ginga Documentation

Release 4.1.0

Ginga Maintainers

Jul 01, 2023

CONTENTS

Part I

About Ginga

Ginga is a toolkit designed for building viewers for scientific image data in Python, visualizing 2D pixel data in NumPy arrays. It can view astronomical data such as contained in files based on the FITS (Flexible Image Transport System) file format. It is written and is maintained by software engineers at the National Astronomical Observatory of Japan, the Space Telescope Science Institute, and other contributing entities.

The Ginga toolkit centers around an image display class which supports zooming and panning, color and intensity mapping, a choice of several automatic cut levels algorithms and canvases for plotting scalable geometric forms. In addition to this widget, a general purpose "reference" FITS viewer is provided, based on a plugin framework.

A fairly complete set of "standard" plugins are provided for features that we expect from a modern FITS viewer: panning and zooming windows, star catalog access, cuts, star pick/FWHM, thumbnails, etc.

Part II

Copyright and License

Copyright (c) 2011-2023 Ginga Maintainers. All rights reserved.

Ginga is distributed under an open-source BSD licence. Please see the file `LICENSE.txt` in the top-level directory for details.

Part III

Requirements and Supported Platforms

Because Ginga is written in pure Python, it can run on any platform that has the required Python modules and has a supported widget set. The basic Ginga display class supports the Qt (4 and 5), PySide, Gtk (3), Tk widget sets natively as well as any Matplotlib Figure, and HTML5 canvases in a web browser. The full reference viewer supports Qt and Gtk variants. Ginga can also be used in Jupyter notebooks.

Part IV

Getting the Source

Clone from Github:

```
git clone https://github.com/ejeschke/ginga.git
```

To get a ZIP file or tarball instead, see the links on About Ginga.

Part V

Building and Installation

Download and install from `pip`:

```
pip install ginga
```

Or `conda`:

```
conda install ginga -c conda-forge
```

The reference viewer can then be run using the command `ginga`.

For detailed instructions, see:

ONE

INSTALLATION

1.1 Dependencies

Ginga is written entirely in Python, and only uses supporting Python packages. There is nothing to compile (unless you need to compile one of the supporting packages).

In recent Linux, Mac, and Windows versions, all of the packages are available in binary (installable) form. It should not be necessary to compile anything, but as always, your mileage may vary.

1.1.1 REQUIRED

- python (v. 3.7 or higher)
- setuptools-scm
- numpy (v. 1.14 or higher)
- astropy
- pillow

Strongly recommended, because some features will not be available without it:

- scipy
- opencv-python (also distributed as opencv or python-opencv, depending on where you get it from)
- exifread
- beautifulsoup4
- docutils (to display help for plugins)

For opening FITS files you will need one of the following packages:

- astropy
- fitsio

For WCS resolution you will need one of the following packages:

- astropy
- kapteyn
- astLib
- starlink

1.1.2 BACKENDS (one or more)

Ginga can draw its output to a number of different back ends. Depending on which GUI toolkit you prefer (and what you want to do), you will need at least one of the following:

- QtPy (PyQt5 or PyQt6)
- PySide (pyside2 or pyside6)
- pygobject (gi) **AND** pycairo (GTK 3)
- tkinter
- matplotlib
- tornado
- aggdraw
- Pillow (PIL fork)

1.1.3 RECOMMENDED

Certain plugins in the reference viewer (or features of those plugins) will not work without the following packages:

- matplotlib (required by: Pick, Cuts, Histogram, LineProfile)
- scipy (required by: Pick, some built-in *auto cuts algorithms* used when you load an image)
- astroquery (required by Catalogs)

To save a movie:

- mencoder (command line tool required by: Cuts)

Helpful, but not necessary (may optimize or speed up certain operations):

- opencv-python (speeds up rotation, mosaicing and some transformations)
- pyopengl + pycairo (for using OpenGL features; very useful for 4K or larger monitors)
- filemagic (aids in identifying files when opening them)

1.2 Notes on Supported Widget Sets

In the discussion below, we differentiate between the Ginga viewing widget, such as used in the `examples/\*/` `example\*.py` programs and the full reference viewer, which includes many plugins (`ginga`).

Note: For the full reference viewer, Mac and Windows users should probably install the Qt version, unless you are the tinkering sort. Linux can use either Qt or GTK fine.

1.2.1 Qt/PySide

Ginga can use either PyQt or PySide, for Qt version 5 or 6. It will auto-detect which one is installed, using the `qtpy` compatibility package. There is support for both the basic widget and the full reference viewer.

Note: If you have both installed and you want to use a specific one then set the environment variable QT_API to either "pyqt" or "pyside". This is the same procedure as for Matplotlib.

1.2.2 GTK

Ginga can use GTK 3 (with `gi`). (If you have an older version of `pycairo` package, you may need to install a newer version from `pycairo`).

1.2.3 Tk

Ginga's Tk support is limited to the viewing widget itself. For overplotting (graphics) support, you will also need one of:

- Pillow
- opencv-python
- aggdraw

1.2.4 Matplotlib

Ginga can render directly into a Matplotlib figure. Support is limited to the viewing widget itself. Any of the backends that Matplotlib supports is usable. Performance is not as good as to one of the "native" backends listed above, but oh, the overplot options!

1.2.5 HTML5 web browser

Ginga can render into an HTML5 canvas via a web server. Support is limited to the viewing widget itself. See the notes in `examples/pg/example2_pg.py`. Tested browsers include Chromium (Chrome), Firefox, and Safari.

1.3 Basic Installation

You can download and install via `pip` by choosing the command that best suits your needs (full selection is defined in setup configuration file):

```
pip install ginga   # The most basic installation

pip install ginga[recommended,qt5]   # Qt5

pip install ginga[recommended,gtk3]   # GTK 3
```

Or via `conda`:

```
conda install ginga -c conda-forge
```

The reference viewer can then be run using the command `ginga`.

