unrotated rectangle, which just encloses it. Mouse input is not rotated, and it is delivered to the topmost label. So a mouse event handler must first send the mouse hit through an inverted transform to find its rotated position, then be tested against the label's rectangle, not the widget's.

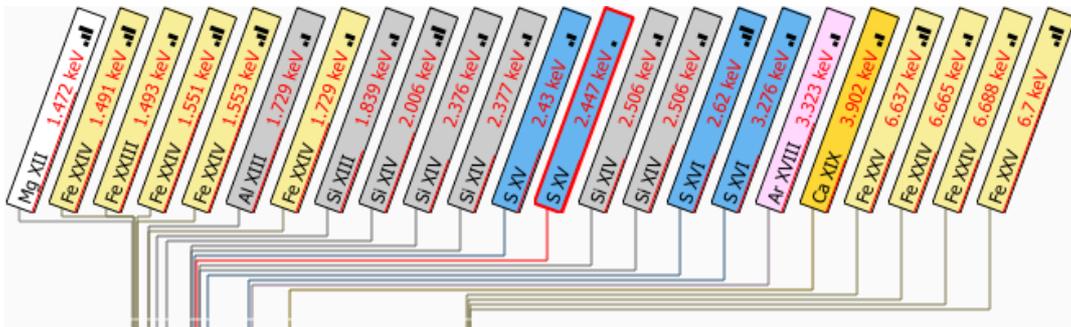
The rotation is done by saving the each label's painter's inverse transform at paint time (see `AnchoredLabel::rotatePainter()`):

```
mInverseTransform = painter.combinedTransform().inverted();
```

This tells us if the mouse point is in the label part of the widget. If it is, the mouse hit is for this widget. But if not, we cannot just set the event status to `ignore()`, because the event then propagates to the *parent* widget (the `PlotLineLabels`) instead of to sibling widgets that might be partially covered (the other `AnchoredLabels`).

The best solution I have found so far starts with marking the `AnchoredLabels` as transparent with respect to mouse events (`WA_TransparentForMouseEvents`). This way, any mouse events in the pane fall through to the `PlotLineLabels`. Then I do all the mouse event processing in the `PlotLineLabels`. For example, the `mouseMoveEvent` is overridden here to highlight the current label as the mouse passes over it, as an aid to tracing the

line that connects it to a peak. To make this work we first set mouse tracking on for the `PlotLineLabels`. Then, when a `MouseMoveEvent` arrives, we move backward through the list of labels, selecting the one that first matches by calling `AnchoredLabel::pointInLabel` to find it. In the example below, an S XV line is highlighted in red.



### 7.18.8.2.1.11 Selections

Users can choose a selection set to constrain the region for which peaks are found. One such selection set is the ignore region, if any. Users can create others. The following events trigger label updates:

- When the `CPPeakFinder` is initialized, it contains only “Entire spectrum,” as a selection set choice.
- When a selection set is added, all `CPPeakFinders` are notified so they can update their selection set popup menus. (One time when this happens is when the user enables ignore regions.)
- When a selection set is deleted, all `CPPeakFinders` are notified so they can update their selection set popup menus. If a selection that is in use by a `PlotLineLabels` is deleted, the choice reverts to entire spectrum, and the labels are recomputed.
- When a selection set’s associated spectrum is changed, all `CPPeakFinders` are notified so they can update their selection set popup menus. If the change renders a selection set inaccessible because it is no longer associated with a spectrum, any `PlotLineLabels` using it revert to the entire spectrum.
- When the user alters a selection set, for example, by changing a selection or adding/removing selections, a signal is passed to the tool and any `PlotLineLabels` using the selection update themselves.

In an earlier design, selection sets had types that were hints to the peak finder about the types of peaks to look for (emission, emission pair, absorption, etc.). The revised design uses selection sets to determine the subset of peaks to be matched. There are now fewer set types, and the types are just mnemonics for the user and do not presently have any influence on the way in which the peak finder operates (other than to restrict the subset of energy ranges).

Ignore regions are counterintuitive, since they are the inverse of the way other selection sets work, so at some point ATLAS will be changed to work in terms of “include regions” instead. This means, for example, the peak finding is easily restricted to the inverse of a spectrum’s ignore region.

### 7.18.8.2.2 Calculating and Fine-tuning Gaussians

This section discusses how ATLAS requests XSPEC to calculate the Gaussians, and how this interacts with user edits to models and parameters.

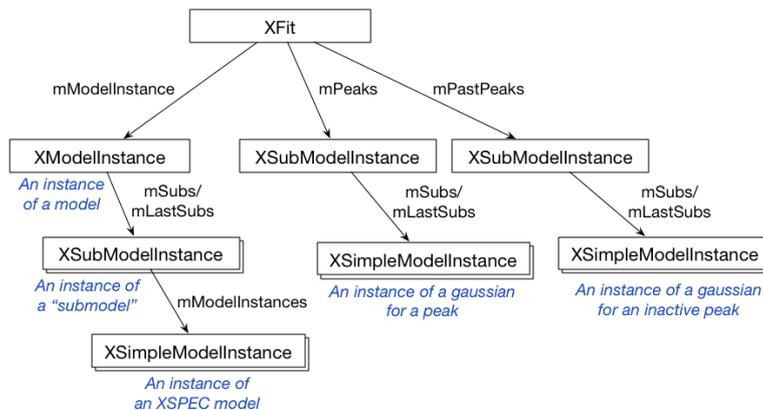
#### 7.18.8.2.2.1 User Models and Line-based Analysis

The Gaussians needed to find peaks are appended to a user model. So, for example, if the user model expression is `phabs*apec`, and we found three peaks, the model expression passed to XSPEC is `phabs*apec+gau+gau+gau`. Each of these five models have trees of parameters. The user should not edit the part of the model expression appended by the peak finder, so it is hidden from view. But the user may wish to edit the gaussian parameters. Fortunately, model editing and parameter editing are handled by different editors, so this asymmetry is easily implemented.

### 7.18.8.2.2.2 Model Handling

Once the user has selected a set of peaks by adjusting the peak sensitivity control, the next refresh command creates a gaussian model for each selected peak, and appends them to whatever the user selected for a model. When the fit completes, the new positions are used to adjust the calculated photon energy of each peak, and new emission lines are matched to the results. Then the user may fine-tune the parameters or make other adjustments to the peak-finding controls, and refresh again.

In order for this process to work, we need a means of establishing a correspondence between each gaussian model and its peak. Since the model expression processed by XSPEC is a combination of user-edited parts and automatically-generated parts, the model must be stored in separate pieces and combined only when appropriate. And if the spectrum file itself is changed, we need to discard the old gaussians and their parameters and start fresh. The user-edited model for a fit is stored in `mModelInstance`, below at left. We add `mPeaks`, which stores all the gaussian models for the selected subset of peaks. In addition, `mPastPeaks` stores any gaussian models for peaks that were previously visible (due to a different setting of the peak sensitivity indicator) but are currently not in use. This preserves their parameter settings in case they are used later.



Since the peak sensitivity control can be adjusted at any time, there may not be a one-to-one correspondence between `EmissionLineMatches` and the `XSimpleModelInstances` that represent their Gaussian models. Hence each `EmissionLineMatch` is assigned a peak identifier. This is a UUID (“universal unique identifier”—a large random number). This peak identifier is kept by each `XSimpleModelInstance` to provide the binding. (Pointers aren’t used here because the `EmissionLineMatches` are regenerated whenever a new convolved spectrum is generated.)

By default (i.e. most of the time) the peaks are stored as shown above, separately from the `mModelInstance`. But there are two exceptions:

#### When the user displays the parameter editor:

1. First, we call `preparePeaks()`, to ensure that `mPeaks` are up-to-date with respect to the current spectrum and peak sensitivity setting. This is described below.
2. Next, the peaks are temporarily appended to the list of `XSubModelInstances` in the `mModelInstance`. We add the pointer, not a copy, and `mPeaks` still owns the peaks: they are “on loan” to the model instance.
3. Next, the parameter editor is displayed, and the user sees both the parameters to their own models and, at the end, the gaussian parameters for the peaks. The dialog is modal, a good thing, because the loan is temporary.
4. After closing the dialog, the peaks are removed from the main instance. Any user edits to the user-defined model won’t disturb the peak finder parameters in any way.

#### At refresh time:

1. As above, we first call `preparePeaks()` to ensure that the models are up-to-date with respect to the currently selected peaks.
2. Also as above, the peaks are temporarily appended to the user model, as the last entry.
3. The command sequence for XSPEC is generated from the merged model.
4. When the refresh completes, the peaks are removed from the `mModelInstance`.
5. The `commandSequenceCompleted` processing updates the calculated positions of the `EmissionLineMatches`, and generates a new set of line labels.

The `preparePeaks()` function:

1. Moves all model instances from `mPeaks` to `mPastPeaks`.
2. Purges the `mPastPeaks` list of any instances whose peaks no longer exist in the current `EmissionLinePeaks` list. This happens if the convolved data is updated (typically because a new spectrum is selected).
3. Builds a new set of model instances from the current `EmissionLineMatches` list. (The ELMs include only the subset of peaks that are over the sensitivity threshold, while the ELPs contain all the peaks.) For each `EmissionLineMatch`, if its model instance is present on the `mPastPeaks` list, it is used; otherwise a new one is created. This preserves user parameter changes.

At the end of this process, `mPeaks` will contain an entry for each peak over the threshold, and `mPastPeaks` will contain any that still exist but are below the threshold.

### 7.18.8.2.2.1 Save/restore

When an `EmissionLinePeaks` is saved, the peaks and their UUIDs are stored. Downstream structures are regenerated when ATLAS restarts though. When an `XFit` is saved, its `mPeaks` and `mPastPeaks` are saved, included in the peak IDs.

### 7.18.8.2.2.3 Parameter Editing

In the example below, the `phabs` and `apec` models were selected by the user. The gaussians that follow were created by the peak finder. Peak finder gaussians are so identified (as Peak *n*), and if there is a line label assigned, it is shown as well, to underscore the association between each model and its peak. (Note that the line label is shown even if the Gaussian has not yet been computed, but in this case, it is marked as tentative.) Users can fine tune the values. After a fit is performed, the user can revert to the user default or to the peak finder default.

If the user sets the `LineE` parameter to something wildly different, all hell will probably break loose.

After editing, a refresh computes a new fit and updates the peak labels.

The screenshot shows a software interface for editing parameters. The main window is titled "Parameters for 'Spectrum 0, model0'" and contains a tree view of models. The selected model is "phabs\*apec". Under "phabs", there are parameters for `nH` (value: 4.77642, Delta: 0.001) and `apec`. Under "apec", there are parameters for `kT` (value: 1.61081, Delta: 116050), `Abundanc` (value: 1, Delta: -0.001), and `Redshift` (value: 0, Delta: -0.01). Below these are three Gaussian models. The first is "Peak 1" with "Mg XII 1.472 keV 3→1" and parameters `LineE` (1471.58, Delta: 50) and `Sigma` (100, Delta: 50). The second is "Peak 2" with "Fe XXIII 1.493 keV 52→1" and parameters `LineE` (1492.5, Delta: 50) and `Sigma` (100, Delta: 50). The third is "Peak 3" with "Fe XXIV 1.553 keV 11→1" and parameters `LineE` (1551.5, Delta: 50) and `Sigma` (100, Delta: 50). On the right, the "Parameter Options" panel includes fields for "Name", "Tool Name", and "Comment", and buttons for "Open all models", "Close all models", "Open all details", "Close all details", "Show model documentation", "Restore 'factory' defaults", "Restore last user values", "Restore last fit results", "Save copy as new model...", "Cancel", and "OK".

## 7.18.8.3 Deprecated Line-based Analysis Design

### 7.18.8.3.1 Line-based Analysis

This is a first cut at a design for line-based analysis (LBA). At its heart is a new meta-model, `multigaussian`. `multigaussian` is a wrapper for a set of zero or more `gaussian` models, each of which encompasses a range of interest in a spectrum. The point of interest may be an emission or absorption line, or a group of closely spaced lines.

A user selects `multigaussian` from the list of models in the normal manner. The Fit details... option offers some additional settings specific to this model:

- A sensitivity slider, set by default at the halfway mark, generates a value between 0 and 1 that is used to increase or decrease the number of peaks (and troughs) for which Gaussians will be computed. Increasing sensitivity values will pick out more peaks.
- A popup labelled “Overall range” selects a photon energy range within which peaks are located. This defaults to “Entire spectrum,” but can also be set to any existing selection. (The user must first create a selection on the graph in order to refer to it, and it must refer to the same spectrum associated with the model.)
- A popup labelled “Selection set” selects any existing selection set as the list of ranges for calculating Gaussians. This defaults to “Create new set” at first, and later defaults to the automatically created set. Since the set is created automatically by default, it is rarely necessary to change this. Its main benefit is to be able to reset to “Create new set.”

Since all these settings have rational defaults, the user normally doesn't need to bother with them.

When the user hits refresh, and the model expression includes “`multigaussian`”:

1. ATSAAL decomposes a model expression into its `multigaussian` and “continuum” sections, lumping everything that isn't `multigaussian` into the continuum department. For example, `phabs*(bbodyrad + multigaussian)` is decomposed into `phabs*bbodyrad` and `multigaussian`. **Randall says this decomposition is not enough: “Most flexible would be a system where you could have multigaussian just as one of many models in an expression, e.g. `phabs*(apec + powerlaw + phabs*blackbody + multigaussian)`.”**
2. If the model expression contains a continuum component, and there is no prior data set for the background component, or the background component expression or parameters have changed, the background part of the expression is passed separately to XSPEC for computation, and the resulting curve is cached for subsequent use. Otherwise the existing cache is used.
3. Next, ATSAAL calls `multigaussian(QuPoints* data, QuPoints* background, PESelection* overallRange, double sensitivity)`, where `data` is the convolved data and `background` is the computed background fit. `overallRange` is the range within which peaks should be located, and defaults to the entire spectrum, and `sensitivity` is a value from 0-1, defaulting to 0.5, where higher values select more peaks. The function computes and returns a `SelectionSet`—a list of `PESelections`, one for each selected peak or trough. Each selection is a photon energy range and an indication of the type of selection (e.g. emission line, absorption line, pair of emission lines).
4. The new selection set, if any, is merged into any existing selection set. The merge process is described shortly. Any new selections are assigned UUIDs that will provide a binding to associated `gaussian` models in the `multigaussian`.
5. The selections are sorted into order by their central photon energy. The list of `gaussians` in the `multigaussian` are adjusted to create a one-to-one mapping between selections and their `gaussians`, a process that retains any settings.
6. `setGaussianParameters(XModelInstance* gaussianModel, PESelection* selection, QuPoints* background)` is called for each `gaussian` model in the `multigaussian`. `gaussianModel` is the model instance describing a given Gaussian model, including its current parameters. `selection` is the range within which the model should be applied. And `background` is the separately computed background curve. This updates the `gaussianModel` parameters as needed.
7. Now the entire model expression is passed to XSpec for analysis. The returned data is used to select likely labels. For each `gaussian` model, ATSAAL calls `selectLabels(XModelInstance* model, QuPoints* data, PESelection* selection)`. `model` is the Gaussian model instance. `data` is the curve computed for the entire model, and `selection` is the range of the Gaussian. The function uses the selection type as a hint. So, for example, it will look for two likely labels if the user has marked it to expect a pair of emission lines. The function returns a `QList<EmissionLineCpp*>`—a list of emission or absorption lines—as a result. The selection type is

interpreted as a hint: if the algorithm is instructed to look for two peaks but only one appears to be present, it will return only one line.

8. ATXSAL displays the returned emission lines over the graph.

Users may adjust selection boundaries by hand, or change selection types, or even add and remove selections to the selection set. They can even create selection sets by hand.

### 7.18.8.3.1.1 Merging

When a newly computed set of selections (say, with a new threshold) is merged with an existing set, there is ambiguity about how the merge should proceed, especially given that users can change selections by hand. So I propose a simple algorithm as a start. Both lists are sorted by photon energy order. For each entry on the new list, if a selection on the old list is within  $x$  units, any settings are copied from the old entry to the new one. **What is  $x$ ?**

### 7.18.8.3.1.2 The Functions

The functions described here are needed. They are shown with C++-style arguments, but if the algorithms are written in python, I can convert them to python-compatible objects. **Note that some means is needed for python access to the emission lines database. It might be better to write selectLabels in C++ to avoid this, or write it in Python and I will translate.**

### 7.18.8.3.1.3 Other Issues

- If more than one multigaussian is included in a model expression, the extras are ignored. This should be reported as an error.
- **You mentioned that absorption lines must be combined multiplicatively. I don't fully understand the implications, since a gaussian is additive. If a multigaussian consists of an absorption gaussian and an emission gaussian, how is this represented as a model expression? Randall's reply: "The absorption case is based on the XSPEC gabs multiplicative model, not the additive gauss model."**
- **How should this interact with ignore regions? Randall: "Interesting question...let's leave for the discussion."**

### 7.18.8.3.1.4 Meeting 2016-05-03

**Randall points out that I may be overthinking this tool, that its implementation can be simple and it will still produce useful results.**

Finding peaks isn't as subtle as it seems; don't need to worry about the subtle cases. An example from Root is in the next section. This phase doesn't require interaction with XSpec, so the whole two-phase thing goes away.

Find the peaks first, show as lines instead of ranges.

If  $x$  positions show spans at all, they shouldn't be marked with lines and shouldn't be adjustable, or, if they are adjustable, adjusting one edge adjusts the other too.

Looking up matches shouldn't be too hard either.

### 7.18.8.3.1.4.1 Root Peak Finder

```
// Illustrates how to find peaks in histograms.  
// This script generates a random number of gaussian peaks  
// on top of a linear background.  
// The position of the peaks is found via TSpectrum and injected  
// as initial values of parameters to make a global fit.  
// The background is computed and drawn on top of the original histogram.
```

```

//
// To execute this example, do
// root > .x peaks.C (generate 10 peaks by default)
// root > .x peaks.C++ (use the compiler)
// root > .x peaks.C++(30) (generates 30 peaks)
//
// To execute only the first part of the script (without fitting)
// specify a negative value for the number of peaks, eg
// root > .x peaks.C(-20)
//
//Author: Rene Brun

#include "TCanvas.h"
#include "TMath.h"
#include "TH1.h"
#include "TF1.h"
#include "TRandom.h"
#include "TSpectrum.h"
#include "TVirtualFitter.h"

Int_t npeaks = 30;
Double_t fpeaks(Double_t *x, Double_t *par) {
  Double_t result = par[0] + par[1]*x[0];
  for (Int_t p=0;pRndm()*980;
       par[3*p+4] = 3+2*gRandom->Rndm());
  }
  TF1 *f = new TF1("f",fpeaks,0,1000,2+3*npeaks);
  f->SetNpx(1000);
  f->SetParameters(par);
  TCanvas *c1 = new TCanvas("c1","c1",10,10,1000,900);
  c1->Divide(1,2);
  c1->cd(1);
  h->FillRandom("f",200000);
  h->Draw();
  TH1F *h2 = (TH1F*)h->Clone("h2");
  //Use TSpectrum to find the peak candidates
  TSpectrum *s = new TSpectrum(2*npeaks);
  Int_t nfound = s->Search(h,2,"",0.10);
  printf("Found %d candidate peaks to fit\n",nfound);
  //Estimate background using TSpectrum:Background
  TH1 *hb = s->Background(h,20,"same");
  if (hb) c1->Update();
  if (np < 0) return;

  //estimate linear background using a fitting method
  c1->cd(2);
  TF1 *fline = new TF1("fline","pol1",0,1000);
  h->Fit("fline","qn");
  //Loop on all found peaks. Eliminate peaks at the background level
  par[0] = fline->GetParameter(0);
  par[1] = fline->GetParameter(1);
  npeaks = 0;
  Double_t *xpeaks = s->GetPositionX();
  for (p=0;pGetXaxis()->FindBin(xp);
       Double_t yp = h->GetBinContent(bin);
       if (yp-TMath::Sqrt(yp) < fline->Eval(xp)) continue;
       par[3*npeaks+2] = yp;
       par[3*npeaks+3] = xp;
       par[3*npeaks+4] = 3;
       npeaks++;
  }
  printf("Found %d useful peaks to fit\n",npeaks);
  printf("Now fitting: Be patient\n");
  TF1 *fit = new TF1("fit",fpeaks,0,1000,2+3*npeaks);
  //we may have more than the default 25 parameters
  TVirtualFitter::Fitter(h2,10+3*npeaks);
  fit->SetParameters(par);
  fit->SetNpx(1000);
  h2->Fit("fit");
}

```

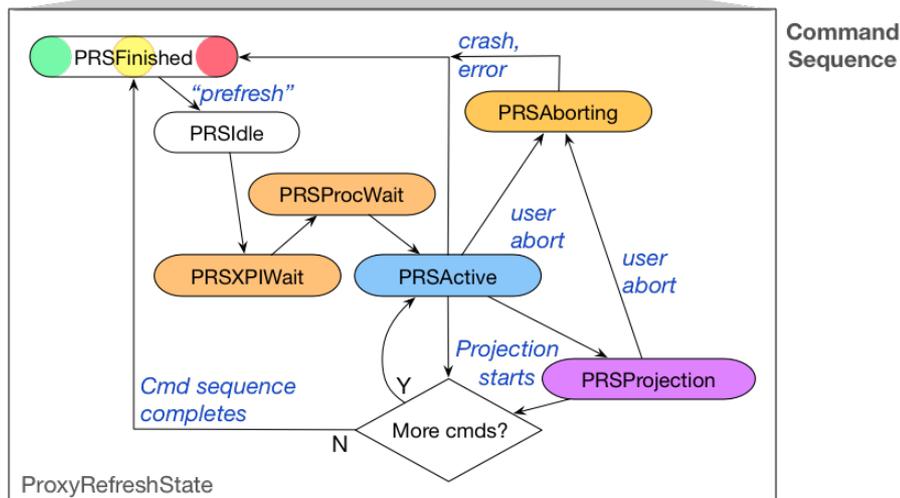
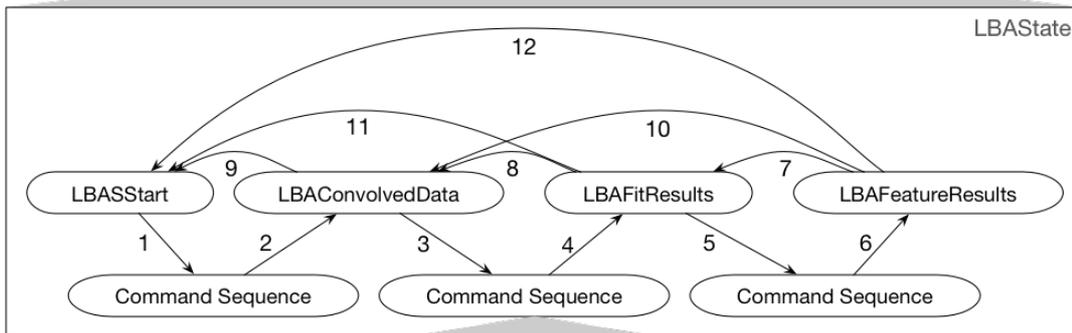
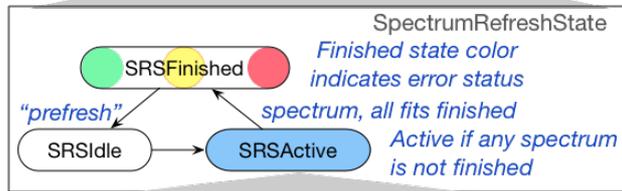
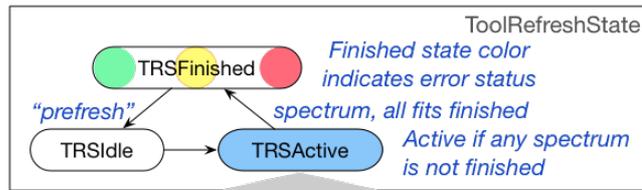
### 7.18.8.3.2 LBA States and Command Processing (Deprecated)

See instead Refresh State Tables, and Refresh Phases.

In addition to the normal tool and command processing states, [also discussed here](#), Line-based Analysis interposes another set of steps for each fit. These states are necessary because some analysis steps cannot proceed until others have completed, and we wish to minimize the computes performed by each refresh cycle. Each time the user hits refresh, we move forward through this table as far as possible. Edits to tags, model expressions, or model parameters move backward through the state table, as far as needed to ensure that the next refresh cycle brings everything up to date. We call these `LBAS` states, but the first three are used for all fits, not just those that include LBA:

1. `LBAS``start` is the initial state for all fits, including those which don't use LBA.
2. When a spectrum or spectrum and model are refreshed, candidate peaks are located in the convolved data. So `LBAS``ConvolvedData` is a state that has this data, but no fit results. If the user has specified a model as well, a refresh command also fits the model expression **except for features**, and enters state `LBAS``FitResults`. This is because the fit results for the rest of the expression must be acquired **and locked** prior to computing the peak Gaussians.
3. `LBAS``FitResults` means that a set of fit results are present and consistent with the current model expression and parameters. If the model includes `features`, entering this state automatically progresses to the next state as well.
4. `LBAS``FeatureResults` is entered only if the model expression includes `features`, the features have been computed, and they are consistent with the input parameters for the features Gaussians.

This diagram is a superficial representation of the process. Colored states are represented to the user with correspondingly colored "LEDs." The red, yellow and green states indicate a finished state and its error status. The finished state is also the starting state. Blue is busy, orange is waiting, and purple is projection. LBA states are not represented by state indicators, hence are uncolored. Each tool, spectrum, and fit has its own state table.



LBAS state transitions are as follows:

1. LBASstart→PRSFinished. This transition happens if a spectrum does not yet have a model and the user hits refresh. It issues the command necessary to obtain raw counts and convolved data. The refresh runs the whole command sequence loop.
2. PRSFinished→LBAConvolvedData. Occurs when convolved data is successfully received. If an error or warning occurred, this state transition does not occur. If the spectrum contains a model, state transition 3 begins automatically.
3. LBAConvolvedData→PRSFinished. Occurs when user refreshes after adding or modifying a model expression or model parameters. The refresh runs the whole command sequence loop. **Only the portion of the model expression that does not include features is executed, since these results must be received and the parameters locked before executing the full model expression.**
4. PRSFinished→LBAFitResults. Occurs only when the fit finishes successfully. (An error remains in PRSFinished.) Results, if any, are displayed. If the model expression contains features, model fit parameters are locked, features Gaussians are appended to the model, and state transition 5 begins automatically.
5. LBAFitResults→PRSFinished. Occurs when user refreshes when fit results are available but features have not

yet been (re-)computed.

6. `PRSFinished`→`LBAFeatureResults`. Occurs when results for fit and Gaussians become available. Emission line labels are computed and displayed for the results.
7. `LBAFeatureResults`→`LBAFitResults`. Occurs when the user modifies any parameter associated with the features model, or any LBA selection parameter that requires recalculation of the features Gaussians (e.g. change in peak sensitivity). This transition ensures that the next refresh command will recompute the features Gaussians. (Modifying Gaussian parameters has no effect if the state is not `LBAFeatureResults`.)
8. `LBAFitResults`→`LBAConvolvedData`. Occurs when the user modifies the model expression or any model parameter except for a features parameter. This ensures that the next refresh will recompute the fit (and possibly the features as well).
9. `LBAConvolvedData`→`LBASStart`. Occurs when the user alters the selected tag or selects a new tag, thus changing the spectrum. The next refresh will recompute the convolved data and all associated fits and features.
10. `LBAFeatureResults`→`LBAConvolvedData`. Occurs when the user modifies the model expression or any model parameter except for a features parameter. Ensures that the next refresh will recompute the fit and features.
11. `LBAFitResults`→`LBASStart`. Occurs when the user modifies a tag or selects a new one. Recomputes everything associated with the spectrum on the next refresh.
12. `LBAFeatureResults`→`LBASStart`. Occurs when the user modifies a tag or selects a new one. Recomputes everything associated with the spectrum on the next refresh.

## 7.19 Persistent Pointers

In the past ATXSAL has used UUIDs for persistent pointers, but the UUIDs, essentially just a typedef for `qulonglong`, are not typesafe, and ATXSAL was not rigorously consistent in how they were saved and restored. They are used for reconstructing pointers after reading a notebook. They are also often used in signaling in place of pointers. When a signal is delivered to an owning object, the ID is looked up, and if it isn't found, the signal is ignored. This reduces the chance of a crash as a result of a tangle of pointers at delete time.

The generic nature of the UUIDs made it hard to determine the kind of tag in some contexts. This could not be solved with typedefs alone, since multiple identically mapped typedefs cannot be declared as metatypes.

Now ATXSAL uses typed UUIDs, trivial classes that combine a data type and a UUID. For example, `FileTagID` is an ID that represents a `FileTagSet`, and `DataSetID` represents any `DataSet`, including subclasses `XPlottable` and `XFit`. When you construct a typed ID, it defaults to invalid (represented internally by a 0 UUID). Usually this state is analogous to a null pointer. Use `ID.init()` to assign a random UUID, or `ID.setInvalid()` to set it back to an invalid state.

The classes are generated via template, and designed to be saved/restored, used in queued signals, and used in `QVariants`. See `UUID.cpp/h`. To use a typed ID in a `QVariant`:

```
FileTagID ID;
ID.init();
QVariant v;
v.setValue(ID);

FileTagID ID2;
ID2 = v.value<FileTagID>(v);
```

To save/restore a typed ID:

```
void MyObject::write(QTextStream& stream, int level)
{
    stream << indent(level) << "<MyObject myID=\"\" << mID.typedID() << \"\" />\n";
}

void MyObject::read(QXmlStreamReader& reader, const Version& notebookVersion)
{
    while (!reader.atEnd())
    {
        reader.readNext();
        if (reader.isStartElement())
        {
            if (reader.name() == "MyObject")
            {
```

```

    QDomStreamAttributes attributes = reader.attributes();
    FileTagID ftcID = FileTagID(attributes.value("myID").toString());
}
}
else if (reader.isEndElement())
{
    if (reader.name() == "MyObject")
    {
        return;
    }
}
}
}
}

```

The saved line looks like this:

```
<MyObject myID="FileTagID 223b4c911bf003a0" />
```

To add a new typed ID:

- Add the type to the `UType` enum in `atsal.enum`.
- In `UUID.h`, define the type, e.g. `DEFINE_UATYPE(FileTagID);`
- Declare the metatype, e.g. `Q_DECLARE_METATYPE(FileTagID);`
- Add the implementation to `UUID.cpp`, e.g. `IMPLEMENT_UATYPE(FileTagID)`
- Register the metatype in `ATBaseApp::ATBaseApp`, e.g. `qRegisterMetaType("FileTagID");`

## 7.20 Plots

### 7.20.1 Coordinate Systems

It is tempting to use Qt's world transform to handle graph scaling, but there are two reasons why this doesn't work. First, the scaling is asymmetric: the width and height of the graph are altered independently. So, for example, text would not have a normal aspect ratio. Second, we want line thickness to remain consistent regardless of scale.

But if we don't use Qt's affine transformations to adjust scale, this leads to a second problem: how to get reasonable results on screens with varying pixel density. At the time of this writing, most monitors hover around 96 pixels/inch; a few are over 200. So a line 96 pixels in length appears about an inch in length on the current crop of monitors, but half an inch or less on high rez monitors. So we use the default transformation (1 pixel is about 1/96th of an inch) for low rez screens, and we scale "pixels" to screen pixels on higher rez devices. This means that many of today's screens will render text and graphics in sizes that vary with the actual screen resolution, but a 1-pixel wide line will remain a single pixel wide, improving legibility. On future screens, the mapping will scale text and graphics to remain the same actual size on any display.

Another rendering issue is apparent line width. Suppose a 1-pixel wide vertical line is drawn at an  $x$  coordinate of 10 pixels, and another at 20.5 pixels. The first is rendered one pixel wide on a low rez device. The second is antialiased, appearing two pixels wide, with translucent pixels. The two are as perceptually identical as possible, given the limits of a low resolution rendering device, but they look quite different. For example, a graph graticule looks awkward when rendered in this way. To work around this problem on low-rez devices, we round the calculated offset to a pixel boundary, drawing lines that are not quite uniformly spaced but appear uniform in thickness. This adjustment is not made for graph lines though, since positional accuracy is more critical here and aliasing variations from different line angles swamp this effect. At higher resolutions, pixel rounding is never performed.

Line thickness of plot traces is user-specified, and may be fractional. The value is scaled to produce roughly the same result on monitors of any resolution. That is, a value of 1.5 points corresponds to  $1.5/72$ s of an inch in width. On low rez devices, values that are close to an integral number of pixels are rounded to an integral value. On high rez devices, no rounding is necessary.

So the world transform is set to map points to pixels only for higher rez output devices, in an attempt to improve readability on lower rez devices. And graphs are scaled "manually," independently, in the horizontal and vertical directions as needed.

### 7.20.2 Monitor Resolution and Screen Size

Most of today's graphical user interfaces operate in pixel coordinate systems. On the relatively low-resolution devices in widespread use today (<120 pixels per inch or thereabouts), pixels are perceptible, and scaling produces objectionable artifacts. Hence the intuitive solution—using a floating point coordinate system and scaling to each display's actual resolution—produces poor results. This issue is major enough to have stalled monitor resolutions at around 100 ppi for many years.

Apple, the only vendor in control of both hardware and software, introduced “retina” displays with exactly twice the resolution of their previous displays, and redefined a point to correspond to two pixels. Since multipliers like 1/2, 2, 4, ... do not produce artifacts, the doubling produces extremely good results for all existing programs. (Scaled photos look fuzzy, but not distorted.) New programs can be written to take better advantage of the increased resolution, by supplying higher rez artwork or working at subpixel resolution. So this is a stopgap.

The other vendors cannot take this approach, because monitors are available in a range of resolutions. Microsoft introduced several options for handling this. (See *Qt and High-resolution Displays*.) The options amount to automatic scaling, with attendant aliasing; or enough information to perform your own scaling. Linux offers a similar approach.

None of which matters ... right? After all, Qt's job is to hide these pesky platform-specific solutions behind a flawless abstraction. Except that Qt also grew up in the pixel coordinates era, and has the same constraints. You don't draw a text box that is 6 cm wide and 1 cm high; you draw it using pixel coordinates. At the time of this writing (Qt 5.4), there is experimental support for a scale factor. But using this won't produce consistently good results on its own. As long as low-rez monitors are around, ATLAS needs to treat them as a special case. When all monitors are over 200 ppi in resolution, scaling works extremely well.

Qt supports two distinct mechanisms for imaging. All `QWidget`s employ an integer coordinate system which normally has a 1-to-1 correspondance to pixels. It also has a low-level graphics drawing system that has all the flexibility of PostScript, including a floating point world coordinate transform that produces excellent scaled results on even low-rez monitors. A new `QScreen` class provides information on the physical size of the screen. This should be enough information to (re-)write ATLAS to produce optimal results on any monitor, running under any OS.

So here are the requirements:

- Avoid scaling on low-rez devices in order to minimize scaling artifacts. ATLAS will turn off scaling on such devices, with the consequence that user interface objects will vary somewhat in actual size on various monitors.
- Activate Qt's scaling on high-rez devices in order to keep user interface elements about the same actual size regardless of monitor resolution. (This may force a switch from vendor-specific user interface elements to Qt's “fusion” user interface elements, which are designed to scale more cleanly.)
- Design ATLAS's user interface elements to scale themselves to the actual font metrics, reducing cases where fonts are clipped or truncated. (In many cases this logic is already present, to simplify translation.)
- Apply scaling based on the number of pixels per inch, *not* the monitor size. That is, a window shouldn't appear larger simply because it is on a larger monitor. Its actual height should remain the same, since most people with larger monitors want more real estate, not larger objects.
- Use vector artwork or high resolution pixmaps to get good results at any resolution.
- Support multiple monitors with varying resolutions.
- Scale line widths on all devices, but on low rez devices, round where possible to an integral number of pixels.
- When painting directly, for example, when drawing graphs, set an affine transformation to get the desired scale factor (unless Qt does this automatically).

We will define a function, `lowResolution(QWidget*)`, which returns true if the monitor on which a widget is displayed is considered to be low in resolution. This returns true for monitors below about 120 pixels per inch, which encompasses most of the monitors available at time of this writing. Only “Retina”-style monitors, to borrow Apple's marketing term, will return false. Presumably at least, within five years most monitors will return false. Qt 5 introduces a [QScreen class](#) that includes physical size information.

So ATLAS will use the (currently experimental) scaling mechanism to position user interface elements, and an affine transformation to scale low level graphics. It will use workarounds as described above to achieve optimal results on low-rez displays.

## Qt and High-resolution Displays

### [Qt and High-resolution Displays](#)

(Remote URLs are not yet included inline in PDFs.)

## 7.20.3 Zone Synchronization

Any zone may be synchronized to another zone, meaning that the horizontal axis units and range are synchronized. This happens by default whenever a new zone is created and at least one existing zone is present that uses the same horizontal axis. A plot zone may be set via popup to unsynchronized, or synced to a different zone. It is not necessary for synced zones to be adjacent. If multiple zones are synced, and a target zone is deleted, the remaining zones remain synced to each other. It is legal to mix any combination of synced and unsynced zones in a multizone plot. (It is legal for the user to link zone A to B, and B to A; the software needs to watch out for this case.)

A synced zone:

- Pans and zooms if any other synced zone pans or zooms.
- Does not scale vertically in response to vertical scaling of other synced zones.
- Shows the current selection range established in any synced zone.
- Shows the current graph cursor of any synced zone.
- Displays the same units as other synced zones.
- Shows a horizontal scroll bar only for the bottommost synced zone.
- Shows only its own plot line labels and user annotations.
- Responds to an added or deleted layer to any synced zone by adjusting its total range to encompass all the layers in all the synced zones.

Unsynced zones:

- Have their own horizontal scroll bar.
- May contain any graph type.

When selecting a zone for displaying a layer, users can choose only a new zone or an existing zone with the same horizontal axis. If all the plot layers for a zone are deleted, the zone disappears. Layers may be moved to other zones with the same units after creation.

To implement the synchronized pane, a `PlotMultizone` contains up to four `SyncPanes`. Each `SyncPane` contains up to four `PlotZone` instances, one per synchronized pane. All the `SyncPanes` are superimposed and transparent, and each `SyncPane` has a single horizontal scroll bar. `QRegions` are used to restrict each overlapping `SyncPane` to the `PlotPanes` it contains, so mouse events are delivered to the correct widgets.

When the first zone is added to a `SyncPane`, the data to be graphed determines the `SyncPane`'s horizontal and vertical physical type, as well as the default units used to label the axes. Later zones must match these types, and their default units are adjusted to match. A change to the units for a `SyncPane` is applied to all its zones (and all their layers).

## 7.20.4 Resize Protocol

Since many widgets must change in unison, ATSA uses a two-phase process when resizing. Many widgets are fixed in one dimension, so a `resize()` step sets those dimensions first. (For example, a horizontal widget displaying the axis units has a height dictated by the style's font and size.) Then a `resizeContents()` pass sizes each widget around its contents.

First, the width of the `PlotPane` is adjusted by any of several zoom controls, and should be set prior to starting the refresh cycle. Similarly, the `PlotLineLabels` and `PlotPane` height is set manually by the user, by dragging the line along the top of each widget, and these pane heights should be set before beginning a refresh.

- Call `PlotMultizone::resize()`. This function's job is to tell all the widgets to set any fixed dimensions, for example, to adjust the height of the horizontal widget that displays units (but not its width) to fit the text in the font and size specified by the style information. This starts by telling the `PlotAnnotations` to set their heights based on their contents. (Though this will probably happen automatically during text entry.)
- Next, this calls `PlotZone::resize()`. This determines the width of the `PlotVertAxisUnits`, `PlotVertAxisNumbers`, and `PlotVertAxisTicks`, defining the space left for the visible portion of the `PlotPane`.
- Next, we call `PlotPane::resize()`. This tells the `PlotSelectionBar`, `PlotHorizAxisTicks`, `PlotHorizAxisNumbers`, and `PlotHorizAxisUnits` to set their sizes vertically, based on display styles. (The `PlotLineLabels` widget is vertically sized manually.) Then it resizes these elements horizontally to match the `PlotPane`.
- Next, `PlotMultizone::resizeContents()` begins adjusting outer elements to conform to inner settings. First it

- calls `PlotPane::resizeContents()` to adjust the height of the pane if needed to fit the elements within.
- Next, `PlotMultizone::resizeContents()` adjusts each `SyncPane`'s width to lie inside the vertical axes. It sets the `PlotAnnotations` widths to match.
- Now that the `PlotPane` settings are known, the `PlotZone` is told to `resizeContents()`. Querying the `PlotPane` settings, it positions and sizes the vertical axes vertically so they line up with the `PlotLayers` portion of the `PlotPane`.
- Next, `PlotMultizone::resizeContents()` calls each `SyncPane`'s `resizeSyncPane()`. This positions the `PlotPanes` and `PlotZones` within a `SyncPane`, and also adjusts the `PlotAnnotations`.
- Finally, the `PlotMultizone` is sized.

### 7.20.4.1 Those Damned Tick Marks!

There are several supported styles of tick marks, and they generally require different kinds of handling. A single plot may use tick marks, a graticule, or neither. The cases are listed below. The horizontal and vertical axes are set separately. A plot can have only a single horizontal axis, but it may have two dissimilar vertical axes, one on each side.

<i>Style</i>	<i>Description and Implementation</i>
<code>TSNone</code>	No tick marks. All tick marks and numbers are hidden.
<code>TSInner</code>	Tick marks appear inside the left axis. (In the special case of two dissimilar plot layers, the second axis appears inside the opposing axis.)
<code>TSOuter</code>	Tick marks appear outside the left axis. (In the special case of two dissimilar plot layers, the second axis appears inside the opposing axis.)
<code>TSInnerBoth</code>	Like <code>TSInner</code> , but the tick marks are mirrored by both opposing axes.
<code>TSOuterBoth</code>	Like <code>TSOuter</code> , but the tick marks are mirrored by both opposing axes.
<code>TSGraticule</code>	Lines crossing the entire plot area, like graph paper, are shown. It is legal (if weird-looking) for one axis to be tick marks and the other graticule lines.

Case `TSGraticule` is handled by the `PlotGraticule` class, which contains both a `PlotVertAxisTicks` and a `PlotHorizAxisTicks`. These classes display the ticks as lines for this case. The other cases employ separate instances of `PlotVertAxisTicks` (one for the left and right edge) and `PlotHorizAxisTicks`, placing them inside or outside of the plot axis as needed. These instances are part of the `PlotZone`.

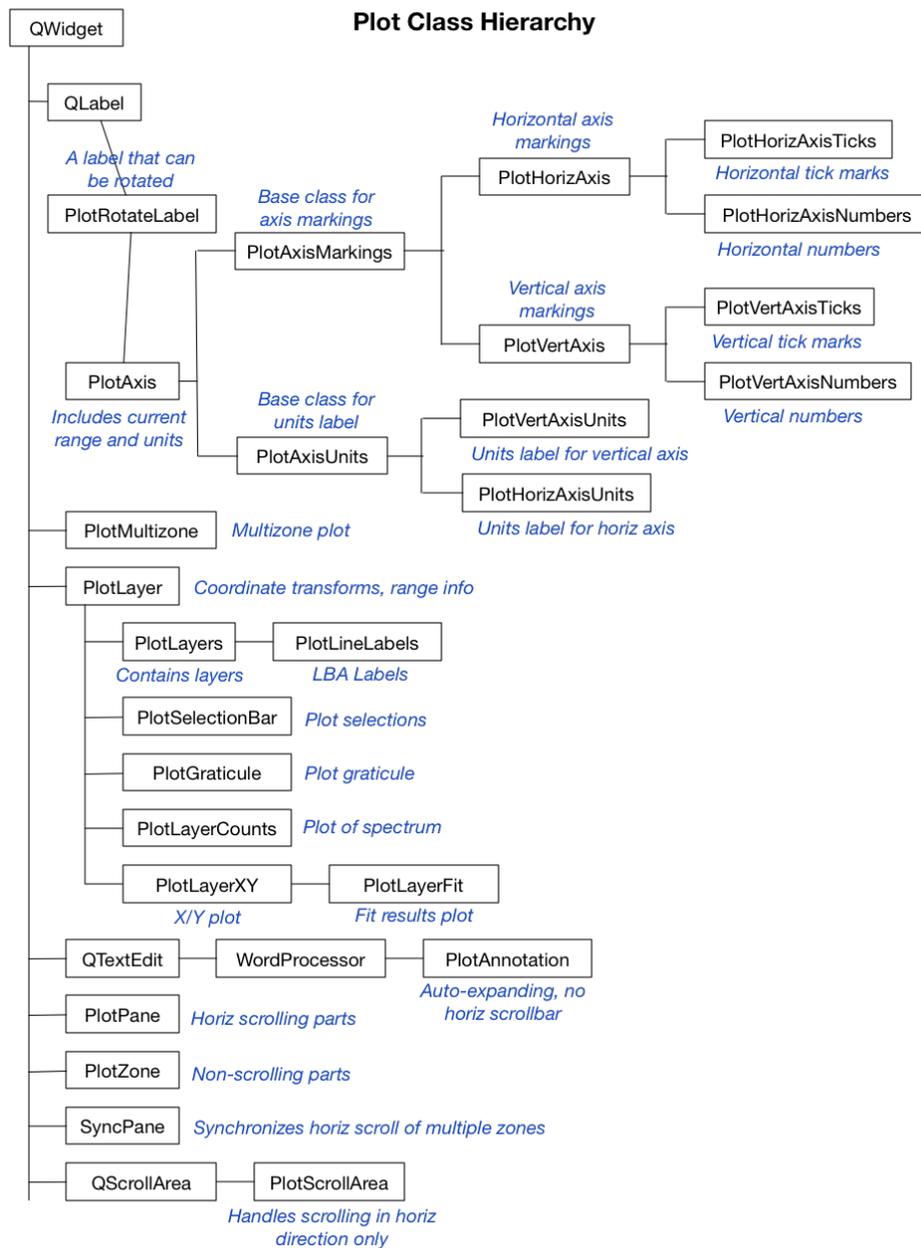
`PlotAxis` is the base class for the `PlotHorizAxisTicks`, `PlotHorizAxisNumbers`, `PlotVertAxisTicks`, and `PlotVertAxisNumbers`. `PlotAxis` contains the logic for placing the marks and labels. `PlotAxisUnits` is the base class for `PlotHorizAxisUnits` and `PlotVertAxisUnits`.

## 7.20.5 Plot Classes

This section describes the plot tool classes.

### 7.20.5.1 Plot Class Hierarchy

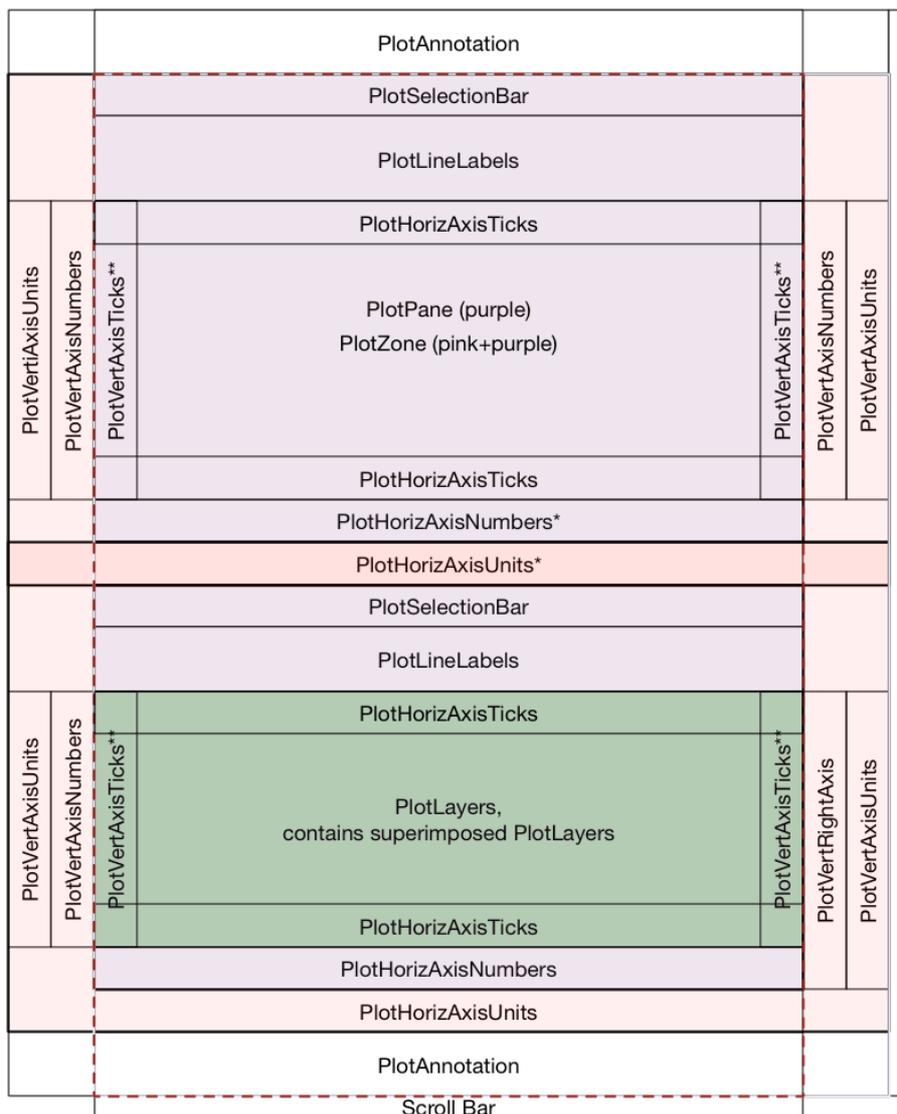
Most of these are self-explanatory. But `PlotLayer` is a base class for classes that produce plots, i.e., need to do coordinate transforms, while `PlotLayers` is a widget that contains multiple overlaid `PlotLayer` instances.



## 7.20.5.2 Plot Tool Ownership Graph



### PlotMultizone with 2 zones



*\*These panes appear only if the units in both zones do not match.*

*\*\*These may appear inside or outside the plot pane, but are always part of the plot zone. There is also a PlotGraticule layer (not shown) that appears in place of ticks in graticule mode.*

*A MultiScrollArea (dotted red) contains all the PlotPanes. There is a separate horizontal scroll bar for each zone.*

## 7.20.6 Spectra, zones, and layers

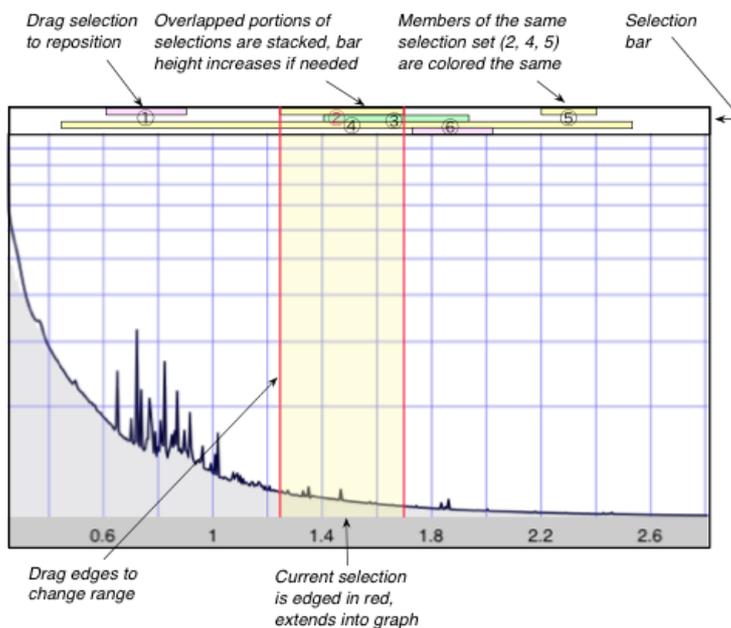
A multizone plot is a plot that contains up to four zones for displaying plots. Each zone can display any number of overlaid plots, providing that they have common axes. A spectrum, represented by the data class `xSpectrum`, is a source file and a set of fits (`xFit`) that represent models, associated XSPEC commands, and fit results.

The plot associated with each spectrum and each fit may be assigned to any of the zones in a multizone plot. The top/up/down/bottom controls allow the position of a plot to be adjusted within a zone. There is no correlation between the order of items in the sidebar and the plot layer order, because the two don't correspond.

A given element (a spectrum or a fit) is uniquely identified by its spectrum number and fit number (both zero-based). Spectrum 1 is identified as (1, -1), where -1 indicates that this is a spectrum, not a fit. Spectrum 2, fit 3 is (2, 3). Spectrum and fit numbers are renumbered as needed when elements are added or removed. Order in the sidebar always corresponds to this order.

## 7.20.7 Selections and the Selection Bar

The selection bar is a bar at the top of a plot zone, used to make and display various named selections. A *selection* (`PSelection`) is just a contiguous range of energies. A *selection set* (`PSelectionSet`) is a group of selections that together specify a possibly discontinuous set of ranges. Selections are identified by set number and selection number—for example, 2.0 is the first selection of selection set 2, and 2.1 is the second. Since a selection is just a range of wavelengths, it is logically independent of the plot for which it is defined. Hence all selections and selection sets are stored at the `PlotZoneInfo` level. (This is no longer true: each selection set is now bound to a single spectrum. But they continue to be stored in the `PlotZoneInfo`.)



Selection sets and selections are numbered from 0 onward. Selection sets appear only in zone 1. Each selection set is uniquely colored. Selection sets are not deleted—unused ones are placed on a free list. This is a simple way to ensure that all are uniquely colored.

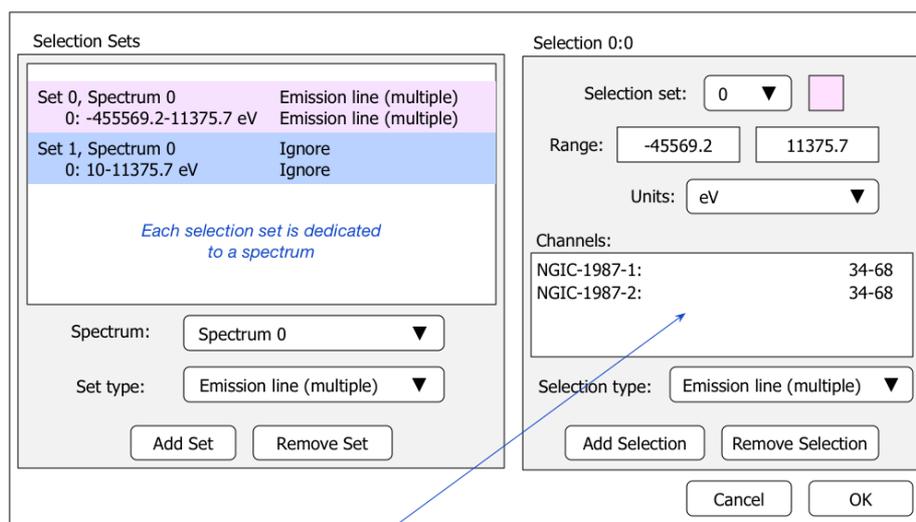
The `PSelectionSet` class is used in two significantly different ways: as a container for the selections in a *set*, and as a container for the selections in a *row*. Selections in a selection set are owned by the set, and deleted when they are no longer needed. But when selections are displayed in the `PlotSelectionBar`, the bars representing each selection are placed in rows. The selections here are shared copies of those in the sets, so they are not owned by the `PSelectionSets` representing each row. `SelectionSets` for sets are constructed with `PSelectionSet::SOOwns`, while row sets are constructed with `PSelectionSet::SOShares`.

Where possible, each `PSelectionSet` occupies its own row in the `PlotSelectionBar`. But if selections in a set overlap, they are shown in different rows to make them clearly discernible. Also, if there are many `PSelectionSets`, rows may be consolidated to save space. This is why color-coding is needed to verify selection membership in a set.

So to summarize, the `PlotZoneInfo` contains a list of all the selection sets in use, and the `PlotSelectionBar` contains a list of selection sets, one per row. Selection sets and selections are not renumbered, but low-numbered ones are recycled first, so user-visible numbers tend to remain low. When a selection is transferred from one selection set to another, it is renumbered to be unique within the target set.

### 7.20.7.1 Update, 2016-02-10

There are some changes implied by the ability to overplot data from multiple instruments. The previous design assumed each selection applied to *all* spectra, or to a user-settable subset, but this has morphed into a design where each selection set is dedicated to a particular spectrum. This solves a number of user interface conundrums and is more intuitive. This is the re-designed dialog:



*If a spectrum applies to a file tag group, the channel range for each group member is listed*

A selection set is applied by default to the “most recent” spectrum, a somewhat ambiguous term whose definition is still evolving. A mistaken association can be overridden by the user. However, since a “spectrum” may be associated with a file group, the selection set’s scope may still include multiple files. Thus there is still a box listing all the member files and their channel ranges.

When a selection is added, it is assigned to the current (last used or selected) selection set. If no set exists, a new one is created, and the new set is assigned to the current (last used or selected) spectrum. It may later be reassigned by the user to another spectrum. Because there is a one-to-one mapping between a selection set and a spectrum, commands that affect spectra (e.g. show/hide, isolate/deisolate, delete) also affect their corresponding selection sets.

### 7.20.7.1.1 Ignore Enable/Disable

Each spectrum has an “Ignore” checkbox. When enabled, ATSAI ignores a subset of the spectrum’s channels in order to improve results for the remaining channels. If the user enables ignore for a spectrum, and an ignore selection set already exists for the spectrum, it will be used on the next refresh. If it does not exist, a new selection set and selection are created from the file metadata. (The user is warned if multiple files are involved and there is an ignore conflict.) If the file(s) lack ignore regions, arbitrary boundaries are chosen initially, and the user is instructed to set them manually. If the user establishes more than one ignore region for the same spectrum, they are combined.

If the user disables ignore for a spectrum, the ignore selection set remains unchanged, and will be applied again if it is later re-enabled. To revert to the ignore region specified by the file’s metadata, the user can delete the ignore selection set, and a new one will be generated.

### 7.20.7.1.2 Storing Selection Sets

A selection set has a `SpectrumID`. Selection sets are stored with the `PlotZoneInfo`. The spectrum IDs are re-evaluated when needed, so a deleted spectrum won’t break anything (the selection set will be discarded if its spectrum has disappeared). Reassigning a selection set to a new spectrum just changes its ID. If the reassigned selection set is an ignore region, and the new spectrum involves a different instrument, the user is warned. Each `xSpectrum` lacks a direct pointer to its selection sets; instead, it called a `PlotZoneInfo` function to look up all the associated selection sets. This allows selection sets to be reassigned or deleted without stale pointers in the `xSpectrum`.

When ATSAI generates commands for fits, it creates a temporary selection set that is a union of all ignore selection sets for the spectrum (just in case there is more than one such set). It then transforms the values to channel numbers for the spectrum. When selection sets are shown in the plot pane or selection bar, they are visible only if their corresponding `xSpectrum` is visible.

If a selection set is marked “unknown,” it still has an associated spectrum. An unknown set performs no function by

default, but it is accessible to python programs for special applications.

## 7.20.7.2 Selections and Channels

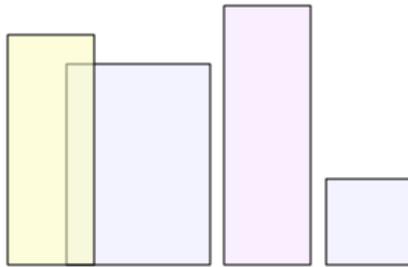
This discusses some rules about how plot selections and channels are related. I need a clear understanding of how channels work in order to allow users to express channel selection. For example, in a counts graph, the selection cursor should snap to the points between channels, and channel selections should fully enclose a set of adjacent channels. Rules are needed to map between pure photon ranges and channel ranges.

I assume that data in the FITS file about the range of each channel:

- Supplies a starting and ending wavelength for each channel, or allows these values to be computed using a (hopefully) sensor-independent rule.
- Supplies a "center" wavelength for each channel, or allows this value to be computed using a (hopefully) sensor-independent rule. For this purpose, "center" means that an arbitrary wavelength that falls within the channel will be rounded to the starting wavelength if it is lower than the center, otherwise to the ending wavelength. More on this shortly.

I further assume that:

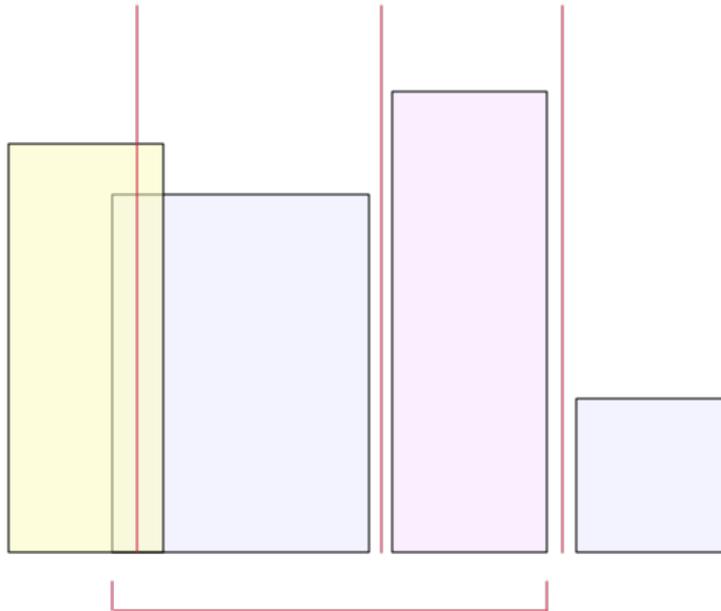
- The width of each channel may be non-uniform.
- Channels are not necessarily exactly adjacent—the start of one might occur a bit before or after the end of another.



When a user clicks in a counts graph, I wish to constrain the selection to a point that lies halfway between the two closest channels. So if the wavelength is before `channel[n]`, I want to compute

```
wavelength = center(channel[n-1].end(), channel[n].start());
```

But if the user selects a range, I need a slightly different rule. First I figure out all the channels that are fully included (by rounding) in the selection. The effective range extends between the beginning of the first channel and the end of the last channel. Below are three cursor positions (vertical lines), and one selection. The cursor positions aren't quite equivalent to the selection ranges.



I lack a precise definition of "center." I can't just use the geometric center, because this will produce different results depending upon the current units. And since the relationship between channels is more logarithmic than linear, one presumes the centering calculation should take this into account.

In order to display the counts as a histogram with channels of uniform width, the horizontal axis is non-uniform, perhaps significantly so. Hence I suppress the horizontal axis and tick marks, but the graph still has horizontal units, so this is displayed. The choice of units is used when providing feedback to the user, so it is meaningful for this to be settable.