1.4 Installation from Source

1. Clone from Github:

   ```
   git clone https://github.com/ejeschke/ginga.git
   ```

   Or see links on this page to get a ZIP file or tarball.

2. Unpack, go into the top level directory, and run:

   ```
   pip install -e .
   ```

1.5 Platform Specific Instructions

1.5.1 Linux (Debian/Ubuntu)

If you are on a relatively recent version of Debian or Ubuntu, something like the following will work:

```
apt install python3-ginga
```

If you are using another distribution of Linux, we recommend to install via Anaconda or Miniconda as described below.

1.5.2 Mac/Windows/Linux (others)

Anaconda

For Mac/Windows or other Linux users, we recommend installing the Anaconda distribution (or Miniconda). This distribution already includes all of the necessary packages to run Ginga.

After installing Anaconda, open the Anaconda Prompt and follow instructions under *Basic Installation* via `conda`.

1.6 Running tests

1. Install the following packages:

   ```
   $ pip install -e .[test]
   ```

2. Run the tests using `pytest`:

   ```
   $ pytest
   ```

1.7 Building documentation

1. Install the following packages:

```
$ pip install -e .[docs]
```

2. Build the documentation using `make`:

```
$ cd doc
$ make html
```

Part VI

Documentation

WHAT'S NEW

2.1 Ver 4.1.0 (2022-06-30)

- change implementation of splash banner to a pop-up modal dialog with version string

- fixed menubar integration on Mac OS X

- fixed an issue with auto cuts saved parameters not being loaded correctly

- fixed an issue with certain auto cuts methods not getting enough samples when the image size is very small

- removed old, deprecated StandardPixelRenderer

- RGB mapping has been refactored to use a pipeline

- Removed deprecated preload feature

- Fix for focusing plugins that have no GUI

- Added feature to reset viewer attributes between images (can be used synergistically with the "remember" feature). See "Reset (Viewer)" and "Remember (Image)" settings in the "Preferences" plugin.

- Fixed bug with pg backend loading icons

- Pick plugin settings now has options to set the autozoom and autocuts settings for the "Image" and "Contour" viewers (see "plugin_Pick.cfg" file in . . ./ginga/examples/configs)

- Fixed an issue with readout of values under the cursor when an image is plotted at a non-zero origin

2.2 Ver 4.0.1 (2022-12-27)

- fixed a DeprecationWarning with jupyterw back end

- fixed a bug for Toolbar plugin that prevented N-E and N-W orientation from working as well as contrast restore

- fixes a bug in exporting ginga canvas objects to astropy regions objects and the test suite for it

2.3 Ver 4.0.0 (2022-12-20)

- fixed getattr functionality in Bunch

- removed the "mock" backend; use "pil" backend for similar purposes

- fixed a longstanding issue where events registered on a canvas could be masked by default key/cursor bindings (not associated with a mode). This meant, for example, that only certain keystrokes could be captured by an event handler registered on a ginga canvas, because any keystrokes that had a default binding would take precedence. Now such bindings are only executed if the event is not handled by any active canvas bindings.

- Removed the "Quick Mode" and "From Peak" options in the Pick plugin to simplify operation.

- Many deprecated camelcase (non-PEP8) methods were removed. Use the "snake-case" names instead.

- Fixed an issue with setting the scale manually in the Preferences plugin

- Fixed a bug that can cause an incorrect cropping of image when the window is resized

- Refactor modes from Bindings module into separate modules:

    - modes can now be written and understood as having a very similar structure to a plugin.

    - mode docstrings can be written and maintained better to document the modes and the code implementing a mode is much easier to understand since it is encapsulated rather than all mixed together in one huge file.

- Added an AutoLoad plugin that can monitor a folder for new files and load them.

- Fixed a bug in the Overlays plugin where the overlay value was reported as the value under the cursor rather than the data value

- Fixed an issue with loading RGB images with opencv-python

- Fixed an issue where manual cut levels changes weren't reflected in the Thumbs icon

- ScreenShot now correctly captures the background color of the viewer

- Fixed a DeprecationWarning related to use of entry points

- File loaders are now discoverable under the "ginga_loaders" entry point. Loaders can be registered for MIME types and a `mime.types` file can be added to your `$HOME/.ginga` to identify types by file extension.

- ColorMapPicker plugin can now be launched as a local or global plugin

2.4 Ver 3.4.0 (2022-06-28)

- Added start_server option to RC plugin configuration; can configure whether ginga should start the remote control server when the plugin starts or not

- fixed an error with auto-orientation of RGB images loaded with OpenCv

- fixed a bug where get_channel_on_demand() would throw an error if the channel already exists

- fixed a bug in ingesting the metadata (header) of RGB images

- added support for backends pyqt6 and pyside6; removed support for pyqt4 and pyside

- fix for a bug in ICC profiling with temp file creation

- new option in Collage plugin to select more accurate (but slower) mosaicing using the 'warp' method

- fixed an issue with numpy floats and drawing lines and ploygons with the Qt backend

- add option for suppressing FITS verify warnings when opening files using astropy.io.fits

- worked around a bug in recent versions of aggdraw (for "agg" backend) that caused problems for drawing ellipses
- added ability to read and write astropy-regions shapes in the Drawing plugin

2.5 Ver 3.3.0 (2022-02-16)

- Fixed an issue with image rotation when OpenCv is installed
- Removed support for OpenCL
- Fixed Crosshair plugin to update the plot when image changes in channel
- Fixed an issue where a thumbnail could be generated even if the channel was configured not to generate thumbs
- Fixed an issue where wcs_world2pix() was called instead of all_world2pix() if wcs_astropy was used. This may have affected graphic overlays plotted in ra/dec instead of pixels.
- Closing the reference viewer now stops all plugins first
- Fix to RC plugin for better error handling if another process is using the port
- Fixed a bug where using the fitsio loader the primary header was not set correctly in some instances
- Additions to the "pg" backend to add functionality already in the Qt and Gtk backends
- Fixed a bug with writing FITS files when using fitsio wrapper
- Fixed a bug where creating a new workspace did not set the correct workspace type that was selected in the drop down menu
- Updated pg widgets web backend due to changes to Tornado asyncio handling
- Changes to 'histogram' and 'stddev' autocuts algorithms:
  - choice of sampling by grid now; useful for mosaics and collages
  - for previous parameter of usecrop=True, use sample=crop
- Moved loading of FITS HDUs from AstroImage to io_fits module, encapsulating the details of this file format into the module responsible for loading those files:
  - added loading of FITS tables via the fitsio package in io_fits
  - TableView can now view tables loaded with either astropy or fitsio
  - inherit_primary_header in general.cfg now defaults to True and actually controls the behavior of always saving the primary header if set to False
- Fixed a rounding bug in displaying sexagesimal formatted coordinates:
  - deprecated ginga.util.wcs.{raDegToString,decDegToString}
  - use ginga.util.wcs.{ra_deg_to_str,dec_deg_to_str} instead
- Fixed an issue where the Catalogs plugin would not start correctly if astroquery was not installed
- The keywords `save_primary_header` and `inherit_primary_header` in the `AstroImage` constructor are deprecated. Use these same keywords in `AstroImage.load_hdu()` or `AstroImage.load_file()` methods instead. Several other methods in `AstroImage` are deprecated as well; they were previously pending deprecation.
- Fixed an issue where image might not be redrawn properly if scale or pan is set directly via a viewer's settings object (not the usual case)

2.6 Ver 3.2.0 (2021-06-07)

- Minimum supported Python version is now 3.7

- Fixed some numpy deprecation warnings with numpy 1.19.0

- Canvas shapes can now be copied

- Added an option to make a copy of existing shape in Drawing plugin

- Added an option to make a copy of existing cut in Cuts plugin

- Added new iqcalc_astropy module to handle FWHM fitting and source finding using astropy and photutils

- Added new calc_fwhm_lib configuration item to let Pick switch between iqcalc and iqcalc_astropy

- Fixed a bug where certain plots were not cleared in Pick plugin

- Removed support for matplotlib versions < 2.1

- Added bicubic and bilinear interpolation methods to OpenGL backend

- Fixed a bug where the FWHM labels in the plot didn't match report values in the Pick plugin

- Fixed a bug in gtk/cairo backend where paths were not drawn correctly

- Included a couple of additional bundled fonts to improve legibility of small text

- Fixed a bug in PixTable that reversed pixel indices on display

- Added box sum and median results to PixTable; also improved statistics display

- Fixed an issue for the Tk Ginga widget if PIL.ImageTk was not installed

- Changed splitter widget so that the "thumbs" have a visual indicator

- Fixed an issue with cursor warp in free panning with Gtk3 backend

- Fixed an issue where the cursor was not changed from the default

- Fixed Pick plugin to autozoom the pick and contour images

- Fixed an issue where Thumbs plugin might not show initial thumb(s) when main window is enlarged to certain sizes

- Added "orientation" setting to orientable plugins

- Enhancements to Histogram plugin: ability to click in plot to set low and high cuts levels, scroll plot to expand or contract cuts width

- Crosshair plugin enhanced to have fast X/Y cuts plot feature; cuts plot removed from Pick plugin

- Fixed an issue where the Pick plugin would not start due to a change to matplotlib canvas initialization

- Updates to Catalogs plugin:

  - new astronomical object lookup section for SIMBAD or NED

  - new ability to specify some astroquery catalog and image sources in the `plugins_Catalogs.cfg` configuration file

  - *API not compatible with previous releases, including configuration via `ginga_config.py`*

2.7 Ver 3.1.0 (2020-07-20)

- Zoom and Pan plugins refactored. Now shows graphical overlays.

- Improved performance of rendering when flipping, swapping axes or rotating viewer.

- Fixed a bug where the display was not redrawn if an ICC profile was changed

- Fixed bugs relating to drawing XRange, YRange and Rectangle objects on rotated canvas

- Fixed a bug with fit image to window (zoom_fit) which was off by half a pixel

- Fixed an issue where an error message appears in the log if the scale is so small the image is invisible

- Fixed an issue where the readout under the cursor for value is reported for an empty row to the left and column below of pixels

- Removed dependence on astropy-helpers submodule.

- Fixed an issue where limits were not reset correctly if image being viewed is modified in place (and data array changes size)

- Fixed an issue with Mosaic plugin where images with a PC matrix were not always oriented correctly

- New Collage plugin offers an efficient alternative way to view mosaics

- Fix for a bug where using Zoom and PixTable at the same time can cause wrong results to be displayed in PixTable

- New ability to specify alternative Ginga home directories, with custom layouts and plugin configurations (–basedir option)

- Fix for a bug that caused a crash when closing the Help window with Qt/PySide backend

2.8 Ver 3.0.0 (2019-09-20)

- Dropped Python 2 support. Ginga now requires Python 3.5 or later.

- Fixed an issue with some RGB images being viewed flipped

- Improved accuracy of Qt-based timers

- Pick plugin enhanced with option to center on found object; also default shape changed to a box rather than a rectangle

- Added support for ASDF and GWCS.

- Fixed drag-and-drop functionality in FBrowser plugin on Windows.

- Enabled HDU sorting via config file in MultiDim.