### 7.20.7.3 Ignore Regions

The "Ignore bad data" checkbox causes ATSAL to ignore channels within the regions marked as ignore for this spectrum. The checkbox is disabled by default. When enabled, it scans the associated tag's file for a default ignore region, and creates a corresponding ignore selection set. If problems occur, they are reported to the user, and the user may cancel, or ask ATSAL to create an arbitrary region that may then be edited manually. Like other selection sets, users can edit them by dragging their edges or by double clicking the plot selection bar to type in new range(s). The same approach is used to alter the default ignore region.

If the user chooses a different tag, or the tag itself is changed, "Ignore bad data" is unchecked, and any ignore selection sets for the spectrum are discarded. This is based on the assumption that the change will require a change to the ignore region too. Since a file change may happen when the plot tool editor is not even visible, the ignore region is quietly discarded.

Any of the following changes cause all fits associated with a spectrum to be recomputed:

- A change in the file tag, or in the files that comprise the file tag.
- A change in the ignore selection set. (But not in other selection sets associated with this spectrum, since XSPEC does not need to know about those.)
- A change in the ignore checkbox.

The ignore selection set for a spectrum is recomputed when the checkbox is turned on, and no selection exists. If the user deletes an ignore selection for a spectrum, its "Ignore bad data" state reverts to unchecked. Unchecking this option does not have any effect on the ignore region.

Since the only time an ignore region is recomputed is when the user explicitly checks the select box, any associated errors are reported at that time, rather than later, when the refresh cycle begins.

It is possible, but not recommended, for a spectrum to have multiple ignore selection sets. This can come about by accident, for example, when setting the associated spectrum wrong. If multiple sets exist, their union is used to generate the ignore command.

Saving a spectrum also saves the ignore on/off setting, and the selections.

### **7.20.7.3.1 Include Regions Design Refinement (2017-03-18)**

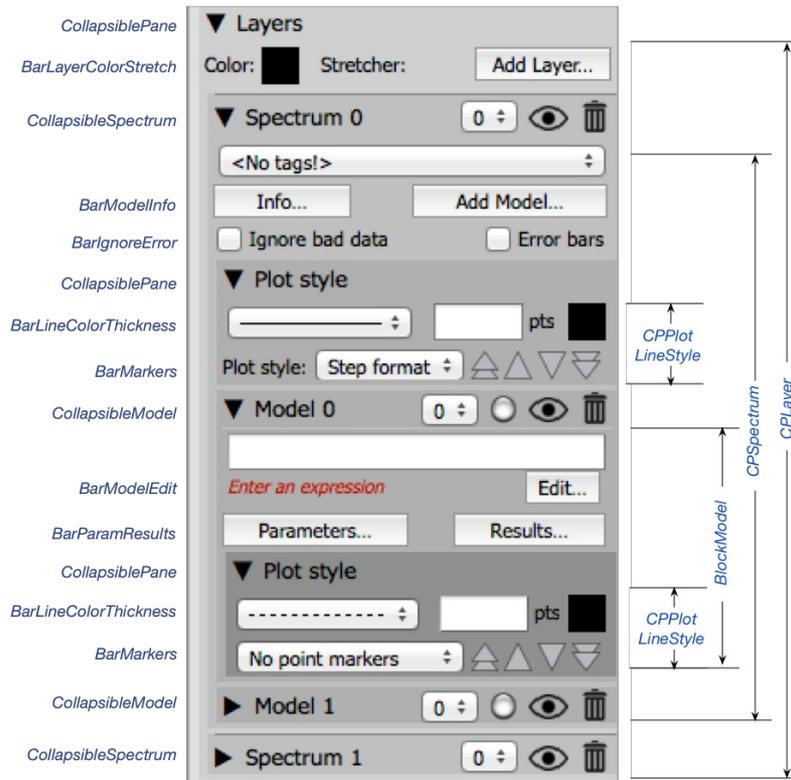
1. Use *include regions*, not ignore regions, on the grounds that it is common to exclude some data at both ends of a spectrum, and that is most easily accomplished with a single region; and for consistency with other types of ATSSAL selections, which are inclusive, not exclusive.
2. Although a given tag may have an associated default include region, it may not, and the user cannot create a manual include region or adjust the default until a convolved spectrum is available, since we need to know the range of the horizontal axis. For this reason, “Use include region” is disabled until convolved data are available, and this option is never checked automatically.
3. If the user selects the “Use include region” checkbox, then:
  - a. If a FITS region exists, its selection set appears, where it may be used as is or further adjusted.
  - b. If there is no region, a prompt explains that there is no region and offers to create a template the user can adjust, or to cancel.
  - c. If there was an error loading the data, it is presented, and ATSSAL again offers to create a template.
  - d. If there is already a previously user-defined include region, it is used. This occurs if a user previously enabled this option, then turned it off and later turned it on again.
4. The include region is associated with the spectrum, not the tag. Hence, re-selecting a previously selected tag does not restore its include region. On the upside, though, two spectra can share the same tag but employ unique include regions.
5. If a user edits a tag’s files, or selects a different tag for a spectrum, the state always reverts to no include region. If checked again, the user starts with the default.
6. In the selection set details dialog, if the include region set is selected, add a button that restores the predefined region.

This is a selection set issue, not an include region issue, but since each tag has its own include region, and multiple such regions would clutter the plot, only the current spectrum’s include region (and any other selection sets) are shown. The user designates a spectrum as current by clicking its label in the sidebar, or by clicking one of its model labels.

## **7.20.8 Plot Sidebar**

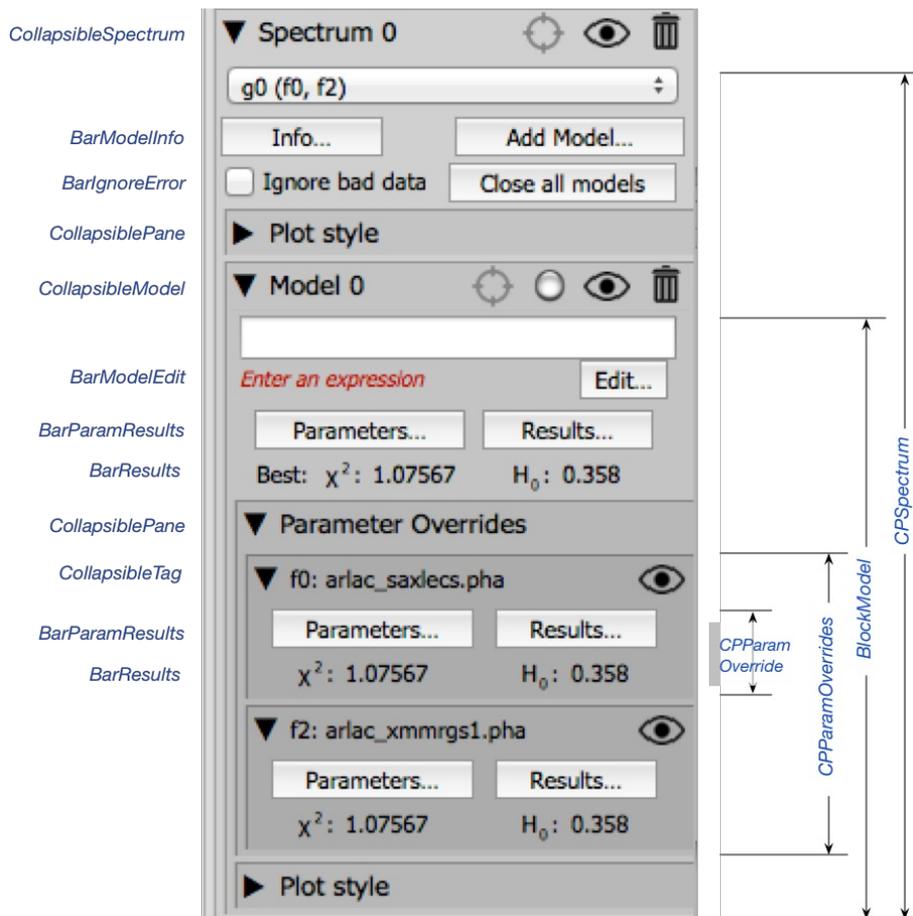
### **7.20.8.1 Layers Section**

## Layers Section of Plot Sidebar



### 7.20.8.2 Layers Section with Tag Groups

When a tag group is selected as an input spectrum, the Parameters... button sets defaults for all the member tags, and the Results... button presents a results summary for each tag. A new Parameter Overrides panel appears as shown below, listing each group member. Here, the Parameters... button sets tag-specific overrides, and Results... presents the results for that file's fits.



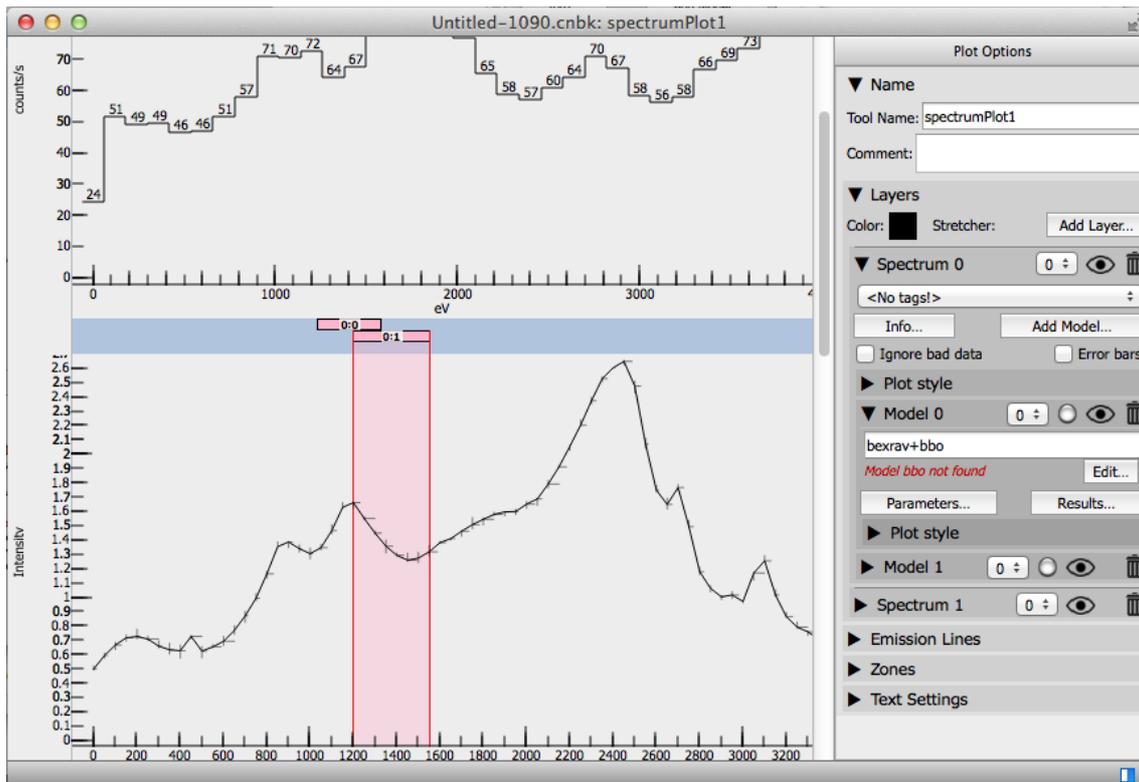
## 7.20.9 Plot Tool Use Cases

I envision three use cases for the plot tool. Calling these “modes” is a bit of a misnomer, because it suggests that the user has to make some significant shift in the use of the tool(s) for each case, and perhaps even that the modes are sequential. The interface is as modeless as possible—users never need to think consciously about modes.

- Analytical mode. In this mode, the user is intent on comparing results of various fits until a suitable match is found.
- Annotation mode. Here, the scientist wants to clean up the notebook a bit, perhaps deleting lines of inquiry that did not pan out, and annotating the line of reasoning that led to viable results. The target of these notes could be the scientist, collaborators, colleagues, or publication.
- Publication mode. If the work is intended for a formal audience, more work may be needed to present unambiguous, attractive data visualizations. ATLAS doesn't attempt to solve all the problems associated with publication, but it tries to simplify this process.

### 7.20.9.1 Analytical Mode

In analytical mode, the user is focused solely on the models, results, and comparisons among fits. ATLAS tries to provide a reasonably uncluttered interface for this, and to avoid excessive trips into subwindows for configuration. This example shows a single raw spectrum and a fit. (Data are randomly generated!)

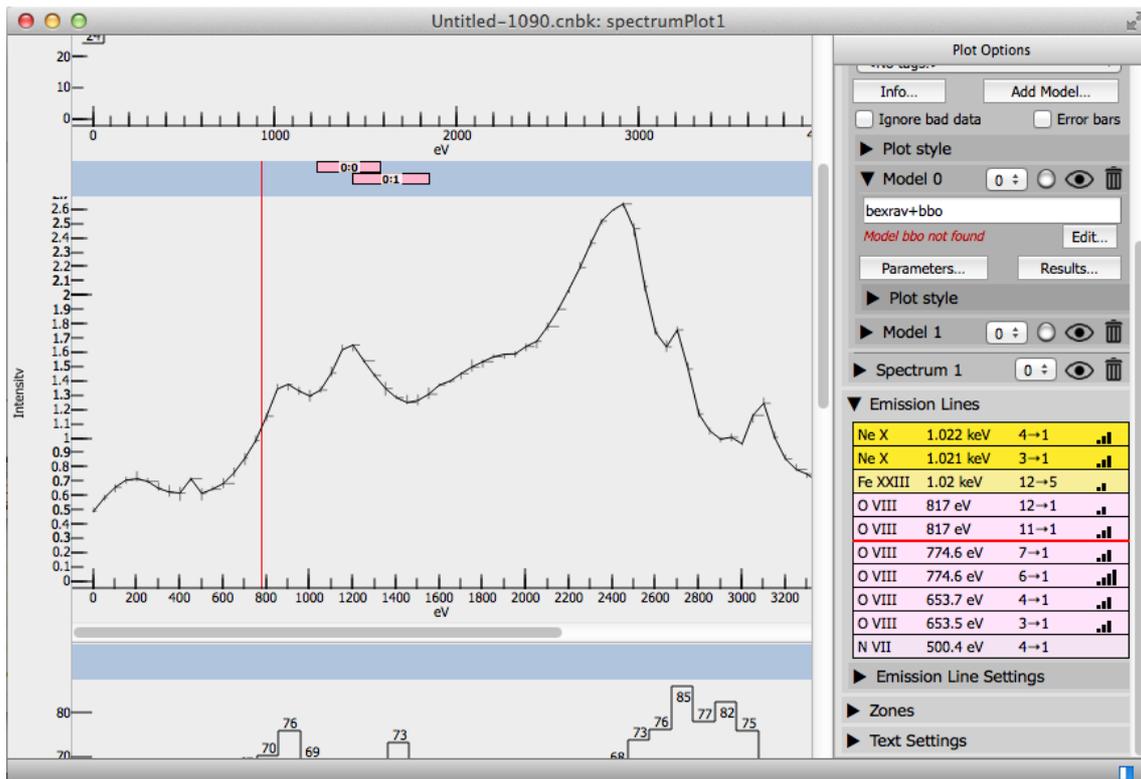


Only the Layers panel is open in the sidebar. Considerable effort has gone into keeping the size of the subpanels for each spectrum and each model as minimal in size as possible, so that even if many models are in use, excessive scrolling won't be needed. The Spectrum panel specifies a file, a few other options, and an option to add a new model. The Model panel allows model editing without the need for the model editor. Only parameter editing demands a visit to another window.

As with Model 1 and Spectrum 1 in this example, even when the panel is closed, it is still possible to reassign the plot to a different zone, view compute progress on a fit (the "LED" in the model panel), hide or show the layer, or delete a model or spectrum that is no longer needed. Even if a dozen models are applied to the same spectrum, the sidebar should remain reasonably uncluttered.

Also in this mode, users can create and edit selections, by clicking on one x-axis endpoint and shift-clicking on the other. Selections appear in the blue selections bar. The current selection extends through the graph as a translucent overlay. Double clicking the bar in the selections bar allows other properties to be set. Bars are members of selection sets; all members of the same set share the same coloring. Selection bars have various uses; for example, clicking "Ignore bad data" produces a selection set of bars that include any data to be ignored in the spectrum.

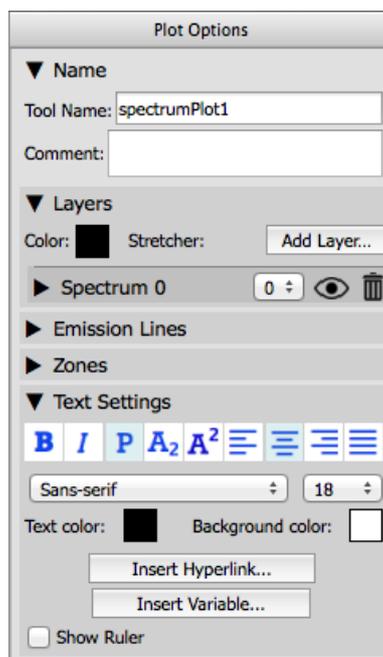
If the emission lines panel is opened, the selection cursor shows nearby emission lines.



This provides a simple way to provide nearby emission line data without performing line-based analysis or modifying the graph.

## 7.20.9.2 Annotation Mode

During or after analyses, users can annotate their thought process. This is done using the text tool, or a couple of features of the plot tool. First, each plot has a header and footer region that is a reasonably flexible mini-word processor. The text panel is used for these.

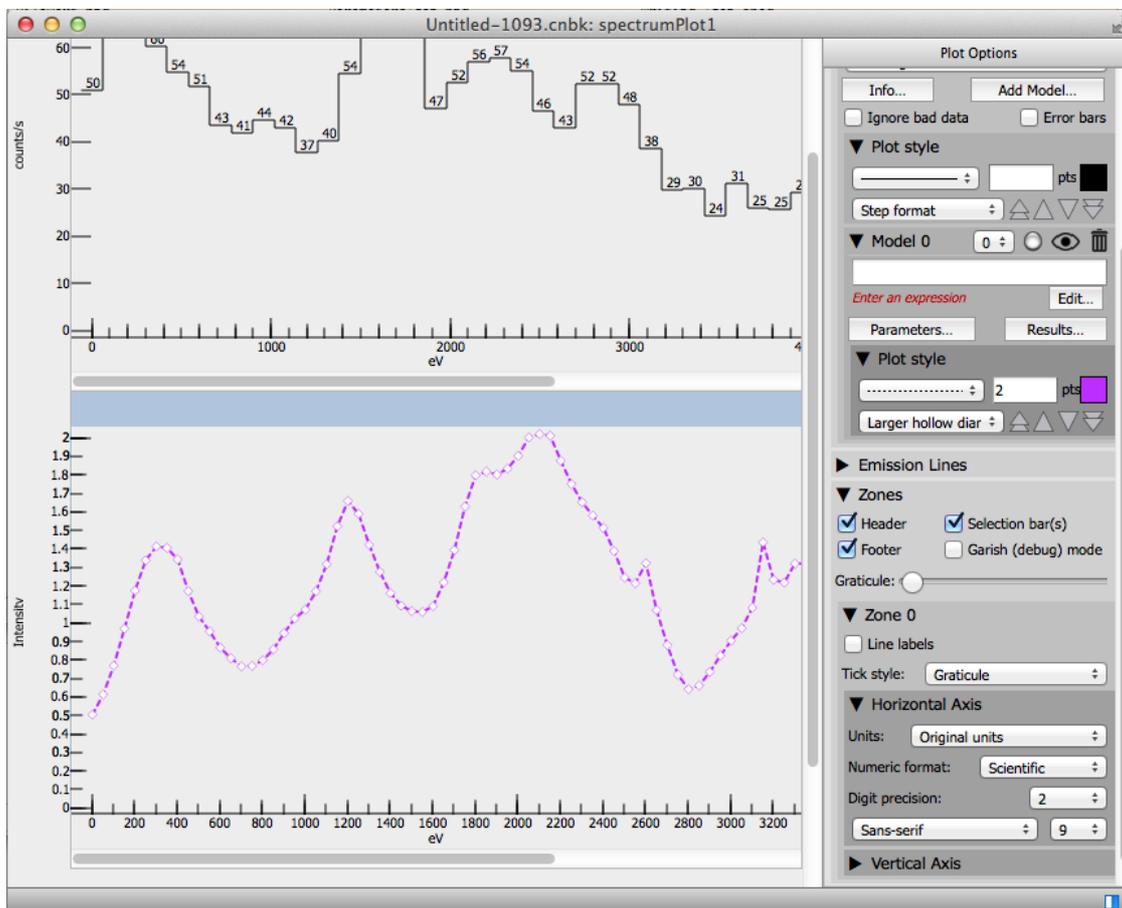


Although this is not yet implemented, users can also enable the line labels plot section. This is a region above the plot and below the plot selection bar, used to display line labels computed by line based analysis. This region can also contain user annotations of ranges or specific x-axis values. The annotations and the labels are shown at an angle like

that in the iPad AtomDB application.

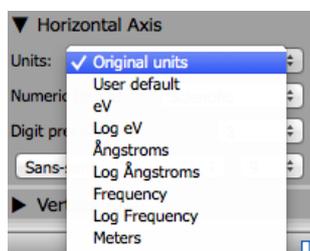
### 7.20.9.3 Publication Mode

Publication mode can loosely be described as a point where the user starts tinkering with plot display options in order to disambiguate among multiple stacked plots, or simply to provide a more pleasing and informative graph. Users can open plot style options at the spectrum or model fit levels. There are also options that apply to plot zones.



Plot style options are intended for the case when several overlaid plots are shown in the same zone. Users can make these distinguishable using color, line style, line thickness, and plot point style. A plot layer may be moved to the top or bottom, or up/down a level.

While plot style options apply to a single plot, the Zones panel controls options for each plot zone. A few such options apply to the zone as a whole; others apply to a particular axis.



### 7.20.9.4 Meeting, Oct. 14, 2015

#### 7.20.9.4.1 More Thoughts on Large-scale Computing

Once it becomes easy to set up multiple models and run multiple observations against all of them simultaneously, the

local computer becomes the limiting factor. Distributing the compute load across multiple server computers solves this problem, but the solution is expensive to implement. We have to solve accounting, load-leveling, security, and version skew issues, as well as port this code to various operating systems.

There is a simpler and very contemporary solution: the "cloud." A scalable set of homogeneous devices, all running the same OS, OS level, and XSPEC server version; with someone else (e.g. Amazon) solving most of the accounting, administrative, security, and load-leveling problems. Since these are (typically) Linux boxes, XSPEC already runs on them, so the XSPEC development process is largely unchanged. An ATLAS user simply obtains a single account with a compute resource quota. This is much cheaper in terms of ATLAS development dollars, and easier for users than maintaining accounts to multiple servers.

Thus any end user can run ATLAS for free on their own computer, but if they have funding, they can run ten or a hundred XSPECs at once in the cloud.

Caveat: observation files are large and moving them around to cloud computers could be a serious bottleneck. XSPEC server would have to implement some kind of aging algorithm for deleting observation files that are no longer needed, but this is fairly easily done.

## 7.20.9.4.2 Spectrum Plot Tool Walk-Through

This is a demo of sorts, intended to explain the design goals behind various plot tool features.

### 7.20.9.4.2.1 Analysis Mode

The main design goal here is to allow the user to focus on the analysis process, by making all decisions involving layout or formatting, as automatically as possible.

1. The initial state of the sidebar is simple, basically only one choice: add a layer. (It is Add Layer, not Add Spectrum, in anticipation of other layer types. More on this later.)
2. If we were working with real data, a "flatline" would appear as a stand-in for the plot, until the user selects a file to plot. We skip this step here because we are working with fake data.
3. The Info... button isn't implemented yet, but will display file metadata.
4. Asymmetric zoom. Adjust sidebar, window width. Adjust plot height for vertical. Note that the same proportion of plot remains visible as window height/width changes. Max zoom 10x window width. Zoom presently done with option click; zoom out is shift-option click. Later I will add icons.
5. Numeric values appear if there is room.
6. Add a model. Again, a "flatline" appears immediately in a new plot zone, as feedback for the pending model results. Once again the data here are synthetic, so after a simulated delay, the plot appears. The default display shows error bars.
7. By default, the plot is scaled to fit slightly within the available space. The idea of the scaling is to make it easier to see maxima and minima, as well as the max and min labels on the plot axes.
8. The little red ball is an experiment. It is easy to distinguish between a plot for which there is no data yet (a flatline) and a real plot. But how to indicate that the displayed plot is stale, because parameters have changed and/or a new fit is being computed? I could dim the stale plot, but this is ambiguous, as will be seen later. So the moving red dot is meant to be a low-key indicator that this plot is currently being (or in need of) recomputation. Note that the activity light in the model pane is different: it shows when XSPEC is actually computing.
9. Clicking establishes a cursor. Dragging modifies the cursor position. If the emission lines display is turned on, it tracks the cursor.
10. The optional emission lines controls need to have reasonable defaults. I chose a temp high enough to have lines dispersed over a wide range of energies, and a fairly aggressive minimum emissivity. Suggestions for refinement are welcome.
11. The red shift has a strong detent at zero. I was wondering about changing the default to match a given observation file, but since the scope of these controls may include multiple files, that isn't workable.
12. Shift-clicking establishes a selection. (Create several.) By default, all selections are added to the same selection group, so they appear in the same color. A maximum of one selection at a time can be the current selection, meaning only that it extends over the plot as a translucent overlay. Clicking outside of selections in the plot selection bar deselects all of them.
13. Selections and ranges appear in the plot selections bar, numbered by selection set:selection.
14. Selection edges may be adjusted by dragging.

15. (Zero-based segue.)
16. A good time for the selections-*vs.*-counts discussion. Selection snapping in count selections. Mapping between count-based selections and "continuous" selections. Determining centers.

From Randall:

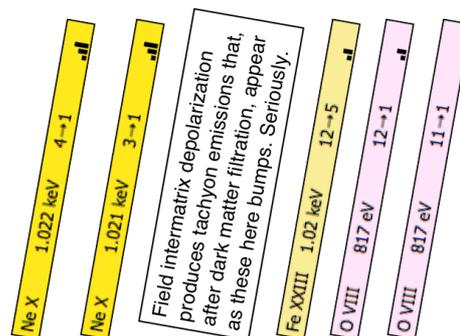
- a. Each channel has a start and end energy (ENERG\_LO, ENERG\_HI in the RMF or RSP file).
- b. Channels are by definition adjacent. The channel number can start at 0 or 1 (1 is preferred but not required) and goes up in unit steps. Some channels may be marked 'bad' in the PHA file, and they may be marked as part of a 'group', but they always have an independent existence in the file.  
In some sense, channels are the rawest form of data; after an event (aka 'X-ray') passes through the detector's analog-to-digital converter, the channel number is just a value, expressed as a number in 1..N. Making a bunch of event channels into a histogram gives us the rawest form of a spectrum.  
...and most importantly...
- c. People very rarely work in channel space. The only real reason to do so is to see about fixing some detector calibration problem. We could likely leave channels out entirely in v1 and, while we'd get complaints about it, they wouldn't even rise to the top of the pile.

Anyway, as a result of all this x-ray histogram in channel space must be linear. There could be a \*second\* axis that shows the energy or wavelength — and that will potentially be wildly non-linear — but in channel space it's linear. But it's just occurred to me that we may be speaking about two things. I've been thinking of my point (3) above, but I suspect you're thinking of the more general problem of plotting data in (say) wavelength space. In this case, we still have binned data (which is ultimately derived from the aforementioned channels) but the histogram data has associated wavelength information which could be spaced in almost any old fashion. In general the relationship between channel and energy (or wavelength) is linear or nearly so, but often people want to plot in log axes, so this relationship isn't all that helpful.

Thinking on this now, I know that channels cannot overlap — by definition your case with the yellow and blue channels overlapping is forbidden. I \*believe\* (but am not certain, will have to check with Keith) that gaps are equally forbidden. However, 'ignored' channels are allowed, which has more or less the same impact since they should not be included in the plot.

May have to talk about this on the phone to make sure we're not talking past each other. I think I figured it out mid-email, but am not certain. I left the earlier stuff in here just for documentation...

17. Selections have multiple purposes. For example, Ignore bad data...
18. Double-clicking a selection opens the selection dialog. This allows selections and selection sets to be created, deleted, and edited.
19. Selection types: LBA hints, ignores, and unknowns. Currently defaults to multiple emission lines. Details will change when we design the LBA support.
20. Each zone has an independent horizontal scroller, capable of synchronized scrolling. Several design iterations, include MouseTrap and HoleyScroller classes, were abandoned in favor of the current design.
21. Line labels and user annotations. Along with plot height, this is the only other user-resizable region. User annotations created here are anchored like line labels to a particular energy (or energy range?). User can choose between LBA labels, user annotations, or both as shown.



22. Insert a second model. It appears in a different color because we are in autocolor mode, so ATSAAL assigns colors that override any user settings. The model name in the sidebar is colored to match.
23. We can have up to four zones, but ATSAAL chose to stack the two fits in the same pane. Why? First, because stacking permits close comparison. Second, because it is scalable: you can create ten fits without exhausting the four-zone limit. Third, it is a simple general rule: whenever you add a plot, it appears by default in the first zone with matching axes.
24. But this makes disambiguation a significant issue, in fact a central design goal. Lets create a few more models. I will close most of the model panels, as a user might do as they experiment with various fits.

25. Note that although unproductive fits are easily discarded, they are also easily suppressed, so keeping them around to guide subsequent model choices is painless. Also note that since results are (*will be*) saved with the notebook, they persist indefinitely without the need for recomputation.
26. Only the top fit displays error bars, to reduce clutter.
27. Note hovering over model names.
28. Open the plot style panel and select a thick or dotted line.
29. Show plot stacking order. (Note: maybe only need top and bottom...)
30. Hide some fits.
31. Hide the spectrum. All plots disappear too.
32. Other point marker styles.
33. The stretcher. The idea here is to take advantage of some of our motion-sensitive neural circuitry—the moving lines appear more distinct, as well as moving apart. The control is spring-loaded so the traces settle back into their proper locations on release.
34. Zone reassignment. In effect, the "extra" zones are temporary places to isolate and examine plots of interest.
35. Zones can be hidden. Right now, hiding a zone hides the zone in both the notebook and the plot tool editor, but it might be better to allow the zones to be controlled independently, because only plot zone 1 is usually desired in the notebook.
36. Synchronized scrolling revisited. Not yet fully implemented. Any subset of plot zones, provided they have matching horizontal axes, may be synched together, panning and zooming together. This is implemented internally via the `MultiScrollArea` class, but not yet exposed to the user interface.

#### 7.20.9.4.2.2 Annotation Mode

1. Enter a multi-line annotation, show text panel.
2. Show ruler, talk about physical page size.
3. Note that the header expands automatically: lots of work to keep user from having to think about layout. Only two resizable entities: plots and line labels.
4. Line label user annotations already mentioned.
5. Another "Add Layer..." option is an annotation pane that does not scroll, which contains arrows, boxes with text, etc. This allows a user to construct detailed annotations for a plot, assuming a fixed pan and zoom. However, this is a very low priority, since it could be done in an external vector drawing editor like Affinity Designer or Adobe Illustrator.
6. Compare notebook representation to tool editor representation. Notebook does not show animation features or selections, and its view properties (plot size, scrolling, etc.) are independent. This means the notebook can be fine-tuned for summary views somewhat independent of the plot tool editor.

#### 7.20.9.4.2.3 Publication Mode

1. We already talked about using disambiguation tools. They can also be used to fine tune plots for presentation.
2. Shut off plot selection bars, which collapse to separator lines.
3. Shut off autocolor.
4. Open zone settings. Note difference between settings for all zones vs. settings for a particular zone.
5. Open zone 1 settings. Again, settings for both axes vs. settings for a specific axis.
6. Show units. Mention that units will default to user prefs.
7. Show emission line display response to units changes.
8. Show graticule, mention antialiasing and thin lines.
9. Thinking about option-clicks on these popups to apply the change to all zones at once.
10. Export panel.
11. Note that SVG is displayed by modern browsers, so these are superior to JPEGs or PNGs.
12. I dropped PDF support, at least temporarily, because Qt simply exports a pixmap wrapped up as a PDF, not a vector image. I will eventually figure out how to convert the SVG to a "real" PDF.

#### 7.20.9.4.2.4 Questions

1. Should error bars be shown for counts data?

2. How should selections in counts zones be handled?
3. Other ideas for indicating stale plots?
4. Eventually I will need to permit model panel cut/paste and duplication of models.
5. Default photon units?

### 7.20.9.4.3 ... One more thing ...

ATSAL is out of the incubator! I transferred it to a Retina laptop running a much newer Mac OS, and encountered no problems at all! A large dark cloud has lifted... This means that I could place a copy of ATSAL on your computer.

Caveats:

- I didn't release the baby from the incubator, I sent the incubator (>1GB) home too. Still lots of packaging issues to work out.
- High rez screen issues that work well on the Mac may fail miserably on Linux, which is still struggling to support high rez displays with non-integral scale factors.

## 7.20.10 Plot Types, Specialized

### 7.20.10.1 Types and Priorities

This is a very incomplete table that tries to assess the relative importance of each of the specialized plot types.

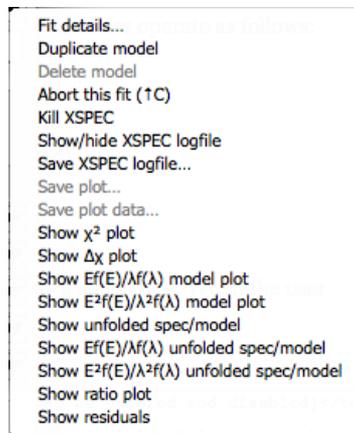
Plot	Vert units	Horiz units	Plot Style	Importance	Sep. zone	Compute intensive	
background	Emissivity	Photon	x/y	med	no	no	
chain			non-xy	low			
chisq				med	y		+
contour			contour	med-lo			
counts				hi			
data				hi			
delchi				med	y		+
dem				low			
emodel	photon/cm2		xy	med-hi	y		
eemodel	photon/cm2		xy	med-hi	y		
efficiency	cm2		xy	low			
eufspec	photon			med	y		+
eeufspec				med	y		+
foldmodel				low			
goodness	weird			low			
icounts				low			
insensitivity				low			
lcounts			xy	hi			
ldata			xy	hi			
margin				low			
model				med-hi			
ratio				med			+
residuals				med			
sensitivity				low			
sum				low			
ufspec	photon			med-lo	y		+

## 7.20.10.2 Fit Sub-plots

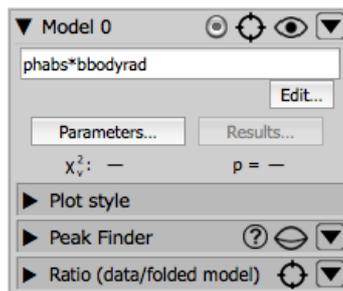
I define a *fit sub-plot* here as a plot associated with a particular fit, but not the fit itself. Because a single plot tool can display multiple spectra, and each spectrum may contain multiple fits, and each fit has ten or so associated plots, the variety of plot types far exceeds the number of plot zones available. My latest thinking on this is that the four zones operate as follows:

Zone	Purpose
0	Raw counts data for all spectra (individual spectra can be enabled and disabled)
1	Fit results for all spectra and all their models (again, individual layers may be adjusted)
2	Temporary slot for a fit sub-plot.
3	Temporary slot for a fit sub-plot.

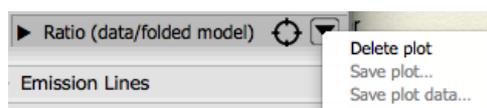
Thus zones 0 and 1 contain any number of overlaid plots, while zones 2 and 3 always display a single plot. When the user selects a sub-plot, it replaces the oldest plot currently displayed in zone 2 or 3. In the plot sidebar, where screen real estate is at a premium, the new sub-plot is selected from the fit details menu:



(I haven't verified that all these plot types are fit sub-plots, as opposed to other kinds of plots, but that is my working assumption at the moment.) When a sub-plot is selected, it is granted a small panel for the plot style settings. In this example, I added a ratio plot.

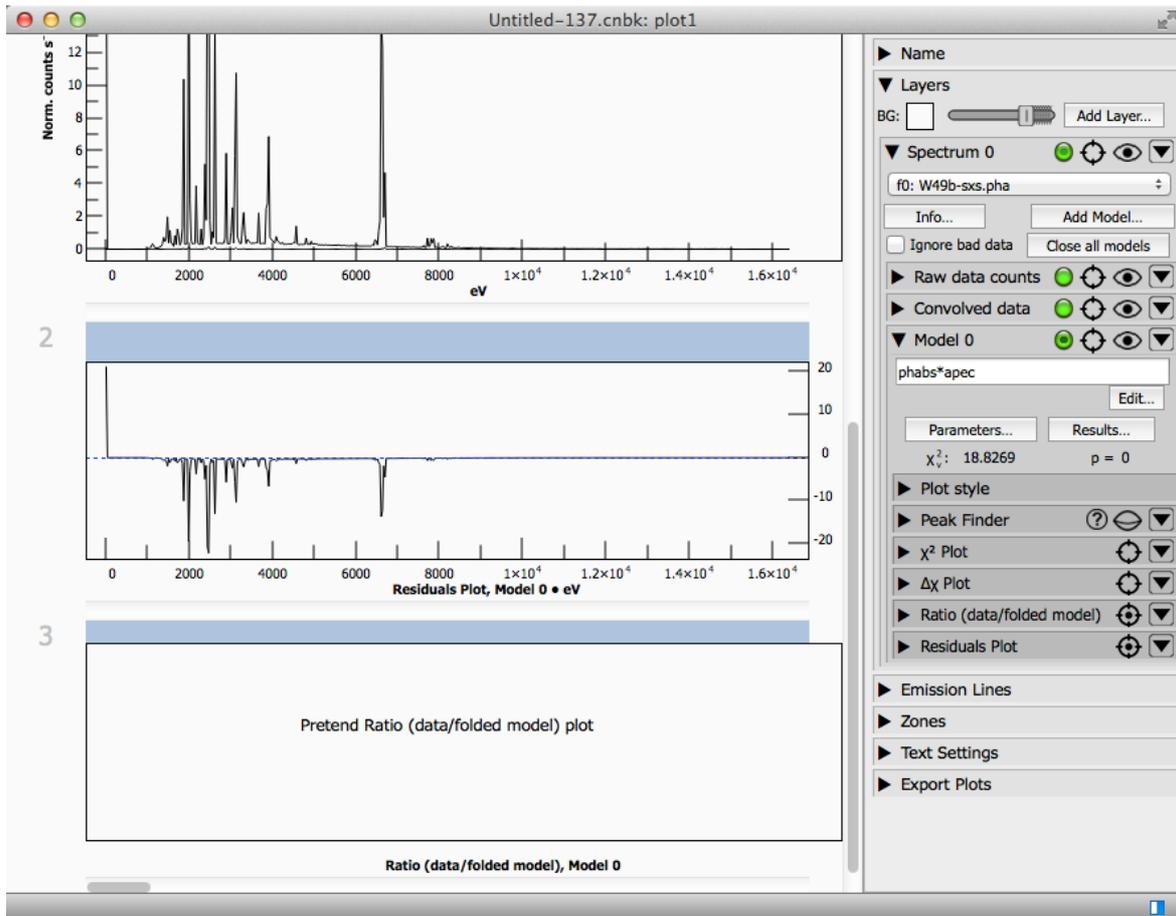


As the user selects multiple such graphs, they first fill the two temporary zones, then begin replacing the oldest plots. The panels representing the plots remain visible as the plots are replaced, making it easy to reselect them for display by clicking the target icon. Sub-plots have their own option menus:



Plots may be deleted via this menu. Deleting them this way also removes the panel. Regardless of whether a given plot is removed by replacement or by deleting the plot, its plot style settings are preserved if the same plot is later displayed. So in this way, users can select subplots freely, for either a given model or multiple models, without thinking about where they should be displayed, where they are in a layer hierarchy, or whether they are visible or not.

In this example, the user inserted four sub-plots, two of which are currently selected:



In this example, the four selected sub-plots have a settings panel in the sidebar, with the “target” icon enabled for the most recent two, which appear in zones 2 and 3. Only the residuals plot is implemented—the others appear like the ratio plot above. Each model has its own list of sub-plots, but all models share access to the pair of plot zones.

There are a lot of assumptions here, perhaps not all correct. First, I assume that interest in sub-plots is relatively ephemeral, so that the automatic replacement is intuitive. Second, I assume that it isn’t necessary to overlay plots of the same plot type for these sub-plots. If you want to compare residuals for two models, you select the residuals plot for both models and they appear in adjacent zones. I also assume that two sub-plots max is an acceptable constraint. (The zone limit is of course arbitrary, but there are good reasons to leave it fixed at four.)

## 7.20.11 Channels, Zones, and Layers

The plot tool lies at the heart of ATLAS, and its detailed implementation is in constant flux. This section summarizes how a trivial-seeming issue led to significant design changes.

### 7.20.11.1 Channels

Photon energy vs. amplitude plots are ATLAS’s most common plot type. My first order assumption was that a channel mode display, being a plot with a unique set of units (channels vs. amplitude), had to appear in its own zone. So if, for example, you started with a channel mode display of convolved data and then decided upon an energy mode display, you would need to create a second plot in a separate zone.

Intuitively, though, it seems clear that a channel mode display is simply another way of viewing an energy mode display, just as an energy display might be viewed in either Ångstroms or log(keV). To accomplish this I created a new physical type, a `QuChanPhoton`. This is exactly equivalent to a `QuPhoton`, except that each `QuChanPhoton` is initialized with a channel number as well as an energy, and can therefore be converted to/from a channel number. This works cleanly with the existing scheme. For example, the cursor can be dragged along a channel axis, producing both the current keV and the current channel during the drag. Similarly, the emission lines list continues to work during a cursor drag in

channel mode. This scheme also allows plot selections, such as ignore regions, to work properly regardless of display mode. You can make a selection in channel mode, switch to Ångstroms, and preserve the selection.

But it breaks down completely if two channel plots from *different* instruments are overplotted. Such an overplot is (I assume anyway) perfectly natural in energy mode, since there is now a compatible  $x$  axis for all the plots.

So I have been thinking about how best to handle this. I could:

1. Refuse to allow plots from different instruments to be overplotted in the same zone.
2. Move any incompatible instrument plots to another zone, upon entry into channel display mode.
3. Display only the current plot and all other plots from the same instrument in channel mode, temporarily hiding any plot layers from other instruments.

I speculate that (1) doesn't work, because overplotting data from multiple instruments in energy mode is a fairly common requirement. (2) is awkward from both a human factors point of view and an implementation point of view. So (3) is the front runner.

Here is a possible use case. Consider two spectra from different instruments, both of the same object, the first with one fit, the second with two. In energy mode, this means there are five plots in one zone, the two convolved plots for the spectra, and the three fits. Now the user shifts to channel mode. At this point, the current spectrum's plots remain visible, while the others are suppressed. (More precisely, the current instrument's plots remain visible, while the others are suppressed.) If the user shifts back to energy mode, the suppressed plots reappear. (Maybe the first time this happens the user is warned, since it is confusing.)

The same issue occurs with raw counts data. Overplotting results from more than one instrument produces meaningless results. So once again, I would use the rule that in such a case, only the current instrument's plot layers would be displayed.

### 7.20.11.2 Plot Layers and Zones

This ties into another issue, one that seems trivial but has emerged as a significant human factors issue. Where do new plot layers go by default, and what should the user be able to do later? Here is progression of choices I have made so far:

1. Each new layer gets a separate zone. When the zones are exhausted, new layers are added to the first zone with compatible axes. The user can then reassign any plot layer from its current zone to any other zone that is empty, or has compatible axes. This isn't really workable, because the pool of available zones is quickly exhausted, leaving nowhere to put plots with new axes. It isn't very intuitive either, since plot comparison is so central to the analysis process.
2. Make the number of zones unlimited. This breaks down quickly as well: too much scrolling, related data too far apart, etc.
3. Add an "isolate" command to each plot layer, which moves a selected plot layer to a free zone. Again, this consumes the list of free zones quite quickly.
4. Redefine the "isolate" command to mean "hide all other plot layers in this zone." A second click on isolate restores them all, or users can click the isolate button on other plot layers to make only their layer(s) visible. My plan is to adopt (4), but like all the solutions, this has pros and cons. A possible downside is that *all* the plots with a particular set of axes are stacked in the same zone, like it or not. A pro is that the four zone limit is now capacious enough to handle most plotting requirements.

The precise operation of the show/hide and isolate/deisolate icons is described in the following section, **Isolationism**.

Summarizing,

- Zone 0 is a counts zone.
- Zone 1 is a `QuChanPhoton` vs. `QuNormalizedCounts` zone, where all convolved data and fits appear.
- Zones 2 and 3 are used for special plots, like residuals.
- The four-zone limit is still enforced.

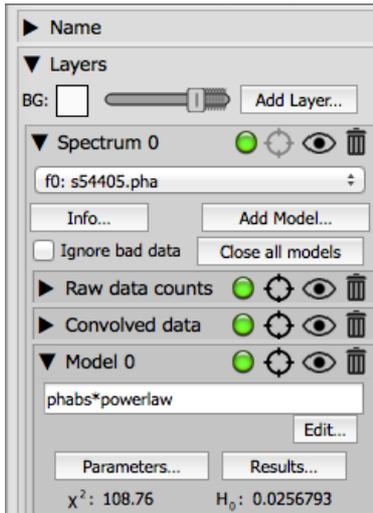
### 7.20.11.3 Isolationism

It isn't unusual for a single zone in a plot tool to contain a dozen or more overlaid plots, so ATLAS needs controls for viewing subsets of the plots. Each plottable object in the hierarchy has a show/hide button () and an "isolate" button (). Although not shown, there is also a show/hide button at the layer level. This section explains how they will

interact. The controls are not exactly complementary, but in combination, they handle most of the use cases that are likely to develop. “Isolate” is a state that is applied to a single object (and its subordinates) at a time, implicitly “deisolating” others. Isolation temporarily overrides the previously set show/hide settings, restoring them upon deisolation. Show/hide commands issued when an isolate state is in effect override the isolate state for a particular object and its subordinates. The cases are described in more detail below.

### 7.20.11.3.1 Show/hide and Isolate/deisolate Controls

The eye icon isn't shown beside the Layers level yet, but it will be present.



#### 7.20.11.3.1.1 Layer Controls

Show enables the visibility attribute for all spectra and plots in the plot tool, and disables any isolate state. This turns on all plot layers in all zones.

Hide disables the visibility attribute for all spectra and plots in the plot tool. This turns off all layers in all zones. This might be done prior to adding a new fit or plot, or before turning on a single plot of interest.

#### 7.20.11.3.1.2 Spectrum Controls

Show enables visibility of all plots associated with this spectrum, even if they are presently suppressed as a result of a prior isolate command.

Hide disables visibility of all plots associated with this spectrum.

Isolate constrains the display to plots associated with the current spectrum, suppressing others, in all zones. It does not enable visibility of the current spectrum's plots though, so if any of these were previously hidden, they must be shown again. If a different spectrum or plot was in the isolate state, it reverts to the not-isolated state.

Deisolate restores the prior state of visibility of plots for spectra other than the current one, so if other plots were previously visible, they become visible again.

#### 7.20.11.3.1.3 Models

Show makes all fits and any associated plots, such as residuals, visible, even if a prior isolate command suppressed them.

Hide hides the fits and associated plots.

Isolate suppresses all other models, spectra, and associated plots, preserving plots only for this model.

Deisolate restores the prior state of visibility of the other plots.

#### 7.20.11.3.1.4 Plottables

Show/hide flips visibility of the selected plot, even if an isolate command is in effect.

Isolate turns off everything in the current zone except for the selected plot. (Most other isolate commands effect multiple zones, but this is restricted to this plot's zone.)

Deisolate restores the previous state of visibility for all plots.

### 7.20.11.3.1.5 Channel Display

In channel display mode, only plots associated with the current instrument are visible, since each instrument's channels map differently to photon energies. The current instrument is set each time a new fit is created, or a plot is shown or isolated. This can lead to counterintuitive results. For example, if isolate is clicked to show a set of fits for one spectrum, and the user clicks the show button for a plot for a different instrument, all plots associated with the prior instrument are suppressed, and those for the current instrument revert to their previous show/hide state. And the selected plot is made visible. Hence turning on a single plot may implicitly shut off several others.

I am not sure how to make this less confusing. Maybe each time selecting a plot hides others for a different instrument, a dialog could explain what is happening, with an option to disable the dialog.

### 7.20.11.3.2 Implementation

Three booleans are associated with each object in the hierarchy. `mHiddenByIsolate` and `mHiddenByShowHide` are set by their respective buttons. `mActuallyHidden` is set based on the state changes happening in the other two. Show/hide sets both `mHiddenByShowHide` and `mActuallyHidden` in unison. Isolate sets `mHiddenByIsolate` and `mActuallyHidden`, without disturbing the state of `mHiddenByShowHide`. Deisolate clears `mHiddenByIsolate` and copies `mHiddenByShowHide` into `mActuallyHidden`, restoring prior visibility.

A fourth bool, `mHiddenByChannelMode`, is set when a user clicks show/hide or isolate/deisolate, to shut off displays for other instruments. The attribute is used to set `mActuallyHidden` only if the zone is in channel display mode.

## 7.21 Qt Trivia

This contains some notes on Qt peculiarities.

### 7.21.1 Copying QObjects

Thou shalt not copy `QObject`s! Yeah I knew that, but I got blindsided. I started with a simple data (vs. widget) class, `ZoneSet`:

```
class ZoneSet
{
public:
    ZoneSet();
    ...
};
```

Later I wanted it to receive signals, so I made it a `QObject`:

```
class ZoneSet : public QObject
{
    Q_OBJECT
public:
    ZoneSet();
    ...
public slots:
    void doSomething();
};
```

This generated an obscure-seeming error, that `QObject` has a private copy constructor. It didn't make sense because I wasn't defining a copy constructor in the derived class, nor was I using a copy constructor. So with a little looking around, I found out how to keep C++ from generating default copy constructors:

```
class ZoneSet : public QObject
{
    Q_OBJECT
public:
```

```

ZoneSet();

ZoneSet(const ZoneSet&) = delete;
ZoneSet& operator=(const ZoneSet&) = delete;
...
};

```

That's when the compiler isolated my implicit use of a copy constructor elsewhere in the code:

```

QList<ZoneSet> list;
list.append(ZoneSet());

```

The fix was simply to use pointers:

```

QList<ZoneSet*> list;
list.append(new ZoneSet());

```

## 7.21.2 QSharedPointer

Qt offers about half a dozen mechanisms for shared pointers. `QSharedPointer` is one such, handling the common case of sharing a pointer using a reference count stored externally to the `QSharedPointer`. These are conceptually simple, but I found the docs a bit lacking on how to use them. I'll use pointers to strings for this example. Do you declare a `QSharedPointer<QString>` or a `QSharedPointer<QString*>`? Turns out it is the former, not the latter.

### 7.21.2.1 Initialization

Okay, so can you do this?

```

QSharedPointer<QString> foo;
foo = new QString("Hello, World");

```

No. You have to do this:

```

QSharedPointer<QString> foo(new QString("Hello, World"));

```

### 7.21.2.2 Assignment

What about this?

```

QString string = *foo;

```

No, that doesn't work, which makes sense. You need to do this:

```

QString *string = foo.data();

```

Assigning the shared pointer a new value is also awkward:

```

foo.reset(new QString("A new value"));

```

### 7.21.2.3 Propagation

So suppose you have this:

```
QSharedPointer<QString> p1(new QString("Thing1"));
QSharedPointer<QString> p2(p1);
QDebug() << *p1.data() << *p2.data();
```

As expected, p1 and p2 both point to the same string, so we get:

```
Thing1 Thing1
```

Now suppose I say:

```
p1 = QSharedPointer<QString>(new QString("Thing2"));
QDebug() << *p1.data() << *p2.data();
```

I would expect this to produce:

```
Thing2 Thing1
```

and it does, but I was a little worried it would produce:

```
Thing2 Thing2
```

## 7.21.2.4 Efficiency

There are a few places in the code where a chain of classes pass around shared pointers to a particular object:

```
class A
{
public:
    QSharedPointer<BigItem> bigItem() { return(b->bigItem()); }
};

class B
{
public:
    QSharedPointer<BigItem> bigItem() { return(c->bigItem()); }
};

class C
{
public:
    QSharedPointer<BigItem> bigItem() { return(d->bigItem()); }
};

...
```

That's a lot of shared pointers getting constructed and destroyed just to provide a chain of access. The overhead could be bypassed by using a *pointer* to a `QSharedPointer` instead, but this is syntactically and even conceptually a bit awkward. So the question is, how efficient is this chain of references? I don't know the answer to this yet.

## 7.21.3 Installing Event Filters

Event filters get first dibs at events before they are dispatched to the widget tree. I adapted this example to trap key press/release cycles of modifier keys, because I needed to trap these transitions without acquiring the keyboard focus in order to enable specific cursors for modifier combinations.

```
class Widget : public QWidget
{
    Q_OBJECT
public:
```

```

Widget(QWidget* parent = 0) : QWidget(parent)
{
    QVBoxLayout* l = new QVBoxLayout();
    QLineEdit* a = new QLineEdit();
    a->installEventFilter(this);
    l->addWidget(a);
    QLineEdit* b = new QLineEdit();
    b->installEventFilter(this);
    l->addWidget(b);
    QLineEdit* c = new QLineEdit();
    c->installEventFilter(this);
    l->addWidget(c);
    statusL = new QLabel();
    l->addWidget(statusL);
    setLayout(l);
}
protected slots:
void resetStatus()
{
    statusL->setText("");
}
protected:
bool eventFilter(QObject *obj, QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast(event);
        if (keyEvent->key() == Qt::Key_Shift)
        {
            statusL->setText("Shift pressed");
        }
        if (keyEvent->key() == Qt::Key_Control)
        {
            statusL->setText("Ctrl pressed");
        }
        QTimer::singleShot(500, this, SLOT(resetStatus()));
    }
    // standard event processing
    return QObject::eventFilter(obj, event);
}
private:
    QLabel* statusL;
};

```

## 7.21.4 Qt and Threads ... and Sockets

### 7.21.4.1 QThreads and Clients

ClientXSpec may look a little confusing to the uninitiated. It looks that way to me, and I've done quite a lot of looking at it. Partly because it has been diced into little pieces and rearranged half a dozen times, and partly because I'm still not sure I understand how it works. Or even how it is *supposed* to work.

There are several Qt classes that deal with threads. I'll focus on `QThread`. `QThread` is not a thread, it is a thread **wrapper**. Thus a subclass of `QThread` runs in the main thread, not the new thread. Older Qt docs recommended subclassing `QThread` to create a user-defined thread. Maya Posch makes a good argument to the contrary in [How To Really, Truly Use QThreads; The Full Explanation](#).

Since then, Qt docs for versions newer than the one I am presently using have been updated to describe both Maya's approach and subclassing. Here, I use Maya's approach:

```

class ClientXSpec : public QObject
{
    Q_OBJECT
public:
    ClientXSpec(...);

private:
    QThread mThread;
};

```

instead of subclassing `QThread`:

```
class ClientXSpec : public QThread
{
    Q_OBJECT
public:
    ClientXSpec(...);
};
```

(In the real code, `ClientXSpec` is a subclass of `ClientBase`, in turn a subclass of `QObject`: for this purpose, the same thing.) The important thing here is that `ClientXSpec` is a wrapper for the `QThread`, not the `QThread` itself. And in both cases, the clients are `QObject`s in order to be able to use signals and slots for orderly exchange of information between threads.

So what parts of `ClientXSpec` actually run in the newly created thread? **The constructor doesn't**, since it is constructed by the main thread. You have to be careful, for example, not to instantiate a `QTcpSocket` in the constructor, since it needs to be owned by a thread object. Any functions called by the constructor are similarly running in the main thread. Any slots, though, are running in the client thread; and any signals are emitted from the client thread. You can connect a signal to a main thread target and it will be placed on the main thread's event queue and delivered in an orderly manner to the main thread. (I think it is fair to say that this use of signals and slots for orderly communications is really the most important thing `QThreads` do to ease thread usability.)

And there's one more function that runs in the client thread, here called `doWork()`. This is the main entry point for everything the client thread does. It is designated as such by this call in the controller constructor (controllers are described below):

```
connect(this, SIGNAL(operate(const QString&)), mClient, SLOT(doWork()));
```

## 7.21.4.2 Controllers

A Qt article I read early on advocated creating a controller associated with each client. For example, `ControllerXSpec` (a subclass of `ControllerBase`) acts as the main thread object that controls the client thread. Broadly speaking, this is intuitive, because it implies that everything in the client class runs in the client thread, and everything in the controller class is in the main thread. The controller is the place that manages orderly transfer of information between the threads.

If the data exchange between threads is fairly complicated, the controller idea is almost certainly justified, but in these relatively simple classes, it is a little ungainly, just another intermediary in the chain. Nevertheless, I have stuck with this idea in all the ATSA threads in anticipation of later added complexity.

So the controller creates a client, and the client has a `QThread`. The application talks to the controller and the client talks to, in this case, the `XSpec` process.

## 7.21.4.3 Sockets

The `ClientXSpec`'s job is to establish a socket connection to a previously created `XSpec` process, then sit in a loop transferring commands and data between the processes. So in pseudoCode:

```
void doWork()
{
    mSocket->connectToHost(...);
    if (mSocket->waitForConnected(timeout))
    {
        while (forever)
        {
            processCommandsAndReplies(...);
        }
    }
}
```

Simple enough, right? Well this doesn't work quite as one might hope. First, `waitForConnected()` "doesn't always work under Windows," according to the Qt docs, and ATXSAL will eventually be ported to Windows. Second, `connectToHost()` fails immediately if XSpec isn't initialized yet, with a "connection refused" error. So we have to either insert a long enough time delay to make sure XSpec is running, or we have to repeat the `connectToHost()` call until the connection succeeds or a timeout expires.

We cannot do this synchronously, we have to rely on signals to convey the state of the connection. For inscrutable reasons, some of those signals arrive while waiting for the `connectToHost()` function to return, while others are placed on the event loop queue for the client, only to be delivered if we return from `doWork()`. (In particular, I found that the `connected` signal is not delivered, so the pending connect always timed out.) So we need an asynchronous design with a state table.

The `ConnectState` state table replicates Qt's own similar table pretty much completely, but I used my own to better understand what was happening. In the current design, state changes are delivered to slots that simply transfer the state to the corresponding `ConnectState`, reissuing the `connectToHost()` as needed until the connect succeeds or we time out. In between each state transition during the connect process, we return from `doWork()` so the event loop can crank. Once the connection is established, we don't need the client event loop any more, and can sit in a while loop in `doWork()`.

At least I think that is true.

So the present implementation is a bit overcomplicated, but it appears to be reliable and does not impose extra delays at startup time.

At the time of this writing, the design does not yet address orderly shutdown.

## 7.22 Version Numbering

Version numbers are assigned in `versions.h`, and the `Version` class represents a version number. The original purpose of this trivial class is to allow such numbers to be compared, when implementing backward compatibility. For example, all ATXSAL-generated XML files contain the version of ATXSAL that created the file.

The `setversion` utility was created when it became necessary to keep a new version number in sync in several places, notably documentation files. As the headers for these files proliferated, this approach became awkward.

Later the `Version` class was extended to contain other information about a build, such as build date, git SHA code, and git commit date. This information ensures that a given executable's source code can be retrieved. Since the build date is included now, comparisons of `Version` instances are now based on build date rather than version number, since the latter is finer grained and automatically updated.

Example:

```
void MyClass::readBlob(QXmlStreamReader& reader, const Version& version)
{
    if (version.buildDate() < Version("2016-12-08T23:11:32"))
    {
        // Read older format
    }
    else
    {
        // Read newer format
    }
}
```

Version numbers are still assigned in an include file, but versions for both ATXSAL and Nexus are assigned in the same file (`versions.h`) so they are available to Nexus when generating documents. And since version numbers are no longer used for version comparison, only build dates are relevant for backward compatibility. The `setversion` utility has been removed.

## 7.23 Resource Files

ATXSAL now employs two different mechanisms for handling resource files: Qt's resource compiler, and a separate

resource manager.

### 7.23.1 Qt Resource Compiler

Qt includes a resource compiler that converts a list of files into part of the executable image by representing them as byte arrays. Each ATSAAL application has a `.qrc` file that lists the resources. Originally resources for each ATSAAL application were stored in separate resource file directories, but a limitation in the resource compiler design prevents this from working when resources are shared among several applications. Resource file locations are specified relative to the location of the `.qrc` file, and may be in the same directory or a subdirectory, but *not in a parent directory*. Hence resource files for all ATSAAL apps are now stored in `atsaal/resources`, allowing any needed sharing.

### 7.23.2 ATSAAL Resource Manager

When Nexus generates a web site or a PDF book, it needs to deploy a series of supporting files: artwork, CSS and Javascript files, and header files. Originally these files were included in Nexus as Qt-style resources and deployed from there. However, this means that a re-compilation is necessary to include these files and re-deploy them, inconvenient when making tweaks to the files. So I added a web resources mechanism. The file `WebResources.xml` is loaded into Nexus as a Qt resource, and it contains a list of other resource files for deployment. This allows changes to be deployed without a recompilation.

Nexus formats its content for several different uses:

- for the Nexus help utility itself
- for a free-standing web site's top level pages, which fill the window width
- for a free-standing web site's help sub-trees, where window width is split between a table of contents and the content
- for a book generated in raster format
- for a book generated in vector format

Somewhat surprisingly, all these cases need adjustments to headers and styles in order to scale content for readability. In addition, MathJax must be invoked only when pages contain math content, since the code is loaded from a remote server and very slow otherwise. For this reason, a bewildering array of HTML headers and style sheets are currently in use.

## 8 Documentation Tools

This section describes Nexus, a documentation manager that is part of ATXSAL, and Doxygen, used to generate documentation from source code.

### 8.1 Nexus Document Tree Manager



Nexus was developed as part of ATXSAL. It is a simple help system implemented as a table of contents pane wrapped around a [Qt WebKit](#) browser window. It assists in the creation of ATXSAL docs, both for developers and end users, as well as in managing links to the many external resources upon which ATXSAL depends.

Nexus supports several types of web pages:

- External web pages (those beginning with "http:") are added as read-only entries.
- Local web pages that are external to the Nexus-managed doc tree (those beginning with "file:") are also added as read-only entries.
- Pages that are part of a Nexus-managed doc tree (created and named with relative pathnames). These pages may be edited by Nexus.

Nexus opens only one topic at a time, but Display in Browser opens the topic in a separate web browser. This only works if the topic is an external web page at this point.

A Nexus document (".nexus") contains only a directory name. It serves to bring up Nexus operating on a particular doc tree. When creating a new Nexus document, this directory can be supplied as absolute or relative. If relative, it is relative to the Nexus document location.