- Fixed a bug where display would get corrupted when adjusting interactive cuts or contrast on rotated image

- Improved smoothness and updates of Zoom plugin image

- Improved smoothness and updates when rotating or shifting color map

- Fixed broken banner

- Improved `pip` installation commands for different backends.

- Fixed a bug where identically named HDUs could not be loaded by MultiDim

- Fixed a bug where compressed HDUs could not be loaded by MultiDim

- Plugins with splitter type panels now remember their sizes when closed

- LineProfile plugin's default Y-axis label is now "Signal", to be more scientifically accurate.

- Simplified plugins Colorbar, Contents, Cursor, Errors, Header, Info, Log, Pan, and Thumbs plugins. Made all of these restartable. Subclasses of these plugins may require refactoring in a couple of cases.

- Selecting item in FBrowser now populates its text box properly.

- Support opening all extensions of given extension name from a FITS file (e.g., `filename.fits[SCI,*]`) from Ginga command line or FBrowser.

- New Downloads plugin for monitoring/managing URI downloads

- Supports PySide2 (alternative Qt5 backend)

- Added statistics line to Histogram plugin

- Removed support for gtk2, since it is not supported for Python 3

- new styles added for Point canvas type: circle, square, diamond, hexagon, uptriangle, downtriangle

- New file opener framework

- Text objects can be resized and rotated in edit mode on the canvas

- Added ellipse and box annulus types as Annulus2R canvas object

- Supports plotting DS9 regions via 2-way conversion between Ginga canvas types and Astropy regions

2.9 Ver 2.7.2 (2018-11-05)

- Fix for linewidth attribute in shapes for AGG backend

- Fix for ellipse rotation in OpenCv backend

- Better text rendering for OpenCv backend (loadable fonts)

- enhancements to the Ruler plugin for reference viewer

- supports quick loading from astropy NDData (or subclassed) objects

- Support for scaling fonts on high-dpi displays

- Fixed a bug where adjusting autocuts parameters in Preferences would crash the Qt backend

- Fixed a bug that caused windows to disappear when changing workspace to MDI mode under Gtk3 backend

- Fixed a bug where local plugins were not properly closed when a channel is deleted

- Fixed a bug in which the ColorMapPlugin canvas was not scaled to the correct size

- Improvements to synchronous refresh feature to reduce jitter and increase frame rate

- Fix for navigating certain data cubes with MutltiDim plugin

- Added new percentage transform and coordinate mapper type (allow placement of objects as a percentage of the window size)

- Updates to Compass canvas type and Pan plugin

- Documentation improvements for writing plugins

2.10 Ver 2.7.1 (2018-07-09)

- Fix for image rendering bug which shows last row and column of image being drawn twice
- Added option to "Compass" draw type to be in pixels (X/Y) or wcs (N/E)
- Changed Pan plugin to attempt to draw both kinds of compasses
- Log plugin enhanced to show lines logged before it was opened
- Info plugin adds convenience controls for "Follow New" and "Raise New"
- WCSMatch plugin enhanced to offer fine grained control over sync
- fixed an issue in Debian build that caused long start up times
- User can dynamically add scrollbars to channel viewers in Preferences
- Made Gtk backend default to 'gtk3' - "-t gtk" now invokes gtk3 instead of gtk2 - choose "-t gtk2" if you want the gtk2 back end
- Fixed a bug with opening wildcard-type filespec from the command line
- Fixed an issue in Thumbs plugin with opening FITS tables from the command line
- Fixes for some keyboard focus (Gtk) and unintentional channel changes (Qt) when viewer is in MDI mode
- IRAF plugin moved to experimental folder
- Allow setting of initial channel list, local, global and disabled plugins from general configuration file
- Fix for a bug when using OpenCv acceleration on dtype('>f8') arrays
- Fixed a bug where colormap scale markers were sometimes not spaced wide enough
- Workaround for failed PDF build in RTD documentation

2.11 Ver 2.7.0 (2018-02-02)

- Fix for gtk 4.0 (use "gtk3" backend, it works for 4.0)
- Fix for broken polygon containment test
- Addition of configurable zoom handlers for pan gestures
- Fix for some broken tests under python 2.7
- Update to mode handling via keyboard shortcuts
    - addition of a new "meta" mode used primarily for mode switching
    - most modes now initiated from meta mode, which frees up keys for other uses
    - see Ginga quick reference for details on how the new bindings work
- Efficiency update for Thumbs plugin when many thumbs are present
- Default for the save_layout option is now True, so the reference viewer will write out its layout state on exit and restore it on startup. See documentation in the "customization" section of the manual.
- Plugins can now be organized by category and these categories are used to construct a hierarchical Operations menu
- Zoom and Header plugins are now not started by default
- Fix for "sortable" checkbox behavior on Header plugin

- Default keyboard mode type is now 'locked' (prev 'oneshot')

- Fixes for missing CSS file in installation script

- Less confusing behavior for workspace and toolbar arrow buttons

2.12 Ver 2.6.6 (2017-11-02)

- Fix for broken sorting in Contents plugin in gtk backends

- Fix for resize bug in switching in and out of grid view in gtk backends

- Updated to have efficient support for gtk3

    – please install compatible pycairo from github.com/pygobject/pycairo if you get a "Not implemented yet"
      exception bubbling up from a method called cairo.ImageSurface.create_for_data()

- Addition of a "Quick Mode" to the Pick plugin–see documentation

- More consistent font handing between widgets and Ginga canvases

- Bug fix for importing some types of matplotlib color maps

- Add antialiasing for Qt back end

- Bug fixes and enhancements for Qt gestures - holding shift with pinch now keeps position under cursor

- New Jupyter notebooks back end based on ipywidgets - requirements: $ pip install ipyevents - see
  examples/jupyter-notebook/