There are two document trees that are part of ATXSAL. `ATXSAL Dev Help` contains this developer help. The `help` directory contains the end user doc tree. These are accessed by `ATXSAL Dev Help.nexus` and `help.nexus`, respectively. Both of these documents specify the doc tree location as a relative path so that a git checkout into any directory will work properly.

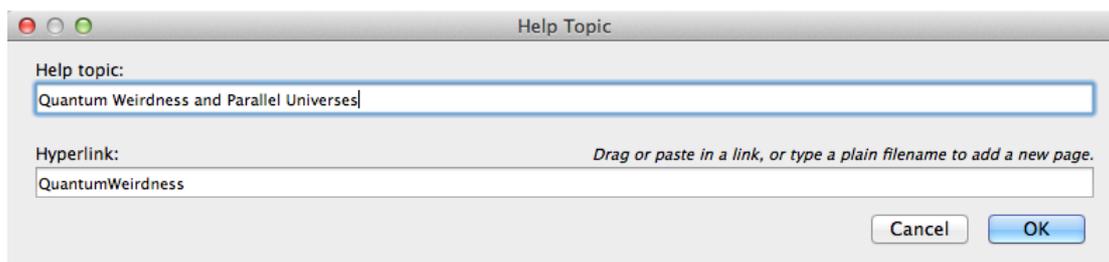
Nexus shows only the currently selected page. If you want to open additional pages, use Display in Browser. This only works for external pages though, because internal pages are constructed on the fly.

#### 8.1.1 Editing Help Files

To add topics to the help tree, use the insert before, insert after, or insert under buttons:



In this example, a local filename has been specified. Nexus appends ".html" if necessary and stores the new file in the document tree:



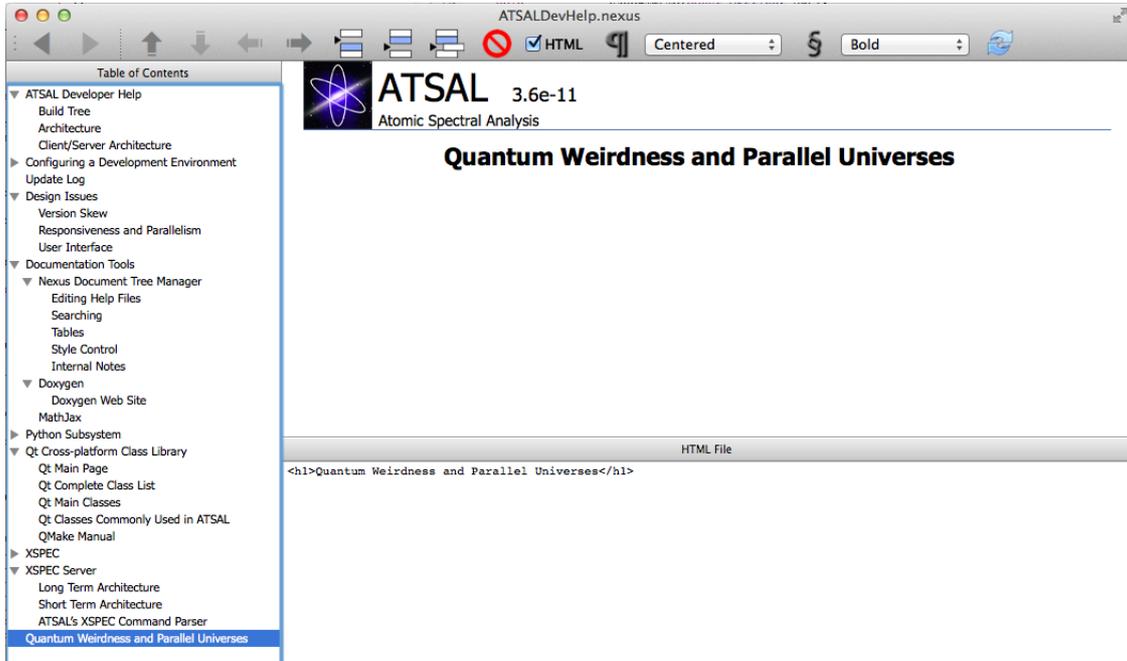
You can also add topics by dragging a URL from a browser window directly into the table of contents pane.

Move existing topic up or down, or promote/demote them, using these buttons:



### 8.1.1.1 The HTML Pane

To edit the text, click the HTML checkbox if needed to display the text in HTML format. (Although you can view the HTML for external web pages, you cannot edit it.)



The HTML entered in the HTML pane is wrapped by necessary headers and footers in order to provide a consistent style among all help pages and make predefined CSS styles available. The actual document looks something like this:

```
<!-- ----- Main Frame ----- -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/xhtml; charset=UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=9">
<meta name="generator" content="Doxygen 1.8.8">
<title>Quantum Weirdness and Parallel Universes</title>
<link href="tabs.css" rel="stylesheet" type="text/css">
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="dynsections.js"></script>
<link href="doxygen.css" rel="stylesheet" type="text/css">
<link href="stylesheet.css" rel="stylesheet" type="text/css">
</head>
<body>
<div id="top"><!-- do not remove this div, it is closed by doxygen! -->
<div id="titlearea">
<table cellpadding="0" cellspacing="0">
<tbody>
<tr style="height: 56px;">
<td id="projectlogo"></td>
<td style="padding-left: 0.5em;">
<div id="projectname">ATSDAL
<span id="projectnumber">3.6e-11</span>
</div>
<div id="projectbrief">Atomic Spectral Analysis</div>
</td>
</tr>
</tbody>
</table>
</div>
```

```
<!-- end header part -->
```

```
<h1>Quantum Weirdness and Parallel Universes</h1>
```

```
</body>  
</html>
```



Refreshes the HTML pane into the web view. A few commonly used style options are available via the paragraph and span menus:



The paragraph and span buttons apply the most recent paragraph and span styles, respectively. Or use Reapply Last Style (Cmd/ctrl-Y) to reapply the most recent style.

To add images, drag the image file into the HTML pane. This generates the HTML to display the image and copies the image into the help tree. You can also drag URLs to create links.

## 8.1.2 Searching

Nexus find and replace deals with a few interesting problems.

### 8.1.2.1 Simultaneous Web and HTML Views

First, the user works with both a rendered web view and the raw HTML simultaneously, so any search support must operate on both views. The current version of Qt does not offer symmetric search support for both views. It will, for example, highlight all instances of a string in a web view, but it won't scroll to the next occurrence of a string and highlight that. Nor does it support whole word searches or regular expression searches. The current version of `QTextEdit` supports `QRegExp` searches but not the newer `QRegularExpression` searches.

So truly symmetric search support is out. Instead, I dispense with regular expressions entirely—they can be added later when I upgrade the Qt level. When a search is performed, I highlight all occurrences of the search string in the web view, including substrings in whole word mode, without scrolling, because that's all I can do. (Not really true: I could parse and edit the HTML, doing more correct highlighting manually, but there is still no mechanism to perform scrolling to the next match. Since a newer `QWebView` debuts in later releases of Qt, it isn't worth solving this problem now.)

### 8.1.2.2 Remote Searching

Nexus displays local HTML pages as well as remote links. Global searches must first download all the remote pages, a potentially slow process. Hence Nexus employs a cacheing scheme, downloading pages on demand and storing them in a local cache (currently in `~/ATFiles/Caches/FindFiles/`). The cache may be purged via `File > Purge cache`, and each time the user views a remote page, its cache entry is updated. Since many downloaded pages are named `index.html`, I use a simple hash to avoid name collisions. `https://x.y.z/index.html`, for example, becomes something like `index-344.html`, where 344 is a simple checksum of the other characters in the URL string. This isn't a very reliable hash of course, suitable only for small cache sizes, but it could be extended later. See the utility function `RemoteURLToCachePath`.

Remote searching introduced some other hassles too. For example, a URL may not specify a filename at all, or a name but no extension. Then there is the guess-the-codec problem.

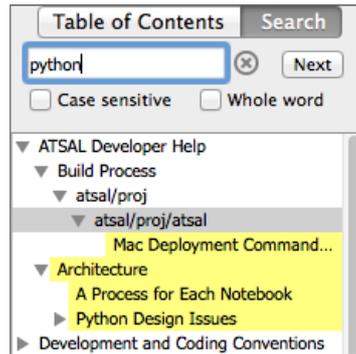
Initially I considered repeating the global search on each new keystroke of the search term, but this requires either reading in every file again on each keystroke, or on buffering all of them in memory. I didn't seriously consider buffering in memory, because in the presence of ample memory, the OS will do pretty much the same thing. But in any case, I initiate the search only when the user hits return.

### 8.1.2.3 View vs. Edit Mode

In edit mode, the user sees both the rendered view and the raw HTML, so it is reasonable to search on terms that are part of the HTML markup, not just the text content of the document. In view mode, though, only the rendered content is visible, so it doesn't make sense to search the markup. Hence in view mode I call a function, `readStrippingHTML`, which accepts a string containing the HTML document and strips out the HTML. It also converts runs of whitespace to a single space. The returned string is suitable for a content-only search. Hence a search for "href" matches most web pages in edit mode, but may match nothing in view mode.

### 8.1.2.4 Local vs. Global Searches

Local searches, restricted to a single page, are implemented with `⌘F`. This displays a dialog that prompts for a search and optional replace string. Global searches (`⇧⌘F`) retask the table of contents pane, showing only the subset of items that match the search criteria:



Global search mode is also selectable via the Search tab. Global searches list all pages containing the search term, as well as their parent topics, which are opened. Since parent topics may not contain the search term, only those that do are highlighted in yellow.

When the user exits global search mode and returns to the table of contents, the original open/close state of topics is resumed, with one exception: the currently displayed page topic remains visible.

Find next (`⌘G`) behaves differently depending upon whether the preceding search was local or global. In both cases, it moves to the next match in the current topic, scrolling the HTML page to bring the match into view. In local mode, it wraps to the start of the page if there are no more topics. In global mode, it moves to the next matching topic or wraps to the first matching topic.

If a set of global matches are displayed, and the user performs a new local search, the new search term replaces both the global search term and the local one. This means that the table of contents pane is still displaying matches for the old term. This seems a bit counterintuitive, but most integrated development environments behave this way.

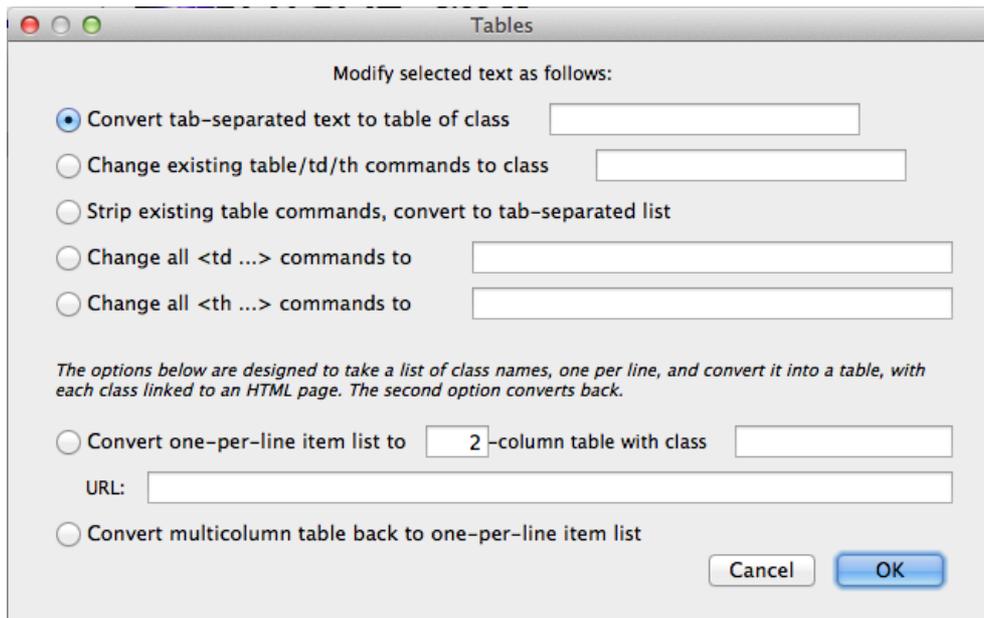
### 8.1.2.5 Replacing

When the user displays a remote URL, its HTML is shown for symmetry with local pages, but the HTML is locked against edits. For this reason, replacing is disabled for this case, and this means that global replacement also doesn't make sense. (It could be applied only to local pages, but this doesn't seem very intuitive.)

### 8.1.3 Tables

Edit > Tables... simplifies some common tasks when formatting HTML tables. Combine these with manual fine-tuning to get the desired effect. **The parsing for these options is not very sophisticated and could have unintended effects on your text, so it is a good idea to operate on a copy.**

All Tables... commands operate on the current selection in the HTML pane.



### 8.1.3.1 Convert tab-separated text

Convert tab-separated text to table of class classname converts tabbed text to a table.

```
Element Atomic Number Atomic Weight
Hydrogen 1 1.008
Helium 2 4.0026
Lithium 3 6.94
Beryllium 4 9.0122
```

Becomes:

Element	Atomic Number	Atomic Weight
Hydrogen	1	1.008
Helium	2	4.0026
Lithium	3	6.94
Beryllium	4	9.0122

If you supplied a class name, the class is applied to the <table ...> element, as well as the <td ...> elements. Note that the first row is considered part of the table rather than a header, but this is easily changed in the generated HTML, by manually editing the tds to ths:

<i>Element</i>	<i>Atomic Number</i>	<i>Atomic Weight</i>
Hydrogen	1	1.008
Helium	2	4.0026
Lithium	3	6.94
Beryllium	4	9.0122

### 8.1.3.2 Change existing table/td/th commands

This option changes the class name associated with each table, td, and th command to the specified name. In this example, the class name has been set to nothing, producing the default formatting.

```
Element Atomic Number Atomic Weight
Hydrogen 1 1.008
```

Helium	2	4.0026
Lithium	3	6.94
Beryllium	4	9.0122

### 8.1.3.3 Strip existing table commands

This option removes all table formatting commands from the selection, leaving the raw text. Table elements in each row are separated by tabs. Applying this to the table above converts it back to its original form.

```
Element Atomic Number Atomic Weight
Hydrogen 1 1.008
Helium 2 4.0026
Lithium 3 6.94
Beryllium 4 9.0122
```

### 8.1.3.4 Change all <td ...> (or <th ...>) commands

These options replace all the arguments associated with each td or th element with the supplied argument list. If, for example, you want to right align all the elements in the table, using class "a", supply the string:

Change all <td ...> commands to

This adjusts everything. At present there is no way to restrict the change to a column.

<i>Element</i>	<i>Atomic Number</i>	<i>Atomic Weight</i>
Hydrogen	1	1.008
Helium	2	4.0026
Lithium	3	6.94
Beryllium	4	9.0122

In this case, the entire td command is supplied, not just an argument. The commands have no effect if there are no such elements present in the selection.

### 8.1.3.5 Convert one-per-line item list

This is a special-purpose item designed to make it easier to create densely packed lists of class names for quick lookups. It takes a list of class names, one per line, and presents them in a table of a specified number of columns.

```
QAbstractItemDelegate
QAbstractItemModel
QAbstractItemView
QAccessible
QAction
QApplication
QButtonGroup
QByteArray
QCache
QCalendarWidget
QCheckBox
QClipboard
QColor
QColorDialog
QColumnView
```

Produces:

[QAbstractItemDelegate](#)
[QAction](#)
[QCache](#)
[QColor](#)  
[QAbstractItemModel](#)
[QApplication](#)
[QCalendarWidget](#)
[QColorDialog](#)

[QAbstractItemView](#)   [QButtonGroup](#) [QCheckBox](#)   [QColumnView](#)  
[QAccessible](#)   [QByteArray](#)   [QClipboard](#)

If a URL is supplied, each item is linked to a URL that is formed by concatenating the URL, the class name, converted to lowercase, and ".html". For example, if the base URL is `http://qt-project.org/doc/qt-5`, `QApplication` links to `http://qt-project.org/doc/qt-5/qapplication.html`.

### 8.1.3.6 Convert multicolumn table back to one-per-line item list

This undoes the conversion listed above. It is intended to make it much easier to insert and remove items in the list.

## 8.1.4 Equations

Nexus detects equation markup and automatically includes the necessary references to the [MathJax](#) server, which formats the equations.

Nexus doesn't presently detect all equation formats. If your equation won't render, add this somewhere in the file. This will force MathJax to be included.

```
<!-- MathJax -->
```

MathJax slows web page rendering significantly, so it is not included unless it is needed.

An identity of Ramanujan, courtesy of the MathJax site:

*[Math Processing Error]*

The Cauchy-Schwarz Inequality

*[Math Processing Error]*

Note: Equation rendering quality by Qt's WebKit is somewhat lower in quality than rendering by some web browsers.

## 8.1.5 Style Control

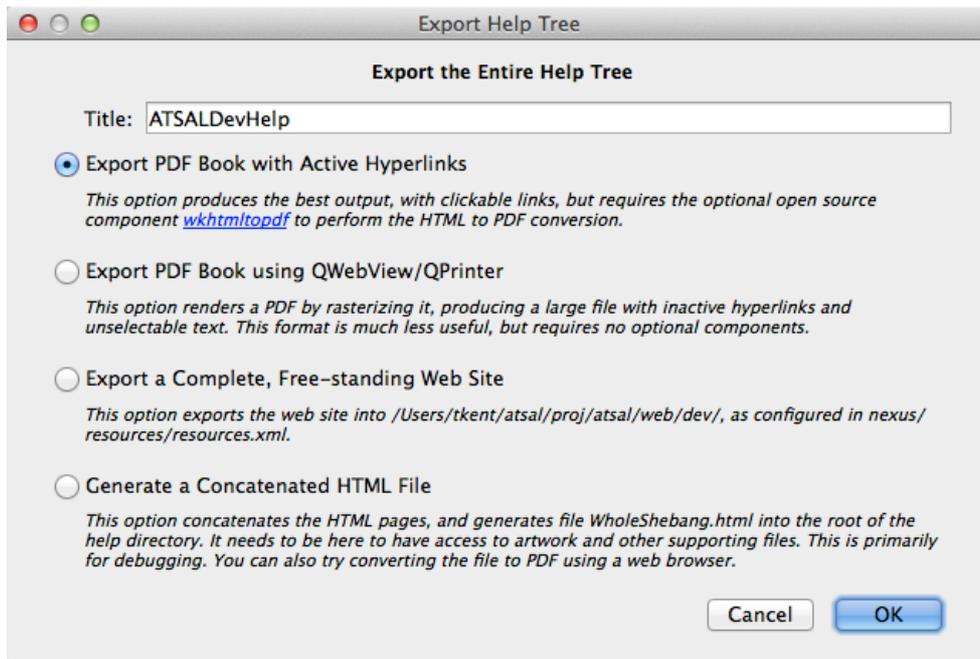
To obtain a uniform look to Nexus-generated pages, Nexus begins with a hard-coded header derived from Doxygen's approach. This header includes `doxygen.css`, and after that, overrides in `stylesheet.css`, to set up the default look. When Nexus generates a new help tree, it adds default versions of these files and several others to the tree. You can override these files as needed to create your own style changes.

## 8.1.6 Printing

Printing is not yet supported, but you can export individual help topics to PDF files via `File > Export Help Topic as PDF...`

## 8.1.7 Exporting the Help Tree

Nexus can export an entire help tree as shown below:



### 8.1.7.1 Exporting a PDF Book with Active Hyperlinks

I experimented with several methods of HTML-to-PDF conversion. This version of Qt can use the classes `QWebView` and `QPrinter` to generate a PDF, but this is done by rasterizing the document. The result is bulky, and its hyperlinks are dead and its text is not selectable. There is little benefit to trying to develop a workaround within Qt, since `QWebView` is deprecated in later Qt releases. However, I retained this option because it introduces no new external dependencies.

I also tried generating the help tree, then loading it into Safari and printing it to a PDF. This produces a better result—text is selectable, and most of the links work—but the self-contained links in the table of contents do not work. You can use this option by choosing the last choice in the dialog to produce a concatenated HTML file, then loading and printing yourself.

After some searching, I found [wkhtmltopdf](#), an open source package that does an excellent job of HTML to PDF conversion. It generates a PDF with an active table of contents that can be displayed by many PDF viewers, and solves various other layout issues. However, I cannot justify adding this as a formal dependency of ATSAAL, so its installation and use are optional.

### 8.1.7.2 Exporting a Complete Web Site

This option generates a full web site into a pre-determined location. The web site can stand on its own, but for use with ATSAAL, the help tree is intended as one part of a larger tree. ATSAAL's use of Nexus-generated web sites is discussed in [ATSAAL Web Site](#).

### 8.1.7.3 Generating a Concatenated HTML File

This is really the first step in generating a PDF book: the files are concatenated into a single large web page, with headers and styles chosen to create a book. You can generate the web page alone for debugging.

### 8.1.7.4 Support Files

Support files, such as icons, header files, CSS and Javascript files, are handled in one of two ways depending upon how Nexus uses them. Masters for all such files are stored in `atsal/nexus/resources`, since they are deployed from there to several destinations. **Hence these files will be overwritten each time Nexus runs or exports a help tree: be sure to edit the masters in `/atsal/nexus/resources`!** If Nexus needs a file to generate support files, it is loaded as a Qt resource in the file `hresources.qrc` as part of the Nexus executable. If Nexus needs to deploy the master file to other destinations,

it follows a deployment process defined in `atsal/nexus/resources/resources.xml`.

Here is a snapshot of the file. The source path is the relative path to directory containing the master files. Each Destination entry creates a shorthand for a destination. Each File entry specifies a filename and a list of destinations. If a File entry specifies a convert keyword, the file is converted in some way during transfer.

```
<!-- Nexus uses this to copy files from the specified source to one or more
destinations as needed. The Nexus Export... or Update Supporting Files...
commands copy all these files to their specified destinations. If
"convert" is specified, the file is converted using the specified
converter. -->

<WorkingDirectory path= "~/atsal/proj/atsal/" version="0.9">
  <!-- Master supporting files are always stored here. -->
  <Source path="nexus/resources/" />

  <!-- These assign a short name to a destination path -->
  <Destination name="root" path="web/" />
  <Destination name="dev" path="web/dev/" />
  <Destination name="help" path="web/help/" />
  <!-- A special destination, ".", is the root of the currently open project. -->

  <File name="robots.txt" dest="root" />
  <File name="doxygen.css" dest="root, dev, help" />
  <File name="stylesheet.css" dest="root, dev, help" />
  <File name="doxygen.png" dest="dev, help" />
  <File name="tabs.css" dest="dev, help" />
  <File name="ATSAL Icon 2 64x64.png" dest="root, dev, help" />
  <File name="dynsections.js" dest="root, dev, help" />
  <File name="jquery.js" dest="root, dev, help" />
  <File name="ATSALSplashScreen.png" dest="root, dev, help" />
  <File name="cover.html" dest="dev, help" convert="expandInserts" />
  <File name="minus.png" dest="dev, help" />
  <File name="plus.png" dest="dev, help" />
  <File name="bullet.png" dest="dev, help" />
  <File name="mktree.css" dest="dev, help" />
  <File name="mktree.js" dest="dev, help" />
  <File name="tabs.css" dest="dev, help" />
  <File name="toplevelstyles.css" dest="root" />

  <File name="Home.png" dest="root, dev, help" />
  <File name="Downloads.png" dest="root, dev, help" />
  <File name="Question.png" dest="root, dev, help" />
  <File name="ContactUs.png" dest="root, dev, help" />
  <File name="Developer.png" dest="root, dev, help" />
  <File name="Documentation.png" dest="root, dev, help" />

  <!-- These files are needed in the project, but not in the generated output. -->
  <File name="PDFStylesheetActive.css" dest="." />
  <File name="PDFStylesheetPassive.css" dest="." />
  <File name="PDFHelpHeaderActive.html" dest="." />
  <File name="PDFHelpHeaderPassive.html" dest="." />
</WorkingDirectory>
```

### 8.1.7.5 Doxygen

In the special case of the ATSAAL developer help, Nexus exports the Doxygen-generated doc tree along with the rest of the help tree. Since this case might recur for other trees as well, Nexus uses the rule that it exports anything in a `dox` directory. This can be any combination of machine- and manually-generated documentation.

ATSAAL's rebuild procedures generate documentation directly into `atsal/ATSALDevHelp/dox`.

### 8.1.7.6 HTML Blocks

I found it a bit confusing to choose between DIVs, FRAMES, IFRAMEs, and FlexBoxes, but finally settled on Flexboxes containing DIVs containing IFRAMEs. IFRAMEs are used to load all web pages, meaning that the formatting for the help viewer is completely independent from that for the loaded pages. This means you can fine-tune the former without fear of disrupting the latter.

Here are some links on the subject.

[DHTML Expandable and Collapsible Tree from JavascriptToolbox.com](#)

[Create expandable TOC \(table of contents or sitemap\) in HTML : Useful and interesting web sites](#)

On FlexBoxes:

[jquery - Why is percentage height not working on my div? - Stack Overflow](#)

<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

## 8.1.8 Internal Notes

### 8.1.8.1 File Organization

A Nexus document contains an absolute or relative pathname for the doc tree on which it operates. Nothing more and nothing less.

A Nexus doc tree contains style sheets provided by default, which may be adjusted as needed, as described in the preceding section.

The doc tree also contains `toc.xml`, which stores the table of contents. This can be edited by hand if desired.

### 8.1.8.2 ATSal Build Process

ATSal's debug build procedure references the help file in its present location. The release build procedure copies the help tree into the ATSal executable bundle.

### 8.1.8.3 To Do

Search and replace is implemented for editing new pages in edit mode, but not for doing multiple-page searches in view mode. Probably best to use a web page indexing tool to create a search index, that way external web pages can be indexed efficiently.

## 8.1.9 Nexus Futures

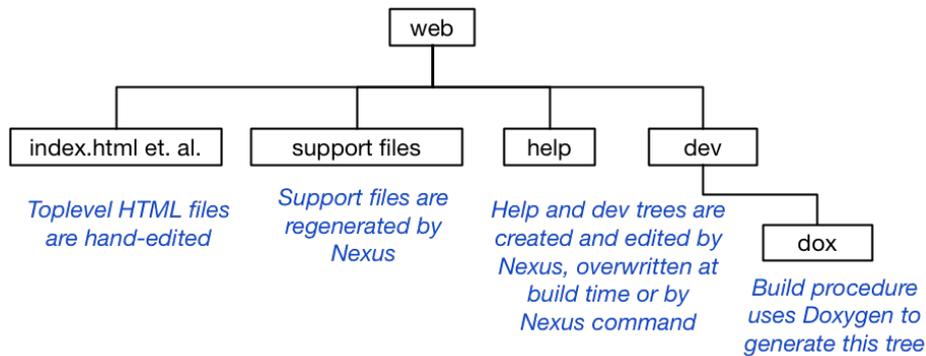
Nexus is designed to act as a free-standing help system, web site, or book. But ATSal combines two Nexus-generated help trees—docs and developer docs. So there is an umbrella web site to tie the two trees together and add other top level items. Currently the top level files are managed largely outside of Nexus, except for the `resources.xml` file that Nexus uses to deploy supporting files to various destinations.

It would be better to include an umbrella web site option in Nexus. The `resources.xml` file is currently loaded into Nexus as a Qt resource. Instead, it could be added explicitly to a Nexus project, and could contain the top level files, as well as pointing to the contained help trees. The container level would allow edits to the top level files analogous to those in the subtrees.

A more general solution would merge `toc.xml` and `resources.xml` to produce a hierarchical help tool that can contain sub-trees. And it would simplify the process of fine-tuning CSS and other files to handle the different layout issues.

## 8.2 ATSal Web Site

ATSal builds its documentation products directly into `atsal/web`, producing a web site that is then uploaded to ATSal.org. Most of this tree is automatically generated by Nexus or Doxygen. **Any changes will be overwritten. The only exception are the half dozen or so HTML files in the `atsal/web` directory.** These are presently managed by hand.



Nexus manages a help tree for user documentation, and an independent tree for developer docs. The former tree is incorporated directly into the ATSAAL application in order to avoid version skew, but the latter is not. As a result, they are managed as separate trees. Nexus can update either tree interactively, and the local web site `help` and `dev` directories are also updated during an ATSAAL build. Similarly, a complete ATSAAL build regenerates the Doxygen doc tree into `web/dev/dox`.

`atsal/updatewebsite.sh` copies the web site to the web host, where it is manually untarred and deployed.

## 8.3 Doxygen

ATSAAL relies on Doxygen to generate docs for ATSAAL's internals for developers. It is not yet determined how end user docs for Python will be generated.

### Doxygen Web Site

[Doxygen Web Site](#)

(Remote URLs are not yet included inline in PDFs.)

### MathJax

[MathJax](#)

(Remote URLs are not yet included inline in PDFs.)

## 9 Python Subsystem

This section discusses build issues and references external python documentation.

### 9.1 Build Issues

#### 9.1.1 Python

I used the Python 3.3.3 build procedure to create a Mac framework, but the build failed, attempting to install into a framework directory that had not yet been created. I modified `Python-3.3.3/Mac/Makefile.in` by adding the lines shown in red.

```
install_pythonw: pythonw
# THK, 2014-11-20: Added line below to prevent build error
mkdir "$(DESTDIR)$(prefix)/bin"
$(INSTALL_PROGRAM) $(STRIPFLAG) pythonw "$(DESTDIR)$(prefix)/bin/pythonw$(VERSION)"
$(INSTALL_PROGRAM) $(STRIPFLAG) pythonw "$(DESTDIR)$(prefix)/bin/python$(VERSION)"
ln -sf python$(VERSION) "$(DESTDIR)$(prefix)/bin/python3"
ln -sf pythonw$(VERSION) "$(DESTDIR)$(prefix)/bin/pythonw3"
ifndef $(LIPO_32BIT_FLAGS),
lipo $(LIPO_32BIT_FLAGS) -output $(DESTDIR)$(prefix)/bin/python$(VERSION)-32 pythonw
lipo $(LIPO_32BIT_FLAGS) -output $(DESTDIR)$(prefix)/bin/pythonw$(VERSION)-32 pythonw
ln -sf pythonw$(VERSION)-32 "$(DESTDIR)$(prefix)/bin/pythonw3-32"
ln -sf python$(VERSION)-32 "$(DESTDIR)$(prefix)/bin/python3-32"
endif
endif
```

#### 9.1.2 PythonQt

The packaged download would not build, too many version skew issues, so I downloaded the latest code via:

```
svn checkout svn://svn.code.sf.net/p/pythonqt/code/trunk pythonqt-code
```

This obtained revision 371.

I added `macbuild.sh` to build PythonQt for AT&T. The procedure generates build scripts for the components, builds them, and then fixes references to shared libraries.

To support XCode 6.1 prior to its formal support by Qt, I modified `qt-install-dir/5.3/clang_64/mkspecs/qdevice.pri` to include the line:

```
!host_build:QMAKE_MAC_SDK = macosx10.9
```

This was done because XCode 6.1 does not include Mac OS X SDKs prior to 10.9, not because it was desirable to switch to 10.9.

## Python Web Site

[Python Web Site](#)

(Remote URLs are not yet included inline in PDFs.)

## PythonQt Web Site

[PythonQt Web Site](#)

(Remote URLs are not yet included inline in PDFs.)

## PySide Web Site

[PySide Web Site](#)

(Remote URLs are not yet included inline in PDFs.)

# PyQt Web Site

[PyQt Web Site](#)

(Remote URLs are not yet included inline in PDFs.)

## 10 Python ATSal Library

This section presents the ATSal functions and classes that are exposed to the Python layer for python programming. In developer mode (marked ) , extra functions are surfaced to assist with testing. Later, when these protocols mature, some of these will be supported for users as well.

The ATSal notebook, and each of its tools, are surfaced via predefined variable names. Other objects are surfaced by obtaining them from these starting points.

Typically, objects created by ATSal are owned by ATSal; those instantiated directly from python are owned by python. `Tools` and `Tables` are examples of objects owned by ATSal even if they were created at the behest of a python program. A `QDialog` created in a python program is owned by the python program. But exceptions occur. For example, `TableRanges` are created by ATSal from `Tables`, but then handed off to python. For these cases, ATSal explicitly transfers ownership of the object to python.

### 10.1 The ATSal Application

The ATSal application object, `app`, is used to create additional notebooks. `app` is pre-defined at startup, and is of type `ATNotebook`.

	<code>quit(path, status)</code>	Quits ATSal. This should be the last line of any notebook test program, since ATSal will otherwise remain open, and the test will not complete until it the timeout expires and it is killed. If <code>path</code> is not empty, saves the notebook to this file; otherwise quits without saving. <code>status</code> is the exit status. The exit status is ignored except when ATSal is running a test suite. In this case, it is <code>at.ESSuccess (0)</code> , <code>at.ESFailure (1)</code> , or <code>at.ESWarning (2)</code> . If a refresh cycle is currently in progress, ATSal waits until the cycle completes to process the quit request.
-----------------------------------------------------------------------------------	---------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 10.2 Notebook Class

The notebook is the toplevel object in ATSal. Since there is one python notebook per process, `notebook` is the predefined global variable used to access it. Functions marked  are experimental, only enabled for developer mode (though they may later be added for users as well). All enumerators are accessed in the `at` scope, e.g. `at.TTObservatory` specifies the observatory tool.

	<code>documentBaseName()</code>	Returns the document name.
	<code>toolFromIndex(index)</code>	Returns the <code>Tool</code> at the specified index. Tools are also available by the name assigned in ATSal, e.g. the variable <code>observatory1</code> .
	<code>toolFromName(name)</code>	Returns the <code>Tool</code> named <code>name</code> .
	<code>toolCount()</code>	Returns a count of tools in the notebook.
	<code>insertTool(toolType)</code>	Inserts a tool and returns it. <code>toolType</code> is one of <code>TTObservatory</code> , <code>TTTag</code> , <code>TTObservatory</code> , <code>TTPlot</code> , <code>TTMatplotlib</code> , <code>TTResults</code> , <code>TTPython</code> , <code>TTText</code> , and <code>TTTable</code> .
	<code>resultsTable(dataSetID, createIfNeeded)</code>	Returns a table with results from a fit.
	<code>showToolEditor(index, show)</code>	Displays or hides the tool editor for the tool at <code>index</code> .
	<code>update()</code>	Updates the user interface in response to changes made via a python program.
	<code>refresh()</code>	Performs a refresh cycle. This function blocks until the refresh cycle completes.

#### 10.2.1.1 XSpec Initialization Functions

These settings apply to all models in the current notebook. Make all necessary calls, then call `saveXSpecInitSettings()` to save them; otherwise XSpec will not see the changes.

	<code>setDummyResponse(beginRange, endRange, bins, responseType)</code>	Sets the dummy response range and number of bins, and specifies a <code>responseType</code> of <code>DRTLinear</code> or <code>DRTLogarithmic</code> .
-------------------------------------------------------------------------------------	-------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------

 <code>setPhotonAbsorptionTable(table)</code>	Sets the photon absorption cross section table to: <ul style="list-style-type: none"> <li>• PACSVern: Verner et al. (1996, ApJ 465, 487)</li> <li>• PACSBcmc: Balucinska-Church &amp; McCammon (1992, ApJ 400, 699) with new He cross-section from (1998, ApJ 496, 1044)</li> <li>• PACSOBmc: BCMC with the old He cross-section.</li> </ul>
 <code>setSolarAbundanceTable(table)</code>	Sets the solar abundance table to: <ul style="list-style-type: none"> <li>• SAFe1d: Feldman, U., 1992. Physica Scripta, 46, 202</li> <li>• SAAng: Anders, E. &amp; Grevesse, N., 1989. Geochimica and Cosmochimica Acta 53, 197</li> <li>• SAAneb: Anders, E. &amp; Ebihara, 1982. Geochimica and Cosmochimica Acta 46, 2363.</li> </ul>
 <code>setFittingMethod(method)</code>	Sets the fitting method, method, to: <ul style="list-style-type: none"> <li>• FMLeven: modified Levenberg-Marquardt algorithm</li> <li>• FMMigrad: Minuit2 migrad method</li> <li>• FMSimplex: Minuit2 simplex method.</li> </ul>
 <code>setStatisticToMinimize( statistic )</code>	Sets the statistic to minimize to MSChi ( $\chi$ ), or MSCStat (cstat).
 <code>setWeightingTechnique(weighting)</code>	Sets the weighting technique to WTStandard, WTGehrels, WTChurazov, or WTModel.
 <code>setNumericalDifferentiation( enable )</code>	If enable is true, uses numerical differentiation.
 <code>setCosmologyParameters(h0, q0, lamda0)</code>	Sets the cosmology parameters $h_0$ , $q_0$ , and $\lambda_0$ .
 <code>setWavePlotUnits(units)</code>	Sets the wave plot units to WPUHz or WPUAngstroms.
 <code>setFitDeltas(deltas, proportion)</code>	If deltas is FDPProportional, sets the fit delta to proportion; if FDPFixed, proportion is ignored.
 <code>saveXSpecInitSettings()</code>	Saves changes made by any of the calls in this section to this notebook's Xspec.init.

## 10.3 Tools

Tools are directly available to python via their names, e.g., `observatory1`, or they are returned from the notebook as discussed in the previous section.

### 10.3.1 Tool Class

<code>toolName()</code>	Returns the tool name.
<code>comment()</code>	Returns the tool comment.
<code>refresh()</code>	Refreshes the notebook. Returns a status of TRSIdle, TRSFinished, or TRSActive. This is called from the main refresh loop.
<code>refreshError()</code>	Returns an error status of RESuccess, REWarn, or REFail.
<code>update()</code>	Updates the user interface to reflect changes made to this tool via a python program. Calling the notebook's <code>update()</code> function calls this for all tools in the notebook.

#### 10.3.1.1 ToolObservatory Class

<code>selectFile(file, select)</code>	Selects or deselects file based on the boolean select.
---------------------------------------	--------------------------------------------------------

### 10.3.1.2 ToolTag Class

 autoTag()	Creates default tags for files selected in the observatory tool.
 addTag(PHAFfile)	Creates a new tag for the specified PHA file and returns its auto-assigned name. The filename is specified as a partial pathname: either a filename alone or a group/filename. To set other associated files (RMF, ARF, COR, or background), retrieve the FileTag from the XSpectrum and assign the filenames.
 update()	Updates the tag tool editor to reflect changes made from python.

### 10.3.1.3 ToolPlot Class

addSpectrum(const QString& tagName)	Adds a new spectrum for tagName, and returns the XSpectrum.
-------------------------------------	-------------------------------------------------------------

### 10.3.1.4 ToolPython Class

 setPythonProgram(program)	Sets the python program to program, a set of newline-separated lines.
-------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------

### 10.3.1.5 ToolMatplotlib Class

TBD

### 10.3.1.6 ToolResults Class

It might be better to make this table readonly.

table()	Returns the Table containing the data.
 update()	Updates the user interface to match any changes made to the table.
 select(plotToolName, spectrumName, fitName)	<p>Selects the specified plot tool, spectrum, and fit results for display in the results tool. Example:</p> <pre>resultsTool.select("plot1", "Spectrum 0", "Model 0")</pre>

### 10.3.1.7 ToolText Class

#### 10.3.1.8

exportAsHTML(path)	Exports the text in HTML format.
--------------------	----------------------------------

### 10.3.1.9 ToolTable Class

table()	Returns the Table containing the data.
update()	Updates the user interface to match any changes made to the table.

## 10.3.2 Plot Tool

This section describes classes used by the plot tool. A ToolPlot contains spectra (XSpectrum). Each spectrum contains fits (XFit). Each fit contains parameters (XModelParamInstance). And each parameter in turn contains quantities (Quantity).

Functions marked  are experimental, only enabled for developer mode (though they may later be added for users as well).

### 10.3.2.1 XSpectrum Class

An `XSpectrum` operates on a tag, or set of input files (spectrum, response file, ancillary response file, and/or background file). It produces some basic plots and contains one or more fits to analyze the data.

	<code>addModel(modelExpression)</code>	Adds a model <code>modelExpression</code> to the spectrum, returning the newly created <code>XFit</code> . This does not initiate any computing.
	<code>fileTag()</code>	Returns the spectrum's <code>FileTag</code> .

### 10.3.2.2 XFit Class

	<code>setExpression(modelExpression)</code>	Changes the fit's <code>modelExpression</code> .
	<code>parameterValue(index)</code>	Returns the value subparameter at <code>index</code> , in native units. (This interface does not return other subparameters like delta or hard minimum—see below.)
	<code>parameter(index)</code>	Returns the <code>XModelParamInstance</code> at <code>index</code> . This contains all the subparameters.
	<code>paramCount()</code>	Returns a count of the parameters for this fit. This is the sum of the parameters for each model in the model expression.

### 10.3.2.3 XModelParamInstance Class

An `XModelParamInstance` represents a single parameter and its (up to) six subparameters.

	<code>name()</code>	Returns the parameter name as found in XSpec's <code>model.dat</code> file.
	<code>paramType()</code>	Returns the parameter type: <ul style="list-style-type: none"> <li>• <code>XMPTParameter</code>: a set of doubles that participate in the fit</li> <li>• <code>XMPTFloat</code>: a single double-precision value, typically a normalization</li> <li>• <code>XMPTCheckbox</code>: a checkbox that generates a boolean value</li> <li>• <code>XMPTRadioButtons</code>: a pair of radio buttons that generate a boolean value</li> <li>• <code>XMPTPopup</code>: a popup menu with a set of enumerations</li> <li>• <code>XMPTFilePath</code>: a pathname.</li> </ul>
	<code>subQuantityCount()</code>	Returns the number of subquantities. This value is six for <code>XMPTParameters</code> and one for the others.
	<code>subQuantity(subParameter)</code>	Returns the <code>Quantity</code> for <code>subParameter</code> .
	<code>isDouble()</code>	Returns <code>True</code> if this parameter contains subparameters of type <code>QuDouble</code> .
	<code>physicalType()</code>	Returns the physical type of the parameter: <code>PTDistance</code> , <code>PTPhoton</code> , <code>PTChanPhoton</code> , <code>PTAmplitude</code> , <code>PTVelocity</code> , <code>PTElapsedTime</code> , <code>PTCountsPerTime</code> , <code>PTNormalizedCountsPerTime</code> , <code>PTEmissivity</code> , <code>PTTemperature</code> , or <code>PTChannel</code> . If the parameter is not a <code>QuDouble</code> , or has no physical type, this returns <code>PTNone</code> .
	<code>physicalTypeString()</code>	Like <code>physicalType()</code> , but returns a string.
	<code>value(units = -1)</code>	Returns the value subparameter. If <code>units</code> is specified, converts to the requested units; otherwise returns native units. <b>Need to list all the possible units in a separate table.</b>
	<code>setValue(value, units = 1)</code>	Sets the value subparameter to <code>value</code> . If <code>units</code> is specified, the value is converted from the specified units to internal units. Otherwise <code>value</code> must be supplied in native units. <b>Need to document each physical type's native units.</b>

 <code>delta(units = -1)</code>	Returns the delta subparameter. If <code>units</code> is specified, converts to the requested units; otherwise returns native units.
 <code>setDelta(delta, units = -1)</code>	Sets the delta subparameter to <code>delta</code> . If <code>units</code> is specified, the argument is converted from the specified units to internal units. Otherwise it must be supplied in native units.
 <code>hardMinimum(units = -1)</code>	Returns the hard minimum subparameter. If <code>units</code> is specified, converts to the requested units; otherwise returns native units.
 <code>setHardMinimum(hardMin, units = -1)</code>	Sets the hard minimum subparameter to <code>hardMin</code> . If <code>units</code> is specified, the argument is converted from the specified units to internal units. Otherwise it must be supplied in native units.
 <code>softMinimum(units = -1)</code>	Returns the soft minimum subparameter. If <code>units</code> is specified, converts to the requested units; otherwise returns native units.
 <code>setSoftMinimum(hardMin, units = -1)</code>	Sets the soft minimum subparameter to <code>softMin</code> . If <code>units</code> is specified, the argument is converted from the specified units to internal units. Otherwise it must be supplied in native units.
 <code>softMaximum(units = -1)</code>	Returns the soft maximum subparameter. If <code>units</code> is specified, converts to the requested units; otherwise returns native units.
 <code>setSoftMaximum(hardMin, units = -1)</code>	Sets the soft maximum subparameter to <code>softMax</code> . If <code>units</code> is specified, the argument is converted from the specified units to internal units. Otherwise it must be supplied in native units.
 <code>hardMaximum(units = -1)</code>	Returns the hard maximum subparameter. If <code>units</code> is specified, converts to the requested units; otherwise returns native units.
 <code>setHardMaximum(hardMin, units = -1)</code>	Sets the hard maximum subparameter to <code>hardMax</code> . If <code>units</code> is specified, the argument is converted from the specified units to internal units. Otherwise it must be supplied in native units.
 <code>setLocked(locked)</code>	Locks or unlocks the parameter. No effect if the parameter is not lockable.

### 10.3.2.4 FileTag Class

See `ToolTag` to create a new `FileTag`. Pathnames are relative to the observation directory. `hrc_pha2.fits` refers to a toplevel file, or `obs_1909_tgid_3678/heg_1.pha` selects a file in an observation group. An empty partial pathname sets the option to none. Call `update()` on effected tools (or the notebook) to reflect changes in the user interface.

 <code>setSpectrumNumber(spectrumNumber)</code>	Selects a spectrum from a file containing multiple spectra. <b>Spectrum-numbering is 1-based, not zero-based.</b>
 <code>spectrumNumber()</code>	Returns the selected spectrum number.
 <code>name()</code>	Returns the tag name, e.g. <code>f3</code> .
 <code>setPHAFile(partialPath)</code>	Sets the PHA file for this tag to <code>partialPath</code> , which is either a filename or a group/filename.
 <code>PHAFile()</code>	Returns the PHA file as a full path.
 <code>setRMFFFile(partialPath)</code>	Sets the response matrix file. An empty <code>partialPath</code> sets it to no ARF.
 <code>RMFFFile()</code>	Returns the response matrix file as a full path.
 <code>setARFFFile(partialPath)</code>	Sets the ancillary response file.
 <code>ARFFFile()</code>	Returns the ancillary response file.
 <code>setCORFile(partialPath)</code>	Sets the correlation file.
 <code>CORFile()</code>	Returns the correlation file as a full path.
 <code>setBGFile(partialPath)</code>	Sets the background file.
 <code>BGFile()</code>	Returns the background file as a full path.

## 10.4 Tables

This interface is not yet fully debugged or implemented.

ATSAL tables often contain blocks with smaller tables. The position and size of these tables may vary, so we need some means of locating and operating on these blocks. Hence there are three types of tables:

- A `TableRange` is a rectangular subset of a `Table`, offering a convenient way to work on some piece of a larger table. A `Table` is a subclass of `TableRange`, and both support the same functions.
- A `Table` is a real table. Any number of `TableRanges` may reference the same `Table`.
- A subtable is a `Table` within a `Table`, overlaid on top of the main table. This is a way to place array data in a larger table.

Functions marked  are experimental, only enabled for developer mode (though they may later be added for users as well). **Many more functions will be added.**

### 10.4.1 TableRange Class

Tables such as the one created by the results tool are fairly complex, composed of multiple sub-blocks of variable size and position. It is not convenient to write python code to access these tables. `TableRanges` make it easy to operate on blocks, or blocks within blocks. The `TableRange` class expresses a rectangular subset of a `Table`. `TableRange` is the base class of `Table`, and a `TableRange` can refer to a cell, a row, a column, or a block. Almost all operations are performed on the entire range by default, though a programmer can also iterate through the cells in a range instead.

Example:

```
# Get the table from the table tool
t = table1.table()

# Create a range encompassing the row of labels starting with 'Distance'
# The -1 says "everything up to the first blank column"
r = t.createRange('Distance', 'Distance', 1, -1)

# Make the label row bold
r.setBold(true)

# Create a second range with top left (13, 0), 20 rows by 5 columns
r2 = t.createRange(13, 0, 20, 5)

# Print the value of the top left cell so we know we're in the right place
cell = r2.cellAt(0, 0)
print(cell.string())

# Extract the first column from the range above: all the rows and one column
col = r2.createRange(0, 0, -1, 1)

# Make the column italic
col.setItalic(True)

# Now update the table so we see the changes. Note that this applies to the table tool,
# not the table itself.
table1.update()
```

A `TableRange` is created via a `Table`'s `createRange()` function. The range is specified by a (row, column) origin and a count of rows and columns. Operations may then be performed on the entire range, or on cells within the range. A `TableRange` can also be created from another `TableRange`. So, for example, a range might express a sub-table within a table, and ranges created from that range would isolate rows and columns of interest in the sub-table.

	Creates and returns a <code>TableRange</code> whose origin at top left is (row, column), with the specified number of rows and columns. Examples:
	<pre># A single cell at (10, 15) table.createRange(10, 15)  # A single cell whose column is the label "Energy" table.createRange(10, 'Energy', 1, 1)  # range is 5 rows by 1 column</pre>

<pre>createRange(row = 0, column = 0, rows = 1, columns = 1, access = at.TATopTable)</pre>	<pre>table.createRange('Alpha Centauri', 15, 5, 1)  # a block bounded by the first blank row and column table.createRange('Alpha Centauri', 'Energy', -1, -1)  # a "Distance" column selected from another range range.createRange(1, 'Distance', -1, 1)  # A cell selected from the main table instead of the uppermost subtable table.createRange(10, 10, 1, 1, at.TABaseTable)</pre> <p>If row or column is a text string, it is the first label cell that matches the string. Label cells must be specially marked, and appear with a light gray background. If the rows and columns arguments are omitted, they default to a single cell. If supplied:</p> <ul style="list-style-type: none"> <li>• A value of -1 extends to the first blank row or column. This provides a way to pick a block out of a larger table without knowing its size in advance. If both rows and columns are -1, the block's row count is terminated by the first blank row all the way across, then the first blank column within that range of rows.</li> <li>• A value of 0 extends to the edge of the table or subtable. This is most often used to specify the rest of a row.</li> <li>• A value of &gt;0 specifies a fixed number of rows or columns.</li> </ul> <p>The access argument:</p> <ul style="list-style-type: none"> <li>• <code>TABaseTable</code>: the base table, i.e., cells underneath any overlaid arrays</li> <li>• <code>TATopTable</code>: the uppermost table cell. If this argument is chosen (default), and the range's origin lies within a subtable, the range is also restricted to the subtable.</li> </ul> <p>For efficiency reasons, a <b>range does not adapt to changes in table size or content</b>. It is calculated once at creation time. If you need to recompute the range, use <code>autosize()</code>, or create a new <code>TableRange</code>.</p>
<pre>setCell(row, column, value)</pre>	<p>Sets the specified cell to value. Examples:</p> <pre># Sets all cells in the range to 0 setCell(0)  # Sets cell at (10, 15) to 'Hello, world!' setCell('Hello, world!', 10, 15)  # Sets cell at (10, 'Energy') to 42 setCell(42, 10, 'Energy')  # Sets cell at ('Alpha Centauri', 15) to '6.022e23 +-3.12e22' setCell('6.022e23 +-3.12e22', 'Alpha Centauri', 15)</pre>
<pre>cellString(row, column, access)</pre>	<p>Returns the specified cell. access is:</p> <ul style="list-style-type: none"> <li>• <code>TABaseTable</code>: the base table, i.e., cells underneath any overlaid arrays</li> <li>• <code>TATopTable</code>: the uppermost table cell</li> </ul>
<pre>cellDouble(row, column, access)</pre>	<p>Returns a cell as a double. Returns zero if the cell does not contain an integer or a double. access is:</p> <ul style="list-style-type: none"> <li>• <code>TABaseTable</code>: the base table, i.e., cells underneath any overlaid arrays</li> <li>• <code>TATopTable</code>: the uppermost table cell</li> <li>• <code>TABlock</code>: a block, or member of a current subtable.</li> </ul>

<code>labelRow(label)</code>	Returns the first row number with a cell whose text matches <code>label</code> , or -1.
<code>labelColumn(label)</code>	Returns the first column matching <code>label</code> , or -1.
<code>rowCount()</code>	Returns the number of rows in the table.
<code>columnCount()</code>	Returns the column count.
<code>clear(row = -1, column = -1)</code>	Clears range, does not change table size. If <code>row</code> and <code>column</code> are omitted, the entire range is cleared.
<code>setSignificantDigits(digits, row = -1, column = -1)</code>	Sets the number of digits to display.
<code>setAlignment(alignment, row = -1, column = -1)</code>	Sets the text alignment.
<code>setBold(enable, row = -1, column = -1)</code>	Enables or disables boldface.
<code>setItalic(enable, row = -1, column = -1)</code>	Enables or disables italic.
<code>setWrapText(enable, row = -1, column = -1)</code>	Enables or disables text wrapping.
<code>setDateFormat(format, row = -1, column = -1)</code>	Sets the date/time format: <ul style="list-style-type: none"> <li>• <code>DTFCurrent</code>: Original units</li> <li>• <code>DTFDateTime</code>: Date and HH:MM</li> <li>• <code>DTFDateTimeSec</code>: Date and HH:MM:SS</li> <li>• <code>DTFDate</code>: Date only</li> <li>• <code>DTFTime</code>: Time (HH:MM)</li> <li>• <code>DTFTimeSec</code>: Time (HH:MM:SS).</li> </ul>
<code>setExponentMode(mode, row = -1, column = -1)</code>	Sets the exponent mode: <ul style="list-style-type: none"> <li>• <code>EM10Format</code>: Use <math>x10^n</math> format</li> <li>• <code>EMFormat</code>: Use <math>E_n</math> format</li> <li>• <code>EMUnits</code>: Use metric multiplier.</li> </ul>
<code>setShowUnits(show, row = -1, column = -1)</code>	Enables or disables inclusion of units.
<code>setShowErrors(show, row = -1, column = -1)</code>	If enabled, appends error(s) to value, if any.
<code>setShowZero(show, row = -1, column = -1)</code>	If disabled, cells containing zero are blank.
<code>setTextStyle(style, row = -1, column = -1)</code>	Selects text style, a string.
<code>setTextSize(size, row = -1, column = -1)</code>	Sets the font size. <code>size</code> is a double.
<code>setBackgroundColor(color, row = -1, column = -1)</code>	Sets the cell background color. <code>color</code> is a <code>QColor</code> .
<code>setTextColor(color, row = -1, column = -1)</code>	Sets the text color. <code>color</code> is a <code>QColor</code> .
<code>sum()</code>	Sums the elements in the range.
<code>average()</code>	Averages the elements in the range.

## 10.4.2 Table Class

So far at least, a `Table` inherits all its functions from the `TableRange` class, preceding.

To create a table, create a table tool (or results tool), then obtain its table. For table tool `table1`:

```
table = table1.table()
```

### 10.4.3 TableCell Class

This hasn't been thought through at all yet. This may not be necessary at all.

 TableCell()	Returns a new TableCell.
string()	Returns the value as a string.
asFloat()	Returns the value as a double, or returns 0 if the cell is non-numeric.

# 11 XSPEC Server

`xspecserver` is an extension of XSPEC designed to accept commands and return results over a socket connection. This allows one or more instantiations of XSPEC, running on the local or remote systems, to be used by ATSAL or by other programs, with a general goal of more efficient use of compute resources.

At this stage, server activation and use is a manual process, but the long term goal is to make this process automatic, allowing a user to configure secure access to one or more copies of the server running on a given machine. The server uses an XML protocol for executing remote procedure calls (RPC). XML is chosen to make it easier to debug client/server transactions and to avoid problems in passing binary data between dissimilar computer architectures. This protocol has an escape for binary data blobs, encoding them in base 64. Large data files are transferred separately, so the relative inefficiency of XML is avoided.

We use a mature open source package called XMLRPC to implement this communication.

The XMLRPC source tree includes variants for several types of interconnect. We use “pstream protocol,” an informal protocol defined by the XMLRPC project, for the specific reason that it establishes a session—a long-lived connection—rather than establishing a new connection for each transaction. This is more natural and efficient for this application, since `xspecserver` can address only one client at a time, and XSPEC is inherently session-oriented. XSPEC has an internal C++ API that is brought out to its built-in Python interpreter. Initially we considered replicating this part of the API on the client, but this approach was abandoned because of likely severe performance problems. Instead, we adopted a more structured version of the command line itself, together with a series of proxy classes.

During the exploratory phase of ATSAL development, the goal is to prove the viability of the client/server architecture, rather than to refine it by resolving all the security issues. But we start here with the long term goals before returning to the short term goals.

## XSPEC

[XSPEC](#)

(Remote URLs are not yet included inline in PDFs.)

## Frequently Asked Questions

[Frequently Asked Questions](#)

(Remote URLs are not yet included inline in PDFs.)

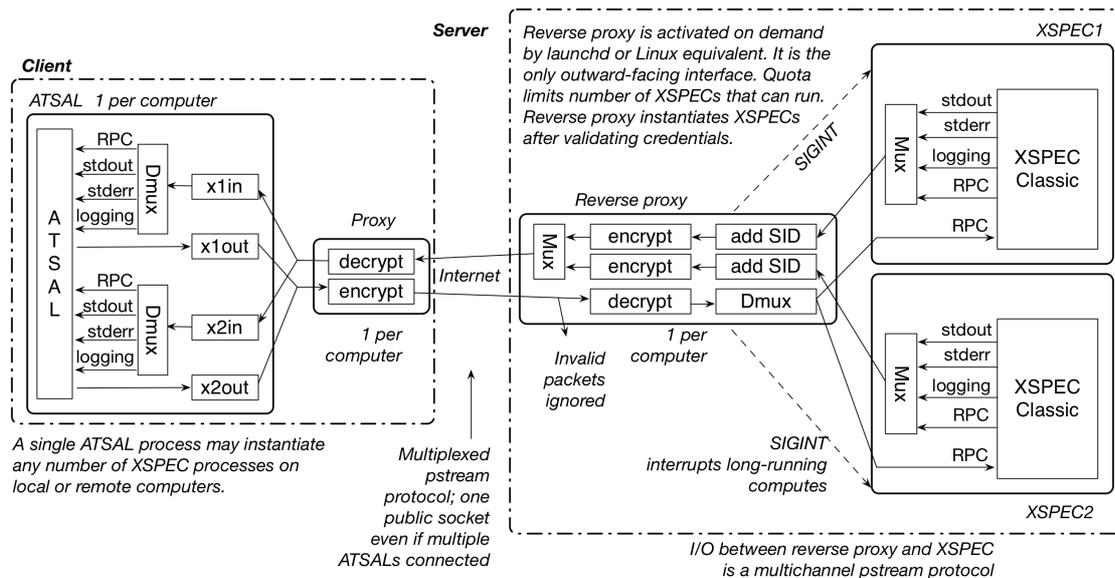
### 11.1 Long Term Architecture

A user will be able to designate any supported system (initially Mac OS X and Linux) as a server, by installing a server package and supplying some simple parameters, including:

- Usernames and passwords of users authorized to access the server
- A quota controlling the total number of XSPEC instantiations permissible
- A quota restricting the total that may be used by an individual user
- A port, in the event that the default port is in use

At boot time, `launchd`, or its equivalent on Linux systems, will launch an XSPEC listener daemon that also acts as a reverse proxy. There is one listener per system, and its purpose is to receive and validate connection requests, instantiate XSPEC servers, and manage their shutdown or restart. The listener also processes all network-facing traffic, acting as a first line of defense in order to reject invalid packets before passing them to XSPEC

On each client, a similar proxy process isolates ATSAL from the network connection. Since there is a maximum of one ATSAL process and one proxy per system, ATSAL itself instantiates the proxy when it starts up. Encryption of traffic is not very important for an application of this sort, except as a hedge against attempts to use the server as an attack vector. Shown below as part of the proxies, it will instead be implemented by optional use of tunneling through ssh.



XSPEC performs many long-running fits that the user may elect to interrupt. This is implemented by sending SIGINT to the XSPEC process. The interprocess communication link shown here handles basic RPC—commands to XSPEC and responses. This traffic level is usually quite low, at the most, arrays of moderate size. Large files such as response matrix files are not sent through this connection, but are sent separately via HTTP.

In addition to the pure RPC connection, stdout, stderr, and logging outputs are shown. Data sent to stdout is parsed to determine command completion status. Data sent to the other channels is available in optional panes for monitoring or debugging.

ATSA L's preference settings will include a list of candidate servers, accessed in the order listed as XSPEC instantiations are needed.

## 11.2 Short Term Architecture

During the prototyping phase, an XSPEC server is instantiated manually, using a public domain utility called `socketexec` to create a socket on a port and pass it via stdin to XSPEC server. ATSA L implements the client side of the connection by opening a socket, using a preference setting for the port. Client and server talk directly rather than via proxy. For debugging and development, they typically run on the same machine.

## 11.3 Server Notes

These are notes made during server implementation. They might come in handy at some point.

`Tcl_PutsObjCmd` calls `Tcl_GetChannelFromObj` to convert channel name to channel object. `Tcl_GetChannelNamesEx` gets channel names.

Status:

1. Output from XSPEC commands redirects as expected.
2. puts from TCL don't redirect ever
3. TCL believes there are only three channels associated with the interpreter, stdin, stdout and stderr.
4. Need a way to redirect TCL stdout too.
5. A `TclStream.cxx` contains the same methods (read/write, etc.) as an `xSChannel`. Also a `Tcl_ChannelType`, which is a C-level driver for a TCL stream.

```
XStream XSUtil/Utils/Xstream pure virtual, has an XStream*
|
+- TclIO (TclStream.h) Uses Tcl_Channels instead of XStreams, assigned post construction.
Writes to the channel.
|
+- ServerIO. Writes to server instead.
```

```

+- TKIo (TclStream.h)

std::iostream
|
+- XSstream (constructs with XSchannel)

XSGlobal::textChan has the server XSChannel, which contains the Tcl_Channel

Tcl_ChannelType: http://tmml.sourceforge.net/doc/tcl/CrtChannel.html#type_Tcl_ChannelType

./configure CFLAGS="-g -O0" CPPFLAGS=-DDEBUG CXXFLAGS="-g -O0"
http://linux.about.com/library/cmd/blcmdl3_Tcl_QueueEvent.htm
http://computer-programming-forum.com/57-tcl/5221f2ffdd733170.htm
Tcl_QueueEvent(Tcl_Event* eventPtr, TCL_QUEUE_TAIL)
return 0 if flags says not to process event, or if no input, otherwise 1
http://stackoverflow.com/questions/14563192/read-all-input-in-event-handler
http://tmml.sourceforge.net/doc/tcl/
Read a key
Tcl_AppendResultVA(...)

http://tclreadline.sourceforge.net
http://tmml.sourceforge.net/doc/tcl

```

Potentially useful TCL functions:

```

Tcl_Eval
Tcl_Async
*Tcl_ReadChars
Tcl_Read
Tcl_Gets
static int TclReadlineCmd(ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST
objv[])

```

## 11.4 ATSAI's XSPEC Command Parser

Each XSPEC command produces one or more lines of output, possibly including optional information, variable length dumps (such as an array), or several lines that match a given pattern. XSPEC output is parsed using a parser that is driven from a table in `xparser.xml`. The parser table is arranged as a series of patterns that match possible outputs for each XSPEC command. A single command's output is considered to fully match the set of patterns if each line of output matches at least one pattern.

The parse table file looks like this. Although designed for XSPEC, a parser for a different application's output could be designed using the same syntax.

```

<?xml version= "1.0"?>
<!-- This file lists XSPEC commands by command names, followed by
possible responses for each command. Each item corresponds to one or more
lines of text, with embedded variable names to which variables in the
response are assigned. -->

<XspecParseTable>
<commands>

<command name="cmd-1">
<block>
block-1
</block>

<block>
block-2
</block>

...
</command>

<command name="cmd-2">
...
</command>

```

```
</commands>
</XspecParseTable>
```

Each block consists of patterns that match one or possibly more than one consecutive lines. Hard returns are interpreted as non-existent, not even considered to be whitespace. Use “\n” to include a return character in a pattern. Whitespace at either end of a pattern is significant.

```
<block>
  Hello, world
</block>
```

matches:

```
  Hello, world
```

but not

```
Hello, world
```

And

```
<block>
Hello,
world
</block>
```

matches

```
Hello,world
```

while

```
<block>
Hello,\nworld
</block>
```

matches

```
Hello,
world
```

Whitespace must match exactly. When you want to accept a flexible amount of whitespace, use <w /> to accept one or more spaces or tabs (but returns are still ignored). For example, three columns of numbers like this:

```
  27 1029  12
73221 198 332
```

could be matched with:

```
<w /><int name="v1"><w /><int name="v2"><w /><int name="v3">
```

However, there is a special case. If an entire pattern is whitespace, it is interpreted the same as <w />, matching any amount of whitespace. Thus:

```
<int name="v1"> <int name="v2"> <int name="v3">
```

is identical to the prior pattern. So is:

```
<int name="v1"> \t <int name="v2"> <int name="v3">
```

A special case, mandated by XML, is that you must encode “<” as `&lt;`; and “>” as `&gt;`;

A command has a command name, which is the command whose output will be parsed. A block consists of a series of patterns that match successive portions of the command output. For example:

```
<block name="specInfo">
Spectral Data File: <filename name="spectrumFile" pattern="*.pha" /> Spectrum <int
name="spectrumCount" />.
```

matches output such as this:

```
Spectral Data File: xyzzy.pha Spectrum 1.
```

After the match is performed, an `XCmdResults` is created. It contains any values that were parsed out by the pattern specifications. If a pattern has an associated name, its value is stored in the `XCmdResults`, accessible by name. For example, `results->structure("specinfo")` returns a data structure named “specinfo.” An optional index accesses the *n*th occurrence of this structure, if the output has more than one line that matches the pattern.

Similarly, `results->variable("specinfo.filename")` returns the filename variable for the first `specinfo` structure.

The following pattern types are accepted:

<i>Pattern</i>	<i>Description</i>
int	64-bit integer with optional leading sign.
posint	64-bit integer with no leading sign.
double	double precision value with optional sign and exponent
doubleerr	a double followed by "+/-" (or "±") followed by another double that is the symmetrical error value, e.g. 4.21572 +/- 1.90059E-02. The "+/-" may be surrounded by zero or more spaces. For example, <code>doubleerr</code> parses the pair of successive minus signs in 4.21572+/-1.90059E-02 correctly.
bool	accepts 0/1 or true/false
optional	patterns until <code>&lt;/optional&gt;</code> are optional. Nestable.
raw text	Must match exactly, except for case
word	Matches a single word (contiguous non-whitespace)
regex	Matches a Unix-style regular expression, supplied as: <code>&lt;regex name="pattern-name" pattern="regular-expression" /&gt;</code>
parameter	Matches a line of values associated with an XSpec parameter. This is treated as a special case because some of these parameters are optional. Described in detail below.
filename	Matches a filename that may include spaces. <code>&lt;filetype name="file" pattern="*.pha" /&gt;</code> matches <code>Spectrum File 4.pha</code> as well as <code>specfile4.pha</code> . If no pattern attribute, picks up only a single word filename.
w	Matches one or more spaces or tabs (hard returns are ignored)
wo	("Whitespace optional")—matches zero or more spaces or tabs (hard returns are ignored)

Example:

```
<block name="specInfo2">
Net count rate (cts/s) for Spectrum:<int name="spectrum" /> <double name="countRate" /> +/-
<double
name="variance" /><optional> (<double name="totalFlux" /></optional>
```

matches:

Net count rate (cts/s) for Spectrum:1 3.783e+00 +/- 1.367e-01

The match occurs despite a missing total flux. `specInfo2.spectrum` will return the spectrum number, an integer. `specInfo2.countRate` contains the count rate, and `specInfo2.variance` contains the variance. `specInfo2.totalFlux` defaults to 0. Test for an optional parameter with `exists("specInfo2.variance")`. If the block is not named, variable values are retrieved without block specifiers. For example, `countRate` retrieves the count rate. While this is legal, it can lead to name conflicts. Since variables are scoped to a particular command, such conflicts are limited to a single command. Some optional output, such as error messages, contain no variables of interest, but it is interesting to know whether the message occurred.

```
<block name="fileNotFound" repeat="0+">>
***Error: File <word> not found.
</block>
```

You can test for the presence of this error by testing the block name with `exists("fileNotFound")`. Alternatively, you can assign a constant:

```
<block repeat="0+">>
***Error: File <word> not found.<bool fileNotFound="true" />
</block>
```

In this case, the variable `fileNotFound` defaults to false if the pattern is not found in the output, but if it is present, it is assigned the value true. A constant does not correspond to anything in the output, it is simply an assignment made if the pattern occurs. A constant assigned within an optional block is assigned only if the optional block occurs.

The block may specify the number of times the block is expected to repeat. If this is not supplied, it defaults to 1. For example:

```
<block name="data[0+]">>
```

means the block is optional, and may occur any number of times. In general, a repeat count of  $n$  specifies  $n$  occurrences, and a trailing plus sign means “ $n$  or more occurrences.” `command->results("data", 3)` retrieves the fourth matching result.

If multiple patterns match an output line, the first pattern listed will match. Sometimes output contains several related consecutive lines. You can match these with a single block by embedding “\n”s between lines.

```
<block name="dataGroup">
Assigned to Data Group <int name="dataGroup" /> and Plot Group <int name="plotGroup"
/>\n Noticed Channels: <int name="startChannel" />-<int name="endChannel" />
</block>
```

matches:

```
Assigned to Data Group 1 and Plot Group 1
Noticed Channels: 1-125
```

When output is parsed, each output line or group of matching lines is marked as matched when a pattern is found to match it. If any unmatched lines remain, the user is notified that the command may have unrecognized output.

### 11.4.1 XSpec Parameters

Some XSpec commands, notably the `fit` command, produce a list of new parameter settings, like this:

```
=====
Model phabs*powerlaw Source No.: 1 Active/On
Model Model Component Parameter Unit Value
par comp
  1 1 phabs nH 10^22 4.21572 +/- 1.90059E-02
```

2	2	powerlaw	PhoIndex	2.75901	+/-	6.20650E-03
3	2	powerlaw	norm	19.0457	+/-	0.208843

The three lines near the bottom correspond to three model parameters. These cannot be parsed by the pattern matcher because the Unit column may be empty, and the pattern matcher is based on finding tokens, not positional offsets. While positional support could be added, it is error-prone because column widths could change. So instead, the three parameter lines are matched by this block:

```
<block name="newparams[1+]">
<parameter />
</block>
```

The `parameter` pattern matches an entire parameter line if:

- The line contains seven or eight tokens
- The next-to-last token is a "+/-"
- The first two tokens are integers
- The value and error fields contain floating point values

After a match, the contents of the parsed line is returned as a structure with fields `modelpar`, `modelcomp`, `model`, `parameter`, `units`, and `value`, respectively. The error is part of the `value` field. If the units column was empty, `units` is empty.

## 11.4.2 Regular Expressions

Regular expressions are conventional Unix-style expressions that provide finer-grained control over matching. Complete documentation for these patterns is available for Qt's `QRegExp` class, upon which this is implemented.

```
<regexp name="fit" pattern="[A-Za-z]*" />
```

matches a word composed exclusively of letters, no numbers or symbols. As with other patterns, a regexp is named only if the characters that match the pattern are needed by the program.

## 11.4.3 Arrays

A series of consecutive data of the same type may be matched with an array:

```
6.022e23 2.71828 3.14159 channel 1
```

could be matched with:

```
<p class="code">
<double name="i1" /> <double name="i2" /> <double name="i3" /> channel <int name="channel" />
```

or with:

```
<p class="code">
<double name="items[3]" /> channel <int name="channel" />
```

or with:

```
<p class="code">
<double name="items[]" /> channel <int name="channel" />
```

The first match expects 3 elements and assigns them to `i1`, `i2` and `i3`. The second assigns them to array `items`. The last of these also assigns to `items`, and matches any number of consecutive doubles on the same line.

A 2D array like this:

```
324 1110 24 388 901
9300 8121 711 991 12
```

is matched by:

```
<block name="array[1+]">
<int name="col[]" />
</block>
```

array[1+] matches one or more lines of array output, and variable col matches all the elements in each row. Matching is terminated by a line that starts with a non-integer. The array syntax is legal for int, posint, bool, double, doubleint, and word.

ATSAL accesses arrays with optional arguments. Instead of

```
command->results("array.col")
```

you supply a row and column:

```
command->results("array.col", 1, 2)
```

accesses row 1, column 2. Row and column numbering is zero-based, and default to 0, 0. One way to use arrays is to skip words:

```
<word name="words[2]" />
```

skips two words. If you don't want to examine the words, omit the variable name:

```
<word name="[2]" />
```

also means "skip 2 words."

## 11.4.4 Skipping Lines

A block with the skipUntil option set causes all unmatched lines in the input to be skipped until the pattern is matched. This provides a way to skip over blocks of highly variable data, if there is nothing that needs to be extracted from the data. For example:

```
<block skipUntil="">
_____
</block>
```

Skips over the echoed parameter inputs for a model command:

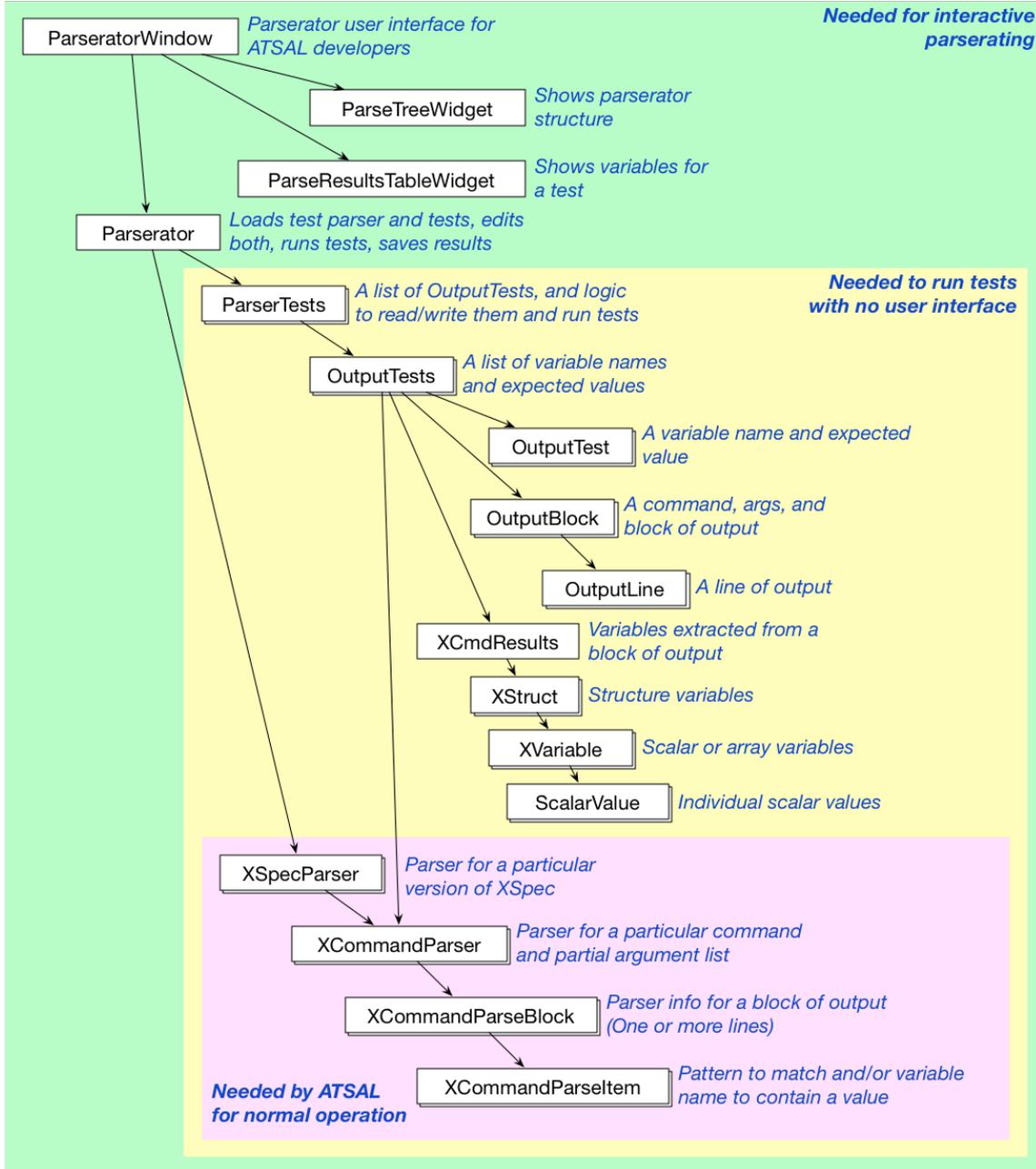
```
=====
Model bbody<1> Source No.: 1 Active/On
Model Model Component Parameter Unit Value
par comp
1 1 bbody kT keV 3.00000 +/- 0.0
2 1 bbody norm 1.00000 +/- 0.0
_____
```

ATSAL already knows the parameter settings, so this information is superfluous. The next block will start processing after the pattern detected by the skipUntil block.

## 11.4.5 The Parserator

ATSAL's "parserator" is an interactive tool, hidden in ATSAL itself, to assist in the development and testing of parsers for XSPEC (and, potentially, for other servers). The classes are shown below. The parserator user interface is shown in the following section.

## Parser Classes Ownership Graph



Here is the parserator user interface.

Parserator

Left to right: Add parser, Add test, What's a "parserator?," Compile and run tests, Save results, Delete item

Parsers and Tests		Parser
# ...	No tests	<command name="fit ...">
▼ data ...	Pass	<block>
data s54405.pha	Pass	<w />Parameters
▼ fit ...	Fail	</block>
fit 100	Fail	<block>
fit 30	Fail	Chi-Squared<w /> beta /N<w /><words />
model ...	No tests	</block>
puts ...	No tests	<block name="parameters" repeat="1+">
renorm	No tests	<double name="cols[]" />
set atsal_temp \$xspec_tclout	No tests	</block>
setplot ...	No tests	<block>
source dumpplot.tcl	No tests	<regexp pattern="[= +]" />\n
tclout plot ...	No tests	
		<b>Test Case</b>
		-----
		Covariance Matrix
		1 2
		2.877e-04 9.508e-09
		9.508e-09 2.269e-11
		-----
		Model bbody<1> Source No.: 1 Active/On
		Model Model Component Parameter Unit Value
		par comp
		1 1 bbody kT keV 0.890425 +/- 1.69607E-02
		2 1 bbody norm 2.78587E-04 +/- 4.76334E-06
		-----
		Fit statistic : Chi-Squared = 187.10 using 85 PHA bins.
		Test statistic : Chi-Squared = 187.10 using 85 PHA bins.
		Reduced chi-squared = 2.2542 for 83 degrees of freedom
		Null hypothesis probability = 5.284338e-10
		-----
		<b>Variables</b>
		Variable Expected Actual Ignore
		results.Fit $\chi^2$ 187.10 187.10 <input type="checkbox"/> Ignore
		results.Fit bins 85 85 <input type="checkbox"/> Ignore
		results.Reduced $\chi^2$ 2.2542 2.2542 <input type="checkbox"/> Ignore
		results.Degrees of freedom 83 83 <input type="checkbox"/> Ignore
		results.H0 5.284338e-10 <input type="checkbox"/> Ignore
		variancesAndPAs.axes 2 <input type="checkbox"/> Ignore

The variables section includes a special variable, "command output," whose actual value is `true` if the command produced some output, or `false` if not. If an XSPEC command should produce no output, set the expected value to false to verify this.

## 11.5 XSPEC Commands

This section lists XSPEC command documentation taken directly from the XSPEC 12 docs. At the end of each description, or occasionally interspersed, are **annotations on how they are (or will be) incorporated into ATSAAL, shown in this "remarks" style**. Alongside each command name in the table of contents is a parenthetical pair indicating my current estimate of priority and implementation difficulty. A developer should be able to consult this list to understand the mapping between any XSpec command and its ATSAAL support, and to see progress in implementation.

### 11.5.1 Description of Syntax

The individual commands are treated in alphabetical order in the following section. The novice would be well-served by reading the treatments of the `data`, `model`, `newpar`, and `fit` commands, in that order, then the other commands as needed. The write-up for each command includes a brief description of the purpose, an outline of the correct syntax, a more detailed discussion of the command assumptions and purpose, and a series of examples. Some commands have one or more subcommands that are similarly described following the command.

In the command description, the syntax uses the following conventions.

```
<arg>           ! an argument to the command
<arg c> ::= <arg a>   ! defines <arg c> as <arg a> followed by <arg b>
<arg b>
```

`<arg>...`                   ! a repeated string of arguments of the same type  
`[<arg>]`                       ! is an optional argument.  
`{ <arg a> | <arg b> }`       ! indicates a choice between an argument of type `<arg a>` or `<arg b>`

Exceptional responses to the command prompt are :

an empty line   - Nothing performed, prompt repeated  
/               - Any remaining arguments will have the values given on the last invocation of the command  
<EOF>           - same as quit  
                  otherwise use /\*  
? (or anything else)   - Write a list of the commands

## 11.5.2 Control Commands

### 11.5.2.1.1 autosave: set frequency of saving commands

Set or disable autosave, which saves the XSPEC environment to a file periodically.

**autosave** `<option>`

where `<option>` is either `off` or a non-zero positive integer `N`. If the option is `off`, then auto-saving is disabled. If the option is `N`, the XSPEC environment is saved every `N` commands. The saving of the environment is equivalent to the command

```
XSPEC12>save all xautosav.xcm .,
```

i.e. both the file and model information is saved to the file `xautosav.xcm`, placed in the directory `~/ .xspec/cache`. Thus in case of an unexpected crash, the state of XSPEC before the crash can be restored by running `@xautosav.xcm`. The default value for the auto-save option is 1.

Not needed in ATSAI; all work is saved with the notebook.

### 11.5.2.1.2 chatter: set verbosity level

Control the verbosity of XSPEC.

**chatter** `<chatter level>` `<log chatter>`

where `<chatter level>` and `<log chatter>` are integer values. The initial value for each argument is 10. Higher values will encourage XSPEC to tell the user more, lower values tell the user less, or make XSPEC “quieter.” `<chatter level>` applies to the terminal output, while `<log chatter>` controls the verbosity in the log file. Currently, the maximum chattiness is 25. Values below five should be avoided, as they tend to make XSPEC far too obscure. Some commands may temporarily modify the chattiness, such as the **error** command. A chattiness of 25 will generate a lot of debug output.

**Examples:**

```
XSPEC12> chatter 10
// Set the terminal chattiness to 10, same as the initial value.
XSPEC12> chatter ,0
// Set the chattiness for the log file to very low.
// This setting essentially disables the log file output.
XSPEC12> chatter 5
// Make XSPEC very quiet.
XSPEC12> chatter 10 25
// Restore the terminal chattiness to the initial level,
```

```
// while in the log file XSPEC will tell all
// (particularly when new data files are read in)
```

A user preference in ATSAAL, along with enabling logging. This would usually be off, but might prove useful for debugging ATSAAL.

### 11.5.2.1.3 exit, quit: exit program

The command to end the current XSPEC run.

#### **exit**

After an `exit`, the current plot files are closed. An `<EOF>` will have an identical result.

Quitting ATSAAL shuts down any XSPECs and offers to save any intermediate results to the notebook.

### 11.5.2.1.4 help: display manual or help for a specific command/theoretical model component

Obtain help on the XSPEC commands, their syntax, and examples of their use.

**help** [`<topic list>`]

On the first invocation of the help command, an instance of a pdf file reader (by default Adobe Acrobat Reader) is started (a shortly delay may ensue), or the XSPEC manual is accessed online. Please see the subsection “Customizing XSPEC” in the XSPEC Overview section for details on how to control this behavior. The Acrobat reader must be in the user’s path. If this default is used, then subsequent calls to help will use this instance to display other help pages. help without arguments displays the XSPEC manual, with a bookmark index that allows random access to the help system, or in the online mode will open to the XSPEC manual homepage.

The design allows for users to add help files for local models and scripts to the help system if they are placed in the help search path.

#### **Examples:**

```
XSPEC12> help
// Show the entire manual.
XSPEC12> help fit
// Go to the help text for the fit command.
XSPEC12> help model pow
// Go to the help text for the powerlaw model. (Entering just “XSPEC12> model” will
// produce a scrolled-text list of all available model components.)
XSPEC12> help appendices
// Show the manual appendices (which document the user interface, the Cash statistic,
// how to add models to XSPEC, a summary of PLT commands, and associated FTOOLS and
// other programs for manipulating data).
XSPEC12>help appendix local
// Show the appendix describing how to add local models
```

Help also displays the following information as scrolling text:

```
XSPEC12> help ?
// Show a list of all available commands.
XSPEC12> help ??
// Show a brief summary and usage syntax of all available commands.
XSPEC12> <command> ?
// Show brief summary and syntax of <command>.
```

In ATSAAL, this help will be rearranged and built right into ATSAAL. In addition to being able to browse the help facility via the Nexus window, many options in the user interface will have a “?” icon that brings up the help system on a particular topic.

### 11.5.2.1.5 log: log the session output

Open a log file.

```
log [STAMP] <log file>
```

where <log file> is the name of the file to be opened (default extension is `.log`). If no arguments are on the line, then the default file name is `xspec.log`. If <log file> matches the string `none`, then the current log file is closed. If the string `STAMP` is given as an argument then the log filename will include a date and time stamp. If <log file> has no suffix then the stamp is appended to the name and a `.log` suffix added. To change the chattiness level for the log file (i.e. the amount of information written to the log file) use the `chatter` command. The default chatter level for the log file is 10.

#### Examples:

```
XSPEC12> log
// Turn on the log file (default xspec.log).}
XSPEC12> log none
// Close the log file.
XSPEC12> log mylog
// Open the log file (mylog.log)
XSPEC12> chatter ,, 12
// Set the log file chattiness to 12.
```

In ATXSAL, this will be a user preference, usually defaulted, but useful to ATXSAL developers.

### 11.5.2.1.6 parallel: enable parallel processing for particular tasks in XSPEC

```
parallel <task> <max num of processes>
```

where <task> is currently limited to **leven**, **error**, or **steppar**. For best results, it is recommended that you set <max num of processes> to the number of CPU cores on your machine. Set <max num processes> back to 1 to turn parallel processing off for the particular task. To display current settings, type `parallel` with no arguments.

The **leven** option will spawn up to <max num> processes during the Levenberg-Marquardt fitting, specifically to perform the  $N$  independent calculations of the parameter first-order partial derivatives ( $N$  being the number of variable fit parameters). [This will not apply if the `USE_NUMERICAL_DIFFERENTIATION` variable in the user's `xspec.init` file is changed from the default 'false' to 'true'.]

The speed-up that one can expect is highly dependent upon the model in use. For simpler models with quick calculation times, you will probably see little to no speed gain with **parallel leven**. But with multi-core CPUs, gains should be quite noticeable when the model calculation consumes a large fraction of the overall fitting time. For example, with fits using the time-intensive **sedov** model on a 4-core machine, we've typically seen about a 40% reduction in fit time compared with the single processing case.

The **error** option is for running parallel computations within XSPEC's **error** command. This enables the error calculations for multiple parameters to be performed simultaneously. The speed-up here should simply be proportional to the number of cores available. However for cases where complications are reported (such as a new minimum found, or a non-monotonicity in the statistic space), it is recommended that you perform the error calculations in standard single-process mode.

When the **steppar** option is set, XSPEC will divide the  $N$ -dimensional **steppar** grid into <max num> sections of equal size, and spawn a separate process for calculating each section.

If both **parallel leven** and **error** or **steppar** are in use, XSPEC will temporarily disable the lower-level **leven** parallelization when running the higher-level parallel **error** or **steppar** command calculations.

#### Examples:

```
XSPEC12> model cflow
```

```

// Using a model with 5 variable fit parameters.
XSPEC12> parallel leven 4
XSPEC12> fit
// Calculations for the 5 parameters will be divided amongst
// 4 processes during the fit.

XSPEC12> parallel leven 1
// Restores single-process calculation to the Levenberg-Marquardt algorithm.

XSPEC12> parallel error 3
// Allow up to 3 simultaneous 'error' parameter calculations to be performed in parallel.

XSPEC12> error 2 3 6
// Perform error calculations on parameters 2, 3, and 6 in parallel.

XSPEC12> parallel steppar 4
// The following 20x30 steppar grid will be split amongst 4 parallel processes.

XSPEC12> steppar 1 10. 11. 20 2 .5 .8 30
// Display current settings:

XSPEC12> parallel
Maximum number of parallel processes:
  error: 3
  leven: 1
  steppar: 4

```

In ATXSAL, in principle, the multiple core requests need to be taken into consideration in any load-leveling. But I don't yet understand how to determine how many cores these algorithms "prefer." Is there a point of diminishing returns, or will all of these algorithms consume all available cores? If they will take full advantage of all the cores, it is probably best to assign all cores to a single algorithm during refresh, while keeping the other XSpecs idle. How common is it to run these algorithms? Is it worth developing moderately sophisticated load-leveling?

### 11.5.2.1.7 query: set a default answer for prompts in scripts

Switch on/off the continue fitting questions.

**query** <option>

where <option> is yes, no, or on. If on, then the continue fitting question in **fit**, **steppar**, and **error** will be asked when the number of trials is exceeded. Also, when the number of trials to find the error is exceeded a question will be asked. For either of the other two options the questions will not be asked but the answer will be assumed to be yes or no depending on <option>. To ensure that fitting continues without any questions being asked use the command

```
XSPEC12> query yes
```

In ATXSAL, this should be set to no. Instead, some sort of running count should be provided, so the user can decide when to abort something that has been running for too long. Currently ATXSAL is designed to abort all simultaneous activities when a refresh is aborted. Looks like it will be important to be able to abort individual processes too.

### 11.5.2.1.8 save: save the current session commands

Save aspects of the current state to a command file.

**save** <option> <filename>

If no <filename> is given, then the file `savexspec.xcm` is created. If you don't give the extension to the file name the default is `.xcm`. The values of <option> allowed are `model`, `files`, and `all`. The `model` option writes out commands to recreate the current model and parameter values; the `files` option writes out commands to read-in the current spectra, and the `all` option does both. The default option is `model`. To recover the saved context use the command

```
XSPEC12>@filename
```

### Examples:

```
XSPEC12> save model fname
// Write out model commands to the file fname.xcm

XSPEC12> save
// Same as above, but save into file savexspec.xcm.

XSPEC12> save files fname
// Write out data file commands.
```

Not needed in ATSA. Saving the notebook saves both actions to be performed and the last computed results.

#### 11.5.2.1.8.1 script: write commands to a script file

Open a script file.

**script** <script file>

where <script file> is the name of the file to be opened, defaulting to `xspec.xcm` or an extension of `.xcm`. If the filename is none, the current script file is closed. The script file saves all commands that are input. This command is useful for users who use the same set of commands repeatedly. Once a script file is written and saved, the user then can re-run the same set of commands on other data by

```
XSPEC12> source <script file>
```

### Examples:

```
XSPEC12> script
// Turn on the script file (default xspec.xcm)
XSPEC12> script none
// Close the script file.
XSPEC12> script myscript
// Open the script file (myscript.xcm)
```

Not needed in ATSA. Actions are implicitly saved.

#### 11.5.2.1.9 show: output current program state

List selected information to the user's terminal (and the log file, if open).

**show** [<selection>]

where <selection> determines the information to be printed. If omitted, it is the information last asked for. Initially, the default selection is `all`. (Note: to better integrate the usage of OGIP type-II files, much of the information given by `show files` in previous versions is now displayed by `show data`.)

Selections are:

```
XSPEC12> show abund
// show current solar abundance table

XSPEC12> show all
// All the information

XSPEC12> show allfile
```

```

// All file information = files + noticed + rates
XSPEC12> show control
// XSPEC control information

XSPEC12> show data
// File names, associated coefficients, and net count rates, displayed in order of spectrum
number.
// For higher chatter, also displays grouping map.

XSPEC12> show free
// Free parameters

XSPEC12> show files
// Equivalent to "show data" but displayed in order of file name.

XSPEC12> show fit
// Fit information

XSPEC12> show model
// The model specification

XSPEC12> show noticed
// Channel ranges noticed for each file.

XSPEC12> show parameters
// All current parameter values (including gain parameters, if any).

XSPEC12> show parameters <par range>
// Show subset of all model parameters given by <par range>,
// e.g. show parameters 1,3,5-8

XSPEC12> show pha
// Current data, error and model values for each channel.

XSPEC12> show plot
// Current plot settings from setplot command, includes rebinning info.

XSPEC12> show rates
// Folded model, correction rates for each file.

XSPEC12> show response
// Show responses loaded

XSPEC12> show rparameters
// All current gain (response) parameters

XSPEC12> show rparameters <par range>
// Show subset of all gain (response) parameters

XSPEC12> show xsect
// Show description of cross-section table

```

In ATSAI, program state is shown in context by the user interface, so this command is not needed.

### 11.5.2.1.10 syscall: execute a shell command

Execute command in a shell.

**syscall** [<shell command>]

This command executes its arguments by passing them to the user's current shell for execution. Thus file name globbing (*i.e.* wildcard expansion) is performed on the command before execution. This is in contrast to the `exec` command, which executes commands directly, without first passing them on to a shell.

If no arguments are given, then the command will start an interactive subshell.

This command is not needed in ATSAI.

### 11.5.2.1.11 tclout: create tcl variables from current state

Write internal xspec data to a tcl variable. This facility allows the manipulation of xspec data by tcl scripts, so that one can, for example, extract data from xspec runs and store in output files, format xspec output data as desired, use independent plotting software, etc.

**tclout** <option> [<par1>] [<par2>] [<par3>]>

tclout creates the tcl variable \$xspec\_tclout, which can then of course be set to any named variable. The allowed values of <option> are:

?	Show the valid options. Does not set \$xspec_tclout.
areascal n <s b>	Writes a string of blank separated values giving the AREASCAL values for spectrum n. If no second argument is given or it is "s" then the values are from the source file, if "b" from the background file. <b>Probably these should be extracted directly from the FITS files.</b>
arf n	The auxiliary response filename(s) for spectrum n. <b>Information already in ATLAS.</b>
backgrnd n	Background filename for spectrum n. <b>Information already in ATLAS.</b>
backscal n <s b>	Same as areascal option but for BACKSCAL value. <b>Probably these should be extracted directly from the FITS files.</b>
chain best last proposal stat	The best option returns the parameter values corresponding to the smallest statistic value in the loaded chains. The last option returns the final set of parameter values in the loaded chains. The proposal option takes arguments distribution or matrix and returns the name or covariance matrix for the proposal distribution when using Metropolis-Hastings. The stat option returns the output of the last chain stat command. <b>All of this information would probably be retrieved automatically after execution of any chain command.</b>
chatter	Current xspec chatter level. <b>Information already in ATLAS.</b>
compinfo [<mod>:]n [<groupn>]	Name, first parameter number and number of parameters of model component n, belonging to model w/ optional name <mod> and optional datagroup <groupn>. <b>Information already in ATLAS.</b>
cosmo	Writes a blank separated string containing the Hubble constant (H0), the deceleration parameter (q0), and the cosmological constant (Lambdao). Note that if Lambdao is non-zero the Universe is assumed to be flat and the value of q0 should be ignored. <b>Information already in ATLAS.</b>
covariance [m, n]	Element (m,n) from the covariance matrix of the most recent fit. If no indices are specified, then entire covariance matrix is retrieved. <b>Should this be retrieved automatically after each fit? How is this data used?</b>
datagr [n]	Data group number for spectrum n. If no n is given, outputs the total number of data groups. <b>Information already in ATLAS.</b>
datasets	Number of datasets. <b>Information already in ATLAS.</b>
dof	Degrees of freedom in fit, and the number of channels. <b>This is either parsed out of the results of other commands or retrieved after each fit.</b>
energies [n]	Writes a string of blank separated values giving the energies for spectrum n on which the model is calculated. If n is not specified or is 0, it will output the energies of the default dummy response matrix. <b>I think this is redundant with the plot data options. If not, could it be taken directly from the FITS files?</b>
eqwidth n [errsims]	Last equivalent width calculated for spectrum n. If "errsims" keyword is supplied, this will instead return the complete sorted array of values generated for the most recent eqwidth error simulation. <b>This is retrieved automatically after an eqwidth is run.</b>
error [<mod>:]n (for gain parameters use: error [<sourceNum>:]n	Writes last confidence region calculated for parameter n of model with optional name <mod>, and a string listing any errors that occurred during the calculation. The string comprises nine letters, the letter is T or F depending on whether or not an error occurred. The 9 possible errors are:  <ol style="list-style-type: none"> <li>1. new minimum found</li> <li>2. non-monotonicity detected</li> <li>3. minimization may have run into problem</li> <li>4. hit hard lower limit</li> <li>5. hit hard upper limit</li> </ol>

)	<p>6. parameter was frozen  7. search failed in -ve direction  8. search failed in +ve direction  9. reduced chi-squared too high  10. So for example an error string of "FFFFFFFFT" indicates the calculation failed because the reduced chi-squared was too high.</p> <p>This is retrieved and transformed into a readable message automatically when needed.</p>
expos n <s b>	Same as <code>areascal</code> option but for EXPOSURE value. Retrieved from FITS files.
filename n	Filename corresponding to spectrum n. Already in ATSA.
flux [n] [errsim]	Last model flux or luminosity calculated for spectrum n. Writes a string of 6 values: <code>val errLow errHigh</code> (in ergs/cm <sup>2</sup> ) <code>val errLow errHigh</code> (in photons). Error values are 0 if flux was not run with "err" option. If the "errsim" keyword is supplied, this will instead return the completed sorted array of values generated during the most recent flux error calculation. Retrieved automatically after a <b>lumin</b> command is issued.
ftest	The result of the last <b>ftest</b> command. Retrieved automatically after an <b>ftest</b> command.
gain [<sourceNum>:] <specNum> slope   offset	For gain fit parameters, value, delta, min, low, high, max for the slope or offset parameter belonging to the [ <code>&lt;sourceNum&gt;</code> : ] <code>&lt;specNum&gt;</code> response. For nonfit gain parameters, only the value is returned. Already present in ATSA.
goodness [sim]	The percentage of realizations from the last goodness command with statistic value less than the best-fit statistic using the data. If optional "sim" keyword is specified, this will instead give the full array of simulation values from the last goodness command. Retrieved automatically after a <b>goodness</b> command.
idline e d	Possible line IDs within the range [ <code>e-d</code> , <code>e+d</code> ]. Retrieved after an <b>identify</b> . The retrieved data are converted to selections on the plot.
ignore [<n>]	The range(s) of the ignored channels for spectrum <code>&lt;n&gt;</code> . Already in ATSA.
lumin [n] [errsim]	Last model luminosity calculated for spectrum n. Same output format as flux option, in units of 1.0×10 <sup>44</sup> erg/s. Retrieved automatically after a <b>lumin</b> command.
margin probability   [<modName>:] <parNum>	The probability option returns the probability column respectively from the most recent <b>margin</b> command. Otherwise, the parameter column indicated by <code>&lt;parNum&gt;</code> is returned. Note that for multi-dimensional margin the returned parameter column will contain duplicate values, in the same order as they originally appeared on the screen during the margin run. Retrieved automatically after a <b>margin</b> command.
model	Description of current model(s). Already in ATSA.
modcomp [<mod>]	Number of components in model (with optional model name). Already in ATSA.
modpar [<mod>]	Number of model parameters (with optional model name). Already in ATSA.
modval [<specNum> [<mod>]]	Write to Tel the last calculated model values for the specified spectrum and optional model name. Writes a string of blank separated numbers. Note that the output is in units of photons/cm <sup>2</sup> /s/bin. If I understand this correctly it is presently retrieved by dumping the associated plot values.
nchan [<n>]	Total number of channels in spectrum <code>n</code> (including ignored channels). Already in ATSA.
noticed [<n>]	Range (low, high) of noticed channels for spectrum <code>n</code> . Already in ATSA.
noticed energy [<n>]	The noticed energies for spectrum <code>n</code> . Already in ATSA.
nullhyp	When using chi-square for fits, this will retrieve the reported null hypothesis probability. Retrieved after each fit.
param [<mod>:]n	(value, delta, min, low, high, max) for model parameter <code>n</code> . Already in ATSA.
peakrsid n [lo, hi]	Energies and strengths of the peak residuals (+ve and -ve) for the spectrum <code>n</code> . Optional arguments <code>lo</code> , <code>hi</code> specify an energy range in which to search. I think these are retrieved when the user requests a residuals plot. Not sure I understand this though.
pinfo [<mod>:]n	Parameter name and unit for parameter <code>n</code> of model with optional name. Already in ATSA.
	Information on parameter linking for parameter <code>n</code> . This is in the form true/false (T or F) for

plink [<mod>:]n	linked/not linked, followed by the multiplicative factor and additive constants if linked. <a href="#">Already in ATSAAL.</a>
plot <option> <array> [<plot group n>]	Write a string of blank separated values for the array. <option> is one of the valid arguments for the <b>plot</b> or <b>iplot</b> commands. <array> is one of x, xerr, y, yerr, or model. xerr and yerr output the 1-sigma error bars generated for plots with errors. The model array is for the convolved model in data and ldata plots. For contour plots this command just dumps the steppar results. The command does not work for genetic plot options. <a href="#">Used to fetch all plot data for ATSAAL.</a>
plotgrp	Number of plot groups. <a href="#">Already in ATSAAL.</a>
query	The setting of the query option. <a href="#">Already in ATSAAL.</a>
rate <n   all>	Count rate, uncertainty and the model rate for the specified spectrum n, or for the sum over all spectra. <a href="#">Retrieved via tcl plot.</a>
rerror [<sourceNumber>:]n	Writes last confidence region calculated for response parameter n of model with optional source number, and a string listing any errors that occurred during the calculation. See the help above on the error option for a description of the string. <a href="#">Retrieved after operation is performed.</a>
response n	Response filename(s) for the spectrum n.
sigma [<modelName>:]n	The sigma uncertainty value for parameter n. If n is not a variable parameter or fit was unable to calculate sigma, -1.0 is returned. <a href="#">I imagine a "Sigmas" checkbox in the parameter editor pane. If enabled, after each fit, a tcl script would return the sigmas for all the parameters in a single transaction with XSpec, displaying them beside the parameters. If the values are frequently used, the checkbox could be omitted and the values always retrieved and displayed.</a>
simpars	Creates a list of parameter values by drawing from a multivariate Normal distribution based on the covariance matrix from the last fit. This is the same mechanism that is used to get the errors on fluxes and luminosities, and to run the goodness command. <a href="#">I need a better understanding of how these values are used.</a>
solab	Solar abundance table values. <a href="#">Already in ATSAAL.</a>
stat [test]	Value of statistic. If optional 'test' argument is given, this outputs the test statistic rather than the fit statistic. <a href="#">Retrieved when the corresponding operation is performed.</a>
statmethod [test]	The name of the fit stat method currently in use. If optional 'test' argument is given, this will give the name of the test stat method. <a href="#">Already in ATSAAL.</a>
steppar statistic   delstat   [<modelName>:] <parNum>	The statistic and delstat options return the statistic or delta-statistic column respectively from the most recent steppar run. Otherwise, the parameter column indicated by <parNum> is returned. Note that for multi-dimensional steppars the returned parameter column will contain duplicate values, in the same order as they originally appeared on the screen during the steppar run. <a href="#">Retrieved automatically after a steppar run.</a>
varpar	Number of variable fit parameters. <a href="#">Already in ATSAAL.</a>
version	The XSPEC version string. <a href="#">Already in ATSAAL.</a>
weight	Name of the current weighting function. <a href="#">Already in ATSAAL.</a>
xflt n	XFLT#### keywords for spectrum n. The first number written is the number of keywords and the rest are the keyword values. <a href="#">Retrieved from FITS files.</a>

### Examples:

```

XSPEC12>data file1
XSPEC12>model pha(po)
...
XSPEC12> fit
...
XSPEC12>tclout stat
XSPEC12>scan $xspec_tclout "%f" chistat
XSPEC12>tclout param 1
XSPEC12>scan $xspec_tclout "%f"par2
XSPEC12>tclout param 2
XSPEC12>scan $xspec_tclout "%f"par3
XSPEC12>tclout param 3

```

In this example, `scan` is a tcl command that does a formatted read of the variable `$xspec_tclout`. It reads the first floating point number into the variable given by the last argument on the line. This sequence creates a simple model, fits it, and then writes the  $\chi^2$  statistic and the three parameters to tcl variables `$chistat`, `$par1`, `$par2`, and `$par3`. These can now be manipulated in any way permitted by tcl. Examples of using **tclout** and **tcloutr** can be found in the `Xspec/src/scripts` directory.

This is already used by ATSAAL to retrieve some of XSpec's internal state. Almost all of this information is either (1) already present in ATSAAL; (2) retrieved automatically when the operation that generates the information is performed; or (3) retrieved (more quickly) from the FITS files. As a result, there is no need to provide an access path to this.

Accessing these values from Python will take some work. First, since there are multiple `xProxys`, the user must specify the proxy of interest, probably by referencing the spectrum and model numbers. Array data needs to be transformed properly into Python-style data formats. Physical unit information needs to be transformed into ATSAAL internal classes that can perform easy unit conversion. The Python calls need to gracefully handle requests for information that don't exist, because the operation that generates the information hasn't been enabled. So there is a substantial amount of work needed to support Python access.

A low-level back door to **tclout/tcloutr** should probably be available to Python as well. This would queue a particular request using one of these commands, and return the raw data for parsing by the user. This would be an escape for a case ATSAAL overlooked or hasn't yet supported. (One could argue that such a back door should be more general, allowing any XSpec command. But this would be highly likely to disrupt ATSAAL's assumptions about XSpec state, and would break down when a different XSpec instance was assigned.

#### 11.5.2.1.12 tcloutr: tclout with return value

```
tcloutr <option> [<par1>] [<par2>] [<par3>]>
```

**tcloutr** is identical to **tclout** except that it also provides what is stored in `$xspec_tclout` as a return value. Therefore it can be used in tcl scripts like this:

```
set var1 [tcloutr energies 1]
```

**tcloutr** may produce quite a lot of unwanted screen output, which can be avoided by using **tclout**.

See **tclout** for ATSAAL comments.

#### 11.5.2.1.13 time: print execution time

Get some information about the program run time.

##### **time**

The time command prints out elapses CPU time attributed to the user and to the system. Two output lines are given, one for user/system time since the time command was last called, and one for time elapsed since the program started.

This is a low priority. Such information would have to be collected on an `xProxy` basis, not an XSPEC process basis. If we support it, it would probably appear somewhere in the `xProxy` log window.

#### 11.5.2.1.14 undo: undo the previous command

Undo the effects of the previously entered xspec command.

##### **undo**

New for xspec version 12, the undo command will restore the state of the xspec session prior to the most recently entered command. The current implementation does not allow restoration to more than one command back, so calling undo repeatedly will have no effect. Also, a plot command cannot be undone.

No undo support in ATSal at this time. If implemented, it could not take advantage of XSpec's undo support, because backing up would disrupt the dependency tree analysis and because XSpec presently offers only one level of backup.

### 11.5.2.1.15 version: print the version string

#### version

**version** prints out the information about version number and build date and time (not current date, time) displayed when XSPEC is started.

This could be placed in the title area of the xProxy log pane. Low priority.

## 11.5.3 Data Commands

### 11.5.3.1.1 arf: change the efficiency file for a given response

Read in one or more auxiliary response files (ARF). An ARF gives area versus energy and is used to modify the response matrix for a spectrum. The file must be in the OGIP standard format.

**arf** [ <filespec>...]

where <filespec> =:: [[source #:]<spectrum num>] <filename>[{ranges}]... and where <spectrum num> is the spectrum number for the first filename specified, <spectrum num> plus one is the spectrum number for the next file (or next entry in {ranges} specifier for Type II multi-ARF files), and so on. <filename> is the name of the auxiliary response file to be used with the associated spectrum. The optional source number defaults to 1, and for ARFs stored in OGIP Type II files, {ranges} specifies the row numbers of the desired ARF(s). See the **data** command for allowed range specification.

If no <spectrum num> is given in the first <filespec> it is assumed to be 1. If no file specifications are entered, then none of the spectrum responses are modified. An error message is printed if the spectrum number is greater than the current number of spectra (as determined from the last use of the **data** command). A file name *none* indicates that no auxiliary response is to be used for that spectrum. If a file is not found or cannot be opened for input, then the user is prompted for a replacement auxiliary response file. An <EOF> at this point is equivalent to *none*. See the **data** command for ways to completely remove the dataset from consideration.

Note: The ARF command is currently not implemented for data formats which use multiple RMFs per spectrum, such as Integral/SPI data.

#### Examples:

It is assumed that there are currently three spectra:

```
XSPEC> arf a,b,c
// New files for the auxiliary response are given for all three spectra.
XSPEC> arf 2 none
// No auxiliary response will be used for the second spectrum.
XSPEC> arf ,d.fits
// d.fits becomes the auxiliary response for the second spectrum.
XSPEC> arf 2 e.fits{3-4}
// Rows 3 and 4 of multi-ARF file e.fits become the auxiliary responses for the second and
third spectra.
XSPEC> arf 2:1 f.fits
// f.fits becomes the auxiliary response for the second source of spectrum 1.
```

This is handled by ATSal's tags and tag groups. Automatically created tags can be customized as needed, and all downstream operations are applied against the altered tags.

### 11.5.3.1.2 backgrnd: change the background file for a given spectrum

Modify one or more of the files used in background subtraction.

**backgrnd** [<filespec>...]  
**backgrnd** <spectrum number> none

where <filespec> ::= [ <spectrum num>] <filename>... and where <filename> is the name of the PHA file to be used for background subtraction. The numbering scheme is as described for the **data** command, except that the <spectrum num> must have previously been loaded.

An error message is printed if <spectrum num> is greater than the current number of spectra (as determined from the last use of the data command. **backgrnd** <n> none indicates that no background subtraction is to be performed for that spectrum. If a file is not found or cannot be opened for input, then the user is prompted for a replacement background file (an <EOF> at this point is equivalent to **backgrnd** <spectrum number> none). The current ignore status for channels is not affected by the **bkgrnd** command. (See the **ignore** and **notice** commands). Finally, any grouping specification will be overridden by the grouping in the source spectral file so that the source and background are binned in the same way.

The format of the background file must match that of the spectrum file: for this purpose OGIP Type I and II are considered to be the same format.

For details of how to remove spectra see the **data** command documentation.

#### Examples:

Suppose there are currently three spectra. Then

```
XSPEC12> backgrnd a,b,c
// New files for background subtraction are given for all
// three spectra.
XSPEC12> backgrnd 2 none
// No background subtraction will be done for the second spectrum.
XSPEC12> backgrnd ,d
// d.pha becomes the background for the second spectrum.
XSPEC12> backgrnd 2 e{4-5}
// Rows 4 and 5 of Type II file e.pha become the background for
// the second and third spectrum respectively.
```

This is handled by ATLAS's tags and tag groups. Automatically created tags can be customized as needed, and all downstream operations are applied against the altered tags.

### 11.5.3.1.3 corfile: change the correction file for a given spectrum

Reset the files used for background correction.

**corfile** [<filespec>...]

where <filespec> is the same as for the **backgrnd** command. The correction file can be associated with a spectrum to further adjust the count rates. It is a PHA file whose count rate is multiplied by the current associated correction norm (see the **cornorm** and **recornrm** command) and then subtracted from the input uncorrected data. The correction norm is not changed by running the **corfile** command. Default values for the correction file and norm are included in the data PHA file. Unlike the background file, the correction data does NOT contribute to the measurement error. A file name of none is equivalent to no correction file used. If an input file cannot be opened or found, an error message is printed and the user prompted for a replacement. As with the **backgrnd** command, the correction file is checked against the associated spectrum for number of channels, grouping status, and detector ID. The current ignore status for channels is not affected by the **corfile** command. Note that correction files have the same format as the PHA files used by the **data** command.

#### Examples:

It is assumed that there are currently three spectra:

```
XSPEC12> corfile a,b,c
```

```
// New correction files are used for all three spectra.
XSPEC12> corfile 2 none
// No correction will be done for the second spectrum.
XSPEC12> corfile ,d
// The 2nd file now uses d.pha as its correction.
XSPEC12> corfile 2 e{4-5}
// Rows 4 and 5 of Type II file e.pha become the correction files for the second
// and third spectrum respectively.
```

This is handled by ATSA's tags and tag groups. Automatically created tags can be customized as needed, and all downstream operations are applied against the altered tags.

### 11.5.3.1.4 cornorm: change the normalization of the correction file

Reset the normalization used in correcting the background.

```
cornorm [[ <spectrum range>...] [ <cornorm>]] ...
```

where <spectrum range> ::= <first spectrum no.> – <last spectrum no.> is a range of spectra to which the correction is to be applied and <cornorm> is the value to be used for the normalization. A decimal point (.) is used to distinguish a correction norm from a single spectrum <spectrum range>. If no correction norm is given, then the last value input is used (the initial value is 1). If no range is given, then the last single range input is modified. (See the **corfile** command.)

#### Examples:

Assume that there are four spectra, all with associated correction files already defined, either by default in their PHA file, or explicitly by using the **corfile** command.

```
XSPEC12> cornorm 1-4 1.
// The correction norm for all four is set to 1.0
XSPEC12> cornorm 0. 1-2 0.3
// The correction norm for the last input range (which was 1-4)
// is set to 0., then files 1 and 2 are reset to 0.3.
XSPEC12> cornorm 4
// file 4 has the correction also set to 0.3.
XSPEC12> cornorm 1 4 -.3
// files 1 and 4 are set to -.3.
XSPEC12> cornorm .7
// file 4 (as the last input single range) is set to 0.7.
```

This looks like an extension to file tags and tag groups. Might be a **Normalization...** button with a value beside it. The button would allow default (1.0 or inherit from tag group), or a value. In a tag group, it could just be an edit box with a 1. in it by default.

### 11.5.3.1.5 data: read data, background, and responses

Input one or more spectra, together with their associated (background, response) files.

```
data <file spec1>[,...] [ <file spec2>...] [ / ]
data none
data <spectrum #> none
```

where the file specification is

```
<filespec> ::= [[ <data group #:>] <spectrum #>] <filename>[ {ranges} ]
```

If a particular file is not found or cannot be opened for input for some reason, then the user is prompted for a replacement file name. An <EOF> at this point is equivalent to typing none. The default extension for all files is .pha, so all other extensions, (e.g. .fak) must be entered explicitly. The default directory is the current user directory when XSPEC is invoked. When a new file is input, by default all its PHA channels are considered good channels for fitting and plotting purposes (see the **ignore** and **notice** commands).

XSPEC's "native" data format is the OGIP standard. The standard specifies the representation of spectrum and all related datasets. XSPEC12 is explicitly designed to be able to work with other data formats as required: for example, the Integral/SPI spectral data format, although based on OGIP TypeII, deviates slightly. This was necessary because 3 response/arf pairs are required per spectrum. XSPEC12 has the ability to specify how response and other data are stored on disk, composed, and combined within the spectral fitting problem by adding new data modules at run-time. In XSPEC12, unlike XSPEC11, the channels that are ignored are a property of the spectrum, and therefore must be reset when the spectrum is replaced by another.

If the file contains multiple spectra, such as an OGIP Type II PHA file, then the desired spectrum can be specified by appending {*ranges*} to the end of the filename, where *n* is the row number of the spectrum in the file.

XSPEC12 allows any combination of multiple ranges in the parentheses delimited by commas. The wildcard characters \* and \*\* may also be used. A \*\* on either side of a hyphen indicates either the first or last row in the file, based on whether it is to the left or right of the hyphen. (If a \* is entered on the left or right side of a hyphen, it is substituted by the most recently entered left or right value respectively.) All rows in the file may be selected simply with a single \* or \*\* between the brackets with no hyphen.

### Examples:

```
XSPEC12> data pha2data{1,3,5-8,14-26,75-**}  
// In addition to the various specified rows between 1 and 26,  
// also load rows 75 through the end of the file.  
XSPEC12> data pha2data{*}  
// Select all rows in the file.
```

For files with multiple spectra the data may either specify a header keyword specifying the response, auxiliary response, background and correction files, or these may be string-valued columns specifying a different filename per row.

Consult the <http://heasarc.gsfc.nasa.gov/docs/software/ftools> package documentation for details of how to modify the file.

The individual spectral data files are created outside of XSPEC by detector-specific software. They are organized as XSPEC data files, but often referred to as PHA files. Whatever its format, the PHA file contains such information as integration time, detector effective area, and a scaling factor (BACKSCAL in the OGIP standard) that estimates the expected size of the internal background. The data file also contains the names of the default files to be used for background subtraction and for the detector sensitivity versus incident photon energy (the response and arf files). A data file has the total observed counts for a number of channels and a factor for the size of any systematic error. Each channel is converted to a count rate per unit area (assumed cm<sup>-2</sup>). The default background file is used for background subtraction. An error term is calculated using Poisson statistics and any systematic error indicated in the file. (For OGIP files, any FITS NULL values will be converted to the value 1.E-32. This should have no practical effect because any channels with NULL values will presumably be marked as bad or otherwise ignored.)

#### 11.5.3.1.5.1 Spectrum numbering

Multiple `filespec` clauses can be input on a single data command, or also on multiple data command. Within XSPEC, each set of data is referred to by its associated spectrum number. `<spectrum #>`, as determined by the following rules. For convenience, we denote the number of spectra that have been previously read in by data command as  $N_s$ .

Spectra in XSPEC are numbered sequentially from 1.

If no spectrum number is specified by the user, the spectrum in the first filename specified is assigned to 1. If spectra have already been loaded at this point, they will be replaced, deleted, or added to depending on the command. For example, if there are 3 spectra loaded ( $N_s = 3$ ) and the user types

```
XSPEC12> data multidatafile{1-2}
```

then spectra 1 and 2 will be replaced and 3 deleted. The command

```
XSPEC12> data multidatafile{1-4}
```

will replace all three spectra and add the fourth.

If the user specifies a “load point,” i.e. the first spectrum number to be created by the new command:

```
XSPEC12> data 3 multidatafile{1-4}
```

then that load point may not exceed  $N_s + 1$ . If it does, XSPEC will correct the number and issue a warning.

A skipped-over argument can be effected by a comma, for example

```
XSPEC12> data 3 spectrum1, , spectrum2
```

indicates that the spectrum for that position, as input in an earlier invocation of data, will continue to be used (in this example, spectrum 3 is replaced, 4 is left untouched, and 5 is either replaced or added. Any spectra with numbers great than 5 are removed.

If the filename input is none, that spectrum is removed, and so are any higher-number spectra unless none is terminated with a / character. For example:

```
XSPEC12> data 3 none
```

removes all spectra numbered 3 or higher,

```
XSPEC12>data 3 none/
```

removes only spectrum 3 and renumbers the rest.

The **data** command determines the current total number  $N_T$  of spectra: either  $N_T$  spectra are implied by the command line, or the highest spectrum number added (after XSPEC has made corrections as mentioned above) is  $N_T$ . This is true UNLESS a / character terminates the data command.

If the line is terminated by a slash (/), then the current number of spectra is the previous total number of datasets  $N_s$  or the number as determined from the command line, whichever is greater.

Both of these commands

```
XSPEC12> data
XSPEC12> data ?
```

print the one-line help summary.

### 11.5.3.1.5.2 Data groups

XSPEC allows the user to specify separate data groups for different spectra. Each data group has its own set of parameters for the defined model. These parameters can be either independent from data group to data group, or they can be linked across data groups using the standard XSPEC syntax (see the newpar command). This facility can be used for, say, analyzing extended sources.

Note that the data group number *precedes* the spectrum number: in the example

```
XSPEC12> data 2:3 spectrum4
```

which assumes that at least two spectra are already present, the data group number is 2 and the spectrum number is 3. XSPEC will not allow the data group number to exceed the spectrum number: for example

```
XSPEC12> data 3:2 spectrum4
```

is invalid. XSPEC will correct this and issue a warning.

### More examples:

```
XSPEC12> data a
// The file a.pha is read in as the first (and only) spectrum.
XSPEC12> data ,b
// b.pha becomes the second spectrum, the first spectrum is
// unmodified (i.e. it is still a.pha)
XSPEC12> data c 3 d,e,f
// c.pha replaces a.pha as the first spectrum;d.pha, e.pha, and
// f.pha provide the, third, fourth, and fifth spectra.
XSPEC12> data g/
// g.pha replaces c.pha as the first spectrum; the slash (/)
// indicates that the 2nd through the 5th spectra remain as before.
XSPEC12> data 2 none/
// the string none indicates that the 2nd spectrum (b.pha) is to be
// totally removed. The current total number of datasets thus becomes
// one less (4). The current spectra are g.pha,d.pha, e.pha, and f.pha.
XSPEC12> data h,,
// The current total number of spectra becomes 2, the current data
// sets are from h.pha and d.pha.
XSPEC12> data
// There is no change in the data status.
XSPEC12> data 1
// The number of spectra is set explicitly to one, that being from h.pha.
XSPEC12> data 1:1 a2:2 b 3:3 c
// Read a.PHA into data group 1, b.pha into data group 2, and c.pha
// into data group 3
XSPEC12> data 1:1 a 1:2 b 2:3 c
// Read a.pha and b.pha into data group 1, and c.pha into data group 2
XSPEC12> data a{3}
// Read the third spectrum in the file a.pha.
```

In ATLAS, this is done by creating a spectrum and selecting a tag. Note that selecting doesn't issue any actual commands. This happens when the user hits Refresh. Note that ATLAS does not yet have any mechanism for selecting individual spectra from a file containing multiple spectra. Such a mechanism will need to be added to the tag tool.

### 11.5.3.1.6 **diagrsp**: set a 'perfect' response for a spectrum

Diagonalize the current response matrix for ideal response.

#### **diagrsp**

This command diagonalizes the current response matrix. The response matrix is set so that the channel values are mapped directly into the corresponding energy ranges, based on the channel energies and energy response range of the current response matrix.

This command is very similar to running **dummyrsp** in mode 1, with two important differences. Unlike **dummyrsp**, usage of this command requires that an actual response is currently loaded. It takes its energy range and channel binning information from this currently loaded response rather than user input parameters. Secondly, this does not change the efficiency (i.e. effective area) as a function of energy stored for the current detector. Invoking this command will simulate a detector with perfect spectral resolution. If you wish to simulate a detector with perfect resolution *and* perfect efficiency, use the **dummyrsp** command.

The previous response matrices can be reimplemented with the **response** command, with no arguments. Any use of the **data** and **notice** commands will replace the dummy diagonal response with the correct set of matrices.

I don't understand this well enough to know how to implement it in ATLAS.

### 11.5.3.1.7 fakeit: simulate observations of theoretical models

Produce spectra with simulated data.

**fakeit** [nowrite] [<file spec>...]

where <file spec> ::= [ <file number>] <file name> [{ranges}]... is similar to the syntax used in the **backgrnd**, **corfile**, and **response** command. The **fakeit** command is used to create a number of spectrum files, where the current model is multiplied by the response curves and then added to a realization of any background. Statistical fluctuations can be included. The integration time and correction norm are requested for each file. The filenames input as command line arguments are used as background. The number of faked spectra produced is the maximum of the number of spectra currently loaded and the number of file specifications in the command line arguments. The special case `fakeit none` makes one fake spectrum for each spectrum loaded (or one fake spectrum if there are none loaded). See the examples below for a clearer description.

If **fakeit** is immediately followed by the `nowrite` specifier, no actual output files will be generated. In this case the fake spectra will exist just for the duration of the Xspec session (or until they are unloaded).

If a faked spectrum is based on a currently loaded spectrum, then by default the background, response, correction file, and numerical information are taken from the currently-defined data, unless a background file is specified on the command line in which case it becomes the background. The `fakeit none` case prompts for the `rmf` and `arf` filenames and sets the default numerical data to 1.0, except the correction norm, which is set to zero. If the output file is type II then the exposure time and correction scale factor will be the same for all spectra in the file.

For each output file, the user is prompted for an output file name. If a background file is in use then `fakeit` will also simulate a new background for each spectrum. Background files are given the same names as output spectrum files but with `_bkg` appended.

The simulated spectra automatically become the current data files. The ignore status is completely reset.

#### 11.5.3.1.7.1 Statistical Issues

The statistical fluctuations used to create the simulated spectra will depend on whether the current spectra have Poisson or Gaussian errors. If a spectrum file has a `STAT_ERR` column and the `POISSERR` keyword is set to false then `xspec` assumes Gaussian errors with sigma from the values in the column. Otherwise, errors are assumed to be Poisson based on the number of counts. Note that it is possible for the spectrum and background files to have different error types. For `fakeit` cases when there is no current file to use, Poisson errors are assumed.

#### 11.5.3.1.7.2 Type I vs. Type II Output

`fakeit` determines whether to place its fake spectra and background data into type I or type II files based on the following rules.

If fake spectra are based on currently loaded spectra then the output files will have the same format as those loaded. For example, assume 3 spectra are currently loaded, spectrum 1 from file `typeIdata.pha` and spectra 2 and 3 from file `typeIIdata.pha`. Then,

```
XSPEC12> fakeit
```

will produce 3 fake spectra in 2 output files with names prompted from the user. The first file will be type I, the second type II containing 2 spectra. The same is true for any background files produced.

If the user asks for more fake spectra to be created than the number of spectra currently loaded, for example by typing the following when the same 3 spectra above described are loaded:

```
XSPEC12> fakeit 5
```

then fake spectra 1-3 will be placed in the two files as before. For the additional fake spectra (4 and 5), **fakeit** uses the following rule: if any of the originally loaded spectra were in a type II file, then all of the additional fake spectra will be placed in 1 type II file. Otherwise, they will each be placed in a separate type I file. In this example, since a type II file was originally loaded (`typeIIData.pha`) when **fakeit** was called, spectra 4 and 5 will be placed together in a type II output file, in addition to the type I and type II files for the first 3 fake spectra.