- Fixes to various reference viewer plugins

2.13 Ver 2.6.5 (2017-07-31)

- Coordinate transforms refactored for speed and code clarity

- Some canvas shapes refactored for better code reuse

- Allow max and min scale limits to be disabled (by None)

- Fixed a bug that prevented the reference viewer from resizing correctly with Qt back end

- Refactored WCS wrapper module for code clarity

- Set minimum astropy version requirement to 1.X

- Fixed a bug in NAXIS selection GUI (MultiDim plugin)

- Fixed MDI window resizing with Gtk back ends

- Fixed an error where zoom 100% button did not correctly zoom to 1:1 scale

- Several fixes for astropy 2.0 compatibility

- Fixed a bug in the FBrowser plugin when channel displaying a table and attempting to load a new file

- Fixed a bug when setting the pan position manually by wcs coordinates

- Updates for changes in PIL.ImageCms module

- Fix for window corruption on certain expose events

- New default bindings for touch pads and differentiation from wheel zoom

2.14 Ver 2.6.4 (2017-06-07)

- Added new ScreenShot plugin to take PNG/JPEG snaps of the viewer window
- Enhancements to the Pick plugin
  - Added ability to make shapes besides rectangles for enclosing pick area. Masks out unwanted pixels. Choose the shape in the Settings tab.
  - Changed behavior of pick log to only write the log when the user clicks the save button.
  - Changed the name of the save button to "Save as FITS table" to make it clear what is being written.
  - If "Show candidates" is selected in Settings, then ALL of the candidates are saved to the log.
  - Added documentation to the manual
  - Bug fix for error when changing radius
- Improvements to layout of Operations menu (plugin categories)
- Colorbar scale now placed below the color wedge and is more legible
- Bug fixes for LineProfile plugin
- Slit function for Cuts plugin can be enabled from GUI
- Bug fixes for Slit function
- Info plugin can now control new image cut/zoom/center settings
- Fixed an issue with the MultiDim plugin that could result in a hang with some back ends
- New canvas type for displaying WCS grid overlay and new WCSAxes plugin that uses it
- Bug fixes to scrolling via scrollbars and vert/horiz percentages
- Enhancements to the LineProfile plugin
  - several new shapes besides the standard point
  - plot multiple lines

2.15 Ver 2.6.3 (2017-03-30)

- Fix for issue that stops ginga startup when loading externally distributed plugins that have errors
- Fix for an issue loading plugins from the command line when they are nested in a package
- Added bindings for moving +/- pixel delta in X or Y and centering on the pixel
- Fixes for some key mappings for tk, matplotlib and HTML5 canvas backends
- Fixes for IRAF plugin under python 3
- Fix for a bug using remote control (RC) plugin from python2 client to python 3 ginga
- Documentation updates

2.16 Ver 2.6.2 (2017-02-16)

- Added some colormaps from ds9 that don't have equivalents in Ginga or matplotlib
- Fix for recognizing CompImage HDU type when using astropy.io.fits
- Add new experimental OpenGL back end
- Fixes for Tk back end on python 3
- You can now write separately distributed and installable plugins for the reference viewer that Ginga will find and load on startup
- Added –sep option to load command line files into separate channels
- New help screen feature available for plugins
- Lots of updates to documentation
- Fixed a stability issue with drag and dropping large number of files under Linux
- Fixes for python3 and several example programs
- Fix for interactive rotation bug under matplotlib back end

2.17 Ver 2.6.1 (2016-12-22)

- Added a working MDI workspace for gtk2/gtk3.
- Added scrollbar frames. See examples/qt/example1_qt.py for standalone widget. Can be added to reference viewer by putting 'scrollbars = "on"' in your channel_Image.cfg preferences.
- Reorganized reference viewer files under "rv" folder.
- Improved Pick plugin: nicer contour plot, pick log uses table widget, pick log saved as a FITS table HDU
- Pick and Zoom plugins can now use a specific color map, rather than always using the same one as the channel window
- gtk3 reference viewer can now be resized smaller than the original layout (gtk2 still cannot)
- added ability to save the reference viewer size, layout and position on screen
- gtk MDI windows now remember their size and location when toggling workspace types
- Fixes for problems with pinch and scroll gestures with Qt5 backend
- Fixed a bug where scale changes between X and Y axes unexpectedly at extreme zoom levels
- Fixed a bug where cursor could get stuck on a pan cursor
- Added ability to define a cursor for any mode
- Added documented virtual methods to ImageView base class
- Added a workaround for a bug in early versions of Qt5 where excessive mouse motion events accumulate in the event queue

2.18 Ver 2.6.0 (2016-11-16)

With release 2.6.0 we are moving to a new versioning scheme that makes use of github tagged releases and a "dev" versioning scheme for updates between releases.

This release includes many bugfixes and improvements, new canvas types (XRange and YRange), a Command plugin, WCSMatch plugin, dynamically configurable workspaces, OpenCv acceleration, an HTML5 backend and much much more.

2.19 Ver 2.2.20160505170200

Ginga has merged the astropy-helpers template. This should make it more compatible management-wise with other astropy-affiliated packages.

2.20 Ver 2.2.20150203025858

Ginga drawing canvas objects now can specify points and radii in world coordinates degrees and sexigesimal notation.

- default is still data coordinates
- can play with this from Drawing plugin in reference viewer

2.21 Ver 2.1.20141203011503

Major updates to the drawing features of ginga:

- new canvas types including ellipses, boxes, triangles, paths, images
- objects are editable: press 'b' to go into edit mode to select and manipulate objects graphically (NOTE: 'b' binding is considered experimental for now–editing interface is still evolving)
- editing: scale, rotate, move; change: fill, alpha transparency, etc.
- editing features available in all versions of the widget
- updated Drawing plugin of reference viewer to make use of all this

2.22 Ver 2.0.20140905210415

Updates to the core display and bindings classes:

- improvements to interactive rotation command–now resume rotation from current value and direction is relative to horizontal motion of mouse
- most keyboard modes are now locking and not oneshot (press to turn on, press again (or hit escape) to turn off
- additional mouse button functionality in modes (see quick reference)
- some changes to default keyboard bindings (see quick reference)
- changes to auto cuts parameters always result in a new autocut being done (instead of having to explicity perform the autocut)–users seem to expect this
- autocenter preference changed from True/False to on/override/off

Reference viewer only: new global plugin "Toolbar" provides GUI buttons for many operations that previously had only keyboard bindings

2.23 Ver 2.0.20140811184717

Codebase has been refactored to work with python3 via the "six" module. Tests can now be run with py.test as well as nosetest.

2.24 Ver 2.0.20140626204441

Support has been added for image overlays. It's now possible to overlay RGB images on top of the canvas. The images scale, transform and rotate wrt the canvas.

2.25 Ver 2.0.20140520035237

Auto cut levels algorithms have been updated. "zscale" has been reinforced by using the module from the "numdisplay" package, which does a fair sight closer to IRAF than the previous one Ginga was using. Also, the algorithm "median" (median filtering) makes a comeback. It's now fast enough to include and produces more usable results.

2.26 Ver 2.0.20140417032430

New interactive command to orient the image by WCS to North=Up. The default binding to 'o' creates left-handed orientation ('O' for right-handed). Added a command to rotate the image in 90 deg increments. Default binding to 'e' rotates by 90 deg ('E' for -90 deg).

2.27 Ver 2.0.20140412025038

Major update for scale (mapping) algorithms

The scale mapping algorithms (for mapping data values during rendering) havebeen completely refactored. They are now separated from the RGBMap class and are pluggable. Furthermore I have redone them modeled after the ds9 algorithms.

There are now eight algorithms available: linear, log, power, sqrt, squared, asinh, sinh, histeq. You can choose the mapping from the Preferences plugin or cycle through them using the binding to the 's' key (Use 'S' to reset to linear). There is also a mouse wheel mapping than can be assigned to this function if you customize your bindings. It is not enabled by default.

The Preferences plugin has been updated to make the function a little clearer, since there was some confusion also with the intensity map feature that is also part of the final color mapping process.

2.28 Ver 2.0.20140114070809

- The SAMP plugin has been updated to work with the new astropy.vo.samp module.
- The Catalogs plugin has been updated to allow the user to define the radius of the conesearch or image search by drawing a circle (as well as the previous option–a rectangle).

2.29 Ver 2.0.20131218034517

The user interface mapping just got a bit easier to use. Ginga now provides a way to do most UI remapping just by placing a simple config file in your ~/.ginga directory. An example for ds9 users is in the new "examples" folder.

Many simple examples were moved out of "scripts" and stored under subdirectories (by GUI toolkit) in "examples".

2.30 Ver 2.0.20131201230846

Ginga gets trackpad gestures! The Qt rendering class gets support for pinch and pan gestures:

- The pinch/rotate gesture works as expected on a Mac trackpad
- The pan gesture is not a two-finger pan but a "non-standard", Qt-specific one-finger pan. These are experimental for now, but are enabled by default in this release.

Also in this release there has been a lot of updates to the documentation. The developer and internals sections in particular have a lot of new material.

2.31 Ver 2.0.20131030190529

The great renaming

I really dislike it when developers do this, so it pains me to do it now, but I have performed a mass renaming of classes. FitsImage ended up being the View in the MVC way of doing things, yet it shared the same naming style as the model classes AstroImage and PythonImage. This would have been the source of endless confusion to developers down the road. Also, PythonImage needed to get renamed to something more akin to what it actually represents.

So the renaming went like this:

- FitsImage -> ImageView
- FitsImage{XYZ} -> ImageView{XYZ}
- PythonImage -> RGBImage

So we have:

- M: BaseImage, AstroImage, RGBImage
- V: ImageView{XYZ}
- C: Bindings, BindMap

I did this in the brand new 2.0 version so at least devs have a heads up that things will not be backward compatible.

And I apologize in advance for any renaming and support issues this may cause for you. Fire up your editor of choice and do a query/replace of "FitsImage" with "ImageView" and you should be good to go.

2.32 Ver 1.5-20131022230350

Ginga gets a Matplotlib backend!

Ginga can now render to any Matplotlib FigureCanvas. The performance using this backend is not as fast as the others, but it is acceptable and opens up huge opportunities for overplotting.

See scripts/example{1,2,3,4,5}_mpl.py

Also merges in bug fixes for recent changes to astropy, and support for other python WCS packages such as kapteyn and astLib.

2.33 Ver 1.5-20130923184124

2.33.1 Efficiency improvements

Efforts to improve speed of entire rendering pipeline and widget specific redrawing

- Decent improvements, Ginga can now render HD video (no sound) at 30 FPS on older hardware (see scripts/example1_video.py). This translates to a slightly speedier feel overall for many operations viewing regular scientific files.
- Fixed a bug that gave an error message of Callback.py:83 (make_callback) | Error making callback 'field-info': 'Readout' object has no attribute 'fitsimage'
- Version bump

2.34 Ver 1.4.20130718005402

2.34.1 New Agg backend

There is now an Agg rendering version of the ImageView object.

- uses the python "aggdraw" module for drawing; get it here –> https://github.com/ejeschke/aggdraw
- this will make it easy to support all kinds of surfaces because the graphics drawing code does not have to be replicated for each toolkit
- see example code in /scripts/example1_agg_gtk.py
- currently not needed for Gtk, Qt versions of the object

2.34.2 New Tk backend

There is now a Tk rendering version of the ImageView object.