If there are no currently loaded spectra all output files will be type I unless either of the following situations exist:

1. Any of the background files entered on the command line are type II, as indicated by row specifiers in brackets.
2. The first response file used clearly belongs to a format associated with type II data, such as SPI/Integral with its multiple RMF format (see section on SPI/Integral usage).

Overall, though the method of determining output format for additional spectra may seem quite complicated, it can be easily summed up: **fakeit** will place all additional spectra and backgrounds (i.e. those not based on already loaded data) in type I output files, unless it detects any evidence of type II file usage amongst the command line input, in which case it will produce type II output.

### 11.5.3.1.7.3 Note on Grouped Spectra

If an input spectrum has grouping information (i.e. a `GROUPING` column telling XSPEC how to bin up the data) then **fakeit** will simulate the number of counts in each of the grouped bins. However, the spectrum that is written out must have the ungrouped number of channels (and a copy of the `GROUPING` column from the original spectrum). The solution that XSPEC adopts is to place all the counts from a grouped bin in the first channel which goes to make up that bin. This is of no consequence for future uses of the simulated spectrum provided that the `GROUPING` column is not changed. So, in this case `grppha` or similar tools cannot be run on the simulated spectrum.

### 11.5.3.1.7.4 Note for SPI/Integral Format

Since the SPI/Integral format builds its responses from a combination of multiple RMFs and ARFs, it must use a different scheme than the OGIP type I and II formats for storing RMF and ARF file location information. This information is stored in a FITS extension, named “`RESPFILE_DB`”, added to the PHA file. Therefore, when **fakeit** prompts the user for the location of the response file, simply enter the name of a FITS file which contains a `RESPFILE_DB` extension pointing to the RMFs and ARFs to be applied. When prompted for an ARF name, enter nothing.

The prompts will only appear for the first spectrum in the data set, and the ARFs will be assigned row by row 1 to 1 with the spectra. For example, if no data is currently loaded, to create 3 fake SPI spectra from the RMFs and ARFs named in the `RESPFILE_DB` extension of the file `realSpiData.pha`:

```
XSPEC12> fakeit 3
// ...(various prompts will follow)...
For fake spectrum #1 response file is needed: realSpiData.pha
// ...and ancillary file: <Ret>
// ...(more fakeit prompts)...
```

This will create 3 fake spectra, each making use of the same RMFs/ARFs, spectrum 1 using the first row of the ARFs, spectrum 2 using the second etc.

### CAUTION – SPI/Integral

As currently implemented, the `RESPFILE_DB` method of storing ARF locations does not retain specific row information. The assumption is that the rows in the ARF correspond 1 to 1 with the rows in the spectral data extension. Therefore, much confusion can arise when the row numbers of the loaded spectra do not match that of the fake spectra. For example:

```
XSPEC12> data my_spi_data.pha{3-4}
// my_spi_data.pha contains a RESPPFILE_DB table pointing to arf1.fits,
// arf2.fit, arf3.fits.
// ...(fit to some model(s))...
XSPEC12> fakeit
```

This will produce two fake spectra generated from the model\*response operation, where the model has parameters based on a fit to the original spectra in rows 3 and 4 of `my_spi_data.pha`, which used ROWS 3 AND 4 of the 3 arf files for their own responses. However, the responses used above to generate the two fake spectra will use ROWS 1 AND 2 of the 3 arf files. This is necessary since the fake spectra will be placed in rows 1 and 2 of their fakeit output file.

### Examples:

#### 11.5.3.1.7.4.1 Type I files

```
XSPEC12> fakeit
```

This will produce 3 fake spectra each in its own type I output file, and the user will be prompted for the file names. The response file information will come from each of the original spectra. If any response information is invalid, the user will then be prompted. A fake background file will be produced for the second spectrum.

```
XSPEC12> fakeit 4
```

Produces 4 fake spectra, the first 3 created as in the previous example. The fourth will be created with no background spectrum, and this user is prompted for response information.

```
XSPEC12> fakeit backa,,none 4
```

Produces 4 fake spectra. For the first spectrum, a fake background file will be generated from the file `backa`. The second uses its own background file as before. The third fake spectrum will no longer use the response information from loaded spectrum 3, the user will be prompted instead, and its default numerical data will be reset to 1. The fourth spectrum will be created as in the previous example.

If no data is currently loaded:

```
XSPEC12> fakeit 2
```

Produces 2 fake spectra in separate type I files, unless the first user entered response file belongs to a format that is explicitly type II (i.e. SPI/Integral).

#### 11.5.3.1.7.4.2 Type II Files

Assume four spectra with no backgrounds have been loaded from one type II file:

```
XSPEC12> data original_type2_data.pha{5-8}
```

Then, after model(s) have been entered and a fit:

```
XSPEC12> fakeit
```

This will produce 4 fake spectra in rows 1 to 4 of one type II output file, with responses and arfs taken from the columns of `original_type2_data.pha`.

```
XSPEC12> fakeit ,,backb{1-3}
```

This produces 5 fake spectra in two type II output files, and 3 fake background spectra also placed in two type II output

files:

The first 4 fake spectra are placed in one output file since that is how the 4 spectra they were based on were originally organized. The default numerical data for this file are taken from the original spectra. Fake spectra 3 and 4 now have backgrounds, based on `backb{1}` and `backb{2}` respectively. These will generate 2 fake background spectra, placed in rows 3 and 4 of the first output fake background file. Rows 1 and 2 of this file will just consist of zeros since the first 2 spectra have no backgrounds.

The fifth fake spectrum will be placed in the second type II PHA file. Response and numerical data will not be based on the existing loaded spectra. A fake background will be generated from `backb{3}` and placed in row 1 of the second type II fake background file.

Now assume no data is currently loaded:

```
XSPEC12> fakeit 2 backb{1}
```

Two fake spectra in one type II output file are produced, as is a corresponding fake background file with 2 rows. The fact that the user has entered a type II background file on the command line tells `fakeit` to produce type II output. The first fake spectrum will have no associated background, so row 1 in the fake background file will be all zeros. Row 2 will consist of the fake background generated from `backb{1}`.

I need to better understand the uses of these in order to support them in ATLAS.

### 11.5.3.1.8 ignore: ignore detector channels

Ignore data channels. (See also **notice**.)

```
ignore <range1> [<range2>] ... [<rangeN>]  
ignore bad
```

where

```
<rangeI> ::= <spectrum range>: <channel range> | <channel range>.
```

If no `<spectrum range>` is given, then the previous range is used (the initial default range is file one (1) only). The form of `<spectrum range>` is

```
<spectrum range> ::= <init spectrum> - <last spectrum> | <spectrum>
```

where `<init spectrum>`, `<last spectrum>`, and `<spectrum>` are spectrum numbers, in the order that they were input with the **data** command. The form of channel range is

```
<channel range> ::= <initial channel> - <last channel> | <channel>
```

If integers are given for the channel ranges then channels will be used while if reals are given then energies (or wavelengths if `setplot wave` has been specified). Energy and wavelength units are determined by the `setplot energy` and `wave` settings. If only the last channel is indicated, then a default value of one (1) is used for the initial channel. Channels remain ignored until they are explicitly noticed with the **notice** command, or if a spectrum is replaced.

Examples:

Assume that 4 spectra have been read in, the first 2 with 100 channels and the last 2 with 50 channels.

```
XSPEC12> ignore **:1-10  
// The first 10 channels of all 4 spectra are ignored  
XSPEC12> ignore 80-**  
// An attempt will be made to ignore channels ≥80 in all four data sets (as that was the last  
// spectrum  
// range specified). As a result, only channels 80-100 will be ignored for spectra 1 and 2. No  
// change  
// will occur for spectra 3 and 4, as they have no channels greater than 50.
```

```

XSPEC12> ign 4:1-20 3:30-40 45-**
// Channels 11-20 for spectrum 4 are ignored (1-10 were ignored already)
// while channels 30-40 and 45-50 of spectrum 3 are ignored.
XSPEC12> ignore 1:1-5
// No channels are ignored, as these were ignored at the beginning.
XSPEC12> ignore 2:1.-5.
// Ignore all channels between 1 and 5 keV in the second dataset

```

This is done with selections and selection groups in ATSAI.

### 11.5.3.1.9 notice: notice data channels

Notice data channels. (See also **ignore**.)

```

notice <range1> <range2> ... <rangeN>
notice all

```

where

```

<rangeI> =:: <spectrum range>: <channel range> | <channel range>

```

If no <spectrum range> is given, then the previous range is used (the initial default range is file one (1) only). The form of <spectrum range> is

```

<spectrum range> =::<init spectrum> - <last spectrum> | <spectrum>

```

where <init spectrum>, <last spectrum>, and <spectrum> are spectrum numbers, in the order that they were input with the data command. The form of channel range is

```

<channel range> =:: <initial channel> - <last channel> | <channel>

```

If <channel range> are integers then channels will be used or if reals then energies (or wavelengths if `setplot wave` has been specified). Energy and wavelength units are determined by the `setplot energy` and `wave` settings. If only the last channel is indicated, then a default value of 1 is used for the initial channel. Channels remain noticed until they are explicitly ignored with the **ignore** command. When a spectrum is replaced by another spectrum, all input channels automatically are noticed.

```

XSPEC12> notice all

```

resets all the channels to 'noticed'.

Examples:

Assume that 4 spectra have been read in, the first 2 having 100 channels and the last 2 having 50 channels. Assume also that channels 1-10 of all four spectra are ignored and that channels 80-100 of spectra 1 and 2 are ignored. In XSPEC12, **notice** does not force the detector response to be reread (see RESPONSE DESCRIPTION).

```

XSPEC12> notice **:1-10
// The first 10 channels of all 4 spectra are noticed.
XSPEC12> notice 80-**
// an attempt will be made to notice channels ≥80 in all 4 spectra
// (as that was the last spectrum range specified) but the result is that
// only channels 80-100 will be noticed for spectra 1 and 2, with no
// change for spectra 3 and 4 as they have no channels greater than 50.
XSPEC12> notice 1:1-5
// No channels are noticed, as these channels were noticed
// in the beginning.

```

This is done with selections and selection groups in ATSAI.

### 11.5.3.1.10 response: change the detector response for a spectrum

Modify one or more of the matrices used to describe the response(s) of the associated spectrum to incident X-rays.

**response** [<filespec>...]

**response** [<source num>:]<spectrum num> none

where

<filespec> ::= [[<source num>:]<spectrum num>] <file name>...,

and <file name> is the name of the response file to be used for the response of the associated spectrum. If <file name> ends in a {n} specifier then the nth response will be read from the file. <spectrum num> is the spectrum number for the first file name in the specification, and follows similar rules as described in the **data** command description. An important difference however is that the response command may only be used to modify the response of a previously loaded spectrum: an error message is printed if the <spectrum num> is greater than the current number of spectra (as determined from the last use of the **data** command).

An optional <source num> may be specified to attach additional responses to a spectrum, and should be paired with <spectrum num> separated by a ':'. This allows the user to assign multiple models, each with their own response file, to a particular spectrum. See the **model** command for more information. If no <source num> is specified, it always defaults to 1. Source numbers do not need to be assigned consecutively to a spectrum, and gaps in numbering are allowed. The additional response may be removed with a response <source num>:<spectrum num> none command. Both the **show data** and **show response** commands will display current information regarding the response(s) to spectrum assignments.

A file name none indicates that no response is to be used for that spectrum. This situation means that any incident spectrum will produce no counts for those particular channels. If a file is not found or cannot be opened for input, then the user is prompted for a replacement response file. An <EOF> at this point is equivalent to using none as the response. See the **data** command for ways to totally remove the spectrum from consideration. The user is also prompted for a replacement if the response file has a different number of PHA channels than the associated spectrum. A warning will be printed out if the response detector ID is different from the associated spectrum's. The current ignore status for channels is not affected by the command. (See the **ignore** and **notice** commands).

Examples:

It is assumed that there are currently three spectra:

Single source usage:

```
XSPEC12> response a,b,c
// New files for the response are given for all three files.
XSPEC12> response 2 none
// No response will be used for the second file.
XSPEC12> response ,d{2}
// The second response in d becomes the response for the second file.
```

Multiple source usage:

```
XSPEC12> response 2:1 e
// A second source with response e.pha is now added to the first
// spectrum. A second model can be assigned to this source.
XSPEC12> response 2:2 f 3:2 g
// A second and third source is assigned to spectrum 2.
XSPEC12> response 2:2 none
// The second source is now removed from spectrum 2.
```

In ATSAI, this needs to be an extension to the tag tool.

## 11.5.4 Fit Commands

### 11.5.4.1.1 bayes: set up for Bayesian inference

**bayes** [*<option>* | *<mod par #>*] {*<prior type>* *<hyperparameters>*}

where *<option>* =: [off | on | cons]. If a parameter number is given as the first argument then this command sets up the prior for the specified model parameter but does not turn Bayesian inference on. If the first argument to the **bayes** command is not a parameter number then one of the options *off*, *on*, or *cons* is used. The first two turn Bayesian inference off or on, while *cons* turns Bayesian inference on and gives all parameters a constant prior. The options for prior types are as follows.

Prior type	Log(prior)
cons	0
exp	$-par/hpar_1 - \log(hpar_1)$
jeffreys	$-\log(par)$
gauss	$-0.5\log(2\pi hpar_2) - 0.5(hpar_1 - par)^2/hpar_2^2$

Where *par* is the parameter value and *hpar#* the hyperparameter values. *jeff* is an abbreviation for the Jeffreys prior, which is  $1/x$  for an assumed Gaussian distribution of the parameter.

I don't fully understand this yet. My reading is that any parameter may have Bayesian inference set to none (default), cons, exp, jeffreys, or gauss, as well as up to two hyperparameters. This could be implemented with a Bayesian... button that presents a dialog with a popup for the five options. Depending upon the option selected, edit boxes for up to two hyperparameters appear. When dismissed, the selected options are summarized beside the button.

In addition, the model as a whole has a checkbox that turns Bayesian inference on or off, and a button that gives all parameters a constant prior.

#### 11.5.4.1.2 chain: run a Monte Carlo Markov Chain.

**chain** [burn *<length>*] [clear] [filetype *fits|ascii*] [info] [length *<length>*] [load *<filename>*] [proposal [*<distr>* *<source>*]|*<user-defined>*] [rand on|off] [run [*>*]*<filename>*] [stat *<par num>*] [temperature *<value>*] [type *mh|gw*] [unload *<range>*] [walkers *<value>*]

If the proposal source is set to use the fit correlation matrix (the default), you must perform a fit before running any chains.

When chains are loaded (and their parameters correspond to the currently loaded model), they will be used by the various XSPEC commands that require distributions of parameter values, such as **eqwidth** or **flux** when calculating error estimations. The **error** command itself will also use the loaded chains, determining the error range from a central percentage of the sorted chain values. This is likely to be faster than the **error** command's standard algorithm when not using chains.

burn <i>&lt;length&gt;</i>	Specifies that the first <i>&lt;length&gt;</i> steps should be thrown away prior to storing the chain.
clear	Does a reset and removes all chains from the list.
filetype <i>fits ascii</i>	Chooses the format of the output chain file. <i>fits</i> (the default) writes the chain to a binary table in a FITS file. <i>ascii</i> writes the chain to a simple text file. Either format is readable when using the <b>load</b> command.
info	Prints information on the current chains.
length <i>&lt;length&gt;</i>	Sets the length for new chains.
load <i>&lt;filename&gt;</i>	Loads a chain which has been run earlier, stored in file given by <i>&lt;filename&gt;</i> .
proposal <i>&lt;distr&gt;</i> <i>&lt;source&gt;</i>	Selects the proposal distribution and source of covariance information to be used when running new chains. The default is <i>proposal gaussian fit</i> . Currently implemented <i>&lt;distr&gt;</i> options are <i>gaussian</i> and <i>cauchy</i> . <i>&lt;source&gt;</i> options are: <i>chain</i> Covariance is taken from the currently loaded chains. <i>diagonal</i> The values of a diagonal covariance matrix are entered directly on the <i>&lt;values&gt;</i> command line, separated by commas and/or spaces: <i>c_11 c_22 ...</i>

	<p><code>C_nn.</code>  <code>&lt;filename&gt;</code> Covariance is read in from a user-specified text file. The file must contain the values of an <math>N \times N</math> matrix where <math>N</math> is the current number of freely varying parameters. The values of each matrix row should be entered on one line with whitespace separation. Since this matrix is always symmetrical, values above the diagonal may be omitted. For example a <math>2 \times 2</math> matrix could be entered as:</p> <pre>0.98 0.15 0.96</pre> <p><code>fit</code> Covariance is taken from the correlation information produced by the current fit.</p> <p><code>matrix</code>  <code>&lt;values&gt;</code> The lower half and diagonal of a symmetrical square covariance matrix are entered directly on the command line, separated by commas and/or spaces: <code>C_11 C_21 C_22 C_31 C_32 C_33 ... C_nn</code></p> <p>Typing <code>chain proposal</code> with no other arguments will show a list of all available proposal options.</p> <p>For an alternative to XSPEC's <code>&lt;distr&gt; &lt;source&gt;</code> proposal options, the user may instead want to provide their own custom randomization algorithm. This can be done by writing their own C++ class(es) derived from an XSPEC randomizer base class. The custom class is added at runtime using the same <b>initpackage/lmod</b> command sequence as for local models, and is specified by <code>proposal &lt;name&gt;</code>, where <code>&lt;name&gt;</code> is the unique name attribute the user provides for their class. Please see Appendix G for more information on writing a custom randomizing class, and <b>initpackage</b> for building and loading it.</p>
<code>rand on off</code>	Specifies whether the chain start point will be randomized, or taken from the current parameters.
<code>recalc</code>	A <b>deprecated</b> option that performs the equivalent of <code>proposal gaussian chain</code> .
<code>run [&gt;]&lt;filename&gt;</code>	Runs a new chain written to the specified file, or append to an already loaded file if the ">" character precedes the filename. The chain is written to the file as it runs so its performance can be monitored by examining the file. For high-chatter settings, additional information is printed to the screen. A long run may be interrupted with Ctrl-C, in which case the chain file will still exist but will not be automatically loaded. If appending to a file, the current <code>filetype</code> setting must match the format of the file or XSPEC will prevent it.
<code>stat [&lt;modName&gt;:]&lt;parIdx&gt;</code>	Writes out statistical information on a particular parameter of the chain, specified by the parameter index number (with optional model name). The information displayed is: <ol style="list-style-type: none"> <li>1. The mean of the parameter over each chain file.</li> <li>2. The parameter mean over all chain files and the variance between chain means.</li> <li>3. The variance within the chains.</li> <li>4. The Rubin-Gelman convergence criterion.</li> <li>5. The fraction of repeats, defined as the number of lines in the chain file for which all parameter values are identical to the previous line, divided by the number of lines in the file.</li> </ol>
<code>temperature &lt;value&gt;</code>	Sets the temperature parameter used in the Metropolis-Hastings algorithm for the proposal acceptance or rejection. The default value is 1.0 and zero or negative values are forbidden. By using the <code>run append</code> option, it is possible for different sections of the chain file to use different temperatures. The temperatures and the line numbers to which they apply are stored in the header of the FITS format chain files, or in the metadata section at the top of the ASCII text format files.
<code>type mh gw</code>	Determines the algorithm used to generate the chain. Choices are <code>mh</code> (Metropolis-Hastings) or <code>gw</code> (Goodman-Weare, the default). If using Goodman-Weare, you must also set the <code>walkers</code> parameter.
<code>unload &lt;range&gt;</code>	Removes the chains specified by <code>&lt;range&gt;</code> from the list in <code>xspec</code> . Note that this does <i>not</i> delete the chain files.
<code>walkers &lt;value&gt;</code>	Sets the <code>walkers</code> parameter for the Goodman-Weare chain algorithm (see <code>type</code> ). This must be an even integer, and both the chain length and burn length should be divisible

by it (xspec will adjust the lengths to make them so if necessary).

All loaded chains must contain the same fit parameters. xspec will prevent the loading of a chain with a different number of parameters from the currently loaded chains.

### Examples:

```
XSPEC12>chain length 100
// Sets length of chains produced by the run command to 100.
XSPEC12>chain run chain_file1.out
// Runs a chain based on current valid fit parameters, output to
// chain_file1.out
XSPEC12>chain run >chain_file1.out
// Appends another run of length 100 to the end of chain_file1.out
XSPEC12>chain load chain_old.out
// Loads a pre-existing chain file, the result of an earlier run
// command. Warning is issued if not the same length as
// chain_file1.out
XSPEC12>chain stat 3
// Prints statistical information on the third parameter of the chain.
XSPEC12>chain proposal gaussian myfile.txt
// New chain proposals will be a normal distribution using
// covariance values stored in myfile.txt rather than fit correlation matrix.
XSPEC12>chain prop gauss diag .1 .001 .0001
// New chain proposals will be a normal distribution using a 3x3
// diagonal covariance matrix with the values from the command line.
XSPEC12>chain temperature .8
// Sets the Metropolis-Hastings temperature value to .8 for
// future chain runs, replacing the default 1.0.
XSPEC12>chain clear
// Removes the 2 loaded chains from xspec's chain list.
```

ATSAL support for this will be very special case. I don't understand it well enough to design an interface yet.

#### 11.5.4.1.3 error, uncertain: determine confidence intervals of a fit

Determine the confidence region for a model parameter.

```
error [[stopat <ntrial> <toler>] [maximum <redchi>] [nonew] [<delta fit statistic>] [<model param range>...]]
```

where

```
<model param range> ::= [<modelName:>]<first param> - <last param>
```

determines the ranges of parameters to be examined, and <delta fit statistic> (distinguished from the model parameter indices by the inclusion of a decimal point), is the change in fit statistic used.

**For response parameters** (see **gain** command), use **error** with identical syntax except:

```
<response param range> ::= [<sourceNum:>]<first param> - <last param>
```

The **error** command uses one of two algorithms. If Monte Carlo Markov Chains are loaded (see **chain** command) the error range is determined by sorting the chain values, and then taking a central percentage of the values corresponding to the confidence level as indicated by <delta fit statistic>. This is likely to be the faster of the two algorithms.

When chains are **not** loaded, **error**'s algorithm is as follows:

Each indicated parameter is varied, within its allowed hard limits, until the value of the fit statistic, minimized by allowing all the other non-frozen parameters to vary, is equal to the last value of fit statistic determined by the **fit** command plus the indicated <delta fit statistic>, to within an absolute (not fractional) tolerance of <toler>. Note that before the **error** command is executed, the data must be fitted. The initial default values are the range 1-1 and the <delta fit statistic> of 2.706, equivalent to the 90% confidence region for a single interesting parameter. The number of trials and the tolerance for determining when the critical fit statistic is reached can be modified by

preceding them with the `stopat` keyword. Initially, the values are 20 trials with a tolerance of 0.01 in fit statistic.

If a new minimum is found in the course of finding the error, the default behavior is to abort the calculation and then automatically rerun it using the new best fit parameters. If you prefer not to automatically rerun the **error** calculation, then enter `nonew` at the start of the command string. The `maximum` keyword ensures that **error** will not be run if the reduced chi-squared of the best fit exceeds `<redchi>`. The default value for `<redchi>` is 2.0.

Since there are very many scenarios which may cause an **error** calculation to fail, it is highly recommended that you check the results by viewing the 9-letter error string, which is part of the output from the **tlout error** command (see **tlout** for a description of the error string). If everything went well, the error string should be "FFFFFFFF".

### Examples:

Assume that the current model has four model parameters.

```
XSPEC12> error 1-4
// Estimate the 90% confidence ranges for each parameter.
XSPEC12> error 9.0
// Estimate the confidence range for parameters 1-4 with delta fit
// statistic = 9.0, equivalent to the 3 sigma range.
XSPEC12> error 2.706 1 3 1. 2
// Estimate the 90% ranges for parameters 1 and 3, and the 1. sigma range for parameter 2.
XSPEC12> error 4
// Estimate the 1. sigma range for parameter 4.
XSPEC12> error nonew 4
// Same as before, but calculation will NOT automatically restart if a new minimum is found.
XSPEC12> error stop 20,,3
// Estimate the 1-sigma range for parameter 3 after resetting the number
// of trials to 20. Note that the tolerance field had to be included
// (or at least skipped over).
```

At the model level, an **Error...** button displays a dialog that selects confidence range, tolerance, number of trials, etc. By default, this applies to all parameters (vs. parameter 1 in XSPEC). Each parameter also has an **Error...** button that inherits the model setting by default or overrides it for this parameter only. Upon completion, possible failure indications are retrieved via `tlout error` and translated to readable form. At both the model and parameter level, an on/off checkbox enables error calculation. Enabling at the model level enables all parameters that are themselves enabled (vs. enabling all parameter-level error calculations). Enabling any parameter implicitly turns on the model level error calculation. At both levels, the button and the checkbox are the only indication of state shown; the user must hit the button to see details.

#### 11.5.4.1.4 fit: fit data

Find the best fit model parameters for the current data by minimizing the current statistic.

**fit** <fit method parameters>

The arguments to `fit` depend on the fitting method currently in use. See the **method** command for details (and for the usage of the `USE_NUMERICAL_DIFFERENTIATION` option in the user's startup `xspec.init` file). Output from the `fit` command also depends on the fitting method currently in use.

Using the Levenberg-Marquardt algorithm, the parameters accepted are the maximum `<number of iterations>` before the user is prompted, the `<critical delta>`, which is the (absolute, not fractional) change in the statistic between iterations less than which the fit is deemed to have converged, and `<critical beta>`.

The `<critical beta>` provides an optional second stopping criterion, and it refers to the  $|\text{beta}|/N$  value reported during a Levenberg-Marquardt fit. This is the norm of the vector derivatives of the statistic with respect to the parameters divided by the number of parameters. At the best fit this should be zero, and so provides another measure of how well the fit is converging. `<critical beta>` is set to a negative value by default, which renders it inactive.

Including the string `delay` as an argument to `fit` turns on delayed gratification. It is turned off by `nodelay`. Delayed gratification modifies the way the damping parameter is set and has been shown in many cases to speed up convergence. The default is `nodelay`.

If <number of iterations>, <critical delta>, <critical beta>, delay, or nodelay is entered through the **fit** command, it also becomes the future default value for the currently loaded fit method (i.e. Levenberg-Marquardt).

### Examples:

```
XSPEC12> fit
// Fit with the default number of iterations and critical delta chi-squared.
XSPEC12> fit 60
// Fit with 60 as the number of iterations.
XSPEC12> fit 50 1.e-3
// Fit with 1.e-3 as the critical delta.
XSPEC12> fit 50 1.e-3 20.
// Same fit, but will now use |beta|/N = 20.0 as another stopping criterion in addition
// to that of the critical delta.
XSPEC12> fit delay
// Same fit, but will now use delayed gratification.
```

Presently I see this as a Fit options... button that appears at the top of the parameter list for each model's parameters. The button brings up a dialog that fine-tunes the fit parameters as described. I don't know whether <number of iterations> should be an option or set to a high value, thus interrupted only when the user wishes to. The currently selected fit options would be shown beside the Fit options... button.

#### 11.5.4.1.5 freeze: set parameters as fixed

Do not allow indicated model parameters to vary. (See also **thaw**.)

**freeze** [<param range>...]

where

<param range> ::= [modelName:] <param#> | <param#> - <param#>.

**For response parameters** (see **gain** command):

**rfreeze** [<param range>...]

where

<param range> ::= [source number:] <param#> | <param#> - <param#>.

The indicated model parameter or range of model parameters will be marked so they cannot be varied by the **fit** command. By default, the range will be the last range input by either a **freeze** or **thaw** command.

Examples:

Currently there are six parameters, initially all unfrozen.

```
XSPEC12> freeze 2
// Parameter 2 is frozen
XSPEC12> freeze 4-6
// Parameters 4, 5, and 6 are frozen.
XSPEC12> thaw 2 3-5
// Parameters 2, 4, and 5 are thawed, parameter 3 is unaffected.
XSPEC12> freeze
// Parameters 3,4,5 are frozen (the last range input by a freeze or thaw command).
XSPEC12> rfreeze 4-6
// Response parameters 4, 5, and 6 are frozen.
```

This is implemented as a lock icon next to each parameter. Currently there is no way to freeze/thaw *all* parameters for a model. If this is needed, it could be added somewhere at the top of the parameter list.

#### 11.5.4.1.6 ftest: calculate the F-statistic from two chi-square values

Calculate the F-statistic and its probability given new and old values of  $\chi^2$  and number of degrees of freedom (DOF).

```
ftest chisq2 dof2 chisq1 dof1
```

The new  $\chi^2$  and DOF, `chisq2` and `dof2`, should come from adding an extra model component to (or thawing a frozen parameter of) the model which gave `chisq1` and `dof1`. If the F-test probability is low then it is reasonable to add the extra model component. WARNING: it is not correct to use the F-test statistic to test for the presence of a line (see Protassov et al 2002, ApJ 571, 545). WARNING: this command can only be used if the extra model component is additive, this does not give the correct result if the component is multiplicative (see Orlandini et al. 2012, ApJ 748, 86).

How should this be integrated?

#### 11.5.4.1.7 goodness: perform a goodness of fit Monte-Carlo simulation

Perform a Monte Carlo calculation of the goodness-of-fit.

```
goodness [<# of realizations>] [sim | nosim]
```

This command simulates <# of realizations> spectra based on the model and writes out the percentage of these simulations with the fit statistic less than that for the data. If the observed spectrum was produced by the model then this number should be around 50%. This command only works if the sole source of variance in the data is counting statistics. The `sim|nosim` switch determines whether each simulation will use parameter values drawn from a Gaussian distribution centered on the best fit with sigma from the covariance matrix. The `sim` switch turns on this option, `nosim` turns it off in which case all simulations are drawn from the best-fit model. The default starting setting is `nosim`.

If I understand this correctly, a Goodness of fit... button selects the number of realizations and sim/nosim. A checkbox enables/disables the operation. If enabled, the operation is performed after the fit. Results are shown beside the button in the parameter window, rather than in the sidebar. (If this operation is performed frequently, the button and result may belong in the sidebar model pane instead.)

#### 11.5.4.1.8 margin: MCMC probability distribution.

Use the currently loaded MCMC chains to calculate a multi-dimensional probability distribution.

```
margin <step spec.> [<step spec.> ...]
```

where

```
<step spec.> ::= [{LOG | NOLOG}] [<model name>:]<fit param index> <low value> <high value> <no. steps>
```

The indicated fit parameter is stepped from <low value> to <high value> in <no. steps> + 1 trials. The stepping is either linear or log. Initially, the stepping is linear but this can be changed by the optional string `log` before the fit parameter index. `nolog` will force the stepping to be returned to the linear form. The number of steps is set initially to ten. The results of the most recently run **margin** command may be examined with **plot margin** (for 1-D and 2-D distributions only). This command does not require that spectral data files are loaded, or that a valid fit must exist.

#### Examples:

Assuming chain(s) are loaded consisting of 4 parameters.

```
XSPEC12>margin 1 10.0 12.0 20 log 3 1.0 10.0 5
// Calculate a 2-D probability distribution of parameter 1 from 10.0-12.0 in 20 linear
// bins, and parameter 3 from 1.0-10.0 in 5 logarithmic bins.
XSPEC12>margin 2 10.0 100.0 10 nolog 4 20. 30. 10
// Now calculate for parameter 2 in 10 log bins and parameter 4 in 10 linear bins.
```

I don't presently understand the purpose of this calculation. It looks like although its scope is per-parameter, only one such calculation at a time makes sense per model, making it a model level option (thus forcing selection of a single

parameter). It could also be a parameter level option, with the rule that setting it for any parameter implicitly unsets it for the others. There is no way presently in ATLAS to select a model without performing a fit. Is it acceptable to impose the constraint that you can't calculate an MCMC probability distribution without also performing a fit?

#### 11.5.4.1.9 renorm: renormalize model to minimize statistic with current parameters

Renormalize model, or change renorm conditions.

**renorm** [AUTO | NONE | PREFIT]

The **renorm** command will adjust the normalizations of the model by a single multiplication factor, which is chosen to minimize the fit statistic. Such a renorm will be performed explicitly whenever the command is used without a keyword, or during certain XSPEC commands, as determined by the following keywords:

AUTO Renormalize after a **model** or **newpar** command, and at the beginning of a fit  
PREFIT Renormalize only at the beginning of a fit  
NONE Perform no automatic renormalizations, i.e., only perform them when a **renorm** command is given explicitly.

What is the default for this command? What is the difference between renormalizing after a **model** or **newpar** command vs. just before a fit? Should this be a per-model option, or per-notebook, or a preference setting?

#### 11.5.4.1.10 steppar: generate the statistic "surface" for 1 or more parameters

Perform a fit while stepping the value of a parameter through a given range. Useful for determining confidence ranges in situations where greater control is needed than given with the **error** command.

**steppar** [<current|best>] <step spec> [<step spec>...]

where

<step spec> ::= [<log | nolog>] [<modelName>:]<param index> <low value> <high value> <# steps>

or

<step spec> ::= [<log | nolog>] [<modelName>:]<param index> delta <step size> <# steps>

In the first case the parameter is stepped from <low value> to <high value> in <# steps> + 1 trials. In the second case the parameter starts at the best fit value, *bft*. It steps from *bft*-<step size>\*<# steps> to *bft*+<step size>\*<# steps>, i.e. a total of 2\*<# steps>+1 trials. The stepping is either linear or log. Initially, the stepping is linear but it can be changed by the optional string *log* before the parameter index. *nolog* will force the stepping to be returned to the linear form. If more than one parameter is entered, then <# steps> must be entered for each one except the last. Note that every variable parameter whose <param index> is *not* entered in the command will still be allowed to vary freely during each steppar iteration.

To perform a steppar run on **gain** (or response) parameters, the optional [<modelName>:] specifier is replaced by an optional [<sourceNumber>:] specifier, and the letter 'r' needs to be attached as a prefix to the <parameter index>. For example:

```
steppar 2:r3 1.5 2. 10
```

will step the third response parameter belonging to source number 2.

The number of steps is set initially to 10. At each value, the parameter is frozen, a fit performed, and the resulting value of chi-squared given. If *best* is given as an argument then the non-stepped parameters are reset to the best-fit values at each grid point. Alternatively, if *current* is given as an argument then the non-stepped parameters are started at their values after the last grid point (the default).

If multiple <step spec> are given for different parameters, then a raster scan of the parameter ranges is performed. At the end of the set, the parameters and chi-squared are restored to the values they had initially.

If the model is in a best-fit state when a **steppar** run is started and a new best fit is found during the run, the user will be prompted at the end of the run to determine if they wish to accept the new best-fit values for their parameters. This prompting can be disabled by the setting of the **query** flag.

Depending on the machine, a steppar run may be sped up significantly by assigning it to multiple processes. See the **parallel** command with the **steppar** option for more details.

### Examples:

Assume that the current model has four parameters:

```
XSPEC12> steppar 3 1.5 2.5
// Step parameter 3 from 1.5 to 2.5 in steps of .1.
XSPEC12> steppar log
// Repeat the above, only use multiplicative steps of 1.0524.
XSPEC12> step nolog 2 -.2 .2 20
// Step parameter 2 linearly from -.2 to .2 in steps of 0.02.
XSPEC12> step 2 delta 0.02 5
// Step parameter 2 linearly from the best-fit value-0.1 to
// the best-fit value+0.1 in a total of 11 steps.
```

Is this run in addition to or instead of a fit? I'm guessing "instead of." This makes me think a little more about progress reporting. It sounds like during command execution, feedback should be continuous:

<i>Dimensions</i>	<i>Feedback</i>
1	x/y plot of steps vs. $\chi^2$
2	3D plot of steps vs. $\chi^2$
3+	x/y plot where x is the composite iteration value and y is $\chi^2$ . This would just appear as a meandering wave of ups and downs, but at least it provides <i>some</i> feedback.

If **steppar** is conceptually a type of fit as I presently understand it to be, then it should probably appear as an option under the Fit... button for each model.

#### 11.5.4.1.11 thaw: allow fixed parameters to vary

Allow indicated parameters to vary. (See also **freeze**).

```
thaw {[<param range>...]}
```

where

```
<param range> ::= [modelName:]<param #> | <param #>-- <param #>
```

**For response parameters** (see **gain** command):

```
rthaw {[<param range>...]}
```

where

```
<param range> ::= [sourceNum:]<param #> | <param #>-- <param #>
```

The indicated parameter, or range of parameters, will be marked as variable by the fitting commands and treated as a fitting parameter in subsequent fits. By default, the range will be the last range input by either a **freeze** or **thaw** command. See the **freeze** examples for an example of the use of the **thaw** command.

In ATSA, this is handled by a lock icon beside each parameter. There is no way to freeze or unfreeze all of a model's parameters. Is this acceptable?

#### 11.5.4.1.12 weight: change weighting used in computing statistic

Change the weighting function used in the calculation of  $\chi^2$ .

**weight** [standard | gehrels | churzov | model]

*Standard weighting* uses *[Math Processing Error]* or the statistical error given in the input spectrum. *Gehrels weighting* uses *[Math Processing Error]*, a better approximation when  $N$  is small (Gehrels, N. 1986, ApJ 303, 336). *Churazov weighting* uses the suggestion of Churazov et al (1996, ApJ 471, 673) to estimate the weight for a given channel by averaging the counts in surrounding channels. *Model weighting* uses the value of the model, not the data, to estimate the weight.

This is implemented via XSpec settings now.

## 11.5.5 Model Commands

### 11.5.5.1.1 addcomp: add component to a model

Add a component to the model.

**addcomp** [modelName:]n <comp>

where  $n$  is the component number *before* which the new component is to be inserted, and <comp> is the name of the new component. Components are numbered in sequence in order of appearance in the expression entered. The new component is regarded as an operator on the component added if it is not additive.

The optional `modelName` qualifier allows the user to address a named model.

The user is prompted for parameter values for the component. If there are  $m$  components in the current model, then acceptable values for the component number added are  $1$  to  $m+1$ .

XSPEC detects the type of the model (additive, multiplicative etc), checks the correctness of the syntax of the output model, and adds the component if the resulting models obeys the syntax rules documented in the **model** command.

Thus,

```
XSPEC12> mo wa(po)
XSPEC12> addcomp 2 bb
```

Yields the model achieved by

```
XSPEC12> mo wa(bb + po)
```

See also **delcomp** (delete component by number).

Other Examples will serve to clarify **addcomp**'s behavior. Suppose that the current model specification is `ga+po` which using the `show` command would yield the description `model = gaussian[1] + powerlaw[2]`.

The comments give the model expression following the entry of **addcomp** and **delcomp** commands:

```
XSPEC12> addcomp 2 wab
// gaussian[1]+wabs[2](powerlaw[3])
XSPEC12> addcomp 4 pha
// (gaussian[1]+wabs[2](powerlaw[3]))phabs[4]}
XSPEC12> delcomp 1
// (wabs[1](powerlaw[2]))phabs[3]}
XSPEC12> addcomp 2 zg
// (wabs[1](zgauss[2]+powerlaw[3]))phabs[4]}
XSPEC12> delcomp 3
// (wabs[1](zgauss[2]))phabs[3]}

XSPEC12> mo wa(po)
XSPEC12> addcomp 1 ga
// gauss[1] + wabs[2]*powerlaw[3]
```

```

XSPEC12> delcomp1
XSPEC12> addcomp 1 pha
// phabs[1]*wabs[2]*powerlaw[3]

XSPEC12>mo wabs(po)
XSPEC12> addcomp 3 bb
// wabs[1]*powerlaw[2] + bbody[3]
XSPEC12> delcomp 1
XSPEC12> addcomp 3 pha
// wabs[1]*powerlaw[2]*pha[3]
XSPEC12> addcomp 3 po
// ERROR: po (additive) is interpreted as being added to the multiplicative
// model pha[3], which is a context error.

```

For multiply nested models...

```

XSPEC12> mo wa(po + pha(bb + ga))
XSPEC12> addcomp 6 po
// wabs[1](powerlaw[2] + phabs[3](bbody[4] + ga[5]) + powerlaw[6])
XSPEC12> addcomp 5 peg
// wabs[1](powerlaw[2] + phabs[3](bbody[4] + pegpwlw[5] ga[6]) + powerlaw[7])
XSPEC12> addcomp 7 wa
// wabs[1](powerlaw[2] + phabs[3](bbody[4] + pegpwlw[5] ga[6]) + wabs[7]*powerlaw[8])

```

ATSAL supports this by directly editing the model expression.

### 11.5.5.1.2 addline: add spectral lines to a model

Tcl script to add one or more lines to the current model in an optimum fashion.

**addline** [ $\langle nlines \rangle$ ] [ $\langle modeltype \rangle$ ] [{*fit*|*nofit*}] /p>

$\langle nlines \rangle$  additional lines are added one at a time. Line energies are set to that of the largest residual between the data and the model. For each line a fit is performed with the line width and normalization as the only free parameters. The default option is one gaussian line. The other  $\langle modeltype \rangle$  that can be used is *lorentz*. If no third argument is given then the sigma and normalization of each line are fit. If *nofit* is specified then the fit is not performed but if *fit* is specified then all free parameters are fit.

**addline** currently will only work with the default model (i.e. not for named models).

I don't understand how this works yet, but I think it means "find  $\langle nlines \rangle$  new peaks, in order of most- to least-promising, and add them to a [model? spectrum? plot layer?]. If I interpret this correctly, this might work in ATSAL as follows. The user adds a selection set of type "Emission line (multiple)." A **Configure...** button appears. This displays a dialog to specify a count of lines, defaulting to 1; and the Gaussian/Lorentzian choice. This dialog could also select the algorithm to use to identify peaks, and would identify the database from which to extract lines (apec, bearden, mekal). Any change to dialog sets the enabled checkbox.

This command is unusual in that two steps are involved in using it. After configuring the command as described, the selection set remains empty because XSpec hasn't yet computed results. A Refresh is necessary to pass these commands to XSpec. Upon completion, ATSAL adds the specified number of selections to the selection set and then disables the option, having created the necessary selections. The user can modify the count and enable it again if desired.

Once the selections have been created, the user can fine tune them like any other selections, and add or remove some of them. On the next Refresh cycle, each selection is passed to **identify**, and a line label, if found, appears over each line.

### 11.5.5.1.3 delcomp: delete a model component

Delete one or more components from the current model.

**delcomp** [modelName:]<comp num range>

where

<comp num range> is a range of positions in the model specification of the components to be deleted.

Examples:

Suppose that the current model specification is wa(po+ga+ga). Then

```
XSPEC12> delcomp 3-4
// Changes the model to wa(po)
XSPEC12> delcomp 1
// Changes the model to po
```

ATSAL supports this by directly editing the model expression.

#### 11.5.5.1.4 dummyrsp: create and assign dummy response

Create a “dummy” response, covering a given energy range.

```
dummyrsp [<low Energy> [<high Energy> [<# of ranges> [<log or linear> [<channel offset>
[<channel width> [source\_Num:spec\_Num]]]]]]]
```

This command creates a dummy response matrix based on the given command line arguments, which will either temporarily supersede the current response matrix, or create a response matrix if one is not currently present. There are two main uses for this command: to do a “quick and dirty” analysis of uncalibrated data (mode 1), and to examine the behaviour of the current model outside the range of the data’s energy response (mode 2). [Note that mode 2 usage has now been rendered redundant by the more flexible \*\*energies\*\* command.](#)

All parameters are optional. The initial default values for the arguments are 0.01 keV, 100 keV, 200 logarithmic energy steps, 0.0 channel offset, and 0.0 channel width. The default values of the first 5 parameters will be modified each time the parameter is explicitly entered. The channel width parameter however always defaults to 0.0 which indicates mode 2 operation, described below.

[In addition to the 6 optional parameters allowed for versions 11.x and earlier, a seventh optional parameter has been added allowing the user to apply the dummy response to just one particular source of a spectrum. It consists of two integers for \(1-based\) source number and spectrum number, separated by a colon. Either both integers should be entered, or they should be left out entirely; i.e. a dummy response is either made for \*every\* source in every spectrum, or just one source in one spectrum. This parameter always defaults to all sources and all spectra.](#)

For mode 1 usage, simply enter a non-zero value for the channel width. In this instance, one has a spectrum for which typically no response matrix is currently available. This command will create a diagonal response matrix with perfect efficiency, allowing for the differences in binning between the photon energies and the detector channel energies (see example below). The response matrix will range in energy from <low Energy> to <high Energy>, using <# of ranges> as the number of steps into which the range is logarithmically or linearly divided. The detector channels are assigned to have widths of energy <channel width> (specified in keV), the lower bound of the first channel starting at an energy of <channel offset>. Then the data can be fit to models, etc., under conditions that assume a perfect detector response.

For mode 2 usage (channel width = 0.0), one can use this command to examine the current model outside the range of the energy response of the detector. When examining several aspects of the current model, such as plotting it or determining flux, XSPEC uses the current evaluation array. This, in turn, is defined by the current response files being used, which depend on the various detectors. For example, low energy datasets (such as those from the *EXOSAT* LEs) may have responses covering 0.05 to 2 keV, while non-imaging proportional counters can span the range from 1 to 30 keV. If the user wishes to examine the behavior of the model outside of the current range, then he or she temporarily must create a dummy response file that will cause the model to be evaluated from <low energy> to <high energy>, using <# of ranges> as the number of steps into which the range is logarithmically or linearly divided. If one wishes only to set the energy response range, than the <channel width> argument may be omitted. In this case, or in the case where no data file has been read in, all entries of the dummy response matrix are set to zero. Under these circumstances the dummyrsp has no physically correct way of mapping the model into the data PHA channels, so the user should not try to fit—or plot—the data while the dummyrsp is active in this mode. Also, data need not even be loaded when calling this command in mode 2.

The previous response matrices can be reimplemented with the **response** command, with no arguments. Any use of

the **data** and **notice** commands will replace the dummy response with a correct set of matrices, or with no response matrix if none was originally present.

### Examples:

```
XSPEC12> dummyrsp
// Create the dummy response for all spectra and sources with the default limits, initially
// .01,
// 100, and 200 bins.
XSPEC12> dummyrsp .001 1
// Create a dummy response with 200 bins that cover the range from
// 0.001 to 1 keV.
XSPEC12> dummyrsp ,,,500
// The same range, but now with 500 bins.
XSPEC12> dummyrsp ,,,,lin
// The same range, but now with linearly spaced bins.
XSPEC12> dummyrsp ,,,,,0.1
// The same range, but now create a diagonal response matrix, with
// channel widths of 0.1 keV.
XSPEC12> response
// Restore any previous correct responses.
```

Example dummy response matrix:

Assume a spectrum with 4 channels, then

```
XSPEC12> dummyrsp .0 30.0 3 lin 5.0 8.0
```

will produce the following response:

<i>Energies</i>	<i>Detector channel energies</i>			
	5.0-13.0	13.0-21.0	21.0-29.0	29.0-37.0
0.0-10.0	0.5	0	0	0
10.0-20.0	0.3	0.7	0	0
20.0-30.0	0	0.1	0.8	0.1

If I understand correctly, this looks like it should be an option in the response matrix popup menu in the tag tool. Selecting the option would prompt for the parameters. To turn it off, the user would select a response matrix file or "None."

### 11.5.5.1.5 editmod: edit a model component

Add, delete, or replace one component in the current model.

```
editmod [<delimiter>] <component1> <delimiter> <component2> <delimiter> ... <componentN>
[<delimiter>]
```

where

<delimiter> is some combination of (, +, \*, and ), and <componentJ> is one of the models known to XSPEC.

The arguments for this command should specify a new model, with the same syntax as the previous model, except for one component which may be either added, deleted, or changed to a different component type. XSPEC then compares the entered model with the current model, determines which component is to be modified (prompting the user if necessary to resolve ambiguities) and then modifies the model, prompting the user for any new parameter values which may be needed.

Examples:

```
XSPEC12> mo wabs(po)
XSPEC12> ed wabs(po+ga)
```

```
// This command will add the component gauss to model in the specified place and prompt
// the user for its initial parameters.
XSPEC12> mo wabs(po+zg)
XSPEC12> ed po+zg
// This command will delete the component wabs from the model, leaving the other
// components and their current parameter values unchanged.
XSPEC12> mo wabs(po+po)
XSPEC12> ed wabs(po)
// Here an ambiguity exists as to which component to delete. In this case XSPEC will print
// out the current model, showing the component number for each component, and then
// prompt the user for which component he wants deleted.
XSPEC12> mo wabs(po+ga)
XSPEC12> ed wabs(po+zg)
// The component gauss will be replaced by the component zgauss, and the user will be
// prompted for parameter values for the new component.
```

In ATSA, this is handled by directly editing the model expression.

### 11.5.5.1.6 energies: specify new energy binning for model fluxes

Supply an energy-binning array to be used in model evaluations in place of their associated response energies. The calculated model spectra are then interpolated onto the response energy arrays before multiplying by the response matrix. This command replaces and enhances the **extend** command from earlier versions.

**energies** <range specifier> [<additional range specifiers>...]

**energies** <input ascii file>

**energies** extend <extension specifier>

**energies** reset

where the first <range specifier> ::= <low E> <high E> <nBins> log | lin

<additional range specifiers> ::= <high E> <nBins> log | lin

<extension specifier> ::= low | high <energy> <nBins> log | lin

All energies are in keV. Multiple ranges may be specified to allow for varied binning in different segments of the array, but note that no gaps are allowed in the overall array. Therefore only the first range specifier accepts a <low E> parameter. Additional ranges will automatically begin at the <high E> value of the previous range.

The extend option provides the same behavior as the old **extend** command. Models will use associated response energy arrays, with an additional low and/or high array extension. <energy> is the value to which the array is extended, using <nBins> additional log or linear bins.

With the <input ascii file> option, the user can instead supply a customized energy array from a text file. The format requirements are simply that the bin values must appear one to a line and in ascending sorted order. Blank lines are allowed and so are comments, which must be preceded by a '#'. A simple example:

```
# myEnergyBinning.txt
.1
1.0
10. # now some linear bins
15.
20.
25.
```

which would actually produce the same energy array as:

```
energies .1 10. 2 log 25. 3 lin
```

Once an energy array is specified, it will apply to all models, and will be used in place of any response energy array (from actual or dummy responses) for calculating and binning the model flux. It will also apply to any models that are created after it is specified. To turn off this behavior and return all models back to using their response energies, type **energies reset**.

Similarly, an array extension created by the `extend` option will continue to be applied to all models until it is either overwritten by another extension, replaced by a new **energies** array, or removed with the `reset` option. This allows both low and high extensions to exist together.

When a custom-energy binned model flux array needs to be multiplied by a response matrix, `xspec` will temporarily rebin the flux array to match up with the response energy binning. This is done by simply scaling the flux by the fractional overlap between the custom and response bins. If there is no overlap between the custom and response energies, then the response will be multiplied by zero.

The **energies** command saves the most recently entered range and extension specifiers to be used as default values the next time it is called. The initial default range specifier is one range with `<low E> = .1`, `<high E> = 10.`, `<nBins> = 1000`, and `lin`. The initial default extension specifier is `high` with `<energy> = 100.`, `<nBins> = 200`, and `log`.

### Examples:

```
XSPEC12> energies ,50,,log
// Creates an array from .1 to 50. of 1000 logarithmic bins.
XSPEC12> energies ,,,,100. 5 lin
// Modifies previous array by adding 5 linear bins from 50. to 100.
XSPEC12> energies ,,,,200.
// The 2nd range is now 50. to 200. in 5 linear bins.
XSPEC12> energies 1.,,100
// Array is now just 1 range, 1. to 50. in 100 logarithmic bins.
XSPEC12> energies myFile.txt
// Array is replaced with values stored in myFile.txt
XSPEC12> energies extend ,75.,,lin
// Models will go back to using response energies, but with an
// extension of the high end to 75. keV in 100 additional linear bins.
XSPEC12> energies extend low .01
// Add a low-end extension to .01 keV with 100 new linear bins.
XSPEC12> energies reset
// All models will go back to using the original energy arrays
// from responses.
```

This looks like a notebook-level **Energies...** option to me, since it overrides the setting for all models. I think this would be handled by simple text box, into which users could load existing tables or create and optionally save tables. There would also be an **Insert bins...** button which prompts for `lin/log`, number of bins, and increment. These would be inserted at the insertion point. This approach preserves compatibility with the existing implementation, which might be important if users sometimes build complex tables. As with several other commands, the **Energies...** button would be accompanied by a checkbox that enables or disables the override. Error-checking would verify that the file has a single value per line and that they are monotonic.

### 11.5.5.1.7 eqwidth: determine equivalent width

Determine the equivalent width of a model component.

```
eqwidth [[RANGE <frac range>] <[model name:]<model component number>] [err <number> <level> | noerr]
```

The command calculates the integrated photon flux produced by an additive model component (**combined with its multiplicative and/or convolution pre-factors**) (FLUX), the location of the peak of the photon spectrum (E), and the flux (photons per keV) at that energy of the continuum (CONTIN). The equivalent width is then defined as  $\{EW = FLUX / CONTIN\}$  in units of keV. **New for version 12: the continuum is defined to be the contribution from all other components of the model.**

There are certain models with a lot of structure where, were they the continuum, it might be inappropriate to estimate the continuum flux at a single energy. The continuum model is integrated (from  $E(1-\text{frac range})$  to  $E(1+\text{frac range})$ ). The initial value of `<frac range>` is 0.05 and it can be changed using the `RANGE` keyword.

The `err/noerr` switch sets whether errors will be estimated on the equivalent width. The error algorithm is to draw parameter values from the distribution and calculate an equivalent width. `<number>` of sets of parameter values will be drawn. The resulting equivalent widths are ordered and the central `<level>` percent selected to give the error range. You can get the full array of simulated equivalent width values by calling `tc!out eqwidth` with the `errsim` option

(see **tcloout** command).

When Monte Carlo Markov Chains are loaded (see **chain** command), they will provide the distribution of parameter values for the error estimate. Otherwise the parameter values distribution is assumed to be a multivariate Gaussian centered on the best-fit parameters with sigmas from the covariance matrix. This is only an approximation in the case that fit statistic space is not quadratic.

### Examples:

The current model is assumed to be  $M_1(A_1+A_2+A_3+A_4+M_2(A_5))$ , where the  $M_x$  models are multiplicative and the  $A_x$  models are additive.

```
XSPEC12> eqwidth 3
// Calculate the total flux of component  $M_1A_2$  (the third component of the model with its
// multiplicative pre-factor) and find its peak energy (E). The continuum flux is found by the
// integral flux of  $M_1(A_1+A_3+A_4+M_2(A_5))$ , using the range of 0.95E to 1.05E to estimate the flux.

XSPEC12> eqwidth range .1 3
// As before, but now the continuum is estimated from its behavior over the range 0.9E to 1.1E.

XSPEC12> eqwidth range 0 3
// Now the continuum at the single energy range (E) will be used.

XSPEC12> eqwidth range .05 2
// Now the component  $M_1A_1$  is used as the feature, and
//  $M_1(A_2+A_3+A_4+M_2(A_5))$  are used for the continuum. The range has been reset to the original
// value.

XSPEC12> eqwidth 1
// Illegal, as  $M_1$  is not an additive component.
```

I think that this might be done by adding an **Equivalent width...** button to each additive model in the parameters display, alongside an enable checkbox. The button would prompt for the necessary parameters. The results would be displayed beside the controls.

### 11.5.5.1.8 flux: calculate fluxes

Calculate the flux of the current model between certain limits.

```
flux [<lowEnergy> [<hiEnergy>]] [err <number> <level> | noerr]
```

where <lowEnergy> and <hiEnergy> are the values over which the flux is calculated. Initial default values are 2 to 10 keV.

The flux is given in units of photons  $\text{cm}^{-2} \text{s}^{-1}$  and ergs  $\text{cm}^{-2} \text{s}^{-1}$ . The energy range must be contained by the range covered by the current spectra (which determine the range over which the model is evaluated). Values outside this range will be reset automatically to the extremes. Note that the energy values are two separate arguments, and are *not* connected by a dash (see parameter ranges in the **freeze** command).

The flux will be calculated for all loaded spectra. If no spectra are loaded (or none of the loaded spectra have a response), the model is evaluated over the energy range determined by its dummy response. (In XSPEC12, models are automatically assigned default dummy responses when there is no data, so the **dummyrsp** command need not be given.) If more than one model has been loaded, whichever model the user has specified to be the active one for a given source is the one used for the flux calculation.

The results of a flux command may be retrieved by the `tcloout flux <n>` command where *n* is the particular spectrum of interest. If the flux was calculated for the case of no loaded spectra, the results can be retrieved by `tcloout flux` with the <n> argument omitted.

The `err/noerr` switch sets whether errors will be estimated on the flux. The error algorithm is to draw parameter values from the distribution and calculate a flux. <number> of sets of parameter values will be drawn. The resulting fluxes are ordered and the central <level> percent selected to give the error range. You can get the full array of

simulated flux values by calling `tclout flux` with the `errsim` option (see `tclout` command).

When Monte Carlo Markov Chains are loaded (see `chain` command), they will provide the distribution of parameter values for the error estimate. Otherwise the parameter values distribution is assumed to be a multivariate Gaussian centered on the best-fit parameters with sigmas from the covariance matrix. This is only an approximation in the case that fit statistic space is not quadratic.

There is also a model component `cflux` which can be used to estimate fluxes and errors for part of the model. For instance, defining the model as `wabs(pow + cflux(ga))` provides a fit parameter which gives the flux in the gaussian line.

Examples:

The current data have significant responses to data within 1.5 to 18 keV.

```
XSPEC12> flux
// Calculate the current model flux over the default range.
XSPEC12> flux 6.4 7.0
// Calculate the current flux over 6.4 to 7 keV
XSPEC12> flux 1 10
// The flux is calculated from 1.5 keV (the lower limit of the current response's sensitivity)
// to 10 keV.
```

This looks like a model option, but (I am guessing) not a common enough one to add to the model pane in the sidebar. So it might be placed near the top of the model parameters window. For a single spectrum, results would appear beside the **Flux...** button. For multiple spectra, the results would appear as a list inside the **Flux...** dialog. A checkbox enables or disables flux calculation.

### 11.5.5.1.9 gain: modify a response file gain

Modify a response file gain, in a particularly simple way. **CAUTION:** This command is to be used with extreme care for investigation of the response properties. To properly fit data, the response matrix should be recalculated explicitly (outside of XSPEC) using any modified gain information derived.

The `gain` command shifts the energies on which the response matrix is defined and shifts the effective area curve to match. The effective area curve stored by XSPEC is either the ARF, if one was in use, or is calculated from the RSP file as the total area in each energy range. This means that if there are sharp features in the response then these will only be handled correctly by the `gain` command if they are in the ARF or if no ARF is input. The new energy is calculated by

$$E' = E / \langle \text{slope} \rangle - \langle \text{intercept} \rangle$$

where `<intercept>` is in units of keV.

```
gain [<sourceNum>:]<specNum> <slope> <intercept>
gain fit [[<sourceNum>:]<specNum>]
gain nofit { [[<sourceNum>:]<specNum>] | all }
gain off
```

The first variant of the `gain` command shown above will apply the gain shift specified by the `<slope>` and `<intercept>` parameters to the response belonging to spectrum `<specNum>`, and optionally specified `<sourceNum>` if the data is analyzed with multiple models. The initial default `<specNum>` is 1; later, the default is the number of the spectrum last modified. Initially, all responses are assumed to have nominal gains, determined implicitly by the data in the response files. This is equivalent to a `<slope>` of 1 and an `<intercept>` of zero. All responses can be reset back to this original state by entering `gain off`. Note that in this mode of usage, the slope and intercept values do *not* become variable fit parameters. They are simply fixed values used to modify the response.

The `gain fit` mode is used when the user wishes to have the slope and intercept parameters determined by the results of a fit. The `<specNum>` and optional `<sourceNum>` parameters specify to which response the fit gain values are to be applied. These may be omitted only if a single spectrum is loaded, with a single model source. Otherwise at least a spectrum number is required. The user will then be prompted for slope and intercept parameter information in the same way as model parameters are normally entered. These values are then immediately applied to the response, and

will be adjusted the next time a fit is run.

Gain fit parameters belong to the more general category of *response parameters* in XSPEC, and may be modified using an equivalent set of commands to those used for regular model parameters. The command names are the same except prefixed by the letter 'r':

<i>XSPEC commands for editing/viewing model parameters</i>	<i>Equivalent commands for gain (or response) parameters</i>
newpar	rnewpar
freeze	rfreeze
thaw	rthaw
untie	runtie
error	rerror
model	rmodel
show par	show par, show rpar

For example after assigning gain fit parameters to source 1 of spectrum 1 (with `gain fit 1`):

```
XSPEC12> rfreeze 1
XSPEC12> rnewpar 2 .05
XSPEC12> show rpar
```

Response parameters defined:

```
=====
Source No.: 1
Rpar Spectrum Rmodel Rpar_name Unit Value
  1 1 gain slope 1.00000 frozen
  2 1 gain offset 5.00000E-02 +/- 0.0
=====
```

**rnewpar** can also link gain parameters to one another and can adjust the hard and soft parameter limits, as **newpar** does for model parameters. The default gain parameter hard limits are hardcoded in XSPEC, but these can be overridden by setting `GSLOP_MIN`, `GSLOP_MAX`, `GOFFS_MIN`, and `GOFFS_MAX` keywords in the matrix extension of your response file.

The gain operation itself belongs to the category of *response functions*, which in future versions of XSPEC may be defined with **rmodel** just as regular XSPEC model functions are defined with **model**. Though gain is currently the only available response function, the following command will work:

```
// Apply gain to the response belonging to source 2 of spectrum 1
XSPEC12>rmodel 2:1 gain
```

which is equivalent to:

```
XSPEC12>gain fit 2:1
```

The `nofit` argument switches off the fitting and leaves the gain at the current values of the parameters. Unless the argument `all` is given, it is applied to a single response specified by `<specNum>` and optional `<sourceNum>`. As with `gain fit`, both arguments may be omitted if only a single spectrum with one source is loaded. When `all` is specified, fitting is switched off for the gain parameters of all responses. `gain off` will switch off fitting for all gain parameters, and will reset all of them to their nominal value.

Whenever a new response file is defined for a spectrum, the response will return to the `nofit` state with nominal value. The `ignore` and `notice` commands however will not affect the current gain of the response. **The gain command is not currently implemented for dummy responses.**

Examples:

```

XSPEC12>gain 1 0.98
// The response belonging to spectrum 1 is adjusted with a slope of 0.98.
// The 1 may be omitted if only 1 spectrum (with 1 source) is loaded.
XSPEC12>gain 1,,.03
// The offset also is moved now by 0.03 keV.
XSPEC12>gain 2:4 1.1 0.1
// The response belonging to source number 2, spectrum 4, is adjusted with slope 1.1
// and offset 0.1 keV.
XSPEC12>gain off
// The above 2 responses, and any others that have been adjusted, are reset to slope
// 1.0, offset 0.0.
XSPEC12> gain fit 3
// Variable fit parameters are created for spectrum 3 response. User will be prompted
// for starting fit parameter values of slope and offset.
XSPEC12> fit
// Best fit gain values will now be determined for and applied to spectrum 3 response.
XSPEC12> gain nofit 3
// Spectrum 3 response will retain its current gain values, but values will not be
// adjusted during future fits.

```

NOTE: Current gain information may be easily viewed with the `show response` command. Gain fit parameters may also be viewed with the `show par` or `show rpar` commands.

### 11.5.5.1.9.1 Historical Notes

The gain command has been slightly revised for XSPEC12. Previously when a user entered a gain command, it was generally interpreted to apply to an entire **model**. This new implementation clearly defines an applied gain as belonging to a particular **response**. It also offers less ambiguity for dealing with XSPEC12's multiple models scheme. So for example if 2 spectra are loaded, each in its own data group, and the user enters a `gain fit` command, under the old system they would be prompted for 2 sets of parameters since the model is applied to 2 data groups. With the new system, the user specifies which particular response (belonging to either spectrum 1 or 2) they wish to apply the gain fit to, and are then prompted for just the one set of gain parameters for that response. This is more clearly demonstrated with the examples below. The new command options `gain nofit all` and `gain off` are also described below.

**NOTE: Backwards incompatible syntax change.** Beginning with XSPEC 12.5.1, **gain** parameters must be specified as [`<sourceNum>`]:`<specNum>` and *not* `<specNum>`[`:<sourceNum>`]. This reversal was made so that the **gain** command conforms to the [`<sourceNum>`]:`<specNum>` usage in other XSPEC commands, such as **response** and **arf**.

It looks like this option applies to a tag, in ATLAS nomenclature, that is, to the response associated with a single spectrum. Hence a **Gain...** button would be added to the bottom of each tag, as displayed in the tag tool. It displays a dialog that selects either Fit or a pair of manually entered slope/intercept values. A checkbox enables or disables the altered gain value. This approach means that the gain value is applied to all models that use the tag, or use tag groups of which the tag is a member.

### 11.5.5.1.10 identify: identify spectral lines

List possible lines in the specified energy range.

```
identify <energy> <delta energy> <redshift> <line_list>
```

The energy range searched is `<energy> ± <delta energy>` (keV) in the rest frame of the source. If working in wavelength mode, as set by the `setplot` command, then the `<energy>` and `<delta energy>` parameters should be entered as wavelengths (in Ångströms). `<line list>` specifies the list of lines to be searched. The options are `bearden`, which searches the Bearden compilation of fluorescence lines (Bearden, J.A., 1967, Rev.Mod.Phys. 39, 78), `mekal`, which uses the lines from the mekal model (q.v.) and `apec`, which uses the APEC <http://cxc.harvard.edu/atomdb> line list. The `apec` option takes an additional two arguments: the temperature of the plasma (keV) and a minimum emissivity of lines to be shown. If the command `xset` has been used to set `APECROOT` then `identify` uses the `APECROOT` value to define the new atomic physics data files. See the help on the `apec` model for details.

To first order at least, this is done with the emission lines list shown in the plot tool sidebar. However, this lists lines

from AtomDB, not those from bearden mekal if desired. This could probably be added without too much trouble.

The plot tool's selection mechanism is intended in part to allow selection of lines of interest, to which a tool like identify may be applied. Each selection has a selection type. "Single emission line" is one such type, and this can be passed to identify to produce a line label. The line label appears over the peak, rotated almost 90°. Another selection type is "Multiple emission lines," a request to perform Line-Based Analysis over a range and identify as many lines as possible. This is a future. Absorption lines are also planned. Note that the redshift parameter is set via the emission lines display in the plot tool sidebar.

Is this redundant with `tcplot idline`?

### 11.5.5.1.11 `initpackage`: initialize a package of local models

The `initpackage` command initializes a package of local models from their source code and from a model component description file in "model.dat" format which defines the component's name, type, function call, and its parameter names and initial settings. Further details of the file format, function and parameter specifications are given in Appendix C, [Adding Local Models To XSPEC](#). [Note: `initpackage` is now also supported on **Cygwin**. The former Cygwin-only `static_initpackage` command has been removed.]

**initpackage** <name> <description file> [<directory>] [-udmget]

The <name> argument names the package. For internal reasons package names must be lowercase: the `initpackage` command will force lower case and warn the user if the argument contains uppercase letters. Also there should be no numerals in the package name.

The <description file> argument specifies the model component description file. The third argument <directory> is optional and specifies the location of the source code. If it is not given, the value of the setting `LOCAL_MODEL_DIRECTORY` given in the user's `xspec.init` file will be used. Finally, the <description file>, if not specified as an absolute pathname, will be read from the same directory as the source code.

Another optional argument is `-udmget`, for local model libraries containing Fortran code which makes use of XSPEC's now-obsolete `udmget` function for dynamic memory allocation. None of the functions in XSPEC's built-in models library use `udmget` anymore, and the necessary `xsudmget.cxx` file no longer resides there. If a user still requires this code for their own local models, they should add `-udmget` at the end of the command line. `initpackage` will then copy the files `xsudmget.cxx` and `xspec.h` into the user's local model directory.

`initpackage` performs the following tasks:

- reads the model description file
- writes code that will load the new component calculation functions
- writes a makefile that will drive the compilation and installation of the new code
- invokes the compiler and builds the library.

A separate command, `lmod`, actually loads the library. This two step process makes it easier to determine where the user is during the process if compilation failures arise. Further, if the model is complete and working correctly, only the `lmod` command need be invoked.

`initpackage` can also be run as a stand-alone program outside of XSPEC. When used like this however, after `initpackage` has finished the user must manually run `hmake` to build their library. XSPEC performs this part automatically using a script file.

In looking at this, it occurs to me that the existing Python program development environment could be extended to allow creation of C++ and/or Fortran model files. The files could be edited from within ATLAS, and ATLAS would then issue the necessary commands to build and load the model libraries. The same partially realized mechanism that includes necessary Python files in saved archives could be used to make the models available to others, packaged transparently inside the notebook. A recipient could run a notebook containing a custom model without even knowing there *was* such a model, provided that their program development environment supported the necessary build procedures. And since a new model could be included only when all XSpecs are idle, ATLAS could automatically load the new models and pick up execution again without missing a beat. This would be a nontrivial subproject, but the result would be a very seamless model extension mechanism. Models created in this way could also be used by the command line version of XSpec.

### 11.5.5.1.12 **lmod, localmodel: load a package of local models**

The **lmod** command (**localmodel** is an alias for this command) loads a user model package. Further details are given in Appendix C. [Note: This command is now also supported on **Cygwin**.]

**lmod** <name> [directory]

As for **initpackage**, the <name> argument is the name of the model package being loaded, and the <directory> is its location, defaulting to the setting of LOCAL\_MODEL\_DIRECTORY given in the user's `Xspec.init`.

**lmod** performs the following tasks:

- loads the library corresponding to the package named <name>
- reads the model description file supplied by the **initpackage** command for the library
- adds the new model components to the list of models recognized by the model command.

Note that **lmod** requires that the user has write access to <directory> (please see Appendix C for details).

User-defined model support for ATSA is discussed under **initpackage**.

### 11.5.5.1.13 **lumin: calculate luminosities**

Calculate the luminosity of the current model for a given redshift and rest frame energy range.

**lumin** [<lowEnergy> [<hiEnergy>] [<redshift>] [err <number> <level> |noerr]

where <low Energy> and <hi Energy> are the rest frame energies over which the luminosity is calculated, and <redshift> is the source redshift. Initial default values are 2 to 10 keV for 0 redshift. The luminosity is given in units of ergs/s. The energy range redshifted to the observed range must be contained by the range covered by the current spectra (which determine the range over which the model is evaluated). Values outside this range will be automatically reset to the extremes. Note that the energy values are two separate arguments and are *not* connected by a dash (see parameter ranges in the **freeze** command).

The **lumin** will be calculated for all loaded spectra. If no spectra are loaded (or none of the loaded spectra have a response), the model is evaluated over the energy range determined by its dummy response. (In XSPEC12, models are automatically assigned default dummy responses when there is no data, so the **dummyrsp** command need not be given.) If more than one model has been loaded, whichever model the user has specified to be the active one for a given source is the one used for the **lumin** calculation.

The results of a **lumin** command may be retrieved by the `tblout lumin <n>` command, where *n* is the particular spectrum of interest. If **lumin** was calculated for the case of no loaded spectra, the results can be retrieved by `tblout lumin` with the <n> argument omitted.

The `err/noerr` switch sets whether errors will be estimated on the luminosity. The error algorithm is to draw parameter values from the distribution and calculate a luminosity. <number> of sets of parameter values will be drawn. The resulting luminosities are ordered and the central <level> percent selected to give the error range. You can get the full array of simulated **lumin** values by calling `tblout lumin` with the `errsims` option (see **tblout** command).

The parameter values distribution is assumed to be a multivariate Gaussian centered on the best-fit parameters with sigmas from the covariance matrix. This is only an approximation in the case the fit statistic space is not quadratic.

Examples:

The current data have significant response to data within 1 to 18 keV.

```
XSPEC> lumin,,,0.5
// Calculate the current model luminosity over the default range for z=0.5
XSPEC> lumin 6.4 7.0
// Calculate the current luminosity over 6.4 to 7 keV.
```

It looks like this model-specific option is used rarely enough so it shouldn't compete for space in the sidebar's model pane. Instead, it could be added (with several similar options) to the top of the parameters pane. A Luminosities... button prompts for the specific parameters. An enable checkbox beside it enables the calculation. Next, the last calculated luminosity is shown. If the dialog is opened, the set of values are shown as a list, along with other associated information. This stuff is placed in the dialog to conserve screen real estate.

### 11.5.5.1.14 mdefine: Define a simple model using an arithmetic expression

**mdefine** [`<name>` [`<expression>` [: [`<type>`] [`<emin>` `<emax>`]]]

where `<name>` is the name of the model. If `<name>` was previously defined via **mdefine**, the current definition will overwrite the old one, and the user is warned; if it is a built-in model, however, the user will be asked to use a different name.

`<expression>` is an arithmetic expression. Simple rules for expression:

1. The energy term must be 'e' or 'E' in the expression. Other words, which are not numerical constants nor internal functions, are assumed to be model parameters.
2. If a convolution model varies with the location on the spectrum to be convolved, the special variable ".e" or ".E" may be used to refer to the convolution point.
3. The expression may contain spaces for better readability.

`<type>` optionally specifies the type of the model. The valid types are `add`, `mul`, and `con`). (Mix models are not yet implemented as of v12.5.0.) Please note that the character ":" must be used to separate the options from `<expression>`. If `<type>` is not given, the default is `add`.

Users may also specify the minimum and maximum energy values for the model, via `<emin>` and `<emax>`. The default values are `1.e-20` and `1.e+20`, respectively.

Note that **mdefine** can also be used to display and delete previously defined models:

- `mdefine` displays the name, type and expression of all defined models.
- `mdefine <name>` displays the same information for model `<name>`.
- `mdefine <name> :` deletes model `<name>`

#### 11.5.5.1.14.1 Operators

The following operators are recognized in an expression:

- + plus operator
- minus operator
- \* multiply operator
- / divide operator
- \*\* exponentiation operator
- ^ exponentiation operator

#### 11.5.5.1.14.2 Functions

The following internal functions are supported:

EXP( <code>&lt;expr&gt;</code> )	exp of a vector expression
SIN( <code>&lt;expr&gt;</code> )	sine of a vector expression in radians

SIND(<expr>)	sine of a vector expression in degrees
COS(<expr>)	cosine of a vector expression in radians
COSD(<expr>)	cosine of a vector expression in degrees
TAN(<expr>)	tangent of a vector expression in radians
TAND(<expr>)	tangent of a vector expression in degrees
LOG(<expr>)	log <sub>10</sub> of a vector expression
LN(<expr>)	natural log of a vector expression
SQRT(<expr>)	sqrt of a vector expression
ABS(<expr>)	absolute value of a vector expression
INT(<expr>)	integer part of a vector expression
ASIN(<expr>)	sin <sup>-1</sup> of a vector expression in radians
ACOS(<expr>)	cos <sup>-1</sup> of a vector expression in radians
MEAN(<expr>)	mean value of a vector expression
DIM(expr)	dimension of a vector expression
SMIN(expr)	minimum value of a vector expression
SMAX(expr)	maximum value of a vector expression
MAX(<expr1>, <expr2>)	maximum of the two vector expressions
MIN(<expr1>, <expr2>)	minimum of the two vector expressions

### Examples:

```

XSPEC12> mdef dplaw E**p1 + f**E**p2
// define a model named "dplaw" with 3 parameters, p1, p2, f

XSPEC12> mdef junk a*e+b*log(e)/sin(e)
// define a model named "junk" with 2 parameters (a, b)

XSPEC12> mdef junk2 exp(-a*e) : mul
// define a model named "junk2" with 1 parameter, a; the option
// following ":" says that it will be a multiplicative model.

XSPEC12> mdef junk3 0.2+B*e : mul
// define a model named "junk3" with 1 parameter, B, options following
// ":" says that this will be a multiplicative model

XSPEC12> mdef bb E**2/T**4/(exp(E/T)-1)
// try to define a blackbody model with name "bb", you get warning:
// ***Warning: bb is a pre-defined model
// Please use a different name for your model.

XSPEC12> mdef sg exp(-E^(2/(2*A*.E)) / sqrt(6.283*A*sqrt(.E)) : con
// this defines a Gaussian convolution model with sigma varying with square root of energy.

XSPEC12> mdef junk2 :
// delete junk2

XSPEC12> mdef
// display all user-defined models
// -- Name ---- Type ----- Expression -----
// dplaw      add      E**p1+f**E**p2
// junk       add      a*e+b*LOG(E)/SIN(E)
// junk3      mul      a+b*e
// sg         con      EXP(-E^2/(2*A*.E))/SQRT(6.283*A*SQRT(.E))
// -----

```

### 11.5.5.1.14.3 Proposed Unified Syntax

It would be nice if the **mdefine** syntax for defining a simple model could be combined with the **model** syntax for a

model expression. This makes it possible to intersperse references to pre-defined models, user-defined models, and arithmetic expressions defining models in a single model expression, and removes the need for two independent mechanisms for defining and editing models. To that end, I propose some modifications to the syntax rules for both. These rules are presently meant for AT&SAL's model expression editor. The resulting expressions are transformed into the format expected by XSpec. An *arithmetic expression* is an expression composed of the arithmetic operators accepted by **mdefine**, while a *model expression* is an expression composed of models as accepted by the **model** command. A model expression contains models, arithmetic expressions that evaluate to models, and user models connected by model operators (+ and \*), as well as the **bg** function.

An arithmetic expression:

- Conforms to the existing **mdefine** conventions, except as follows.
- Specifies its type via:
  - `add(<expr> [, [<emin>], [<emax>]])`
  - `mul(<expr> [, [<emin>], [<emax>]])`
  - `con(<expr> [, [<emin>], [<emax>]])`
 <emin> and <emax> specify energy range limits. <expr> is a valid **mdefine** expression.
- May refer to other user-defined models, provided they are arithmetic, that is, that they are composed of an arithmetic expression, not a model expression.
- May *not* contain references to models or user models containing a model expression.

A model expression:

- Conforms to the existing rules for a model expression, except as follows.
- May not contain `/b`, appended to a model component, to indicate that the component is a background model. Instead, this is specified via the extension `bg()`.
- May not connect components by implicit operators, so e.g. `m1 a2` is illegal, and must be expressed as `m1*a2`. (I believe this older syntax is already illegal.)
- Does not allow multiplication implied via the use of parentheses. That is, `m1 (a1+a2)` must be expressed as `m1*(a1+a2)`. This makes distinguishing between a multiply and a function invocation easier, leading to better error messages during parsing.
- Cannot abbreviate model names. (There are a couple of reasons for this. Forward compatibility is the obvious one, but abbreviation complicates the model expression editing process by increasing the chance of losing previously supplied parameters.)
- May contain other user-defined models.
- A user-defined model name may contain numerals, although it may not begin with a numeral. `model12` is legal, but `5model` is not.

Examples:

Assume user-defined model `model11` is defined as `E**p1+f*E**2`, and `model12` is `phabs*powerlaw`.

<i>Expression</i>	<i>Expression Type</i>	<i>Description</i>
<code>E**p1+f*E**2</code>	Arithmetic, additive	Defines a model that defaults to additive
<code>a*E+b*LOG(E)/SIN(E)</code>	Arithmetic, additive	Also defaults to additive
<code>mul(a+b*E)</code>	Arithmetic, multiplicative	Defined as a multiplicative model. Differs from <b>mdefine</b> in how the model type is declared.
<code>con(EXP(-E^2/(2*A*.E))/SQRT(6.283*A*SQRT(.E)))</code>	Model, convolution	Convolution model. Differs from <b>mdefine</b> in how the model type is declared.
<code>SIN(E)*phabs</code>	Illegal	Attempt to combine an arithmetic expression with a model expression, '*' is ambiguous.
<code>add(sin(E))*phabs</code>	Model, multiplicative	Above ambiguity resolved, because the type of <code>add(sin(E))</code> is additive model expression, not arithmetic expression.
<code>phabs powerlaw</code>	Illegal	Deprecated syntax.
<code>wabs*phabs*powerlaw</code>	Model, multiplicative	

wabs(phabs)powerlaw	Illegal	Operators must be explicit.
wabs(phabs*powerlaw)	Illegal	Operators must be explicit.
pha(po)	Illegal	Model names cannot be abbreviated.
phabs*(powerlaw)+gauss+powerlaw	Model, additive	
phabs*(powerlaw)+gauss+powerlaw/b	Illegal	/b syntax is not supported, use bg( )
phabs*(powerlaw)+gauss+bg(powerlaw)	Model, additive	Correct way to indicate background component
phabs*(powerlaw)+a*E	Illegal	Cannot combine arithmetic and model expression
phabs*(powerlaw)+ add(a*E)	Model, additive	Resolves above problem
model1+model1	Model, additive	Equivalent to add(E**p1+f*E**2)+add(E**p1+f*E**2)
model2*model1	Model, additive	Equivalent to phabs*powerlaw+add(E**p1+f*E**2)
sin(E)*model1	Model, additive	Equivalent to sin(E)*(E**p1+f*E**2).
sin(E)*model2	Illegal	model2 is not an arithmetic expression, so it cannot be part of an arithmetic expression.

It is possible that a later version of XSpec might incorporate the same syntax.

### 11.5.5.1.15 model: define a theoretical model

Define the form of the theoretical model to be fit to the data.

```

model [<source num>:<name>] [<delimiter>] <component1> <delimiter> <component2> <delimiter>...
<componentN> [<delimiter>]
model [ ? ]
model [<name>|unnamed] none
model clear
model <name>|unnamed active|inactive
rmodel [<source num>:]<spec num> <response function>|none

```

where <delimiter> is some combination of (, +, \*, ); and <componentJ> is one of the model components known to XSPEC. The optional name must be preceded by a source number followed by a colon. To specifically refer to the default model use the string unnamed. Descriptions of these models may be accessed by typing `help models`.

The source argument and name, if present, assign that model to be used with one of the sources found to be in the spectrum during the data pipelining. These 2 parameters allow one to simultaneously analyze multiple models, each assigned to their own responses. The model will be referred to the channel space using a response corresponding to that source number. To create a model for a source number higher than 1, a detector response must first exist for that number. See the examples below and the **response** command for more information about using multiple sources. [This ability to assign multiple models both generalizes and replaces the XSPEC11 method of using '/b' to specify background models.](#)

After the model is loaded, if there are data present the model is attached through the instrumental response to the spectra to be fitted, as in XSPEC11. Unlike XSPEC11, however, if there are no data loaded the model will be attached to a default diagonal dummy response. The parameters of that dummy response (energy range, number of flux points, linear/logarithmic intervals) can be set by the user in the `xspec.init` file using the `DUMMY` setting. Thus any model can be plotted in energy or wavelength space as soon as it has been defined.

The model components are of various types depending on what they represent and how they combine with other models additive, multiplicative, convolution, pile-up, and mixing models. Each component may have one or more parameters that can be varied during the fit (see the **newpar** command writeup).

- Additive model components are those directly associated with sources, such as power laws, thermal models, emission lines, etc. The net effect of two independent additive models is just the sum of their individual emissivities.
- Multiplicative model components do not directly produce photons, but instead modify (by an energy-dependent multiplicative parameter) the spectrum produced by one or more additive components. Examples of multiplicative models are photoelectric absorption models, edges, absorption lines, etc.
- Convolution models components modify the spectrum as a whole, acting like operators rather than simply applying bin by bin multiplication factors. An example of a convolution model is a gaussian smoothing with energy dependent width. Thus, when using convolution models, the ordering of components is in general significant (see below under **syntax rules**).
- The pile-up model is similar to the operation of the convolution models. The only difference is that the flux is multiplied by the effective area on input and divided by the same factors on output.
- Mixing model components implement two-dimensional transformations of model spectra. The data are divided into regions by assigning them to 2 or more datagroups, and the transformation “mixes” the flux among the regions. An example is the `project` (projection) model, which assumes that the regions are 3-dimensional ellipsoidal shells in space, and projects the flux computed from the other components onto 2-dimensional elliptical annuli.

A list of all the currently installed models is given in response to the command

```
model ?
```

(The ‘?’ is not actually required.) This will leave the current model in use.

The new command variants have the following uses:

```
model [<name>] none
```

removes the model of name `<name>` if given. Without the `<name>` argument, the command removes the unnamed “default” model, which is of course the XSPEC11 behavior.

```
model clear
```

removes all models

```
model <name>|unnamed active|inactive
```

makes the model named `<name>` active (fit to data) or inactive. Inactive models are tied to a dummy (unit diagonal) response. Making a model assigned to a given source active makes any previous model assigned to that source inactive. Note that to make the default unnamed model active or inactive refer to it by the string `unnamed`.

See the commands **delcomp**, **addcomp** and **editmod** for details on how to modify the current model without having to enter a completely new model.

```
rmodel [<source num>:]<spec num> <response function>|none
```

assigns or removes a response function to the response belonging to `<source num>` of spectrum `<spec num>`. Currently the only available `<response function>` in XSPEC is `gain`, which makes **rmodel** redundant with the **gain** command usage:

```
gain fit [<source num>:]<spec num>
```

The `rmodel none` option removes the response function and restores the response to its initial state.

### 11.5.5.1.15.1 Syntax Rules

Model components are combined in the obvious algebraic way, with `+` separating additive models, `*` separating multiplicative models, and parentheses to show which additive models the multiplicative models act on. The `*` need not be included next to parentheses, where it is redundant. Also, if only one additive model is being modified by one or more multiplicative models, the required brackets may be replaced by a `*`. In this case the additive model must be the last component in the grouping. Thus

$$M_1*(A_1+A_2) + M_2*M_3(A_3) + M_4*A_4 + A_5$$

is a valid model, where the *M*'s signify multiplicative models and the *A*'s additive models.

The old style syntax for entering models (versions 9.02 and earlier) is not supported in version 12 and will return a syntax error.

XSPEC12's recursive lexical analyzer and expression parser allows, in principle, infinite nesting depth. It has been tested to 3 levels of parentheses, although it should be said that this new behavior is a by-product of the design rather than fulfilling an important need. Thus, expressions such as

$$M_1*(A_1 + A_2*(A_3 + M_2*M_3*(A_6 + A_7))) + C_1*(A_8 + A_9*(A_{10} + M_2*A_6))$$

are supported.

The model expression is analyzed on entry and syntax errors, or undefined models, will return control to the prompt with an error message. XSPEC12's model definition algorithm treats expressions delimited by '+' signs that are not within parentheses as separate "Component Groups". The Component Group comprises a list of components of the different types, and these are in turn calculated and then combined to produce an internal "Sum Component". These Sum Components from each such component group are then added to produce the output model (note that if there is an overall component—for example, a convolution or mixing component—then all of the model will be contained inside one Component Group).

The syntax rules that are checked for are as follows:

- Expression must not begin with a "\*"
- A "\*" must be preceded and followed by words or a brace (redundant braces are removed).
- A standalone component must be additive. A standalone component is defined as a single component model or a single component at the beginning (end) of the expression followed (preceded) by a "+", or in the middle of the expression delimited by 2 "+" signs.
- A convolution or mixing component must not appear at the end, or followed by a closing brace.
- When using convolution or mixing components, the order in which they are applied is in general significant. For example, the two models

$$C_1*M_1(A_1+A_2) \text{ and } M_1*C_1(A_1+A_2)$$

are not necessarily equivalent (here the *C*'s represent convolution models). The way XSPEC handles the ordering of components is by first computing the spectrum for the additive components of a given additive group ( $A_1+A_2$  in the above example). It then applies all multiplicative or convolution components in the additive group from right to left in the order they appear in the model formula.

N.B. Beginning with v12.5.0, convolutions no longer have to precede the source. Parentheses may also be used to specify convolution precedence, so the following two examples are not equivalent:

$$C_1*M_1(A_1+A_2) \text{ and } (C_1*M_1)(A_1+A_2)$$

### 11.5.5.1.15.2 Examples

Note that *po* (= powerlaw) and *ga* (= gauss) are additive models, and that *wabs* and *phabs* (different photoelectric absorption screens) are multiplicative models.

```
XSPEC12> model po
// The single component po (powerlaw) is the model.
XSPEC12> model po+ga
XSPEC12> model (po+ga)wabs
XSPEC12> model phabs(po+ga)
XSPEC12> model wa(phabs(po)+ga)
XSPEC12> model wa po phabs ga //error: old syntax
XSPEC12> model wa*phabs*po
XSPEC12> model (po+po)phabs
// Note that though the first and second components are the same form, their parameters are varied
```

```
// separately.  
XSPEC12> model phabs*wa(po)
```

A complex (and almost certainly unphysical) example is the following:

```
XSPEC12> model wa(po+pha(peg+edge(disk+bbod)))const + pla(pos+hr*step) + not*gau
```

Applying multiple models:

Assume 3 spectra are loaded, each with a single response (source 1 by default).

```
XSPEC12> model wa(po)  
// The unnamed model wa(po) will apply to all 3 spectra, accordingly  
// multiplied by each spectrum's response.  
XSPEC12> response 2:2 new_resp.pha 2:3 another_new_resp.pha  
// Additional responses assigned to source number 2 for spectra 2 and 3.  
XSPEC12> model 2:second_mod ga  
// The model "second_mod" will now apply to source 2, and is therefore  
// multiplied by new_resp.pha and another_new_resp.pha for spectra 2 and 3 respectively.  
XSPEC12> model second_mod inactive  
// "second_mod" will no longer apply to spectra 2 and 3, though they retain responses for  
source 2.
```

Or:

```
XSPEC12> response 2:2 none  
XSPEC12> response 2:3 none  
// No responses exist for source number 2, second_mod is rendered inactive.
```

Handled by the [model expression editor](#).

### 11.5.5.1.16 modid: write out possible IDs for lines in the model.

Tcl script to write out possible IDs for gaussian or lorentzian lines in the current model.

```
modid [<delta> | conf]
```

This script runs the identify command for every gaussian or lorentzian line included in the current model. If a number is given as an argument then that is used as the delta energy for identify. If the string `conf` is given as the argument then the last calculated confidence regions are searched for possible line IDs. If no argument is given then `conf` is assumed.

[ATSAL's implementation of this is described under `identify`](#).

### 11.5.5.1.17 newpar: change parameter values

Adjust one or more of the model parameters.

```
newpar [modelName:]<index range> [<param spec list>]  
newpar [modelName:]<index> = <coupling expression>  
newpar 0
```

where

```
<param spec list> ::= <param value> <delta> <param range spec>  
<param range spec> ::= <hard min> <soft min> <soft max> <hard max>
```

**For response parameters** (created with the `gain` or `rmodel` command):

**rnewpar** [`<param spec list>`]

**rnewpar** [`<sourceNum>:`]`<index>` = `<coupling expression>`

The model parameters are accessed through their model parameter indices. For example, the first parameter of the first model component generally is model parameter 1, etc. The first command line argument, `<index range>`, gives the indices' parameters to be modified by the **rnewpar** command. The default value is the range from the previous invocation of **rnewpar**. The remaining arguments can be used to update the parameter specification. If the parameter specification is omitted from the command line, then the user is explicitly prompted for it.

The `<param spec list>` consists of the following:

<code>&lt;param value&gt;</code>	The trial value of the parameter used initially in the fit.
<code>&lt;delta&gt;</code>	The step size used in the numerical determination of the derivatives used during the fitting process. When delta is set to zero, the parameter is not adjustable during the fit. This value may be overridden for all parameters by the <code>xset delta</code> command option, which will apply a proportional rather than a fixed delta.
<code>&lt;param range spec&gt;</code>	The four arguments of the range specification determine the range of acceptable values for the parameter. The soft limits, <code>&lt;soft min&gt;</code> and <code>&lt;soft max&gt;</code> , should include the range of expected parameter behavior. The parameter is never allowed to have a value outside the hard limits, <code>&lt;hard min&gt;</code> and <code>&lt;hard max&gt;</code> . Between the hard and soft limits, the parameter is made stiffer to adjustment by the minimization routine invoked by the fit command.

A slash (/) will set all the six parameter specification values (value, delta, range specification) to the previous value (default for a new model, current value if the parameter has previously been set or fit).

The sequence `/*` leaves all parameters unchanged (in the case of a new model, to be set to the default).

`rnewpar 0` prints the current parameter settings.

### 11.5.5.1.17.1 Parameter Links

Coupling of parameters allows parameters in a model to always have the same value or to be related by an expression. The expression is a function of the other parameters (XSPEC will reject attempts to link parameters to themselves!). Scale parameters (i.e. never variable during a fit), and switch parameters (i.e. that change the mode in which a component is calculated) can only be linked to other scale and switch parameters, respectively. Details of parameter types are explained in more detail in Appendix C.

The syntax for linking parameters is

`XSPEC12> rnewpar <par> =  $f(\mathbf{par})$ ,`

where  $f$  is a function in the (other) parameters. Parameters can be specified either by the character "p" followed by the parameter number (preferred) or by the parameter number. Integers appearing in  $f$  that are within the range of existing parameter numbers will be interpreted as parameters: to avoid confusion, if a real number is intended it should include a decimal point. Integers larger than the last parameter number will be interpreted as integers. Parameters of named models must be prefixed by `[modelName:]`.

The following operators and functions can be used in  $f$ :

#### Operators:

- + plus operator
- minus operator
- \* multiply operator
- / divide operator
- \*\* exponentiation operator

^ exponentiation  
operator

### 11.5.5.1.17.2 Functions

The following internal functions are supported:

EXP(<expr>)	exp of a vector expression
SIN(<expr>)	sine of a vector expression in radians
SIND(<expr>)	sine of a vector expression in degrees
COS(<expr>)	cosine of a vector expression in radians
COSD(<expr>)	cosine of a vector expression in degrees
TAN(<expr>)	tangent of a vector expression in radians
TAND(<expr>)	tangent of a vector expression in degrees
LOG(<expr>)	$\log_{10}$ of a vector expression
LN(<expr>)	natural log of a vector expression
SQRT(<expr>)	sqrt of a vector expression
ABS(<expr>)	absolute value of a vector expression
INT(<expr>)	integer part of a vector expression
ASIN(<expr>)	$\sin^{-1}$ of a vector expression in radians
ACOS(<expr>)	$\cos^{-1}$ of a vector expression in radians
MEAN(<expr>)	mean value of a vector expression
MAX(<expr1>, <expr2>)	maximum of the two vector expressions
MIN(<expr1>, <expr2>)	minimum of the two vector expressions

If there are multiple data groups present, then the parameters of models associated with datagroups greater than 1 (“secondary models”) are coupled by default to their “primary” counterparts. For example, if there are 5 parameters in the model and 3 datagroups present, then the model command will prompt for 15 parameters. If the user types

```
XSPEC12> model <expression>  
XSPEC12>/*
```

then parameters 1-5 will be set to their values specified in the initialization (model.dat) file. Parameters 6-15 will be linked to their counterparts, i.e. as if the user had typed

```
XSPEC12> newpar 6 = p1  
XSPEC12> newpar 7 = p2  
...  
XSPEC12> newpar 11 = p1
```

And so on.

Examples:

The total number of model parameters for the example is four.

```
XSPEC12> newpar 2 0.1  
// The value of the second parameter is set to 0.1.  
XSPEC12> newpar 3-4  
// The program will prompt for a specification for the 3rd  
// parameter (comp gives the name of the corresponding model component)  
comp:param3> 0.001, 0  
// which has its value set to 0.001 and its delta set to zero, fixing
```

```
// it in later fits. The program now prompts for a specification for
// the 4th parameter
comp:param4> 21
// which is set to 21. As there is no 5th parameter, the program
// displays a summary and returns to command level.
XSPEC12> newpar ,,0.001
// The value of the delta of the 3rd parameter (which is the default
// index as it was the first parameter modified in the previous
// newpar invocation) is set to 0.001, allowing it to be adjusted
// during any fits.
```

The total number of parameters for this example is eight.

```
XSPEC12> newpar 4 = 1
// The value of parameter 4 is set to the value of parameter 1. This has the consequence of
// model
// parameter 4 being frozen at the value of parameter 1 during subsequent fitting procedures.
// If model parameter 1 is a free parameter, then both parameters 1 and 4 change their values
// simultaneously in the fit procedure.
XSPEC12> newpar 4 = p3/5 + 6.7
// The value of parameter 4 is set to the value of (parameter 3/ parameter 5) plus 6.7
XSPEC12> newpar 6 = p3 * 0.1 - 9.5
// The value of parameter 6 is set to 0.1 times the value of parameter 3 minus 9.5
XSPEC12> newpar 5 = 2 + 5.
// The value of parameter 5 is set to the value of parameter 2 plus 5.
XSPEC12> newpar 8 = p1 / 4.6
// parameter 8 is set to parameter 1 divided by 4.6
XSPEC12> newpar 8 = abs(p1^3) / 2.0
// parameter 8 is set to the absolute value of the cube of parameter 1 divided by 2.0
XSPEC12> newpar 5 = cos(p1) + sin(p3)
// parameter 5 is set to the cosine of parameter 1 plus the sine of parameter 3
XSPEC12> newpar 3 = log(mymodel:p1)
// parameter 3 is set to the log (base 10) of parameter 1 in the mymodel model
XSPEC12> untie 6
// Makes parameter 6 independent of parameter 3 and a free parameter.
```

This function is performed using the Parameters... button to edit the current model's parameters. Although not yet implemented, the mechanism to derive parameters from other values will be more general, because any Python expression may be used.

### 11.5.5.1.18 systematic: add a model-dependent systematic term to the variance

**systematic** [<model systematic error>]

Set a systematic error term on the model to be added in quadrature to that on the data when evaluating chi-squared. The default value is zero.

Not sure where this goes.

### 11.5.5.1.19 untie/runtie: unlink previously linked parameters

Untie the specified parameter from any links to other parameters.

**untie** <param range>

where <param range> is of the form

<param range> ::= [modelName:]<param #>

**For response parameters** (see **gain** command):

**runtie** <param range>

where <param range> is of the form

<param range> ::= [sourceNum:]<param #>

Parameters previously linked together with commands such as `newpar <param spec>` are unlinked. The parameter will retain its current value for the next fit.

This is done by editing the parameter directly.

## 11.5.6 Plot Commands

### 11.5.6.1.1 cpd: set current plotting device

`cpd <plot device>`

`cpd <filename>`

`cpd <filename>/{ps,cps,vps,vcps}`

`cpd none`

Set current plot device. The same can be achieved with the `setplot device` command, which takes the same options. In XSPEC12 as in previous versions, the plot device options are those allowed by the PGPLOT library.

When plotting to the screen, the most commonly used devices are `/xs (/xserve)` and `/xw (/xwindow)`. If you select `/xs`, the plot window is persistent: it remains visible and in the selected position even after the XSPEC session is finished. With `/xw` the plot window closes at the end of the XSPEC session. Also note that on some platforms, when using `/xs` in multiple desktops, you might not see the window appear in a second desktop if it is still open in the first.

If the second argument does not start with a '/' character, which indicates that the string represents a PGPLOT device, it is taken to be a filename for Postscript output, and the default postscript driver will be used. The default postscript driver produces a monochrome plot in landscape orientation.

The filename argument can be followed by a '/' that specifies a particular postscript driver variant. Allowable variants are: `cps` (color postscript), `vps` (monochrome portrait orientation), and `vcps` (color portrait orientation), as well as the default, `ps`.

#### 11.5.6.1.1.1 PGPLOT devices

A number of plot device types are supported in XSPEC. PGPLOT devices available on Unix machines are:

<code>/GIF</code>	Graphics Interchange Format file, landscape orientation
<code>/VGIF</code>	Graphics Interchange Format file, portrait orientation
<code>/NULL</code>	Null device, no output
<code>/PPM</code>	Portable Pixel Map file, landscape orientation
<code>/VPPM</code>	Portable Pixel Map file, portrait orientation
<code>/PS</code>	PostScript file, landscape orientation
<code>/VPS</code>	PostScript file, portrait orientation
<code>/CPS</code>	Colour PostScript file, landscape orientation
<code>/VCPS</code>	Colour PostScript file, portrait orientation
<code>/TEK4010</code>	Tektronix 4010 terminal
<code>/GF</code>	GraphOn Tek terminal emulator
<code>/RETRO</code>	Retrographics VT640 Tek emulator
<code>/GTERM</code>	Color gterm terminal emulator
<code>/XTERM</code>	XTERM Tek terminal emulator
<code>/ZSTEM</code>	ZSTEM Tek terminal emulator
<code>/V603</code>	Visual 603 terminal

/KRM3	Kermit 3 IBM-PC terminal emulator
/TK4100	Tektronix 4100 terminals
/VT125DEC	VT125 and other REGIS terminals
/XDISP	pgdisp or figdisp server
/XWINDOW	X window window@node:display.screen/xw
/XSERVE	An /XWINDOW window that persists for re-use

Closes the device. For Postscript output, it flushes the write buffer into the file and closes the file.

Note that in XSPEC12, each plot command produces a separate page in the postscript file, unlike previously where each plot overwrote the previous plot.

Example:

```
// produce a set of color postscript plots in landscape orientation
// ... commands to produce a plot.
XSPEC12> cpd dataplot.ps/cps
XSPEC12> plot data chi
XSPEC12> plot< ufspec
XSPEC12> plot efficiency
XSPEC12> cpd none
```

Will produce 3 plots in the file dataplot.ps.

Note, in contrast, that the hardcopy command will print only the plot that is currently in a graphics frame.

This command does not apply in ATLAS, since the plot tool is the sole plot device.

### 11.5.6.1.2 hardcopy: print plot

Spool the current plot to the printer.

**hardcopy** [<filename>] [mono | color]

This command takes whatever is the current display in you plot window, writes it to a postscript file, and then sends it to a printer using the unix lpr command. It will thus be printed on whatever printer lpr uses as your default printer. If a filename is specified, the postscript file will be saved (e.g. hardcopy dataplot.ps color will produce a color plot saved in the file dataplot.ps). If mono or color is not given, the hardcopy will be monochrome.

In ATLAS, all plots will be generated in color, since a print time option can convert them to monochrome if desired. Users will be able to print the notebook as a whole, or any plot tool window as a whole. In addition, the plot tool or any subset of plot zones may be exported as separate files in a variety of formats, and subsequently printed.

### 11.5.6.1.3 iplot: make a plot, and leave XSPEC in interactive plotting mode

Interactive plotting on the current plot device.

**iplot** <plot type>

This command works like the **plot** command, but allows the user to change the plot and to add text to the plot interactively using the PLT package. See the Overview of PLT in the Appendices for more information.

In normal use, plots are often panned and zoomed, complicating the placement of overlaid text. Header and footer text is supported. Text "anchored" to a particular energy is also supported. It will also (eventually) be possible to generate a snapshot of a plot and incorporate it in an image tool, where it will be a fixed object upon which other text and graphics objects may be layered. This allows further annotations. But these plots will not update automatically in response to changes—they are static objects.

#### 11.5.6.1.4 plot: make a plot

Make one or more plots to the current plot device (see `setplot device`).

**plot** <plot type> [<plot type>] [<plot type>] ...

<plot type> is a keyword describing the various plots allowed. Up to six plot panes can be put on a single page by combining multiple <plot type> options. For example:

```
plot data resid ratio model
```

will produce a 4-pane plot. However contour plots may not be combined with other plots in this manner. When a certain plot type takes additional arguments (e.g. `chain`, `model`), simply list them in order prior to specifying the next plot type:

```
plot chain 3 4 data ufspec
```

In multi-pane plots, XSPEC will determine if two consecutive plot types may share a common X-axis (e.g. `plot data delchi`, or `plot counts ratio`). If so, the first pane will be stacked directly on top of the second. (Note that the small subset of multi-pane plots that were allowed in earlier versions of XSPEC all belonged in this category.)

For changing plot units, see `setplot energy` and `setplot wave`. Also see `iplot` for performing interactive plots.

##### 11.5.6.1.4.1 background

Plot only the background spectra (with folded model, if defined). To plot both the data and background spectra, use `plot data` with the `setplot background` option.

##### 11.5.6.1.4.2 chain

Plot a Monte Carlo Markov chain.

```
plot chain [thin <n>] <par1> [<par2>]
```

Chains must be currently loaded (see `chain` command), and <par1> and <par2> are parameter identifiers of the form [`<model name>: ]n` where *n* is an integer, specifying the parameter columns in the chain file to serve as the X and Y axes respectively. To select the fit-statistic column, enter 0 for the <par> value. If <par2> is omitted, <par1> is simply plotted against row number.

Use the `thin <n>` option to display only 1 out of every *n* chain points. Example:

```
// plot one in five chain points, using parameters 1 and 4 for (X,Y)
plot chain thin 5 1 4
```

The `thin` value will be retained for future chain plots until it is reset. Enter `thin 1` to remove thinning.

##### 11.5.6.1.4.3 chisq

Plot contributions to `chisq`. The contribution is plotted +ve or -ve depending on whether the residual is +ve or -ve.

##### 11.5.6.1.4.4 contour

Plot the results of the last **steppar** run. If this was over one parameter then a plot of statistic versus parameter value is produced while a **steppar** over two parameters results in a fit-statistic contour plot.

```
plot contour [<min fit stat> [ <# levels> [ <levels>]]]
```

where <min fit stat> is the minimum fit statistic relative to which the delta fit statistic is calculated, <# levels> is the number of contour levels to use and <levels> := <level1> ... <levelN> are the contour levels in the deltafit statistic. contour will plot the fit statistic grid calculated by the last **steppar** command (which should have gridded on two parameters). A small plus sign “+” will be drawn on the plot at the parameter values corresponding to the minimum found by the most recent fit.

The fit statistic confidence contours are often drawn based on a relatively small grid (i.e., 5x5). To understand fully what these plots are telling you, it is useful to know a couple of points concerning how the software chooses the location of the contour lines. The contour plot is drawn based only on the information contained in the sample grid. For example, if the minimum fit statistic occurs when parameter 1 equal 2.25 and you use `steppar 1 1.0 5.0 4`, then the grid values closest to the minimum are 2.0 and 3.0. This could mean that there are no grid points where delta-fit statistic is less than your lowest level (which defaults to 1.0). As a result, the lowest contour will not be drawn. This effect can be minimized by always selecting a steppar range that causes XSPEC to step very close to the true minima.

For the above example, using `steppar 1 1.25 5.25 4`, would have been a better selection. The location of a contour line between grid points is designated using a linear interpolation. Since the fit statistic surface is often quadratic, a linear interpolation will result in the lines being drawn inside the true location of the contour. The combination of this and the previous effect sometimes will result in the minimum found by the fit command lying outside the region enclosed by the lowest contour level.

Examples:

```
XSPEC12> steppar 2 0.5 1. 4 3 1. 2. 4
// create a grid for parameters 2 and 3
XSPEC12> plot contour
// Plot out a grid with three contours with
// delta fit statistic of 2.3, 4.61 and 9.21
XSPEC12> plot cont,,4,1.,2.3,4.61,9.21
// same as above, but with a delta fit statistic = 1 contour.}
```

#### 11.5.6.1.4.5 counts

Plot the data (with the folded model, if defined) with the y-axis being numbers of counts in each bin.

#### 11.5.6.1.4.6 data

Plot the data (with the folded model, if defined).

#### 11.5.6.1.4.7 delchi

Plot the residuals in terms of sigmas with error bars of size one. In the case of the cstat and related statistics this plots (data-model)/error where error is calculated as the square root of the model predicted number of counts.

#### 11.5.6.1.4.8 dem

Plot the relative contributions of plasma at different temperatures for multi-temperature models. This is not very clever at the moment and only plots the last model calculated.

#### **11.5.6.1.4.9 eemodel**

See model.

#### **11.5.6.1.4.10 eeufspec**

See ufspec.

#### **11.5.6.1.4.11 efficien**

Plot the total response efficiency versus incident photon energy.

#### **11.5.6.1.4.12 emodel**

See model.

#### **11.5.6.1.4.13 eqw**

Plot the probability density of the most recently run eqwidth calculation with error estimate.

#### **11.5.6.1.4.14 eufspec**

See ufspec.

#### **11.5.6.1.4.15 goodness**

Plot a histogram of the statistics calculated for each simulation of the most recent 'goodness' command run.

#### **11.5.6.1.4.16 icounts**

Integrated counts and folded model. The integrated counts are normalized to unity.

#### **11.5.6.1.4.17 insensitiv**

Plot the insensitivity of the current spectrum to changes in the incident spectra (experimental).

#### **11.5.6.1.4.18 lcounts**

Plot the data (with the folded model, if defined) with a logarithmic y-axis indicating the count spectrum

#### **11.5.6.1.4.19 ldata**

Plot the data (with the folded model, if defined) with a logarithmic y-axis.

#### **11.5.6.1.4.20 margin**

Plot the probability distribution from the results of the most recently run **margin** command. (Must be a 1-D or 2-D distribution.)

#### **11.5.6.1.4.21 model, emodel, eemodel**

Plot the current incident model spectrum (Note: This is NOT the same as an unfolded spectrum.) The contributions of the various additive components are also plotted. If using a named model, the model name should be given as an additional argument. **emodel** plots  $Ef(E)$  or, if plotting wavelength,  $\lambda f(\lambda)$ . **eemodel** plots  $E^2f(E)$ , or if plotting wavelength,  $\lambda^2f(\lambda)$ . The  $E$  (or  $\lambda$ ) used in the multiplicative factor is taken to be the geometric mean of the lower and upper energies of the plot bin.

#### **11.5.6.1.4.22 ratio**

Plot the data divided by the folded model.

#### **11.5.6.1.4.23 residuals**

Plot the data minus the folded model.

#### **11.5.6.1.4.24 sensitvty**

Plot the sensitivity of the current spectrum to changes in the incident spectra (experimental).

#### **11.5.6.1.4.25 sum**

A pretty plot of the data and residuals against both channels and energy.

#### **11.5.6.1.4.26 ufspec, eufspec, eeufspec**

Plot the unfolded spectrum and the model. The contributions to the model of the various additive components also are plotted. **WARNING!** This plot is not model-independent and your unfolded model points will move if the model is changed. The data points plotted are calculated by  $D^*(\text{unfolded model})/(\text{folded model})$ , where  $D$  is the observed data, (unfolded model) is the theoretical model integrated over the plot bin, and (folded model) is the model times the response as seen in the standard plot data. **uufspec** plots the unfolded spectrum and model in  $Ef(E)$ , or if plotting wavelength,  $\lambda f(\lambda)$ . **eeufspec** plots the unfolded spectrum and model in  $E^2f(E)$ , or if plotting wavelength,  $\lambda^2f(\lambda)$ . The  $E$  (or  $\lambda$ ) used in the multiplicative factor is taken to be the geometric mean of the lower and upper energies of the plot bin.

*ATSAL's plot tool provides these fuctions. At least that's the plan. It wouldn't hurt to rank these by importance.*

### 11.5.6.1.5 setplot: modify plotting parameters

Set one of the various plot options.

**setplot** <subcommand string>

where <subcommand string> is a keyword followed in some cases by arguments. Current settings of all **setplot** items can be viewed with **show plot**.

#### 11.5.6.1.5.1 add

Show individual additive model components on the data plots. The opposite is `setplot noadd`.

#### 11.5.6.1.5.2 area, noarea

After `setplot area` is entered, `plot data` and `plot ldata` will show the data divided by the response effective area for each particular channel. `plot residuals` will necessarily also be affected by this. Usual plotting is restored by `setplot noarea`. If data is associated with more than 1 response, the response effective area is calculated by simply summing the contributions from each response.

#### 11.5.6.1.5.3 background, nobackground

When running `plot data` or `plot ldata`, also show associated background spectra (if any).

#### 11.5.6.1.5.4 channel

Change the x-axis on data and residual plots to channels.

#### 11.5.6.1.5.5 command

Add a PLT command to the command list.

```
setplot command <PLT command>
```

where <PLT command> is any valid PLT command. Every time you use `setplot command`, that command is added to the list that is passed to PLT when you use **plot** or **iplot**. The most common use of `setplot command` is to add a common label to all plots produced. You should be careful when using this command, because XSPEC does not check to see if you have entered a valid PLT command. These commands are appended to the list that XSPEC creates to generate the plot and so `setplot command` will override these values (this can either be a bug or a feature, depending on what you have done!) See also `setplot delete` and `setplot list`.

Example:

```
XSPEC12> setp co LA OT Crab {Add the label "Crab" to future plots.}
```

#### 11.5.6.1.5.6 delete

Delete a PLT command from the command list.

```
setplot delete [all | <command #>-<command #> | <command #>]
```

where <command #> is the number of a PLT command that had been entered previously using `setplot` command. This command is used to delete commands from the list passed to PLT when you use the XSPEC **plot** or **iplot** commands.

Example:

```
XSPEC12> setp co LA OT Testing
// PLT label command
XSPEC12> setp co LWidth 5
// PLT line-width command
XSPEC12> setplot lis
// List the PLT command stack.
1: LAbel OT Testing
2: LWidth 5
XSPEC12> setplot del 1
// Delete the first command in the stack.
XSPEC12> setplot lis
1: LWidth 5
```

### 11.5.6.1.5.7 device

Set current plot device.

```
setplot device <plot device>
setplot device <filename>
setplot device <filename>/{ps,cps,vps,vcps}
setplot device none
```

If the second argument does not start with a '/' character, which indicates that the string represents a PGPLOT device, it is taken to be a filename for Postscript output, and the default postscript driver will be used. The default postscript driver produces a monochrome plot in landscape orientation.

The filename argument can be followed by a '/' that specifies a particular postscript driver variant. Allowable variants are: `cps` (color postscript), `vps` (monochrome portrait orientation), and `vcps` (color portrait orientation), as well as the default, `ps`.

Set the device used for plots.

### 11.5.6.1.5.8 PGPLOT devices

A number of plot device types are supported in XSPEC. PGPLOT devices available on Unix machines are:

/GIF	Graphics Interchange Format file, landscape orientation
/VGIF	Graphics Interchange Format file, portrait orientation
/NULL	Null device, no output
/PPM	Portable Pixel Map file, landscape orientation
/VPPM	Portable Pixel Map file, portrait orientation
/PS	PostScript file, landscape orientation
/VPS	PostScript file, portrait orientation
/CPS	Colour PostScript file, landscape orientation
/VCPS	Colour PostScript file, portrait orientation
/TEK4010	Tektronix 4010 terminal

/GF	GraphOn Tek terminal emulator
/RETRO	Retrographics VT640 Tek emulator
/GTERM	Color gterm terminal emulator
/XTERM	XTERM Tek terminal emulator
/ZSTEM	ZSTEM Tek terminal emulator
/V603	Visual 603 terminal
/KRM3	Kermit 3 IBM-PC terminal emulator
/TK4100	Tektronix 4100 terminals
/VT125DEC	VT125 and other REGIS terminals
/XDISP	pgdisp or figdisp server
/XWINDOW	X window window@node:display.screen/xw
/XSERVE	An /XWINDOW window that persists for re-use

Examples:

```
XSPEC12> setplot device /xt
// sets the device to the xterm.
XSPEC12> setplot device none
// closes the plot file.
```

### 11.5.6.1.5.9 energy

Change the X-axis on plots to energies, and optionally change the units.

**setplot** energy [<units>]

where <units> is an optional string for modifying X-axis energy units. Valid choices currently are: keV, MeV, GeV, and Hz, which are case-insensitive and can be abbreviated. Energy units initially default to keV. The selection made here also determines the units in **ignore/notice** energy range specifiers.

Where applicable, Y-axis units will be modified to match the X-axis selection. The exception is for the choice of Hz when emodel/eufspec is in Jy and eemodel/eeufspec in ergs/cm<sup>2</sup>/s.

### 11.5.6.1.5.10 group

Define a range of spectra to be in the same group for plotting purposes only.

**setplot** group <spectrum range>...

where <spectrum range> is a range of contiguous spectra to be treated as a single spectrum for plotting purposes. The spectra still are fit individually. If multiple ranges are given, each range becomes a single group. Initially, all spectra read in are treated as single spectra. (See also **ungroup**.)

Examples:

Assume that there are five spectra currently read in, all of them ungrouped initially.

```
XSPEC12> setplot group 1-4
// The first four spectra are treated as one group, with the fifth spectra on its own. Thus all
// plots will appear to have two spectra.
XSPEC12> setplot group 1 2 3 4
// The spectra are reset to each be in their own group.
XSPEC12> setplot group 2-3 4-5
// Now there are three plot groups, being spectrum 1, by itself, and
```

```
// spectra 2-3 and 4-5 as groups.  
XSPEC12> setplot group 1-**  
// All the spectra are placed in a single plot group.
```

### 11.5.6.1.5.11 id

Switch on plotting of line IDs.

**setplot** id <temperature> <emissivity limit> <redshift>

The IDs are taken from the APEC line list for the temperature given by the first argument. The plot only shows those lines with emissivities above the limit set and the lines are redshifted by the amount specified. Currently the APEC version used is 1.10. If `xset apcroot` has been used to reset the APEC files then `setplot id` uses a filename based on the value of `apcroot` as described in the documentation for the `apc` model.

### 11.5.6.1.5.12 list

List all the PLT commands in the command list.

**setplot** list

See `setplot delete` for an example of use.

### 11.5.6.1.5.13 noadd

Do not show individual additive model components on the data plots.

**setplot** noadd

The opposite is `setplot add`.

### 11.5.6.1.5.14 noid

Switch off plotting of line IDs.

**setplot** noid

The opposite is `setplot id`.

### 11.5.6.1.5.15 rebin

Define characteristics used in rebinning the data (for plotting purposes *only*).

**setplot** rebin <min significance> <max # bins> <plot group> <error type>

In plotting the data from a spectrum (or group of spectra, see `setplot group`), adjacent bins are combined until they have a significant detection at least as large as <min significance> (in s). However, no more than <max # bins> may be so combined. Initial values are 0. and 1, respectively. This argument effects only the presentation of the data in plots. It does not change the fitting, in particular the number of degrees of freedom. The values given are applied to all the plotted data in the plot group specified as the final argument. To change the rebinning simultaneously for all the plot groups give a negative value of the plot group.

The `<error type>` argument specifies how to calculate the error bars on the new bins. The default is `quad` which sums in quadrature the errors on the original bins. `sqrt` uses *[Math Processing Error]*, where  $N$  is the number of counts in the new bin, `poiss-1` uses *[Math Processing Error]*, `poiss-2` uses *[Math Processing Error]*, and `poiss-3` is the arithmetic mean of `poiss-1` and `poiss-2`. If background is present its error is calculated by the same method then added in quadrature to the source error.

Examples:

```
XSPEC12> setplot rebin 3 5 1
// Bins in plot group 1 are plotted that have at least 3σ, or are grouped in sets of 5 bins.
XSPEC12> setplot rebin 5 5
// The significance is increased to 5σ.
XSPEC12> setplot rebin,,10,-1
// All plotted bins can be grouped into up to 10 bins in reaching the 5σ significance
criterion.
XSPEC12> setplot rebin ,,sqrt}
// Uses [Math Processing Error] to calculate error bars.
```

### 11.5.6.1.5.16 redshift

Apply a redshift to the X-axis energy and wavelength values.

**setplot** redshift `<z>`

This will multiply X-axis energies by a factor of  $(1+z)$  to allow for viewing in the source frame. Y-axis values will be equally affected in plots which are normalized by energy or wavelength. Note that this is not connected in any way to redshift parameters in the model (or the `setplot id redshift` parameter) and should only be used for illustrative purposes.

### 11.5.6.1.5.17 splashpage

When set to off, the usual XSPEC version and build date information will not be printed to the screen when the first plot window is initially opened. This is intended primarily for the HERA installation of XSPEC.

**setplot** splashpage `(on|off)`

### 11.5.6.1.5.18 ungroup

Remove previous grouping set up by `setplot group`, resetting all spectra to be in a distinct plot group.

### 11.5.6.1.5.19 wave

Change the x-axis on plots to wavelength, and optionally change the units.

**setplot** wave `<units>`  
**setplot** wave perhz `[off]`

where `<units>` is an optional string for modifying X-axis wavelength units. Valid choices currently are: `angstrom`, `cm`, `micron`, and `nm`, which are case-insensitive and can be abbreviated. Wavelength units initially default to `angstrom`.

Where applicable, Y-axis units will be modified to match the X-axis selection. However this behavior can be changed by the command `setplot wave perhz`, which will cause Y-axis units to be in  $1/\text{Hz}$ . This feature is turned off by `setplot wave perhz off`, and its initial setting is determined by the `WAVE_PLOT_UNITS` setting in the user's

~/xspec/xspec.init file. Also note that when perhz is selected, **emodel/eufspec** and **eemodel/eeufspec** will have the same Y-axis units as for `setplot energy hz`.

This command makes **ignore** and **notice** operate in terms of wavelength rather than energies. The units setting here also determines the units in the **ignore/notice** range specifiers.

### 11.5.6.1.5.20 xlog

`setplot xlog (on | off)`

Set the x-axis to logarithmic or linear respectively for energy or wavelength plots. `xlog` has no effect on plots in channel space (recall that the default for energy plots is logarithmic: `xlog` allows the user to override this setting). `xlog` and `ylog` will not work for model-related plots (e.g. `model`, `ufspec`, and their variants) as their axes are always set to log scale.

### 11.5.6.1.5.21 ylog

`setplot ylog (on | off)`

Set the y-axis to logarithmic or linear respectively for energy or wavelength plots. For plot instructions that are explicitly logarithmic (`plot ldata`, `plot lcounts`) the state of the `ylog` setting is ignored. `xlog` and `ylog` will not work for model-related plots (e.g. `model`, `ufspec`, and their variants) as their axes are always set to log scale.

<code>add</code>	?
<code>area, noarea</code>	?
<code>background, nobackground</code>	Will be added, but if there are multiple superimposed plots, it doesn't seem sensible to allow multiple overplotted backgrounds too. Should it be possible to turn on/off the background for each spectrum separately (allowing for multiple overlaid backgrounds), or should this be a zone option that selects either no background, or any single background from spectra that are currently displayed? Should show/hide spectrum always hide the background too, if any?
<code>channel</code>	Supported as another type of horizontal units.
<code>command</code>	Eventually this will be supported by the image annotation editor.
<code>delete</code>	n/a
<code>device</code>	The plot tool displays all plots, and supports export in various formats.
<code>energy</code>	Each plot zone's horizontal and vertical axis units are settable via the zone panels.
<code>group</code>	This is implicit, in that multiple spectra may be overplotted in the same zone.
<code>id</code>	There is a separate pane for display of line labels, and this can be hidden. I don't understand how this operation interacts with <b>identify/modid</b> though.
<code>list</code>	Not applicable.
<code>noadd</code>	?
<code>noid</code>	Done by closing the line label pane.
<code>rebin</code>	I think this is best grouped with the plot style options, which apply to a plot layer.
<code>redshift</code>	This is currently set in the emission lines controls in the plot tool sidebar.
<code>splashpage</code>	n/a
<code>ungroup</code>	Explicit grouping is not performed, but overplotting allows implicit grouping.
<code>wave</code>	Done by setting the horizontal axis units for the plot zone.
<code>xlog/ylog</code>	Done by setting the horizontal or vertical units for the plot zone.

## 11.5.7 Setting Commands

All these would be probably be implemented at the same time.

### 11.5.7.1.1 abund: set the Solar abundances

Set the abundance table used in the plasma emission and photoelectric absorption models.

**abund** <option>

where <option> is:

angr	from Anders E. & Grevesse N. (1989, Geochimica et Cosmochimica Acta 53, 197)
aspl	from Asplund M., Grevesse N., Sauval A.J. & Scott P. (2009, ARAA, 47, 481)
feld	from Feldman U. (1992, Physica Scripta 46, 202 except for elements not listed which are given grsa abundances)
aneb	from Anders E. & Ebihara (1982, Geochimica et Cosmochimica Acta 46, 2363)
grsa	from Grevesse, N. & Sauval, A.J. (1998, Space Science Reviews 85, 161)
wilm	from Wilms, Allen & McCray (2000, ApJ 542, 914 except for elements not listed which are given zero abundance)
lodd	from Lodders, K (2003, ApJ 591, 1220)
file filename	where filename is an ASCII file containing 30 lines with one number on each line. All abundances are numbered relative to H.