- see ginga.tkw.ImageViewTk
- renders on a Tk canvas
- see example code in /scripts/example{1,2}_tk.py
- you will need the aggdraw module (see above) to use it

2.34.3 AutoCuts

- the ginga.AutoCuts module has been refactored into individual classes for each algorithm

- **The Preferences plugin for ginga now exposes all of the parameters**
  used for each cut levels algorithm and will save them

2.34.4 Etc

- additions to the manual (still incomplete, but coming along)

- lots of docstrings for methods added (sphinx API doc coming)

- many colors added to the color drawing example programs

- WhatsNew.txt file added

QUICK REFERENCE

3.1 Main image window

These keyboard and mouse operations are available when the main image window has the focus.

3.1.1 Mode control commands

About modes

Certain keystrokes invoke a *mode*—modes are usually indicated by the *mode indicator*: a small black rectangle with the mode name in one corner of the view. In a mode, there are usually some special key, cursor, and scroll bindings that override *some* of the default ones.

Modes additionally have a *mode type* which can be set to one of the following:

- `held`: mode is active while the activating key is held down

- `oneshot`: mode is released by initiating and finishing a cursor drag or when `Esc` is pressed, if no cursor drag is performed

- `locked`: mode is locked until the mode key is pressed again (or `Esc`)

By default, most modes are activated in "locked" type. The mode type is indicated in the brackets after the mode name in the mode indicator. The following keys are important for initiating a mode:

Commmand	Description
Space	Enter "meta" mode. Next keystroke will trigger a particular mode.
Esc	Exit any mode.

"meta" mode

Most modes are defined so that they are invoked from a special intermediate mode called "meta". In that case a two-key sequence is required to enter the mode: pressing the key that invokes "meta" and then pressing the key that invokes the desired mode. The following table shows the modes that can be triggered from meta mode.

Commmand	Description
Space	Exit/Enter "meta" mode.
b	Enter draw mode (canvas must be enabled to draw).
q	Enter *Pan mode*.
w	Enter *Freepan mode*.
r	Enter *Rotate mode*.
t	Enter *Contrast mode*.
y	Enter *CMap (color map) mode*.
s	Enter *Cuts mode*.
d	Enter Color Distribution ("dist") mode. See *Dist mode*.
n	Enter naxis (cube axis navigation) mode. See *Naxis mode*.
L	Toggle the normal lock to the current mode or any future modes.

3.1.2 Panning and zooming commands

Commmand	Description
Scroll wheel turned	Zoom in or out.
Shift + scroll wheel	Zoom while keeping location under the cursor.
Ctrl + scroll wheel turned	Pan in direction of scroll.
Digit (1234567890)	Zoom image to zoom steps 1, 2, …, 9, 10.
Shift + Digit	Zoom image to zoom steps -1, -2, …, -9, -10.
Backquote (`)	Zoom image to fit window and center it.
Minus, Underscore (-, _)	Zoom out.
Equals, Plus (=, +)	Zoom in.
Middle (scroll) button click	Set pan position (under cursor).
p	Set pan position (under cursor) for zooming.
Shift + left-click	Set pan position for zooming.
Shift + arrow key	Move pan position 1 pixel in that direction.
c	Set pan position to the center of the image.
Ctrl + left-drag	Proportional pan (press and drag left mouse button.
slash (/)	Set autocenter for new images to *override*.
question (?)	Toggle autocenter for images to *on* or *off*.
apostrophe (')	Set autozoom for new images to *override*.
double quote (")	Toggle autozoom for new images to *on* or *off*.

3.1.3 Cut levels and colormap commands

Commmand	Description
a	Auto cut levels.
D	Reset color distribution algorithm to "linear".
T	Restore the contrast (via colormap) to its original (unstretched, unshifted) state.
Y	Restore the color map to default (gray).
I	Invert the color map.
semicolon (;)	Set autocuts for new images to *override*.
colon (:)	Toggle autocuts for new images to *on* or *off*.

3.1.4 Transform commands

Commmand	Description
Left bracket ([)	Toggle flip image in X.
Left brace ({)	Reset to no flip of image in X.
Right bracket (])	Toggle flip image in Y.
Right brace (})	Reset to no flip image in Y.
Backslash (\)	Swap X and Y axes.
Vertical bar (|)	Reset to no swap of X and Y axes.
R	Restore rotation to 0 degrees and additionally undo any flip/swap transformations.
period (.)	Increment current rotation by 90 degrees.
comma (,)	Decrement current rotation by 90 degrees.
o	Orient image by transforms and rotation so that WCS indicates North=Up and East=Left.
O	Orient image by transforms and rotation so that WCS indicates North=Up and East=Right.

3.1.5 Pan mode

Commmand	Description
left-drag	Pan proportionally to drag.
middle-click	Set pan position.
right-drag	Zoom in/out proportionally to L/R drag.
<Modifier> + arrow key	Pan in direction of arrow key. Adding Ctrl reduces amount, adding Shift reduces more.
p	Pan to position under cursor.
z	Save current scale (see below for use).
backquote (`)	Zoom to fit window and center.
r	Pan to cursor and zoom to saved scale level (or 1:1 if no scale level saved).
c	Set pan position to the center of the image.
slash (/)	Set autocenter for new images to *override*.
question (?)	Toggle autocenter for images to *on* or *off*.
apostrophe (')	Set autozoom for new images to *override*.
double quote (")	Toggle autozoom for new images to *on* or *off*.