The tables are:

Element	angr	aspl	feld	aneb	grsa	wilm	lodd
H	1.00e+00						
He	9.77e-02	8.51e-02	9.77e-02	8.01e-02	8.51e-02	9.77e-02	7.92e-02
Li	1.45e-11	1.12e-11	1.26e-11	2.19e-09	1.26e-11	0.00	1.90e-09
Be	1.41e-11	2.40e-11	2.51e-11	2.87e-11	2.51e-11	0.00	2.57e-11
B	3.98e-10	5.01e-10	3.55e-10	8.82e-10	3.55e-10	0.00	6.03e-10
C	3.63e-04	2.69e-04	3.98e-04	4.45e-04	3.31e-04	2.40e-04	2.45e-04
N	1.12e-04	6.76e-05	1.00e-04	9.12e-05	8.32e-05	7.59e-05	6.76e-05
O	8.51e-04	4.90e-04	8.51e-04	7.39e-04	6.76e-04	4.90e-04	4.90e-04
F	3.63e-08	3.63e-08	3.63e-08	3.10e-08	3.63e-08	0.00	2.88e-08
Ne	1.23e-04	8.51e-05	1.29e-04	1.38e-04	1.20e-04	8.71e-05	7.41e-05
Na	2.14e-06	1.74e-06	2.14e-06	2.10e-06	2.14e-06	1.45e-06	1.99e-06
Mg	3.80e-05	3.98e-05	3.80e-05	3.95e-05	3.80e-05	2.51e-05	3.55e-05
Al	2.95e-06	2.82e-06	2.95e-06	3.12e-06	2.95e-06	2.14e-06	2.88e-06
Si	3.55e-05	3.24e-05	3.55e-05	3.68e-05	3.55e-05	1.86e-05	3.47e-05
P	2.82e-07	2.57e-07	2.82e-07	3.82e-07	2.82e-07	2.63e-07	2.88e-07
S	1.62e-05	1.32e-05	1.62e-05	1.89e-05	2.14e-05	1.23e-05	1.55e-05
Cl	3.16e-07	3.16e-07	3.16e-07	1.93e-07	3.16e-07	1.32e-07	1.82e-07
Ar	3.63e-06	2.51e-06	4.47e-06	3.82e-06	2.51e-06	2.57e-06	3.55e-06
K	1.32e-07	1.07e-07	1.32e-07	1.39e-07	1.32e-07	0.00	1.29e-07
Ca	2.29e-06	2.19e-06	2.29e-06	2.25e-06	2.29e-06	1.58e-06	2.19e-06
Sc	1.26e-09	1.41e-09	1.48e-09	1.24e-09	1.48e-09	0.00	1.17e-09
Ti	9.77e-08	8.91e-08	1.05e-07	8.82e-08	1.05e-07	6.46e-08	8.32e-08
V	1.00e-08	8.51e-09	1.00e-08	1.08e-08	1.00e-08	0.00	1.00e-08
Cr	4.68e-07	4.37e-07	4.68e-07	4.93e-07	4.68e-07	3.24e-07	4.47e-07
Mn	2.45e-07	2.69e-07	2.45e-07	3.50e-07	2.45e-07	2.19e-07	3.16e-07

Fe	4.68e-05	3.16e-05	3.24e-05	3.31e-05	3.16e-05	2.69e-05	2.95e-05
Co	8.32e-08	9.77e-08	8.32e-08	8.27e-08	8.32e-08	8.32e-08	8.13e-08
Ni	1.78e-06	1.66e-06	1.78e-06	1.81e-06	1.78e-06	1.12e-06	1.66e-06
Cu	1.62e-08	1.55e-08	1.62e-08	1.89e-08	1.62e-08	0.00	1.82e-08
Zn	3.98e-08	3.63e-08	3.98e-08	4.63e-08	3.98e-08	0.00	4.27e-08

This will be a preferences option and a notebook-level option. At the notebook level, it can be set to "inherit" to use the user's preferences (default), or to a specific option for this notebook. This way, the option is preserved when someone else runs the same notebook.

### 11.5.7.1.2 cosmo: set the cosmology

Set the cosmology used (i.e.,  $H_0$ ,  $q_0$ , and  $\Lambda_0$ ).

**cosmo** [ $\langle H_0 \rangle$ ] [ $\langle q_0 \rangle$ ] [ $\langle \Lambda_0 \rangle$ ]]

where  $\langle H_0 \rangle$  is the Hubble constant in  $\text{km s}^{-1} \text{Mpc}^{-1}$ ,  $\langle q_0 \rangle$  is the deceleration parameter, and  $\langle \Lambda_0 \rangle$  is the cosmological constant. If the cosmological constant is non-zero then at present XSPEC requires that the universe is flat. In this case the value of  $q_0$  will be ignored and XSPEC will assume that  $\Omega_{\text{matter}} \geq 1 - \Lambda_0$ . The default values are  $\langle H_0 \rangle = 70$ ,  $\langle q_0 \rangle = 0.0$ , and  $\langle \Lambda_0 \rangle = 0.73$ .

Examples:

```
XSPEC12> cosmo 100
// Set <H0> = 100 kms-1 Mpc-1
XSPEC12> cosmo ,0
// Set <q0> = 0
XSPEC12> cosmo ,,0.7
// Set a flat universe with <Λ0> = 0.7
```

This will be a preferences option and a notebook-level option. At the notebook level, it can be set to "inherit" to use the user's preferences (default), or to a specific option for this notebook. This way, the option is preserved when someone else runs the same notebook.

### 11.5.7.1.3 method: change the fitting method

Set the minimization method.

**method** <algorithm> [<# of trials/evaluations> [<critical delta>] [method-specific options]]

where <algorithm> is the method in use and the other arguments are control values for the minimization. Their meanings are explained under the individual methods. The `migrad` and `simplex` methods are taken from the CERN Minuit2 package, with documentation located at <http://seal.web.cern.ch/seal/MathLibs/Minuit2/html/index.html>. If either of these are used, then the **error** command will use the Minuit2 `minos` method to find the confidence regions.

#### 11.5.7.1.3.1 leven

**method** `leven` [<# of eval> [<crit delta>] [<crit beta>]] [delay | nodelay]

The default XSPEC minimization method using the modified Levenberg-Marquardt algorithm based on the CURFIT routine from Bevington. <# of eval> is the number of trial vectors before the user is prompted to say whether they want to continue fitting. <crit delta> is the convergence criterion, which is the (absolute, not fractional) difference in

fit statistic between successive iterations, less than which the fit is determined to have converged.

`<crit beta>` refers to the  $|\text{beta}|/N$  value reported during a fit. This is the norm of the vector of derivatives of the statistic with respect to the parameters divided by the number of parameters. At the best fit this should be zero, and so provides another measure of how well the fit is converging. When this is set to a positive value, it will provide another fit stopping criterion in addition to that of the `<crit delta>` setting.

Including the string `delay` as an argument turns on delayed gratification. It is turned off by `nodelay`. Delayed gratification modifies the way the damping parameter is set and has been shown in many cases to speed up convergence. The default is `nodelay`.

`<# of eval>`, `<crit delta>`, `<crit beta>`, `delay`, and `nodelay` may also be set through the **fit** command.

This method requires an estimate of the second derivative of the statistic with respect to the parameters. By default, XSPEC calculates these using an analytic expression which assumes that partial second derivatives of the model with respect to its parameters may be ignored. This may be changed by setting the `USE_NUMERICAL_DIFFERENTIATION` flag to "true" in the user's startup `xspec.init` initialization file. XSPEC will then calculate all second derivatives numerically, which can be noticeably slower.

### 11.5.7.1.3.2 migrad

**method** [`<# of eval>`]

The Minuit2 migrad method. `<# of eval>` is the number of function evaluations to perform before giving up. Migrad uses an internal convergence criterion.

The current version of Minuit2 included is that from ROOT v5.34. Documentation on Minuit2 can be found at <http://seal.web.cern.ch/seal/MathLibs/Minuit2/html/>.

If migrad is not working well try experimenting with different hard and soft limits on parameters.

### 11.5.7.1.3.3 simplex

**method** simplex [`<# of evaluations>`]

The Minuit2 simplex method. `<# of evaluations>` is the number of function evaluations to perform before giving up. Simplex uses an internal convergence criterion. This method is included for historical interest and is almost always outperformed by migrad.

This will be a preferences option and a notebook-level option. At the notebook level, it can be set to "inherit" to use the user's preferences (default), or to a specific option for this notebook. This way, the option is preserved when someone else runs the same notebook.

### 11.5.7.1.4 statistic: change the objective function (statistic) for the fit

Change the fit or test statistic in use, for one or more spectra.

**statistic** [`chi` | `cstat` | `lstat` | `pgstat` | `pstat` | `whittle[#]`] [`<spectrum range>`]

**statistic test** [`ad` | `chi` | `cvm` | `ks` | `pchi` | `runs`] [`<spectrum range>`]

The fit statistic options are  $\chi^2$  (`chi`), C statistic (`cstat`), Loredo statistic (`lstat`), a statistic for Poisson data with assumed known background (`pstat`), a statistic for Poisson data with Gaussian background (`pgstat`), and the Whittle statistic (`whittle`) for power density functions. If the statistic is given as `whittle` with a number appended (e.g. `whittle5`) then the statistic is appropriate for that number of power density functions averaged together. The test statistic options are Anderson-Darling (`ad`),  $\chi^2$  (`chi`), Cramer-von Mises (`cvm`), Kolmogorov-Smirnov (`ks`), Pearson  $\chi^2$  (`pchi`) and Runs (`runs`). These statistics are described in the appendix on Statistics in XSPEC. If a spectrum number or

spectrum range is given, the chosen statistic will only apply to those spectra. It is therefore possible for a multi-spectrum fit to use more than one fit or test statistic. If no spectrum number or range is given, the chosen statistic will apply to all loaded spectra and will be the default statistic for any future loaded spectra.

Note that if the chosen statistic is not compatible with the currently used **weight** method, the **weight** method will be changed to standard weighting until the conflict is removed.

### Examples:

Assume 3 spectra are currently loaded, all using the  $\chi^2$  statistic, and that  $\chi^2$  is the default statistic.

```
XSPEC12>statistic cstat 2-3
// Spectrum 1 continues to use  $\chi^2$ , 2 and 3 use cstat.
XSPEC12>data 4 spec4.pha
// New spectrum 4 will use  $\chi^2$ .
XSPEC12>statistic cstat
// All 4 spectra now use cstat, cstat is the new default.
XSPEC12>data 5 spec5.pha
// New spectrum 5 will use cstat.
XSPEC12>statistic test ks
// All 4 spectra now use ks as the test statistic.
```

This will be a preferences option and a notebook-level option. At the notebook level, it can be set to "inherit" to use the user's preferences (default), or to a specific option for this notebook. This way, the option is preserved when someone else runs the same notebook. It may also be overridden on a per-tag basis, where it defaults to "inherit."

### 11.5.7.1.5 xsect: set the photoionization cross-sections

Change the photoelectric absorption cross-sections in use.

```
xsect [bcmc | obcm | vern]
```

The three options are: **bcmc**, from Balucinska-Church & McCammon (1992; Ap.J.400, 699) with a new He cross-section based on (1998; Ap.J. 496, 1044); **obcm**, as **bcmc** but with the old He cross-section, and, **vern**, from Verner et. al. (1996 Ap.J.). This changes the cross-sections in use for all absorption models with the exception of **wabs**.

This will be a preferences option and a notebook-level option. At the notebook level, it can be set to "inherit" to use the user's preferences (default), or to a specific option for this notebook. This way, the option is preserved when someone else runs the same notebook.

### 11.5.7.1.6 xset: set variables for XSPEC models.

Modify a number of XSPEC internal switches.

```
xset [abund | cosmo | delta | mdatadir | method | seed | statistic | weight | xsect |
<string_name> ] [<options> | <string_value> ]
```

The arguments **abund**, **cosmo**, **method**, **statistic**, **weight**, and **xsect** just run the appropriate XSPEC commands. **mdatadir** changes the directory in which XSPEC searches for model data files. You probably don't want to change this. The **seed** option requires an integer argument, which will then be used to immediately re-seed and re-initialize XSPEC's random-number generator.

The **delta** option is for setting fit delta values (see the **newpar** command) which are proportional to the current parameter value rather than fixed. For example,

```
XSPEC12> xset delta .15
```

will set each parameter fit delta to  $.15 * \text{parVal}$ . To turn proportional deltas off and restore the original fixed deltas, set **delta** to a negative value or 0.0. The current proportional delta setting can be seen with **show control**.

The <string\_name> option can be used to pass string values to models. XSPEC maintains a database of <string\_name>, <string\_value> pairs created using this command. Individual model functions can then access this database. Note that **xset** does no checking on whether the <string\_name> is used by any model so spelling errors will not be trapped.

To access the <string\_name>, <string\_value> database from within a model function use the fortran function `fgmstr`. This is defined as `character*128` and takes a single argument, the string name as a `character*128`. If the <string\_name> has not been set then a blank string will be returned.

The current <string\_name> options, models to which they apply and brief descriptions are given in the following table :

APECROOT	apec, vapec, bapec, bvapec, equil, vequil, npshock, vnpshock, pshock, vpshock, sedov, vsedov, c6mekl, c6vmekl, c6pmekl, c6pvmekl, cemkl, cevmdl, mekal, vmekal, mkcflow, vmclow	Switch from default APEC input files. <b>Not supported in ATSA.</b>
APECTHERMAL	apec, vapec, bapec, bvapec, equil, vequil, npshock, vnpshock, pshock, vpshock, sedov, vsedov, c6mekl, c6vmekl, c6pmekl, c6pvmekl, cemkl, cevmdl, mekal, vmekal, mkcflow, vmclow	Thermally broaden emission lines in APEC input files. <b>Preferences &amp; notebook option.</b>
APECVELOCITY	apec, vapec, bapec, bvapec, equil, vequil, npshock, vnpshock, pshock, vpshock, sedov, vsedov, c6mekl, c6vmekl, c6pmekl, c6pvmekl, cemkl, cevmdl, mekal, vmekal, mkcflow, vmclow	Velocity broaden emission lines in APEC input files. <b>Preferences &amp; notebook option.</b>
NEIAPECROOT	gnei, nei, vgnei, nvei, equil, vequil, npshock, vnpshock, pshock, vpshock, sedov, vsedov	Switch from default NEIAPEC input files. <b>Not supported in ATSA.</b>
POW_EMIN, POW_EMAX	powerlaw, bknpower, bkn2pow, cutoffpl	Switch to normalize to a flux calculated over an energy range. <b>Preferences &amp; notebook option.</b>
NEIVERS	gnei, nei, vgnei, vnei, equil, vequil, npshock, vnpshock, pshock, vpshock, sedov, vsedov	Switch NEIAPEC version number. <b>Preferences &amp; notebook option.</b>
CFLOW_VERSION	mkcflow, vmclow	Switch CFLOW version number. <b>Preferences &amp; notebook option.</b>
CFLOW_NTEMPS	mkcflow, vmclow	Switch number of temperature bins used in CFLOW model. <b>Preferences &amp; notebook option.</b>
SUZPSF-IMAGE	suzpsf	Set image file to be used for surface brightness. <b>Preferences &amp; notebook option.</b>
SUZPSF-RA	suzpsf	Set RA for center surface brightness map which is taken from the WMAP. <b>Preferences &amp; notebook option.</b>
SUZPSF-DEC	suzpsf	Set Dec for center surface brightness map which is taken from the WMAP. <b>Preferences &amp; notebook option.</b>
SUZPSF-MIXFACT-IFILE#	suzpsf	Set filename to read mixing factors. <b>Preferences &amp; notebook option.</b>
SUZSF-MIXFACT-OFIILE#	suzpsf	Set filename to write mixing factors. <b>Preferences &amp; notebook option.</b>
XMMPSF-IMAGE	xmmpsff	Set image file to be used for surface brightness. <b>Preferences &amp; notebook option.</b>
XMMPSF-RA	xmmpsff	Set RA for center surface brightness map which is taken from the WMAP. <b>Preferences &amp; notebook option.</b>

XMMPSF-DEC	xmmpsfsf	Set Dec for center surface brightness map which is taken from the WMAP. <a href="#">Preferences &amp; notebook option.</a>
XMMPSF-MIXFACT-IFILE#	xmmpsfsf	Set filename to read mixing factors. <a href="#">Preferences &amp; notebook option.</a>
XMMPSF-MIXFACT-OFILE#	xmmpsfsf	Set filename to write mixing factors. <a href="#">Preferences &amp; notebook option.</a>
NSA_FILE	nsa	Change filename used for model data. <a href="#">Preferences &amp; notebook option.</a>
NSAGRAV_DIR	nsagrav	Change directory used for model data files. <a href="#">Preferences &amp; notebook option.</a>
NSMAX_DIR	nsmax	Change directory used for model data files. <a href="#">Preferences &amp; notebook option.</a>
ZXIPCF_DIR	zxipcf	Change directory used for model data files. <a href="#">Preferences &amp; notebook option.</a>

Examples:

```
XSPEC12> xset neivers 2.0
// Set the NEIVERS variable to 2.0
XSPEC12> xset
// List the current string variables
XSPEC12> xset apecroot /foo/bar/apec_v1.01
// Set the APECROOT variable
XSPEC12> xset seed 1515151
// Re-initialize the pseudo random-number generator with the seed value 1515151
```

## 11.5.8 Tcl Scripts

### 11.5.8.1.1 lrt: likelihood ratio test between two models

Tcl script to perform a likelihood ratio test between two models.

```
lrt <niter> <model0_name> <model1_name> [<filename>]
```

Runs <niter> simulations of datasets based on <model0\_name>, calculates the likelihood ratio for <model1\_name> relative to <model0\_name> (calculated by the statistic for <model0\_name> minus the statistic for <model1\_name>), and outputs the fraction of iterations with the likelihood ratio smaller than that for the data. If the optional filename is given then the simulation results are written to the file. The first line of the file contains the results for the data, the other lines the simulations. Each line comprises the statistic values for <model0\_name>, the statistic value for <model1\_name>, and the difference.

Before running this procedure you must have created command files called <model0\_name>.xcm and <model1\_name>.xcm which define the two models. A good way to do this is to set up the model then use save model to make the command file.

**TBD.**

### 11.5.8.1.2 multifake: perform multiple fakeit iterations and save to file.

Tcl script to perform many iterations of fakeit and save the results in a FITS file.

**multifake** <time> <niter> <outfile>

This script runs <niter> iterations of **fakeit** with an exposure of <time> and writes the results to <outfile>. Before running this procedure you have read in one (and only one) dataset along with its response and optional background and arf files. You must also have defined the model.

The output file is a FITS binary table with the columns being the value fit for each parameter in each iteration. The final column is the statistic value for that iteration.

Note that if an error occurs during the fit of a faked spectrum then -999 is written for all parameters and the statistic value for that iteration.

TBD.

### 11.5.8.1.3 rescalecov: rescale the covariance matrix.

Tcl script to rescale the entire covariance matrix used in the proposal chain command.

**rescalecov** <scale>

Rescales the chain proposal distribution covariance matrix by the factor input as <scale>.

TBD.

### 11.5.8.1.4 simftest: estimate the F-test probability for adding a component.

Tcl script to generate simulated datasets and use these to estimate the F-test probability for adding a model component.

**simftest** <model\_comp> <niter> [<filename>]

This script runs <niter> sets of simulated datasets to estimate the F-test probability for adding the additional model component number <model\_comp>. If <filename> is specified then passes this to `lrt.tcl` to save likelihood ratio simulation information. The first line of the file written contains the results for the data, the other lines for the simulations. Each line comprises the statistic value for the model without <model\_comp>, that for the model with <model\_comp>, and the difference.

Before running this script the model should be set up including the additional component to be tested. The script will create temporary files `model_with_comp.xcm` and `model_without_comp.xcm`.

TBD.

### 11.5.8.1.5 writefits: write information about the current fit and errors to a FITS file

Tcl script to dump a lot of useful information to a FITS file.

**writefits** <FITS filename>

This script writes filenames, free parameter values and errors to one row of a FITS file. The error command should have been run on all the free parameters before running this script. If the FITS file already exists then a new row is appended.

TBD.

## 11.5.9 Deprecated Commands

### 11.5.9.1.1 dump

Write out a history package of observed and model spectra.

```
dump    [<option>]
```

Two options are available : `ecdata` and `model`, with the same meaning that they have in the `plot` command. Plotting of unfolded spectra is possible with the XPLOTT plotting program. A `dump ecdata` and a `dump model` write out all the necessary information into the history file.

### Examples:

```
XSPEC>dump          ! The first dump command
                    ! uses ecdata as default. It
                    ! writes a history package
                    ! containing the observed PHA
                    ! spectrum and the folded
                    ! model versus channel energy.
```

```
XSPEC>dump model    ! Write history package for the
                    ! model spectrum in
                    ! photons/cm2 s keV.
```

### 11.5.9.1.2 exec

The command to execute a shell command.

```
exec    <shell command>
```

This command executes a shell (ie. an operating system) command, and then returns control to XSPEC after it is completed. Note that if your system is setup with the standard TCL distribution, shell commands entered at the XSPEC prompt will be executed automatically if they do not match any XSPEC or TCL command.

### 11.5.9.1.3 extend

Extend the energy range over which the model is calculated.

```
extend  <high | low> <energy> <no. energies> <log | lin>
```

where `high` or `low` indicates whether the energy range is to be extended above or below that from the response matrix, `<energy>` is the maximum or minimum energy for the extension, `<no. energies>` is the number of energy bins to add, and `log` or `lin` is whether they should be spaced logarithmically or linearly. The defaults are to extend on the high end with logarithmic binning. Note that all response matrices in use are extended if necessary. This command is intended for use with convolution models which may need to have their input models calculated over a wide energy range. When the response matrix is read in again the extension is lost (note that this occurs when the `notice` command is used).

### Examples:

```
XSPEC>extend high 50. 50    ! Extend the response energy
                            ! to 50 keV in 50 logarithmic
                            ! steps
```

```
XSPEC>extend low 0.1 10 lin ! Extend the response energy
                            ! down to 0.1 keV using 10
                            ! linear steps
```

```
genetic <option> <value>
```

Set parameters used in the PIKAIA genetic global minimization algorithm (see ``method'` for further details). For details see Charbonneau, P., 1995, Ap.J. Suppl, 101, 309.

Option can take the following values.

`npop`      Change the size of the population.  
`ndigits`    Change the encoding accuracy.  
`pcross`     Change the crossover rate.  
`imutate`    Change the size of the population.  
`pmutate`    Change the initial mutation rate.  
`pmutmin`    Change the minimum mutation rate.  
`pmutmax`    Change the maximum mutation rate.  
`fdif`        Change the fitness differential.  
`irepro`     Change the reproduction plans.  
`ielite`     Change the elitism.  
`iverbose`   Change the verbosity.

#### 11.5.9.1.4 improve

Try to find a new minimum.

```
improve
```

This command runs the MINUIT `improve` command which attempts to find a new local minimum in the vicinity of the current minimum. This gives some global minimization capability.

#### 11.5.9.1.5 quit

The same as `exit`.

```
quit
```

#### 11.5.9.1.6 readline

Enable/disable gnu readline facility.

```
readline    [on | off]
```

This command is used to enable or disable the gnu readline command editing/history facility. Giving the command with no arguments will print the current status (enabled or disabled). Readline is enabled by default. For more information on using readline see Appendix D.

#### 11.5.9.1.7 recornrm

**Adjusts the indicated data sets' correction norm by a single multiplicative factor that minimizes the fit statistic. This approach to fitting the correction norm is considered an INTERIM solution.**

```
recornrm    [<file range>...]
```

where `<data set range> := <low data set> [-<high data set>]` All the data sets specified by the one or more data set ranges on a single invocation of the command are multiplied by a single number, which is chosen to best reduce the fit statistic. If no ranges are given, then the last SINGLE range input is used. If you wish to fit a data set's correction norm individually, then refer only to that data set.

Note : The use of the `recornrm` command requires that a response and model be defined. It is perhaps best if a preliminary fit is performed. The user then should alternate between `fit`'s and `recornrm`'s until a stable solution is

achieved. If the model is not a good fit, this process may not converge.

### Examples:

```
XSPEC>recornrm * *      ! All the files' correction norms
                        ! are adjusted by a single
                        ! number.
XSPEC>recornrm 1-3 5    ! Files 1,2,3, and 5 are
                        ! adjusted.
XSPEC>recornrm         ! File 5 (the last range input) is
                        ! adjusted by itself.
XSPEC>recornrm 2       ! File 2 is adjusted
```

### 11.5.9.1.8 source

Add tcl procedures in script file to command set. This command is intended to be used for developing new procedures, after which the script should be added to the user's `$XSPEC_HOME` directory and be source'd automatically on startup.

```
source <file name>
```

The full file name must be specified. There are no default extensions as in previous versions of XSPEC. The script may use the full power of the TCL scripting language. It is recommended that full command names be used when writing scripts. This will cause the script files to run more efficiently.

N.B. The alternative syntax for executing files containing xspec commands,

```
XSPEC>@<scriptname>
```

is recommended for xspec scripts not containing tcl procedures, for example output from the 'save' command. This is because the tcl command processor uses a 'newline' character to determine when a command that is not in curly parentheses is ended. However, this is not correct for xspec's multiple-line commands (i.e. those which when run interactively prompt the user for input such as `model`). Hence scripts that contain such commands may not process correctly. In contrast, the '@' command will detect such conditions and process the script properly.

### 11.5.9.1.9 suggest

Get advice on what to do with bug reports and enhancement requests.

```
suggest
```

If the suggestion is the result of a catastrophic error, please give as much information as possible. Clarity and completeness will improve the response to your suggestion.

### 11.5.9.1.10 thleqw

Determine the expected line equivalent width of a fluorescent line.

```
thleqw [<line energy> <line fluorescent yield> <absorption edge energy> <upper energy limit> <bin
        number> <log or lin> <max edge depth>]
```

The `thleqw` command integrates the photon flux absorbed in an edge with the maximum depth from the edge energy to the upper energy limit. To get good resolution in the edge, the command uses the `dummyrsp` command with the edge energy as low energy, the upper energy limit as high energy and the bin number for the number of ranges (see `dummyrsp` command). The calculation for the edge absorption is done in the same way as in the `edge` command. The fluorescent yield is used to calculate the photon flux going into a fluorescent line emission. The continuum flux is calculated in the same way as it is in the `eqwidth` command. No lines are added to the continuum.

This model assumes a spherically symmetric distribution of absorbing material around the X-ray source with a column density obtained by the `wabs` model. The defaults are 1.0 keV, 0., .01 keV, 100. keV, 200, and 0.0, which gives 0 for the equivalent width. They are chosen this way since the `dummyrsp` command defaults are 0.01 keV, 100. keV and 200.

Since the `thleqw` command uses `dummyrsp`, the old response must be read in again when working further on the same spectrum.

### Examples:

```
XSPEC>thl                ! Calculate the expected equivalent width for a line at 6.4 keV with a fluorescent
6.4, .34, 7.1, 20., .015  ! yield of 34%. The absorption edge is integrated from 7.1 keV to 20. keV in 200
                           ! steps. The maximum depth is .015.

XSPEC>thleqw ,,,,,,100   ! As before, but with 100 steps
XSPEC>thleqw             ! As before.
```

Don't forget to read the response file in again if you want to do further work on the same data.

### 11.5.9.1.11 uncertain

A synonym for `error`.

### 11.5.9.1.12 xhistory

#### Open a file for outputting history records. **[Deprecated.]**

```
xhistory {<filename> | none}
```

where `<filename>` is the name of the file to be opened for output. The string `none` will close the current history file. The history file produces SF format packages that can be read by other programs. For example, the results of the `XSPEC` `steppar` command are placed in the `steppar` package, which in return is used by the `XPLOTT` program to plot confidence contours. The history file is not directly human-readable. A more printable recounting of an `XSPEC` session is produced with the `log` file (see the `log` command). Currently implemented history packages:

data files	produced by	data, fakeit
assoc. files		data, backgrnd, response, corfile, fakeit
channels		data, ignore, notice
model		addcomp, model, delcomp
param def		addcomp, model, define, delcomp, newpar, addcomp, model, freeze
fit		fit
steppar		steppar
error		error
equiv width		eqwidth
flux		flux
flum		lumin
spectrum		dump ecdata
photon		dump model
time		time



## 12 ATSAL Testing

ATSAL's testing is still in its infancy. But there's a plan for several tiers of testing. Each tier tests higher levels of system operation, so tests are implemented from lowest tier upward.

- Unit tests usually test data-only classes.
- A special type of unit testing is aimed at the XSpec parser, which must adjust to different types of special-case output from XSpec. This is handled by the Parserator.
- Python extensions allow a python program to create notebooks, either to aid manual debugging or to perform automated tests.
- At the highest tier, a tool called Squish simulates interactive use by the user,

These are described in the following sections.

### 12.1 Unit Tests

Historically ATsAL has had a number of unit tests for various subsystems, scattered haphazardly through the code. In May 2017 these were largely reorganized to run under the new test framework. This makes their operation more uniform, and moves them from running whenever ATsAL is started up to running as part of a master test suite. At the time of this writing, `unittests.py` runs all the unit tests.

### 12.2 The Parserator

The [Parserator](#), for creating new XSpec command parsers, includes a decent mechanism for developing regression tests for them.

### 12.3 Python Notebook Extensions

In Release 0.1.6, I added a simple syntax for configuring ATsAL tests, and `TestConfig.ats`, a sample file showing the syntax. The immediate goal is to save time during testing, by initializing a particular notebook test configuration in a single keystroke. Then it dawned on me that this should be a python program, not a special-purpose scripting language. I had not previously considered this in part because the python language is meant to help users evaluate data returned from XSpec, not configure the notebooks themselves. Although there is nothing wrong with this "meta level" of control in principle, it exposes more of ATsAL's innards and creates more overall complexity. But supporting this as a test framework:

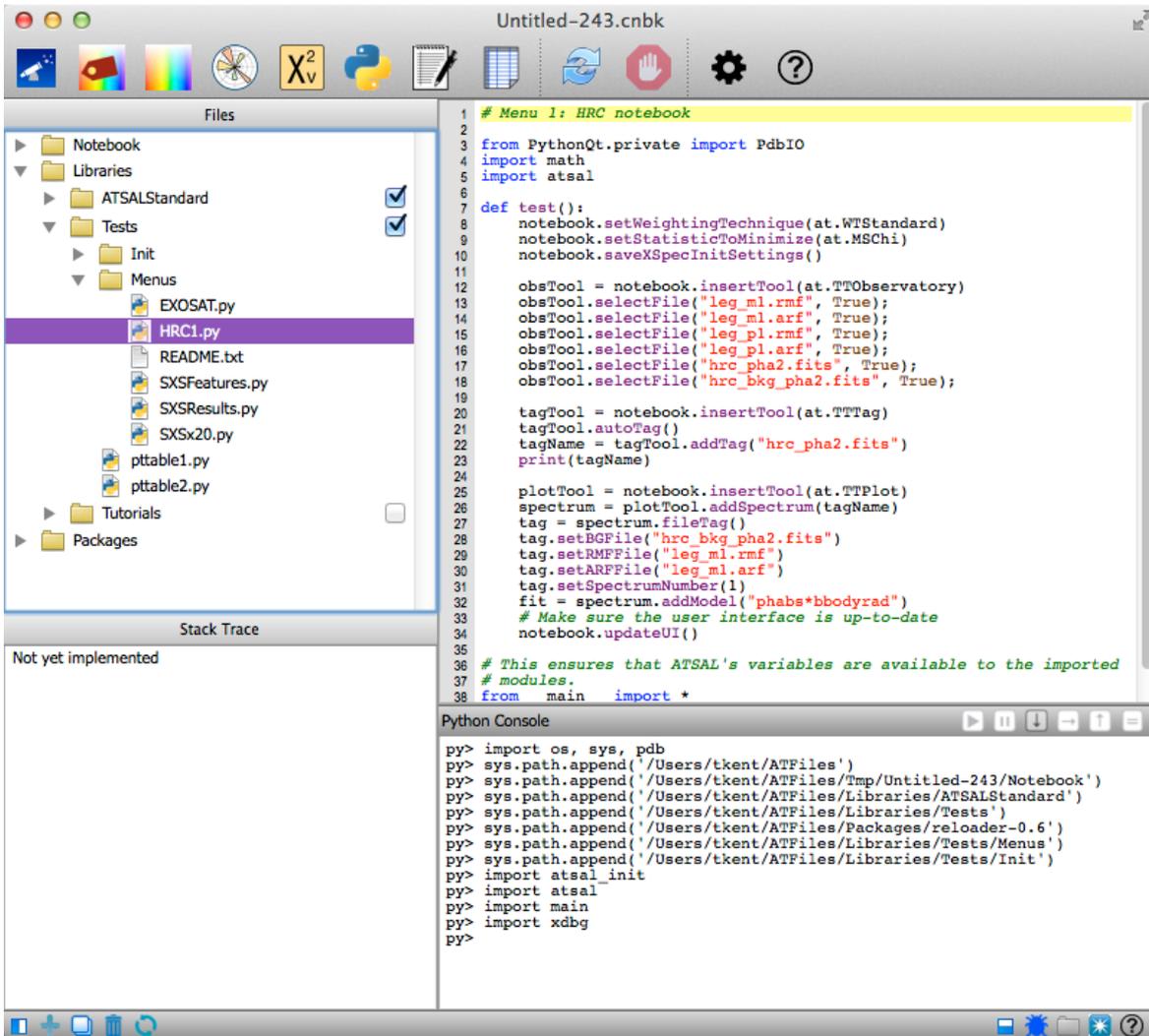
- Solves the immediate problem of speeding up the debugging process.
- Provides the foundation for a flexible higher level of ATsAL testing. The new extensions allow the python user to add tools, create spectra and models, set parameters, etc. The extensions modify the user interface, not just the data classes. This provides a new tier of automated testing.
- Refining the test system provides important experience as to whether and how to surface this same capability to users. If it doesn't turn out to be a good idea, we can enable these bindings only in developer mode. If it does, this further extends the python capabilities. For example, this could make it easy to write a program to run dozens or hundreds of parallel analyses.

See the [Python ATsAL Library](#) for ATsAL's python interface.

The following sections describe these aspects of the extensions.

#### 12.3.1 Speeding up Debugging

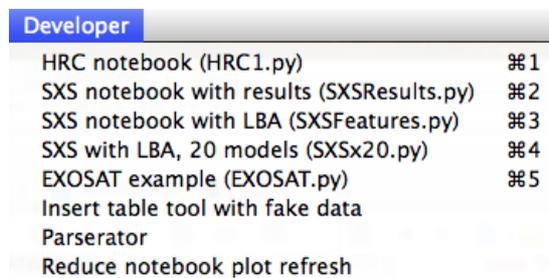
The new mechanism introduces a directory for developer menu extensions, `~/ATFiles/Libraries/Tests/Menus`. A sample program from this directory, `HRC1.py`, is shown below:



At startup, ATLAS scans the python files in `~/ATFiles/Libraries/Tests/Menu`. For each file, if the first line is a comment beginning with "# Menu", the text following is added to the Developer menu, and choosing the menu item runs the program. The special comment format is:

```
# Menu [n]: Menu item text
```

If `n` is supplied, a corresponding keyboard shortcut is created. Each option is added to the beginning of the Developer menu. The file shown above appears as the first item below:



To remove a python file from the Developer menu, simply alter the first line so it doesn't look like a menu entry. There is no limit to the number of files that can be added, though only a maximum of ten may have key bindings.

Note that the Developer menu will eventually appear only if developer mode is enabled.

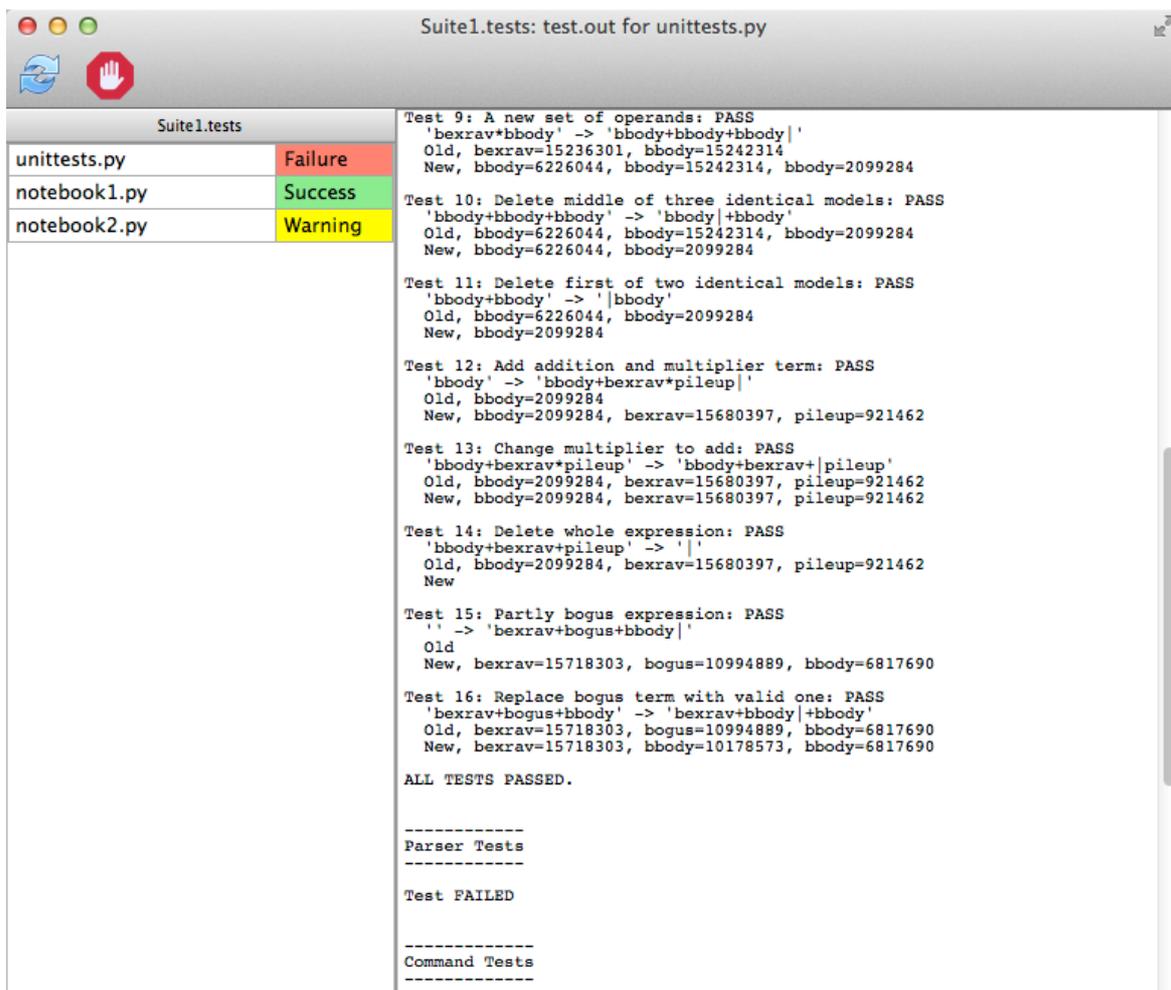
## 12.3.2 Automated Testing

The ATSA supervisor's Developer menu supports a new command, Run Test Suite... This opens a file of type .tests, like this. Eventually a test suite like this may run for several hours, so the intended use case is to kick off a test suite at night and examine the results in the morning.

```
# A test suite is simply a list of python files. Each line starts
# with a timeout (blank = 1 minute), a tab, and a filename. The
# timeout is in HH:MM:SS format. When the .tests file is opened
# via ATSA, ATSA creates a notebook for each python file, and
# runs the specified test. The final status is fed back to ATSA
# for display.
unittests.py
1:10 notebook1.py
notebook2.py
# notebook3.py
```

For this example, notebook1.py has a timeout of 1 minute 10 seconds. unittests.py and notebook2.py have default timeouts of 1 minute. notebook3.py will not run, because it is commented. Timeouts prevent a hung process from stalling the entire test suite. ATSA runs the tests sequentially, and the exit status of each test indicates overall success or failure. Some tests generate a test.out file with detailed results. Others write to the python console, which is saved as python.log.

The test suite viewer is a simple readonly view of the overall status of each test. The test suite is invoked with the Refresh command and cancelled with Abort. As each test completes, its primary results file is shown. The primary file is test.out, if it exists, or python.log. There is no formal format for either file. In the example below, the unittests.py output is shown. The overall status is failure because at least one test failed.



Like the python programs that assist debugging, ATSA's built-in editor may be used to edit the test suite files and the individual tests.

Each run of a test suite creates a directory of the form `YYYY-MM-DD_HH-MM` in `-ATFiles/TestResults`. Within this directory is a directory for each test. The directory contains the uncompressed notebook (which is only up to date if the notebook was saved), together with all the relevant logfiles. Test results are either in `python.log` (if the test writes its output into the python log), or `test.out` (if the file was written by a C++ unit test). If the notebook was saved, the compressed version is not moved into `TestResults`, since the uncompressed files are already present. It is safe to purge the contents of `TestResults` at any time.

## 12.4 Squish Testing

Squish is an automated tool test framework specifically designed for Qt. It performs a higher tier of testing, by simulating clicks on the user interface. This follows code paths that are not followed by any lower test tiers. Among other modes is a one that does pixel comparison of output, so these tests report update glitches, graph anomalies, anything that is out of place. Such tests are not suitable for rapidly evolving interfaces, because screen snapshots have to be updated too often, but for mature interfaces, they exercise the system thoroughly.



# 13 Futures

This section lists longer term activities. At the moment (early 2015) it lists post-demo tasks.

## 13.1 Bits of History

### 13.1.1 Fun with Acronyms

Chispan	
Chi2SPAN	
SPANIELA	SPAN with integrated emission line analysis
SPANEAL	Spectroanalysis with emission and absorption lines
SPANILA	
SPANILBA	SPAN integrated line based analysis
SPANELA	SPAN emission line analysis
SPEALA	SP emission absorption line analysis
SEALBA	Spectro emission absorption line based analysis
SPANIEAL	
LineSPAN	
SPANLEX	SPAN with Line-based-analysis extensions
EALSPAN	Emission Absorption line SPAN
ATOMSPAN	
SPANEAL	Spectrographic analysis with emission and absorption lines
SAWEAL	Spec Analysis with E A Lines
SPECELA	SPECTographic Emission Line Analysis
SPECELBA	
SPEALA	Spec emission/absorption line analysis
XSPECPrime	
SAILBA	Spec Anal Inc LBA
LBASPEC	
SPEC++	
SEAL	Spectro emiss ab line
SPEAL	Same
SPANWADE	Spec Anal D w/ AtomDB Extensions
SPECADE	
SPECALBA	Spec Atom LBA
SAWALBA	
ADESPAN	Atomic DB Enhanced Sp An
SPANADE	Spec Anal w/ Atomic DB Enhancements
SPANIEALA	Sp An inc. emission absorp line analysis
HYPERSPANNER	Extended spectroanalysis tool, play on Star Trek tool
XSPAND	Expanded spectroanalysis tool, or "XSPEC Expanded"
SPANDX	Spectroanalysis w/ Database Extensions
dY/dXSPEC	

## 13.2 General Enhancements

- Hide sidebars for inactive tool editors.

- Update to latest XSpec.
- Update to latest Qt.
- Testing!
- Linux port
- User docs.
- A really good tutorial.

## 13.3 Notebook Enhancements

- If notebook window closed, delete notebook process and any XSPEC subprocesses.
- Export notebook as HTML file. (The value of this is somewhat dubious, but it is the simplest way to support PDF export and printing.)
- Support notebook printing, all tools fully expanded.
- Support notebook printing, tools as shown in notebook.
- Export notebook as PDF.
- Export notebook as LaTeX.
- Export notebook as archive. This looks just like a regular notebook (.cnbk), but includes all “interesting” observations and any custom Python. When an archive is opened, its files are added automatically to the recipients cache of observations. When an archive is saved, the user is prompted as to whether to strip the observations from it.
- Open notebook from a remote machine. This mostly works now, but needs to merge in any included observations and Python programs.

### 13.3.1 Copy/Paste

The unit of exchange between ATLAS users is always complete notebooks. But for a given ATLAS user, there are many other options planned for copy and paste at the tool, spectrum, or model levels. Since the copies may be too large for memory, they are stored in `Clipboard.xml`, somewhere in `~/ATFiles`.

<i>Operation</i>	<i>Description</i>
Copy Observatory Tool	Copies observatory tool so it can be pasted into another notebook, where its search settings and selected files are preserved.
Copy Tag Tool	Copies the tag tool’s tags. When pasted, the tags replace any existing tags. (There can only be a single tag tool per notebook.)
Copy Plot Tool	Copies the plot tool and all its spectra.
Copy Text Tool	Copies the text tool.
Copy Python Tool	Copies the python tool.
Copy Matplotlib Tool	Copies the Matplotlib tool and any program to drive the plot.
Copy Table Tool	Copies the table.
Copy Results Tool	Copies the results tool, but the results themselves reflect the target plots.
Duplicate Notebook	(File menu.) Clone the existing notebook completely, opening it (notebook only) in another window. Allows more refinements to settings while analysis is underway.
Duplicate Spectrum and Models	(Spectrum option.) Inserts a copy of the selected spectrum and all its models after the current one, in the same plot tool.
Duplicate Linked Spectrum	(Spectrum option.) Identical to Duplicate Spectrum and Models, but the duplicate is linked so that all changes to models and parameters (except overrides) are shared between the linked spectra. Used to apply identical models to two or more spectra. Linked spectra may be unlinked, but the

Copy Spectrum and Models	converse is not true. (Spectrum option.) Copies the current spectrum (to Clipboard.xml), where it can be pasted elsewhere, including into another notebook.
Paste Spectrum and Models	(Layers option.) Pastes the clipboard spectrum and models into the current plot tool. Computed results are preserved until changes are made to the duplicate.
Duplicate Model and Params	(Model option.) Clones the selected model exactly for the current spectrum, so the clone can be altered in some way.
Copy Model and Params	(Model option.) Copies the model and its parameters (to Clipboard.xml) so it can be pasted into a different spectrum or a different notebook.
Paste Model and Params	(Spectrum option.) Pastes the model and its parameters into the selected spectrum. Computed results are discarded.

## 13.4 Observatory Tool

- Source for observatories: [http://en.wikipedia.org/wiki/List\\_of\\_space\\_observatories](http://en.wikipedia.org/wiki/List_of_space_observatories)
- This article has information on how to retrieve the “where from” metadata from downloaded files on MacOS: <http://hints.macworld.com/article.php?story=20101206161739274>

## 13.5 Plot Tool and Models

- Implement Duplicate Linked Spectrum.
- How to deal with projection completion?
- Copy/paste: Tool must include full definitions for user models. Gets complicated: requires compares on receiving side, and possible renames if there are conflicts. Need to bring temps across always, but also any perms. Maybe routinely include entire model instance in all saves and copies, then figure out what to do on restore/paste... On paste, if any permanent modelInfos, make sure they exactly match existing, if not, rename. For temps, just re-create.

### 13.5.1 Line-based Analysis

- Implement additional refresh phase to lock model parameters.

## 13.6 Table Tool

- Get rid of selection “shadows”—weird highlighting leftover after a range is de-selected. This is intermittent, so it is probably a bug in `QTableWidget`.
- Accept NumPy arrays returned from Python expressions and display them.
- Accept Python native arrays and display them.
- Surface `Tables`, `TableCells`, and `TableCellFormats` to Python for building better tables.
- Redesign of selection panel: show all options for all units, enabling only those that apply to at least one item in the current selection.
- Add a Set physical type... button which presents a type and units popup. This would be used to enter a bunch of values, then set their type in one operation. This can also be used to strip physical type. No effect on non-doubles.
- Reassign the Physical Type popup to be Interpret as, enabled only when the selection includes values of a uniform physical type such as `QuEnergy`. Then it lists only the legal alternative values without changing the units.
- Support Python lists, display vertically by default, add option to orient the vector horizontally instead.
- Show numpy arrays up to 2D. Need to support complex numbers too (or disallow). Need more optimal table implementation or refuse large arrays. Need to be able to display numbers of any precision. Input not supported, output only.
- `setCell(row, col, '1.5 Å')` fails, Python chokes on the unicode symbol
- If selection spans multiple physical quantities, suppress all physics popups. Currently if it spans two physical quantities, all numeric info vanishes.
- Why are tables slow?

For any selection:

- Cell box (disabled if selection count > 1)

- Show zeros (always enabled, tristate)
- Show units (always enabled, tristate)
- Show errors (always enabled, tristate)
- Significant digits (always enabled, may show mixed selection)
- Exponent style (always enabled, may show mixed selection)
- Set physical type... Dialog shows physical type and units, also says only applies to numbers with no physical type
- Strip physical type... Dialog explains that selected values that have a physical type will have the type stripped, reverting to doubles.
- Interpret as (popup) Enabled only if all in selection have same type/units such as "eV"
- Wrap text: always enabled, tristate
- Label: always enabled, tristate
- Date format: always enabled, tristate
- Settings already stored in TableCellFormat, so they take effect regardless of current cell.

For large selections: Show zeros, show units, show errors, significant digits, exponent style; no physical type or original units. All these settings only effect appropriate data types, so that you can select the whole spreadsheet to make global changes. Also show options for other data types, enabled only if selection contains at least some.

## 13.7 Text Tool

### 13.7.1 Enhancements

- Add ability to paste images into text blocks.

### 13.7.2 Styles

Originally the text tool supported character and paragraph styles, but this is disabled because Qt's word processing code cannot quite be extended to support them. Might work in the future though. So here are some notes I made at the time.

- When a new para style is applied, it does not disrupt masks. When a new char style is applied, mask bits are cleared for each attribute set by the char style. Remove formatting also removes all masks.
- When a new char style is applied, it clears all mask bits that are set in the applied style. So, for example, if a blue color style is applied to a red chemical formula, the text color changes but the subscripts do not.
- Para attributes: 3 margins, space after, alignment, font, size, color, bg color Char attributes: bold, italic, underline, url, sub, super, strikeout, font, size, color, bg color
- When creating a style, each attribute chosen by user sets a corresponding mask bit to cause its setting to be conveyed to the text. Other attributes are inherited if they are set in parents. A single button clears all overridden choices.
- Each style has a master copy and a document copy. When a doc is opened, its styles always override the masters. But when a document style is changed, the master is updated too. Thus the document style replaces the master from then on. The master is saved with prefs; each doc has its own list. For a new document, the master list is cloned to provide the new document's list. For an existing document, only masters that are new are added to the document's list. Edits to an existing doc have no effect on the masters; only edits to an existing document's STYLES make changes.
- Copy/paste between documents. After a paste, the receiving document's styles override the pasted styles.
- User data:
  - Pointer to char style
  - Pointer to para style
  - Char style mask
  - Para style mask
- Copy: everything works if user data preserved.
- Paste: track change in block count, use this to determine which blocks were pasted. Iterate through pasted blocks. If they are untagged, mark them with current styles but block everything, making it a no-op.
- If change non-style attribute, this is an exception to the style, so no marker is needed. Text block is broken into up to 3 blocks. First block's attribute style is unchanged. Second is:
  - If paste and adopt current format, no marker.
  - If paste and retain pasted format, mark with null style to indicate unknown source. Such blocks are not effected by style changes.
  - If convert to HTML, add <style name> markers at begin/end of blocks. How to correlate? If convert from HTML, convert back to markers.
- <http://kernelcoder.wordpress.com/2010/08/25/how-to-insert-ruler-scale-type-widget-into-a-qabstractscrollarea-type-widget/>

```
Para styles
Char styles
CSS mapping
Style management done by block in sidebar
Paragraph Styles
  Parent
  Style
  Margins
  Spacing
  Font family
  Size
Char Styles
  Parent
  Font family
  Font size
  Red
Predefined para styles
  Headings
  Normal
  List-1
  List-2
  Existing...
Predefined char styles
  Monospaced
  Normal
```

## 13.8 Python Integrated Development Environment

- When there is a leading set of four spaces before a tab, the first tab does not expand properly. Looks to be a `QTextEdit` bug.
- Implement stack trace pane.
- Implement breakpoints.
- Implement debugger operations such as single step.
- Need a careful design for which C++ items to surface to python.
- Make sure optional python files are packaged with notebooks.

## 13.9 Nexus Enhancements

No major enhancements planned.



## 14 Release Process

This describes the manual steps necessary to produce and superficially test a new release.

- Bump the version number (`versions.h`).
- Make sure `ATFilesMaster` has up-to-date files.
- Do a complete command line build.
- Test the executable on another machine, either with the existing `~/ATFiles` or with a new version if needed.
- Update the release notes. Mark the release date too.
- Check in the source code.
- Do a second complete build to incorporate the git information.
- Upload the web site.
- STATISTIC: `cstat`, WEIGHT: `gehrels defaults for Xspec.init`.

-- -- -

## 15 Build History

This section records additions and bug fixes for some releases.

### 15.1 ATSAL 0.1.1

#### 15.1.1 Summary

Continued work on Line-based Analysis; miscellaneous bug fixes.

#### 15.1.2 Fixes/additions

- Cmd-R/Refresh command only works in notebook. Needs to work in tool editors and other non-modal windows too.
- Red shift doesn't always default to 0 in Line-based Analysis labels display.
- If `features` is turned on before there is a convolved spectrum is available, don't try to pass any gaussians to XSpec.
- Adding `features` to a model expression turns on peak finder. This happens, but since a refresh hasn't been done, it has no discernible effect. That makes sense, just like other models: you need a refresh to get results.
- Fix display glitches in the parameter editor.
- Add XSpec version to the splash screen.
- Restore `setversion` utility, which alters various header files used to generate the doc set by synchronizing the version numbers with the main build.
- Check that `setversion` works for other generated products.
- Add help for using Line-based Analysis.
- Extend help to describe parameter tweaking and finding detailed info on emission lines.
- Add help for user annotations.
- When `features` is added to a model expression, immediately display peak stubs if convolved data is present.
- Add buttons under peak finder controls to set specific fields.
- Extend label area messages to provide hints as to how to do LBA.
- Added a popup for obscure LBA options. It includes "Restore peak parameter defaults" and "Fine tune." These aren't implemented yet though.
- Model help broke when resources were rearranged.
- When generating help for a model expression, don't repeat help blocks for the same model. Right now the help for Gaussians appears once for each peak.
- Nexus global searches work in debugger but not in release code.
- Fixed some display glitches in the label area.
- "Officially" switch to C++-11 for ATsAL builds, by telling the compiler not to warn about uses of C++-11. This change significantly reduces the spurious warnings in the build logs. (Note that XSpec and PyQt both generate large numbers of spurious warnings.)
- Crashes if refresh with no model. This is a result of the fact that the `xSpectrum` shares access to the first fit's XSpec instance, to avoid the need to create a whole XSpec server session just to produce a convolved spectrum. So for the moment, this has been fixed by refusing to refresh a spectrum that does not have at least one fit. It will be very hard to fix this correctly.
- Save/restore user annotations and spectral lines as part of the notebook. Code is written, and the save file looks right, but reading of this entire file is still on the to-do list.

### 15.2 ATsAL 0.1.2

#### 15.2.1 Changes:

- The makefile build procedure sets the Clang compiler to produce more compiler warnings than the XCode build, so a clean XCode build still produces some warnings in the build log. I removed several dozen such warnings. Note that PythonQt generates hundreds of warnings. These are not worth correcting because the code that generates them is machine-generated. The program's author probably isn't seeing these warnings, as a result of different compiler settings.
- Resolved a bug that caused only the first peak to have a Gaussian computed for it, instead of all of the selected peaks.
- Elaborated the shared spectra redesign. This works out most of the necessary details and solves the problem of exempting some parameters from sharing if desired. `XModelParamTriplet` is now `XModelParamVariants`, and

stores four variant sets of values for each subparameter.

- File Tag Groups have been completely extricated from the source code, in preparation for the [new approach](#).
- ATXSAL uses UUIDs for persistent pointers and signaling. The old mechanism simply used typedefed `qLongLongs`, but this was increasingly awkward. I replaced these with `FooID` classes, trivial classes which combine a type and a UUID, in order to make the code a bit more reliable and maintainable. See `UUID.cpp/h` and [this description](#). This upgrade changed hundreds of bits of code, and may have introduced a few bugs. It also fixed a few.
- Corrected two version skew problems in `ATFilesMaster`, the directory containing the default files used to create a `~/ATFiles` directory for a new ATXSAL user. One was a bug in `default_main.py`, which implements ATXSAL's refresh loop. The other was that the SQLite database lacked a recently added field.
- Corrected a problem establishing library search paths for the deployed version of ATXSAL.
- Corrected a couple of disconnected signals.
- If `~/ATFiles` is present, but `~/ATFiles/Obs/Work/Observations.db.sqlite` (SQLite database) is missing, it is regenerated automatically when the supervisor starts up. This provides a brute force way to force database regen if there was version skew.
- Improved display of peak information in the detailed line display, per suggestions from Randall.
- Added Save Notebook as...
- An open notebook is a directory tree containing (potentially) many files. ATXSAL compresses this tree into a single file at save time. If the user renames the compressed file, when it opens, it expands into the temporary directory with the original name, causing confusing side effects. Now the compressed file is expanded into a quarantine directory, and from there moved to match the current filename.
- Separate `Missions/Instruments` and `MissionProxys/InstrumentProxys` to add necessary flexibility. The proxies add some user interface options to the basic mission info, and multiple proxies share the same mission.
- Save `SearchQuery`s instead of `MissionProxies` as defaults. Introduce two layers of defaults: "factory," and user. The former are loaded from resource file `FactorySearchQuery.xml`; the latter are stored in `~/ATFiles/UserSearchQuery.xml`. The intent here is to allow the user to configure the subset of EM bands and missions/instruments of interest, and use this as a default for new notebooks.
- Added Search Defaults... to the observatory tool. This supports four default setting options.
- Actually wired up the search settings so they generate the correct SQL searches.
- Added a second search mode. Now "quick search mode" accepts a single search string and applies it to all seven (or thereabouts) of the standard database fields. This quick search mode is fine for small collections of observations, but will match too eagerly against larger collections. In "detail search mode," a larger block of query options appear to further constrain the list.
- Some FITS files lack dates, so we decided to include such files in searches constrained to a date range so they do not fall through the cracks. Since these files sometimes lack not only a DATE OBS value but even the empty field, they were not being copied into the MySQL database. Also, when this field is present, its format is inconsistent. Currently I convert any of MM/DD/YY, MM/DD/YYYY, YYYY/MM/DD, YYYY-MM-DD, YYYY-MM-DDTHH:MM:SS.FFFF, or YYYY-MM-DD HH:MM:SS.FFFF into YYYY-MM-DD HH.MM.SS.FFFF, where FFFF is fractional seconds. The supported date range is from 1000-01-01 to 9999-12-31, a limit imposed by MySQL.
- Changed the database format to handle undefined dates. This change requires a new database, `~/ATFiles/Obs/Work/Observations.db.sqlite`. You can force regeneration of the database by deleting the old one prior to running ATXSAL, or by moving the entire `~/ATFiles` directory aside so ATXSAL auto-installs a new one.
- Added drag-and-drop support to the observatory tool. Users can drag archives, files, or folders into the (top) files pane of the observatory tool to copy the files into ATXSAL's `~/ATFiles` area and add them to the database. **When I added drag-and-drop, I disabled the old watched folder implementation, since the two interact. The only way to get new observations into ATXSAL is drag-and-drop, at least at the moment.**
- **The observatory tool is a demo release candidate now.** (This means bugs are actual bugs, not loose ends.)
- **The tag tool is a demo release candidate now.**
- AtomDB is used in an optimized binary format by ATXSAL, as well as in its original format, by XSpec. Only the former was previously bundled into the ATXSAL.app executable. Added the other format, and fixed ATXSAL to point to it. In development mode, ATXSAL depends on the environment variable `ATOMDB_FOR_XSPEC`:

```
ATOMDB_FOR_XSPEC='/Users/tkent/Tom/Projects/ATOMDB/atomdb_v(VERSION)'; export
ATOMDB_FOR_XSPEC
```

(In this string, "(VERSION)" is expanded by ATXSAL, not by the shell.) In release mode, the AtomDB files are located inside the executable bundle.

- Added help buttons to the observatory and tag tools. Previously this button has only been used sporadically, but now the help button is an option for all `CollapsiblePanes`, making it easy to insert context-specific help links at

any level in the sidebar hierarchy. It seems likely that this will become the simplest way to access help in the future.

## 15.3 ATSAL 0.1.3

This release continues to work with notebook save/restore, and moved the text, table, and python tools to demo release candidates.

### 15.3.1 Python Issues

- The Python debugger button, , has gone through several changes in function. When first clicked, it displays the python files pane, the stack trace pane, and the python console. The notebook pane area continues to show the notebook, but the user can select python files for editing instead. Once in this mode, though, it is counterintuitive as to how to return to the notebook. You do this by clicking on the notebook at the top of the files pane, but this doesn't occur to everybody. It doesn't even immediately occur to *me*, and I designed it. Also, you still have to manually collapse the files pane and the console. So the new improved Python debugger button now toggles between full Python mode and normal notebook mode. One-click shopping.
- When opening a saved notebook, python search paths were not restored correctly. This disabled the refresh loop.
- The Python console (part of PythonQt) beeps if the user tries to position the cursor prior to the command prompt, but this prevents copying, so I disabled the beep.
- The Python console accidentally disables command keys for copy/paste. I restored them.
- Python search paths were being set incorrectly for ATsAL's python interpreter. This was traced to a conflict in search path assignment between XSpec's python and ATsAL's, as well as the python interpreter that is pre-installed on the Mac. At development time, the environment variables `ATSAL_PYTHON_ROOT` and `ATSAL_PYTHON_PATH` are now consulted to find relevant files. The deployed application bundle looks in several hard-coded directories inside the bundle. Below is the addition to the developer `.bash_profile`. The listed directories are those self-assigned by python when run under a shell.

```
# At development time, this is used to set up ATsAL's python search path, directed to
# python in the development directory. ATsAL_PYTHON_ROOT is enclosed in parens, not braces:
# it is expanded by ATsAL, not by the shell.
ATSAL_PYTHON_ROOT='/Users/tkent/atsal/proj/atsal/Python-3.3.3' ; export ATsAL_PYTHON_ROOT
ATSAL_PYTHON_PATH='(ATSAL_PYTHON_ROOT)/framework/Python.framework/Versions/3.3/lib/python33

'(ATSAL_PYTHON_ROOT)/Lib:'\
'(ATSAL_PYTHON_ROOT)/plat-darwin:'\
'(ATSAL_PYTHON_ROOT)/build/lib.macosx-10.9-x86_64-3.3:'\
'(ATSAL_PYTHON_ROOT)/lib/python/site-packages' ; export ATsAL_PYTHON_PATH
```

- The python packaging procedure was extended to copy in some dynamic libraries needed by python.

### 15.3.2 Notebook

- In the notebook window, cut/copy/clear/paste/duplicate previously applied only to the notebook. Now these operations apply to the current focus, which may be a console, python console, program editor, or notebook.

#### 15.3.2.1 Notebook Save/Restore

- [Formally specified the format of common data types in XML files](#) in order to smooth out some inconsistencies, and changed XML files and file generators to be consistent.
- Corrected a systematic error in storage and retrieval of `QuDouble` subclasses such as `QuPhoton`. Values were stored in native format (each physical data type has an arbitrarily chosen native units), but were being read back in the units with which they were originally defined.
- Did some refactoring in the `Quantity` classes to resolve some confusion.
- Made many dozens of changes to the save/restore logic to add new information. Work on this is not yet complete.

### 15.3.3 Table Tool

- Fixed a bug that caused an immediate crash on table tool insertion.
- Restored show/hide zeros.

- When a user double-clicks into a table cell to edit it in place, AT&SAL actually creates and overlays a widget for the editing. Previously any text style changes applied to the cell did not apply to the overlaid widget, which is confusing. Now some of those changes do appear, while others do not because they would interfere with editing.
- The preceding change required working out details of how a table cell is represented in [up to four different ways](#) when using the table tool.
- The significant digits popup is wired up.
- The physical type override is wired up. **Needs work though. The purpose of this control is to resolve ambiguity in parsing physical types. For example, if “27 m” is interpreted as meters, but the user meant it as a wavelength, this control sets is appropriately. But the popup menu currently lists *all* physical types, not just those that might be suitable, and if the user setting conflicts with the units, the popup menu is ignored.**
- Physical type changes are now “sticky.” If the user overrides the default guess for an ambiguous physical type (e.g. meters, which could be a distance or a photon wavelength), the new assignment persists through edits.
- Corrected several boundary cases in the logic that decides whether an entered string is a numeric value, a numeric value with errors and/or units, a date, or a string. Added a number of unit tests to verify operation. **I am sure there are more such cases, and this needs additional unit tests.**
- Units ending in “m” are treated as distance, not photon wavelengths, by default. **A better solution might be to make the choice based on magnitude: >1 mm is a distance, otherwise a wavelength.**
- Added a set of [navigator keys](#) to simplify use of the table tool. The key bindings probably won’t match on Mac and Linux, and they need a design review to see if they are reasonably consistent with other programs. The main issue here is that if the user is clicked into a cell for editing, navigator keys implicitly click into the target cell as well. Tab and backtab (⇧-tab) are the most helpful.
- Added help docs for the table tool.

### 15.3.4 Python Tool

- Extended Python tool a bit. The modify flag in the state “LED” works correctly now, and if the Python tool fails because of an error in the Python code, the LED turns red to indicate this. The tooltip also encourages the user to open the Python Debugger to see what went wrong.
- The Python tool (*vs.* the Python development environment) is now a **demo release candidate**.

### 15.3.5 Text Tool

- The text tool plays nice with the refresh logic now.
- Corrected a strange new bug that causes parentless modal dialogs (in this case, both `VariableDialog` and `HyperlinkDialog`) to crash at destruction time. The workaround was to give both dialogs a parent widget.
- URLs added to text tools can be specified in a more lenient format now, and previewed in a web browser.
- Corrected a problem that prevented the notebook pane from updating in response to changes in the text tool editor.
- Added text tool help.
- Text tool refresh status is bumped to warning now if the text contains any undefined or null variables. (Python shows null variables as “∅”.)
- The variable entry dialog previously checked to see if a symbol name being entered exists, and signaled an error otherwise. But this is not reliable because the variable may not exist when it is defined, and because more complex expressions cannot be verified. So I disabled this feature.
- Fixed some boundary cases in the `WordProcessor` implementation.
- Refined the ruler a bit. Among other things, ruler margin settings now snap to 1/8” boundaries.
- As with other tools, there are two copies of each content area: one in the editor, and a readonly copy in the notebook pane. In the editor, clicking a hyperlink displays a dialog for changing it, with a button for following the link. In the notebook pane, clicking the link follows it. In either case, following the link shows it in a web browser, outside of AT&SAL.
- **The text tool is a demo release candidate now.** There are plans to refine this later, with support for images, tables, and possibly styles, but it is adequate for the moment.

### 15.3.6 Tag Tool

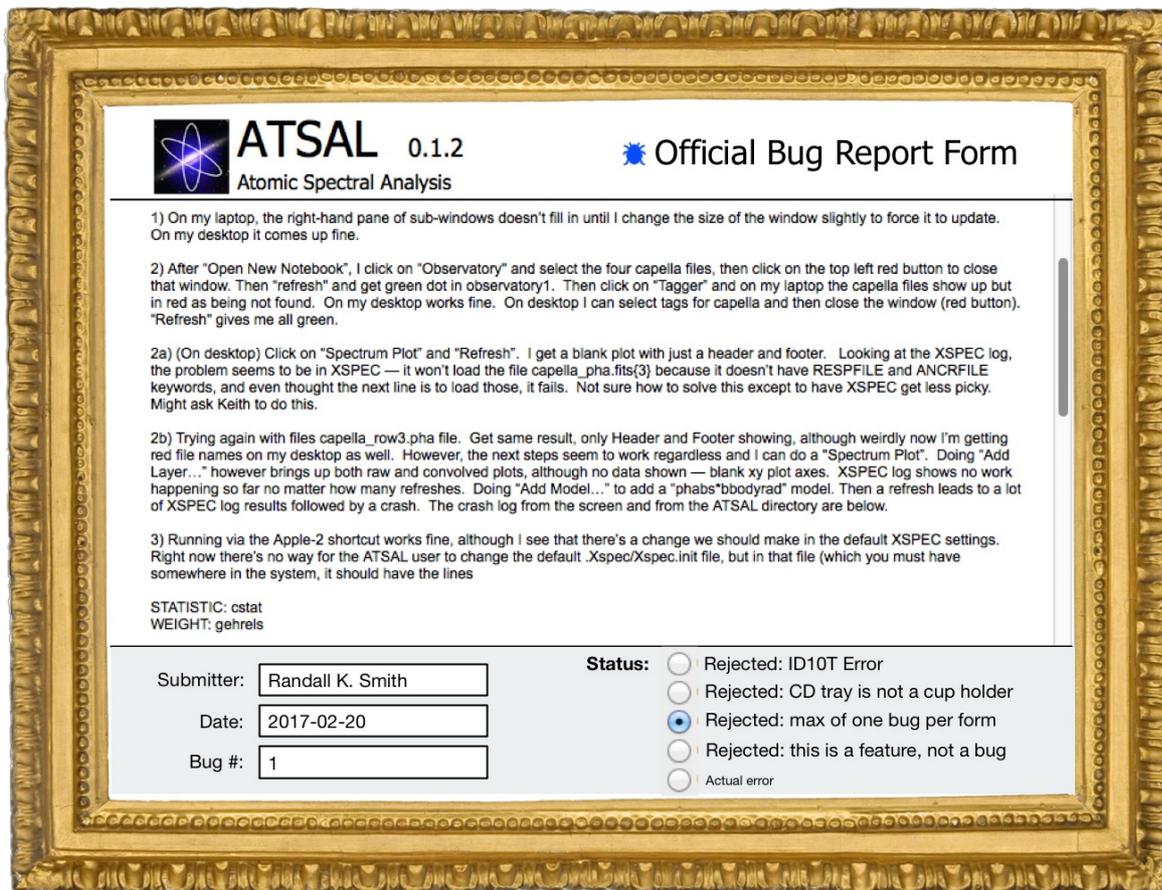
- There is an intermittent bug that causes interesting files that exist to sometimes be highlighted in red, indicating that they do not exist. This appears to have been due to an uninitialized variable, but since it is intermittent, it may not be fixed.

### 15.3.7 XSpec

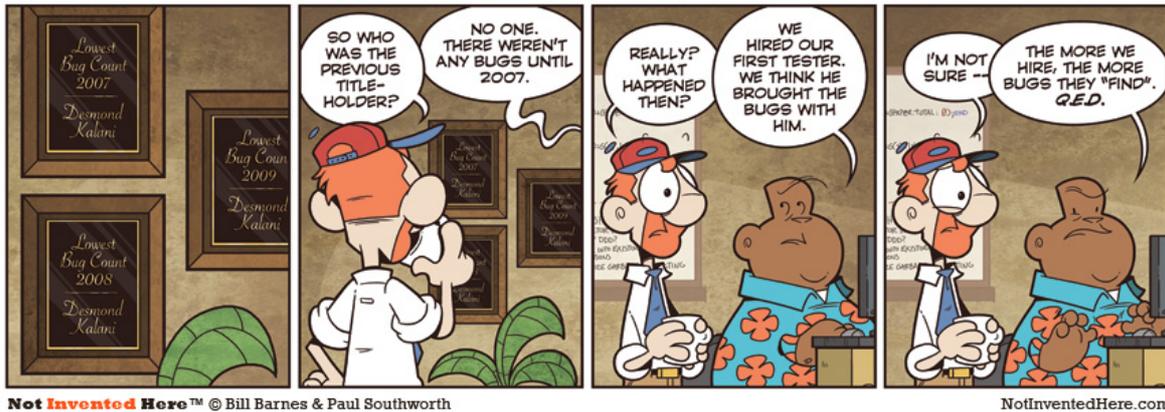
- `xspec.init`, XSpec's initialization file, was not handled properly in previous releases. Now a "factory default" is hard-coded into ATLAS, and there is a user default file in `~/ATFiles/XSpec/default_xspec.init`. Each time a notebook is created, the notebook gets its own copy of the defaults. Values are modified via the new XSpec Settings icon in the notebook window (don't modify by hand!). If you give someone else a copy of your notebook, they will get the same XSpec init values when they run it.
- I mentioned earlier a problem in which the XSpec allocator could transfer an XSpec between notebooks, causing problems if each notebook has an `xspec.init` with conflicting settings. However, I overlooked the fact that the allocator exchanges XSpec `slots`, not actual processes, because `stdio` cannot be reassigned. That is, if one notebook no longer needs an XSpec, the allocator can give the slot to another notebook, but to do so, it deletes the old process and creates a new one. Hence the conflict with `xspec.init` files does not arise.
- The XSpec Settings dialog also provides a way to make the current settings the new user default.
- There is no parameter validity checking at the moment.
- Changes to the current notebook are not presently noticed by XSpec unless they precede the first refresh command. Later I will change this so that a new XSpec is instantiated after any change is made to the init file.
- As a means of verifying that XSpec is seeing the notebook's `xspec.init`, I added a `cosmo` command to the command sequence that is issued when a model is refreshed. So if you change a cosmological constant prior to the first refresh, you will see the changed value in the XSpec logfile.

### 15.3.8 Bug Report #1

It seemed suitable to "bronze" ATLAS's first formal bug report. ATLAS is finally coming of age!



And Randall's reply to my note encouraging him to submit as many bug reports as he can:



## 15.4 ATSA 0.1.4

### 15.4.1 Plots

- Selections are now constrained to remain within the electromagnetic spectrum.
- Corrected several cases where changes in the parser logic were not yet reflected in the parserator tests, so the tests were failing.
- Saving of XSpec logs now has the option, enabled by default, to strip out timestamps, making it easier to use these logs in the parserator.
- Fixed a bug in which XSpec logs repeatedly list a pending long-running command on each status update, making the logs difficult to use in the parserator.
- The use of cstat instead of  $\chi$  as the statistic to minimize uncovered several places where the parser was not flexible enough. And justified the existence of the parserator.

### 15.4.2 Capella Files

#### 15.4.2.1 Case 1: PHA file exists, RMF/ARF do not

If the file `capella_row3.pha` is autotagged, its file dependencies, `acis_hegm1.rsp` and `acis_hegm1.arf`, appear in red because they are not found. This behavior is correct because they do not exist in ATSA's observatory directory. If this tag is selected for a spectrum and refreshed, it should turn the status LED red, and hovering over it should describe the problem. This appears to be working as intended.

#### 15.4.2.2 Case 2: PHA file does not specify RMF/ARF, not specified by user either

If `capella_pha2.fits` is selected, its metadata does not specify any RMF or ARF. Presently this is treated as legal by ATSA. XSpec returns 0 counts and convolved data, though it does return some data for a fit. Previously ATSA crashed if this was refreshed. The crash has been fixed. **What should ATSA do in this case, where no files are specified in the metadata and the user has not specified any either?** The XSpec log is below. (Note that logs truncate the data to a couple of lines.)

```
12:32:07 Initializing log XSPEC plot1 0.0 (ID 4B01) on 2017-03-01, times in UTC
readline off
xset apcroot /Users/tkent/Tom/Projects/ATOMDB/atomdb_v2.0.2/apc_v2.0.2
cosmo

Cosmology parameters: H0 = 70 q0 = 0 Lambda0 = 0.73
data /Users/tkent/ATFiles/Obs/Observations/capella_pha2.fits{1}
setplot energy
tclout plot counts x 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot counts xerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
```

```

tclout plot counts y 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot counts yerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
# Finish
setplot energy
tclout plot data x 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot data xerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot data y 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot data yerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
# Finish
model phabs*bbodyrad & /*

=====
Model phabs*bbodyrad Source No.: 1 Active/Off
Model Model Component Parameter Unit Value
par comp
  1 1 phabs nH 10^22 1.00000 +/- 0.0
  2 2 bbodyrad kT keV 3.00000 +/- 0.0
  3 2 bbodyrad norm 1.00000 +/- 0.0

renorm
***Renormalization cannot be performed when multiple fit statistics are in use.

fit 100
setplot energy
tclout plot model x 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0.106617 0.120728 0.136706 0.154798 0.175285 0.198484 0.224752 0.254498
0.288179 0.326319 0.369506 0.418409 0.473784 0.536488 0.60749 0.68789 0.77893
0.882018 0.99...
tclout plot model model 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0 0 0 0 0 2.24235e-42 1.31714e-30 2.17892e-22 4.81335e-17 9.30087e-14
1.07478e-10 1.06866e-08 3.26626e-07 1.42954e-06 8.92306e-07 7.94623e-06 2.54162e-05
8.42461e-05...
# Finish

```

### 15.4.2.3 PHA doesn't specify RMF/ARF but user does

This is shown for tag fo, below.

Tag	PHA/FITS File (f)	Response Matrix File (r)	Ancillary Response File (a)	Corrections File (c)	Background File (b)
f0	capella_pha2.fits	capella_row3.rmf	capella_row3.arf	<None>	<None>
f1	capella_row3.pha	acis_hegm1.rsp	acis_hegm1.arf	<None>	<None>

I assume that XSpec isn't returning data because of its rejection of the response and arf commands, but I don't understand why it is rejecting them. Here is the log.

```

12:56:56 Initializing log XSPEC plot1 0.0 (ID 7E42) on 2017-03-01, times in UTC
readline off

```

```

xset apecroot /Users/tkent/Tom/Projects/ATOMDB/atomdb_v2.0.2/apec_v2.0.2
cosmo

Cosmology parameters: H0 = 70 q0 = 0 Lambda0 = 0.73
data /Users/tkent/ATFiles/Obs/Observations/capella pha2.fits{1}
response /Users/tkent/ATFiles/Obs/Observations/capella_row3.rmf

XSPEC: Request to modify response for spectra not
yet loaded ... skipped
arf /Users/tkent/ATFiles/Obs/Observations/capella_row3.arf

XSPEC: Request to modify ARF for spectra not
yet loaded ... skipped

setplot energy
tclout plot counts x 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot counts xerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot counts y 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot counts yerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
# Finish
setplot energy
tclout plot data x 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot data xerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot data y 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
tclout plot data yerr 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0
# Finish
model phabs*bbodyrad & /*

=====
Model phabs*bbodyrad Source No.: 1 Active/Off
Model Model Component Parameter Unit Value
par comp
1 1 phabs nH 10^22 1.00000 +/- 0.0
2 2 bbodyrad kT keV 3.00000 +/- 0.0
3 2 bbodyrad norm 1.00000 +/- 0.0

renorm
***Renormalization cannot be performed when multiple fit statistics are in use.

fit 100
setplot energy
tclout plot model x 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0.106617 0.120728 0.136706 0.154798 0.175285 0.198484 0.224752 0.254498
0.288179 0.326319 0.369506 0.418409 0.473784 0.536488 0.60749 0.68789 0.77893
0.882018 0.99...
tclout plot model model 1
set atsal_temp $xspec_tclout
puts server1 $atsal_temp
0 0 0 0 0 2.24235e-42 1.31714e-30 2.17892e-22 4.81335e-17 9.30087e-14
1.07478e-10 1.06866e-08 3.26626e-07 1.42954e-06 8.92306e-07 7.94623e-06 2.54162e-05

```

```
8.42461e-05...
# Finish
```

Now ATSAAL handles this by reporting the problem as a warning, and continuing execution. **Not sure if this is right.**

#### 15.4.2.4 Case 4: Another variant of manually supplied RMF/ARF

XSpec also rejects the case where the manually supplied RMF/ARF is associated with the other PHA file, f1 as shown below:

Tag	PHA/FITS File (f)	Response Matrix File (r)	Ancillary Response File (a)	Corrections File (c)	Background File (b)
f0	capella_pha2.fits	<None>	<None>	<None>	<None>
f1	capella_row3.pha	capella_row3.rmf	capella_row3.arf	<None>	<None>

Now ATSAAL handles this by reporting the problem as a warning, and continuing execution. **Not sure if this is right.**

#### 15.4.3 S54405.pha

- Clicking “Ignore bad data” crashed with a zero-based vs. one-based numbering bug.
- The new plot logic for handling plots with multiple data points per horizontal pixel location worked incorrectly when the number of data points was less than or equal to one per pixel, causing the plot of this data to appear incorrectly clipped.
- The XSpec message “Warning: cstat statistic is only valid for Poisson data. Source file is not Poisson” is now recognized by the parser and brought out to the user by turning the LED yellow. This is mostly just a test of how errors and warnings are plucked out of the output stream and conveyed to the user. ATSAAL continues to blissfully ignore most such warnings.

#### 15.4.4 Miscellaneous

- Added Close Window command for notebooks, all the tool editors, and the parserator.
- Fixed incorrect tooltips in XSpec settings dialog.
- **Bug:** If I put in “phabs\*tbodyrad + features” into the model o field without turning on the peak finder, I get an error that I can’t add multiplicative models. Turning on peak finder makes this go away. Then changing tbodyrad to brems gives the same error until I retype it a few times, removing features and then putting it back. Then it works. Not sure what’s happening here. **Fix:** I fixed a parser bug and added a lot of regression tests for various kinds of model expressions, but it is not clear if I actually resolved the problem. I reproduced the problem prior to the fix, but have not subsequently been able to reproduce it, so I *think* it is corrected. If it recurs, is the failure in the little sidebar box, or in the model editor?
- One more model expression error condition is detected now: multiple operators in succession.
- The results dialog could appear as too large to fit some monitors, particularly those of laptops. Now, the dialog’s height is constrained to somewhat less than the height of the smallest connected monitor. On a Mac, the dialog appears as a “sheet,” dropping from the top of the parent window, but if the parent window is too low on the monitor, it moves up temporarily to expose the entire dialog.
- When the emission line details window was closed via the Close Window menu, the plot tool editor closed instead. To fix this, the `LineDetailsEditor` class was subclassed from `ATSAALMainWindow` in order to retain menu constancy.
- Some Edit menu options were enabled in situations where they should not be.
- Fixed a crash when clicking on a header in a notebook pane.
- Improved the error accumulating system to pick out a few more errors occurring in XSpec output, and to filter out redundant errors.
- **Bug:** Click on each tool icon in left-to-right order, then click the observatory icon again. Refresh crashes. **Fix:** This was a bug in the table tool.
- Refreshing when running orphaned puts up a warning now instead of hanging indefinitely. (This only comes up when debugging the notebook without a supervisor, so users won’t encounter this. The fix is there to prevent me from staring blankly at the screen for a few seconds before realizing my mistake.)
- The Parserator prohibits updates in standalone mode, since this writes into the development tree and this tree may not be present. (Eventually the Developer menu will be hidden in standalone mode.)

### 15.5 ATSAAL 0.1.5

## 15.5.1 Fit Sub-plots

In this release I revisited the issue of how to display various alternative plots associated with a fit. (This came up because the XSpec tutorial shows a residuals plot early on, a plot type not yet supported.) Specifically, I thought about how to solve the problem of displaying lots of plots in the limited space (four zones) available for each plot tool. I settled on an approach [described here](#) (after the table on plot types). [Using the last two zones as temporary buffers for the two most recently chosen plot types is an experiment.](#)

- Added items to the fit details menu for fit plots.
- Choosing an item replaces the least recently used plot in zone 2 or 3 with a stand-in for the plot, except for the residuals plot, which is implemented.
- The residuals plot subtracts the convolved data from the fit using an algorithm implemented in `QuPoints::QuPoints(const QuPoints& minuend, const QuPoints& subtrahend, QuPoints::MatchMinuendPoints)`. The curve subtraction algorithm is [described here](#). [This represents a bit of a departure for ATLAS, in that it relies on ATLAS to do a calculation, albeit a simple one, that would otherwise be performed by XSpec. Upside: the plot appears instantly. Internally, ATLAS can avoid long chains of dependencies that cause a lot of computes from XSpec, slowing response time. Downside: in general, results from ATLAS cannot be considered fully trustworthy without solid testing. Hence there will always be pressure to replicate some of XSpec's functions in ATLAS.](#)

## 15.5.2 General

- Added `nativeUnits()` and `nativeValue()` functions to all physical data types, to simplify some operations involving operating on pairs of values, where the units are not important. By using the native units, precision is better retained because there are fewer superfluous conversions. This also makes it easier to write algorithms that operate on multiple physical quantities whose physical type is known, but whose units are not.
- Fixed a bug in which dates were not being saved in UTC format.
- Improved uniformity of notebook XML file.
- Corrected other problems in XML files.
- C++ defines enumerated data types by giving each entry a name, as in this example:

```
enum PhotoAbsorptionCrossSection
{
    PACSVern,
    PACSBcmc,
    PACSObmc
};
```

The names have associated numeric values, numbered by the compiler. It is often necessary to associate each enumerated type with a string for the user interface, which appears, for example, in popup menus. So I added an enumeration preprocessor to ATLAS. The example above looks like this in the file processed by the enumerator utility:

```
enum PhotoAbsorptionCrossSection
{
    PACSVern,    "Verner et al. (1996, ApJ 465, 487)",
    PACSBcmc,    "Balucinska-Church & McCammon (1992, ApJ 400, 699) with new He cross-section
from (1998, ApJ 496, 1044)",
    PACSObmc,    "BCMC with the old He cross-section"
};
```

The utility generates a converter class for each enum as well as a ready-to-use popup menu. The user-facing string is also used to save a setting in XML and preference files, in order to make the setting readable to a human, and to protect against changes in ordering. So instead of, say:

```
<Settings photoAbsorptionCrossSection="2" />
```

you get:

```
<Settings photoAbsorptionCrossSection="BCMC with the old He cross-section" />
```

But this doesn't work as well in practice as I had hoped. It means that if the phrasing is changed for any reason, for example, to fit a limited display space, the mechanism is no longer backward compatible without modifying the code to accept both the old and new phrasing. So I extended the enumerator utility and the ATSal code base to save the enumerator name instead. For this example, you get:

```
<Settings photoAbsorptionCrossSection="PACSObmc" />
```

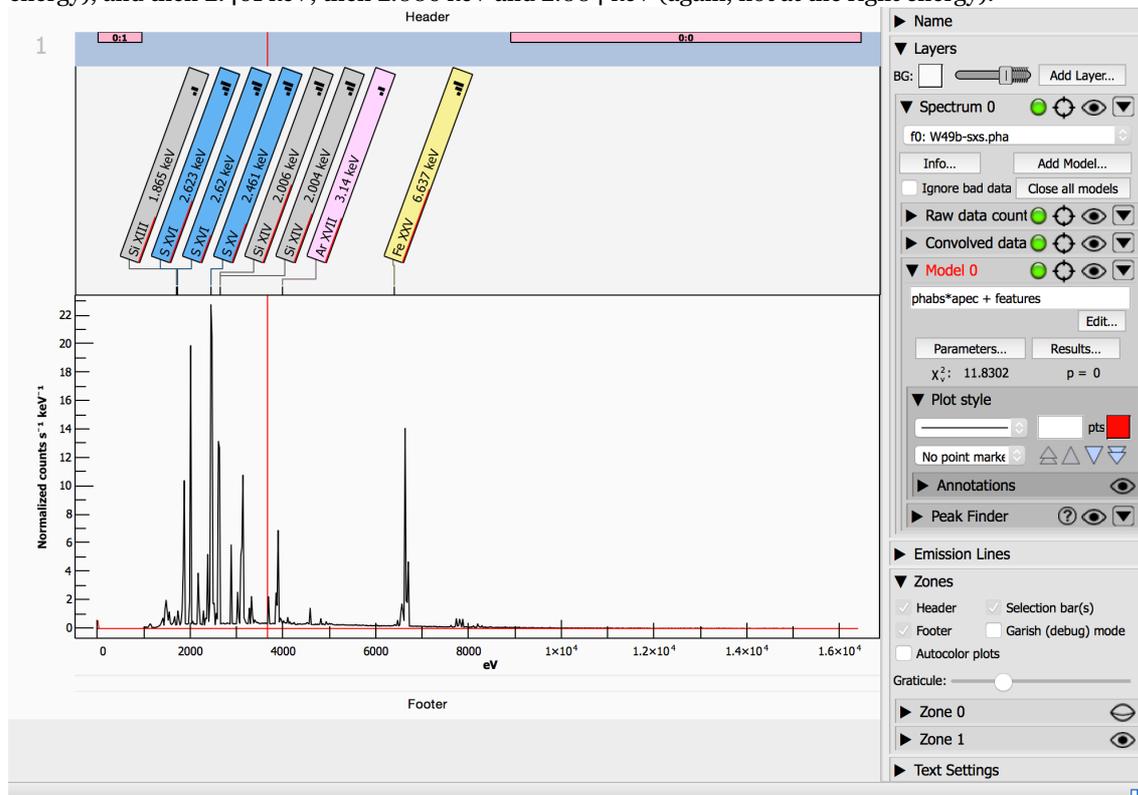
These are far less likely to change (and adding new ones is backward-compatible), so this improves maintainability and is still fairly human readable. (The "PACS" part of the name stands for photo absorption cross section, and is a simple means of reducing name conflicts, since these symbols have global scope.) Other changes made in tandem improve code readability and save/restore efficiency.

This change breaks the file `~/ATFiles/UserSearchQuery.xml`. You can fix it by deleting this file before running this version of ATSal. The file will be recreated, but you may need to recreate your default search settings.

- Corrected a problem in `TableWidget`, where responding to `selectionChanged` events was masking a `QTableWidget` function.
- Got rid of a spurious directory in `~/ATFiles/Logs`. A new notebook, e.g. `Untitled-15`, creates `~/ATFiles/Tmp/Untitled-15` to store files during a session. But a second, empty directory was being created: `~/ATFiles/Logs/Untitled-15`.

### 15.5.3 Line-based Analysis

- In the Peak Info box, the header "Energy (Å)" should be labeled "Wavelength (Å)" or possibly "Lambda (Å)". It is an energy, to be sure, but it's going to confuse folks.
- **Bug:** Using `W49b-sxs.pha`, then fit a `phabs*apec + features` model, using the default values. I also set two ignore regions where we don't have much data. Looking at the identified features, they aren't in energy order. From left to right, they are at 1.865 keV, 2.623 keV, 2.62 keV (although neither seem to be pointing to that energy), and then 2.461 keV, then 2.006 keV and 2.004 keV (again, not at the right energy).



I have seen this unsettling bug previously, but had difficulty reproducing it. The case described here is easily reproduced, and unrelated to ignore regions, making it easier to isolate. **Fix:** This was a problem in calculation

order, leading to an uninitialized variable problem.

- **Bug:** Also with W49b-sxs.pha and phabs\*apec + features, clicking on the yellow Fe XXV brings up the Ar XVII data, no matter where I click. Clicking on Ar XVII brings up Si XIV, and so on. I'm feeling a fencepost error somewhere. **Fix:** This was another manifestation of the bug described above.
- Emission line label placement is sometimes random. This older bug report is also believed to derive from the same problem that caused the previous two reports, but, because it is intermittent, I cannot be sure it is resolved.

## 15.5.4 W49b-sxs.pha

- **Bug:** Clicking on “Ignore bad data” leads to an instant crash. **Fix:** There are a surprising number of boundary cases in the code for handling ignore regions. I fixed a couple of bugs, but a more general solution is planned post demo. See [Include Region Design Refinement](#).

## 15.6 ATSal 0.1.6

### 15.6.1 Notebook Save/Restore

Notebook save/restore is definitely not finished, but it is stable enough so that a saved notebook now has a reasonable chance of opening into a similar state. Notebooks are kept in a directory tree in `~/ATFiles/Tmp` while open; notebook `foo` appears as directory `~/ATFiles/Tmp/foo/*`. After a save, the directory is compressed into `~/ATFiles/foo.cnbk` (“cnbk” means “compressed notebook”). When you quit ATSal, the temporary directory is usually deleted (unless left in place for debugging). So to examine a notebook, save it, then look at `~/ATFiles/Tmp/foo/Notebook.cnbk` before quitting ATSal.

Round-tripping—opening a saved notebook, saving it again, and comparing the two files—verifies that the data are being reread successfully, but this technique does not prove that all the necessary data are being saved in the first place. Many settings pertaining purely to layout, such as window size, plot height, subpanel open/close status, etc., are not presently saved. **It is still too early to support backward compatibility.** But it is far enough along now for experimental use, and crashes should be reported.

Recent changes:

- Fixed a bug restoring values saved in new enum format.
- Corrected a notebook save problem associated with the new fit sub-plots: these plots are now saved and restored.
- Corrected a bug where opening a notebook and saving it again duplicated each spectrum and its contents.
- After opening a saved notebook, LBA labels were placed improperly.

### 15.6.2 General

- Previously counts plots were not resizable. Now all plots are resizable. This is not very intuitive at the moment, but hovering just above the horizontal axis line of a plot turns the cursor into a vertical resizer, and adjusts the height of the plot above.
- Previously, plot window height could not be adjusted if the plot window was empty.
- The plot line label area is resizable now, adjusted by clicking the line at the bottom.
- Make results tables better immune to various parse errors.
- The default height of the line label area now allows for all four fields (ion name, energy, levels, and emissivity). **Might be better to make this shorter though?**
- Many small changes were made to the label rendering logic to produce reasonably intuitive behavior during resizing.
- If a new manual tag was created, and not assigned to any files, the plot tool crashed. Now the crash is averted, and refreshing a notebook with a bad tag fails on the tag tool, never reaching the plot tool.
- Find and Find next are implemented for logfiles now. This required the introduction of the `EditorPlus` mix-in class and subclasses `TextEditPlus` and `PlainTextEditPlus`. The subclasses bring a uniform interface to search/replace. Now searching applies to the current `ProgramEditor` or the current `ConsolePane`, whichever is the focus, and different dialogs are used for read-only and read-write files. The fix applies to Python source files, the Python console, XSPEC logs, the merged log, and the observatory log.
- In the tag tool sidebar, a spectrum count popup no longer appears if no file has been chosen for a manual tag.
- **Bug:** On my laptop, the right-hand pane of sub-windows doesn't fill in until I change the size of the window slightly to force it to update. On my desktop it comes up fine. **This is fixed for a case I observed. If it still happens for other cases, which tool editor fails, and does it happen upon initial display of the editor, or later re-display?**

- Now that additional plot zones are in use, the “photon cursor” (red vertical cursor) can appear in more than one zone, making it ambiguous which one is associated with the emission lines panel in the sidebar. This was changed so that clicking into a given plot shuts off cursors in the other zones. **On one occasion I saw the emission lines panel in the sidebar stop tracking the motion of the photon cursor, but I am having trouble reproducing this. If you happen to notice it and can reproduce it, please let me know.**
- Added line type of “None” to plot line styles. This omits the connecting line segments between points in a plot.
- The parameter editor now performs modify checking and prompts for saving only if there are changes. (This case occurs when the editor is dismissed using the close box, not the OK/Cancel buttons.)
- Parameters are now frozen in the second phase of LBA by negating their signs in the `newpar` commands, instead of using `freeze/thaw` commands.
- The hyperlink color was made brighter, because the prior choice was almost indistinguishable from black.

### 15.6.3 Spectrum-level XSpecs and Phases

The existing design shares an XSpec process between a spectrum and its first fit. This is awkward, for a variety of reasons internal and external. To the user, it means that you cannot obtain counts or convolved data for a spectrum without first creating a model. Internally, it complicates the state table for processing chains of commands. So now I dedicate an XSpec process to each spectrum as well as to each fit.

- Each `xSpectrum` now has its own `xProxy`. A user can add a spectrum and hit refresh, and get the spectrum-level plots.
- The state table has been augmented with “phases,” which represent processing milestones. This was a significant change, and it is not fully debugged. See [Refresh Phases](#). The modified design handles long-running fits, spectrum-level refreshing, and the multiple steps necessary for LBA. **For debugging, the current phase is shown in blue in the area where model expression errors appear. This will probably be removed later.** If there is a model expression error, it takes precedence over the phase information.
- The spectrum options menu now includes Kill XSPEC, Show/hide XSPEC logfile, and Save XSPEC logfile.... These perform the same functions for the spectrum level XSPEC that their counterparts perform in the model panel.
- Adjusting temperature line group or redshift for LBA reverts the phase so that new Gaussians will be fitted on the next refresh cycle.
- Adjusting peak sensitivity for LBA also reverts the phase so that new Gaussians will be fitted on the next refresh cycle.
- Sending XSpec a  $\wedge C$  (via menu) for a fit in progress leaves the state modified so it will be recomputed on refresh.
- At the spectrum level, it is now possible to abort the spectrum-level XSpec, or the spectrum and its fits; or to kill the spectrum-level XSpec, or the spectrum and fits XSpecs. This affords finer-grained control in support of long-running fits.
- Corrected several problems in previously undebugged code that handles shutdown by actual process deletion, and subsequent recovery. There remains a bug involving the OS locking access to a socket after process termination that is unresolved, though.

### 15.6.4 XSpec

Files `outfe.fits` and `xspectr1.6` were among those recently discovered to be mysteriously missing. I looked at the build tree and these files were not present anywhere; neither were they generated by some other program in the build process. Eventually I compared the original master source tree for XSpec against the current tree, discovering that an entire directory, `heasoft/Xspec/src/spectral`, was not present! Even more perplexing, it *never had been* present, since git had no record of it. I restored the missing directory and did a new build, and the missing files reappeared. I traced this to the `.gitignore` file, which blocked a generated directory, with the unintended side effect of blocking a necessary source directory as well. The files are now checked in. No history was lost.

### 15.6.5 Long-running Fits

- Added a design note—see [Fits of Pique](#)—on an extension to ATSA’s current design to handle very long-running fits.
- Implemented support for adding new models while existing fit(s) are in progress. The refresh command is always enabled. New spectra and models may be added and configured, and a refresh adds them to any existing fits. This is not fully debugged, just a proof of concept.
- Resolved a problem parsing fit results for `TBabs*vnei`.

### 15.6.6 Developer Aids

State table debugging is very time consuming, so this release adds some features intended simply to save some development time.

- ATXSAL—and I sincerely hope you will appreciate the irony—now supports command lines, via a new developer menu: “Create Notebook from TestConfig.ats.” It opens `~/ATFiles/TestConfig.ats`, and reads a very simple syntax used to set up a list of observations, tags, and insert spectra and models. At the moment it saves time while debugging, but it will be extended to help with testing in the future. It took 30 minutes to write and will save an estimated 20 minutes per day.

Sample file:

```
// This is a very simplistic mechanism to speed up debugging. It does a bit of
// setup. When the developer menu option Load TestConfig.ats is invoked, this
// file is read from ~/ATFiles/TestConfig.ats.
//
// obs file [, file, ...]
//   add observation tool, marks the specified files
// tags auto
//   add a tag tool, generates automatic tags. Note that tag order may not
//   correspond to file order above.
// plot
//   inserts a plot tool. Following spectrum and model commands apply to this tool.
// spectrum [tag]
//   adds a spectrum. If no tag, the first tag (f0) is selected.
// model [expr]
//   adds a model to the current spectrum. The model expression is set to expr.
// open [tool#] // tool# is zero-based
//   opens the specified tool's tool editor

obs W49b-sxs.pha, s54405.pha, s54405.rsp
tags auto
plot
spectrum
model
model phabs*apec
spectrum f1
model phabs*bbodyrad
open 2
```

- Added a new application to the ATXSAL application suite: `RedButton`. Here it is in operation (that’s part of a waterfall in the background):



Clicking the red button kills all running ATXSAL supervisors, notebooks, and XSPECS. (XSPECS that are not parented by notebooks are left untouched.) Yes, that’s a weird-shaped window. The little tab on top is for dragging. `RedButton` is a debugging aid. It is included in the release package, but hidden inside the `ATXSAL.app` top-level directory. Also makes a very satisfying sound of a 12 Ga Winchester shotgun.

- Added “Reduce Notebook Plot Refresh” to the Developer menu. Each time the plot tool editor plots are refreshed, a the second copy of the plots in the notebook pane are also refreshed. Any breakpoints in the plot classes are entered twice as often as a result. This turns off the notebook refresh to avoid the extra breakpoints. A few drawing artifacts appear in the notebook pane as a result.
- Added “Developer Mode” option to Preferences. This does something to assist debugging. Exactly what is undefined and likely to change without notice. At the moment, it enables “Reduce Notebook Plot Refresh,” described above.

## 15.7 ATXSAL 0.1.7

- Corrected a problem in the logging code that led to crashes, especially when running multiple simultaneous fits. Now the logfile writes are protected by a mutex, since `QTextStream` is not thread-safe. As a result of this fix, **ATSAL can now run multiple simultaneous fits (up to 40 have been tested), apparently reliably.**
- If ATSAAL is shut down abruptly (via `kill -9` or equivalent), any sockets in use produce a socket connection refused error if ATSAAL is restarted within 20 seconds or so. This isn't a common case in normal use, but it slows debugging considerably, since once the error occurs, it is necessary to restart ATSAAL, sometimes more than once. The correction for this problem was fairly obscure, and is presently Mac-specific: see [Socket Connection Refused Errors](#).
- The maximum number of lines per console logfile is now settable via Preferences. The limit applies to the logs as viewed within ATSAAL. Copies of the logs written to disk are not truncated.
- Added a sample `~/ATFiles/TestConfig.ats` file. This file is executed when the user chooses Create Notebook from `TestConfig.ats` from the Developer menu.

## 15.8 ATSAAL 0.1.8

### 15.8.1 General

- Update docs on LBA.
- Fixed the text that appears in the label area when there are no labels because the Gaussians haven't yet been computed.
- A bug in the logic that communicates with XSpec processes caused some commands to be missing from, or redundantly recorded in, logfiles. This bug caused small variations in the logfiles associated with each fit. Now that it has been fixed, these logs can be compared against each other to verify that all the (identically configured) fits produced identical results.
- ATSAAL saved and restored a notebook with results from 24 fits—an 81 MB notebook—successfully. At least it *looks* successful.
- Now that the number of XSpec processes is no longer limited, a setting to limit them was removed from Preferences.
- In Release 0.1.6, I added a simple syntax for configuring ATSAAL tests, and `TestConfig.ats`, a sample file showing the syntax. The immediate goal is to save time during testing, by initializing a particular notebook test configuration in a single keystroke. Then it dawned on me that this **should be a python program**, not a special-purpose scripting language. This is an important change, because it introduces a new tier of testing capabilities. By restricting it initially to developers, there is time to refine the interface before exposing it to end users. And capabilities of this sort will be necessary for users to create larger-scale parallel analysis notebooks. Hence the prior test mechanism has been replaced, a significant sub-project. The python design issues I considered are discussed in [ATSAL Python Approach](#). The extensions are documented starting in [Python Notebook Extensions](#).
- Notebook toolbars are no longer movable or floatable.
- Corrected a recently introduced bug that prevented the text cursor from appearing when clicking into a text editor or log window.
- Corrected a bug in auto-completion in the Python console window. (Completion is not a pre-demo goal, but it was already implemented as part of PythonQt.)
- Deleting a python source file deleted it and removed it from the list, but did not also remove the associated editor, if any.
- A crasher when clicking on an empty graph was corrected.
- The `enum` utility has been extended to optionally publish enums to python as well as to C++. See [Publishing Enums](#). This makes it faster to make enums python public, and guarantees that the python enums remain in sync with their C++ counterparts.
- Added python-visible functions to change all relevant `xspec.init` settings prior to creating a notebook. For example, `notebook.setWeightingTechnique(at.WTStandard)`. Another new function saves the changed settings to the notebook's `xspec.init` file, where they will be seen by XSpec. Example:

```

80 def test5():
81     # For this older data, we have to restore the standard
82     # XSpec defaults.
83     notebook.setWeightingTechnique(at.WTStandard)
84     notebook.setStatisticToMinimize(at.MSChi)
85     # This save is needed, or XSpec won't see the changes
86     notebook.saveXSpecInitSettings()
87     obsTool = notebook.insertTool("Observatory")
88     obsTool.selectFile("s54405.pha", True)
89     tagTool = notebook.insertTool("Tag")
90     tagTool.autoTag()
91     plotTool = notebook.insertTool("Plot")
92     spectrum = plotTool.addSpectrum("f0")
93     fit = spectrum.addModel("phabs(powerlaw)")
94     notebook.showToolEditor(2, True)

```

- Corrected a bug in PythonQt, which converted a string via `toLatin1()` instead of `toUtf8()`, preventing a python program from working properly with Unicode. **I found many more such conversions in the PythonQt source tree, but I am reluctant to make larger scale changes until I understand whether there is a reason for the constraint. It looks as if PythonQt is unnecessarily blocking use of Unicode, which is otherwise supported by the other components. This might be because older versions of python did not support Unicode.**
- Did a first cut at documentation for the **Python ATSal Library**, ATSal functions and classes surfaced to the python layer. Unfortunately, this documentation must be maintained manually, because the transliteration of C++ headers to python is not easily automated. This library layer distinguishes between functions designed for developers for testing, and those designed for end users.
- I found a way around an awkward solution to publishing ATSal classes for use by python. The solution involved pairs of objects: an owning object with python-compatible functions, and an owned object private to ATSal. This approach was based on a misunderstanding of PythonQt. In practice, the python-public class, called a decorator class in PythonQt parlance, need be instantiated only once, not once for each instance. This allowed me to abandon the pairing scheme and spend a brain-numbing afternoon extricating this code. The result is a little simpler and a lot more consistent. **The extraction may have introduced a couple of bugs.**
- I considered the issue of object ownership more carefully. Loosely speaking, if ATSal creates an object, it owns it; and python owns python-created objects. (Ownership here is in the context of who has responsibility for deleting objects when they are no longer needed.) Newer versions of PythonQt allow for transferring of ownership, but the version I am using does not support this. So the problem is solved, but not yet implemented.
- Tools restored from a saved file are now properly surfaced to python programs.
- A new method of adding debug-assisting python programs replaces and extends one introduced in the previous release. Now `~/ATFiles/Libraries/Tests/Menu` contains any number of python programs that set up a new notebook for debug or testing. ATSal scans this directory at startup. A special first-line comment, if present, places the program on the developer menu. For example, if program `ChandraLBA1.py` starts with:

```
# Menu 3: LBA Example with Chandra data
```

ATSal adds **LBA Example with Chandra Data (ChandraLBA1.py) %3** to the Developer menu. Omit the numerical value to add an item without a keyboard shortcut. In the event of a conflict, the first file encountered takes precedence for shortcut key assignment. If the special first-line comment is missing, the program is not added to the menu, making it easy to remove programs that are no longer needed, without actually deleting them.

- The build procedure now syncs the python `Libraries` and `Packages` directories from `~/ATFiles` into the `ATFilesMaster` (using `rsync`) in order to ensure that changes made to these files (new code, tests, tutorials, etc.) are distributed with ATSal releases.
- The first time a new release of ATSal runs on a user's machine, if there is no `~/ATFiles` in place, ATSal creates and populates it. This behavior has been in place for quite some time. Now, if there is a `~/ATFiles`, ATSal automatically updates the python `Libraries` and `Packages` directories, so that tests, tutorials, and other library code are updated. ATSal does not overwrite newer files that have been modified by the user. Once a given release performs an update, it is not repeated.
- The ATSal supervisor now supports `Run Test Suite...`, in the Developer menu. (This is available in both the ATSal supervisor and in any notebook.) It opens a file of type `.tests`, which is a simple list of python programs to run for testing. A test suite viewer appears, and its refresh command runs the entire test suite, sequentially. By convention, test output is written to `test.out` if desired, or, if a separate file isn't necessary, output can be written directly to the python console, which is saved as `python.log`. There is no special format to these files.

Suite1.tests	
unittests.py	Failure
notebook1.py	Success
notebook2.py	Warning

```

Test 9: A new set of operands: PASS
'bexrav+bbbody' -> 'bbbody+bbbody|'
Old, bexrav=15236301, bbbody=15242314
New, bbbody=6226044, bbbody=15242314, bbbody=2099284

Test 10: Delete middle of three identical models: PASS
'bbbody+bbbody+bbbody' -> 'bbbody|+bbbody'
Old, bbbody=6226044, bbbody=15242314, bbbody=2099284
New, bbbody=6226044, bbbody=2099284

Test 11: Delete first of two identical models: PASS
'bbbody+bbbody' -> '|bbbody'
Old, bbbody=6226044, bbbody=2099284
New, bbbody=2099284

Test 12: Add addition and multiplier term: PASS
'bbbody' -> 'bbbody+bexrav+pileup|'
Old, bbbody=2099284
New, bbbody=2099284, bexrav=15680397, pileup=921462

Test 13: Change multiplier to add: PASS
'bbbody+bexrav+pileup' -> 'bbbody+bexrav+|pileup'
Old, bbbody=2099284, bexrav=15680397, pileup=921462
New, bbbody=2099284, bexrav=15680397, pileup=921462

Test 14: Delete whole expression: PASS
'bbbody+bexrav+pileup' -> '|'
Old, bbbody=2099284, bexrav=15680397, pileup=921462
New

Test 15: Partly bogus expression: PASS
'' -> 'bexrav+bogus+bbbody|'
Old
New, bexrav=15718303, bogus=10994889, bbbody=6817690

Test 16: Replace bogus term with valid one: PASS
'bexrav+bogus+bbbody' -> 'bexrav+bbbody|+bbbody'
Old, bexrav=15718303, bogus=10994889, bbbody=6817690
New, bexrav=15718303, bbbody=10178573, bbbody=6817690

ALL TESTS PASSED.

-----
Parser Tests
-----

Test FAILED

-----
Command Tests
-----

```

- `~/ATFiles/Python Libraries` has been renamed `~/ATFiles/Libraries` in order to avoid awkwardness resulting from the space in the pathname. Ditto for `Python Packages`. ATSal looks in the new locations even if the preferences are set to the old ones.
- Previously ATSal's unit tests were run in an ad hoc fashion, producing a variety of output files and lacking consistency. Many of them have now been rewritten to run under the automated test facility. The entire set of unit tests are currently invoked via `app.doTests()`. This function is invoked from the test suite's first test, `unittests.py`. A sample of the output is shown in the window above.
- The refresh command for python (`notebook.refresh()`) now blocks until the refresh cycle completes, unlike its interactive counterpart. This is necessary to keep from moving immediately to the next step in the test.
- Fixed a couple of bugs in the final build and packaging scripts.

## 15.8.2 Table Tool

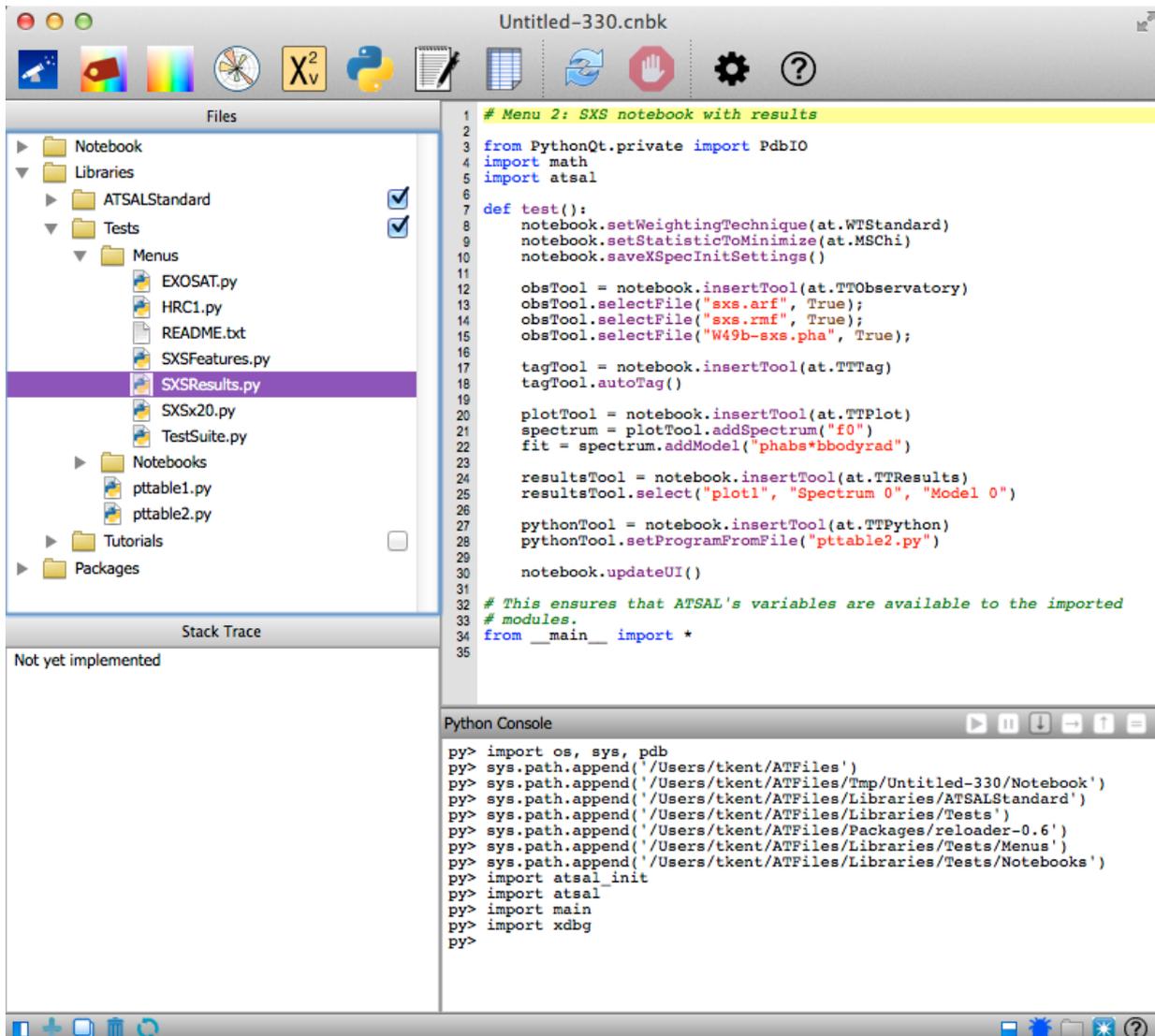
As part of the thought process in deciding how best to surface ATSal internals to python, I did a significant refactoring of the `Table` class and its handling of blocks or ranges. Tables such as the one created by the results tool are fairly complex, composed of multiple sub-blocks of variable size and position. It is not convenient to write python code to access these tables, since blocks are variably sized. In a previous ATSal iteration, I introduced the concept of a block, a temporary region set up by the python programmer to access the contents of a subset of a table. But this is not general enough. Users need not only to access blocks, but blocks *within* blocks: for example, sub-tables, and rows and columns within.

Now the `TableRange` class expresses a rectangular subset of a `Table`. `TableRange` is the base class of `Table`, and a `TableRange` can refer to a cell, a row, a column, or a block. Almost all operations are performed on the entire range by default, though a programmer can also iterate through the cells in a range instead. Instead of creating two nested for loops to set a block to boldface, `range.setBold(True)` suffices. This change substantially reduces the programming steps needed to access and manipulate tables. And it feels reasonably "pythonic" as well.

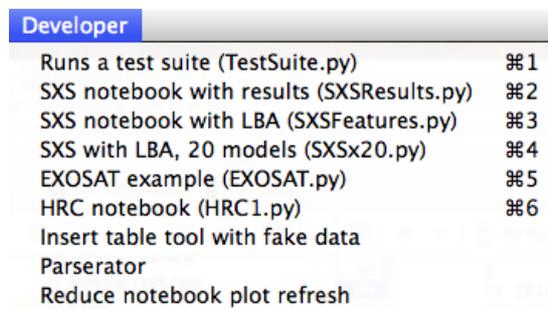
See the [TableRange](#) class discussed in this section, or see an example, below.

### 15.8.3 Debug Support Example

Python programs in ~/ATFiles/Libraries/Tests/Menus can be automatically added to the Developer menu, as with this example. This example also illustrates the utility of these extensions as a higher level test mechanism. The test below is configured and run with only three keystrokes: ⌘N (Notebook), ⌘2 (Configure test 2), ⌘R (Refresh).

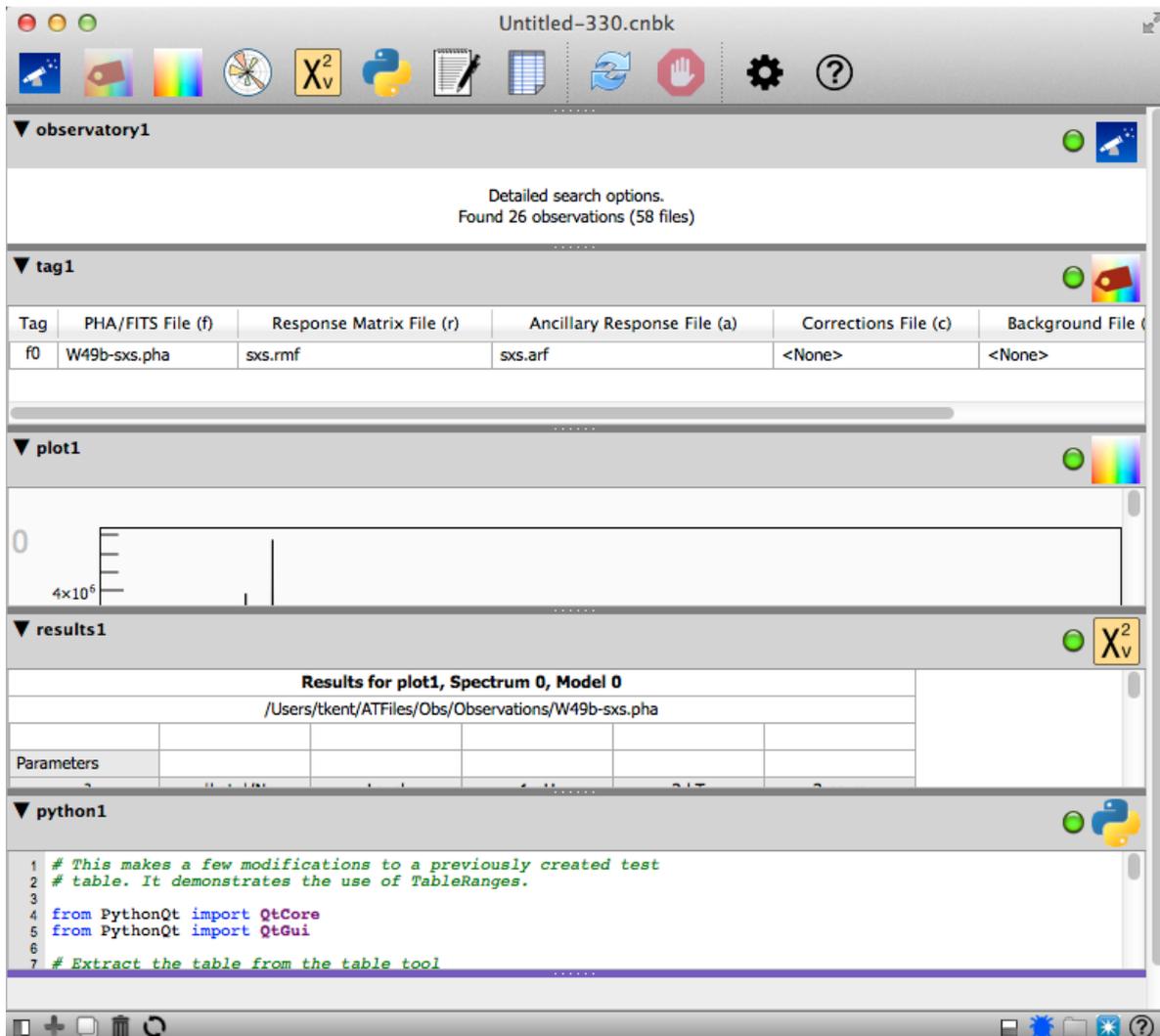


If the first line is a comment conforming to a certain format, ATSAAL uses adds the file to the menu at startup. Here is a menu showing several such programs.



Choosing the option executes the program. In this example, the program sets a couple of xSpec.init settings, then inserts four tools. It relies on a few predefined variables, including notebook and each tool.

After refreshing, the result looks like this:



Notice that the last tool added was a python tool. The program below was loaded into the python tool.

```

1 # This makes a few modifications to a previously created test
2 # table. It demonstrates the use of TableRanges.
3
4 from PythonQt import QtCore
5 from PythonQt import QtGui
6
7 # Extract the table from the table tool
8 t = results1.table()
9 # Create a range starting at 13, 0, terminated by the first blank
10 # row and 5 columns wide "χ²", "χ²"
11 r = t.createRange("χ²", "χ²", -1, -1)
12 # r = t.createRange(4, 0, -1, -1)
13
14 # Highlight the range in yellow
15 yellow = QtGui.QColor('lightyellow')
16 r.setBackground(yellow)
17
18 # Create a new range from the first row of this one
19 row = r.createRange(0, 0, 1, -1)
20 row.setBold(True)
21
22 # Extract the first column from the range above, and show it in
23 # bold
24 col = r.createRange(1, "2:kT", -1, 1)
25 azure = QtGui.QColor('azure')
26 col.setBackground(azure)
27 print("(Useless) average kT = ", col.average(), sep='')
28
29 # Update the spreadsheet view
30 results1.updateUI()
31

```

This tool demonstrates the operation of `TableRanges`. Since the results tool provides a table, we can use its table for this example. Part of it is shown below.

	0	1	2	3	4	5
0	<b>Results for plot1, Spectrum 0, Model 0</b>					
1	/Users/tkent/ATFiles/Obs/Observations/W49b-sxs.pha					
2						
3	Parameters					
4	$\chi^2$	beta /N	Level	1:nH	2:kT	3:norm
5	2.8733×10 <sup>5</sup>	8940.7	-1	0.42634	2.3983	0.10973
6	2.6386×10 <sup>5</sup>	1.3822×10 <sup>5</sup>	-1	0.73358	1.9982	0.20347
7	2.3508×10 <sup>5</sup>	90186	-1	0.85023	1.7936	0.34959
8	2.2064×10 <sup>5</sup>	42678	-1	0.98386	1.6742	0.50007
9	2.1419×10 <sup>5</sup>	19289	-1	1.0666	1.5907	0.63336
10	2.1215×10 <sup>5</sup>	9207.4	-2	1.3364	1.392	0.95578
11	2.0669×10 <sup>5</sup>	28029	-3	1.4374	1.3356	1.2548
12	2.0637×10 <sup>5</sup>	7961.5	-4	1.4238	1.3344	1.2994
13	2.0637×10 <sup>5</sup>	25.456	-5	1.4296	1.3334	1.3041
14	2.0637×10 <sup>5</sup>	13.868	-6	1.4282	1.3334	1.3036
15	2.0637×10 <sup>5</sup>	2.4182	-7	1.4286	1.3334	1.3039
16						
17	Variances and principal axes					
18		1	2	3		
19	4.4383×10 <sup>-7</sup>	-0.0351	0.9652	0.259		
20	3.0443×10 <sup>-5</sup>	0.7936	0.1845	-0.5798		
21	0.0001602	0.6074	-0.1852	0.7725		

The program begins like this:

```
t = results1.table()
r = t.createRange("χ²", "χ²", -1, -1)
```

obtains the results tool's current table, and assigns it to `t`. The next line creates a `TableRange` representing a subset of the table. The subset's origin (top left) is at the label containing "χ²". Only labels (shown in light gray in the table) may be used as coordinates in this way. The next two arguments indicate the number of rows and columns. `-1` extends to the first blank row and column. Thus range `r` isolates a useful portion of the table. The next couple of lines highlight the range in yellow. This relies on PythonQt's access to the entire Qt class library, here with a trivial example of asking the Qt GUI library for a color.

```
row = r.createRange(0, 0, 1, -1)
row.setBold(True)
```

Next, we create a new range, `row`, as a sub-range of `r`. It says the range should begin at the top left of `r`, and be a single row, and it should include all the columns until there is a blank column. This ability to express ranges as subsets of other ranges makes it much easier to work on large tables. In many cases no iteration is needed. Here, we set the row to bold.

```
col = r.createRange(1, "2:kT", -1, 1)
azure = QtGui.QColor('azure')
col.setBackgroundColor(azure)
print("(Useless) average kT = ", col.average(), sep='')
```

In this sequence, we create a second subrange from the original range, `r`. The range is a single column, the "2:kT" column. The starting row number is 1, to skip over the header. First we highlight the column to show what is selected. Next, we compute the average of the column, and print it to the python console. The average of this column is meaningless, but the example demonstrates that spreadsheet-style functions are easily implemented using ranges. The result:

```
py>
(Useless) average kT = 1.5924554545454541
py>
```



# 16 Bugs and Features

## 16.1 High (Pre-demo) Priority

- A weighting/statistic conflict is causing bad results for LBA. Need resolution from Randall. He directs me to [this write-up by Keith](#), and says to use Churazov weighting. **This didn't work either, and remains an open issue.**
- Search in Python window, if return is used to dismiss dialog, it also replaces selection with a return. I spent awhile trying to make sense of this, without success.
- Version number in help is not being brought up to date during builds.
- The problem importing dynamic libraries into python is only half-fixed. "import math" works at the toplevel interpreter, but within a program, it does not. Maybe related to the python reloader.
- Search for other toLatin1() cases.
- Add statistic popup to each model.
- Surface ignore on/off.
- Communications between the supervisor and the notebooks is sometimes prompt, but sometimes plagued by mystery delays. Might be a flush problem?
- Add notebook help.
- The zone 1 conundrum.
- Finish models list; disable models that have "dependencies."
- Bring user help to an acceptable level.
- Create a tutorial analogous to Keith's.
- Create a second tutorial that includes LBA.
- Create a tutorial that focuses on python.
- Web site. **(Skeletal web site is automatically generated now, just needs fleshing out.)**

Demo caveats:

- Python integration severely limited—need to wait for design to settle down before committing ourselves to a Python interface.
- Saved notebooks will not be usable with future releases, format hasn't stabilized yet.
- All results will be suspect, since there has been so little testing; for example, unit conversion glitches may be common.

## 16.2 Medium Priority (Nice for demo, probably won't make it)

- Notebook tool cut and paste.
- When quitting or doing another operation that involves communication between notebook and supervisor, there is often an inordinately long delay.
- Get zoom working in label area.

## 16.3 Low Priority (Post-demo)

- Support more of XSPEC.
- Move file buttons to files pane title area.
- Implement Export Notebook Archive.
- Implement Export Notebook PDF.
- Implement Export Notebook HTML.
- Probably need to make zone visibility directly controllable in graph.
- Top axis sometimes appears at top of label area, not graph. Intermittent.
- Change ignore regions to include regions.
- Make label area size adjustable.
- Add support for syncing multiple spectra.
- Source code install procedure for Mac. In addition to Git checkout, need to install necessary source code components and libraries. Preferably a script to do the whole process. Note that using the machine for other purposes could be disrupted.
- Build procedure for Linux
- Source code install procedure for Linux
- Packaging procedure for Linux
- Serious effort to provide Python integration
- Table tool design enhancement
- Python debugger

Error preparing HTML-CSS output (preProcess)