3.1.6 Freepan mode

Commmand	Description
Turn scroll wheel	Zoom while keeping location under the cursor.
left-click	Set pan position, zoom in a step and warp cursor to pan position (if supported on backend).
right-click	Set pan position, zoom out a step and warp cursor to pan position (if supported on backend).
middle-drag	Pans freely over entire image in proportion to cursor position versus window.
p, z, backquote, r, c, arrow keys	(Same as for *Pan mode*.)

3.1.7 Dist mode

Commmand	Description
scroll	Select distribution from linear, log, etc.
b, up-arrow	Select prev distribution in list.
n, down-arrow	Select next distribution in list.
D	Reset color distribution algorithm to "linear".

3.1.8 Cuts mode

Commmand	Description
left-drag	Interactive cut *both* low and high levels (vertical cuts low, horizontal cuts high).
Ctrl + left-drag	Interactive cut low level only (horizontal drag).
Shift + left-drag	Interactive cut high level only (horizontal drag).
scroll	Coarse (10%) adjustment in/out.
Ctrl + scroll	Fine (1%) adjustment in/out.
a, right-click	Do an auto level to restore cuts.
h	Set the high cut to the value under the cursor.
l	Set the low cut to the value under the cursor.
S	Set cuts to min/max values.
A	Set cuts to 0/255 values (for 8bpp RGB images).
b, up-arrow	Select prev auto cuts algorithm in list.
n, down-arrow	Select next auto cuts algorithm in list.
semicolon (;)	Set autocuts for new images to *override*.
colon (:)	Toggle autocuts for new images to *on* or *off*.

3.1.9 Contrast mode

Commmand	Description
left-drag	Interactive shift/stretch colormap (AKA contrast and bias). Left/Right controls shift, Up/Down controls stretch.
right-click	Restore the contrast (via colormap) to its original (unstretched, unshifted) state.
T	Restore the contrast (via colormap) to its original (unstretched, unshifted) state.

3.1.10 Rotate mode

Commmand	Description
left-drag	Drag around center of window to rotate image.
right-click	Restore rotation to 0 degrees (does not reset any flip/swap transformations).
R	Restore rotation to 0 degrees and additionally undo any flip/swap transformations.
Left bracket ([)	Toggle flip image in X.
Left brace ({)	Reset to no flip of image in X.
Right bracket (])	Toggle flip image in Y.
Right brace (})	Reset to no flip image in Y.
Backslash (\)	Swap X and Y axes.
Vertical bar (|)	Reset to no swap of X and Y axes.
period (.)	Increment current rotation by 90 degrees.
comma (,)	Decrement current rotation by 90 degrees.
o	Orient image by transforms and rotation so that WCS indicates North=Up and East=Left.
O	Orient image by transforms and rotation so that WCS indicates North=Up and East=Right.

3.1.11 Cmap mode

Commmand	Description
scroll	Select color map.
left-drag	Rotate color map.
right-click	Unrotate color map.
b, up-arrow	Select prev color map in list.
n, down-arrow	Select next color map in list.
I	Toggle invert color map.
r	Restore color map to unrotated, uninverted state.
Ctrl + scroll	Select intensity map.
j, left-arrow	Select prev intensity map in list.
k, right-arrow	Select next intensity map in list.
i	Restore intensity map to "ramp".
c	Toggle a color bar overlay on the image.
Y	Restore the color map to default ('gray').

3.1.12 Naxis mode

Note: Naxis mode is used when viewing data that has more than 2 dimensions (e.g., data cubes).

Commmand	Description
scroll	Select previous or next slice of current axis.
Ctrl + scroll	Select previous or next axis as current axis.
left drag	select slice as a function of percentage of cursor/window width.
up-arrow	Select prev axis as current axis.
down-arrow	Select next axis as current axis.

3.1.13 Plot mode

Plot mode is only valid when the viewer is used with a `PlotAide` object to display a graph.

Commmand	Description
scroll	Zoom the X axis. (Sets `autoaxis_x` to `off`, if it was `on`)
Ctrl + scroll	Zoom the Y axis. (Sets `autoaxis_y` to `off`)
x	Toggle `autoaxis_x` between `on` and `off`.
p	Toggle `autoaxis_x` between `pan` and `off`.
y	Toggle `autoaxis_y` between `on` and `off`.
v	Toggle `autoaxis_y` between `vis` and `off`.

The graph can be flipped in X or Y and the axes swapped, using the keystroke commands found in the transform section above (*Transform commands*).

To understand the role of the `autoaxis_x` and `autoaxis_y` settings, please see the chapter on plots (*Plots*).

3.1.14 Autozoom setting

The "autozoom" setting can be set to one of the following: "on", "override", "once" or "off". This affects the behavior of the viewer when changing to a new image (when done in the typical way) as follows:

- `on`: the image will be scaled to fit the window
- `override`: like `on`, except that once the zoom/scale is changed by the user manually it turns the setting to `off`
- `once`: like `on`, except that the setting is turned to `off` after the first image
- `off`: an image scaled to the current viewer setting

(In the *Reference Viewer*, this is set under the "Zoom New" setting in the channel preferences.)

3.1.15 Autocenter setting

The "autocenter" setting can be set to one of the following: "on", "override", "once" or "off". This affects the behavior of the viewer when changing to a new image (when done in the typical way) as follows:

- `on`: the pan position will be set to the center of the image
- `override`: like `on`, except that once the pan position is changed by the user manually it turns the setting to `off`
- `once`: like `on`, except that the setting is turned to `off` after the first image
- `off`: the pan position is taken from the current viewer setting

(In the *Reference Viewer*, this is set under the "Center New" setting in the channel preferences.)

3.1.16 Autocuts setting

The "autocuts" setting can be set to one of following: "on", "override", "once" or "off". This affects the behavior of the viewer when changing to a new image (when done in the typical way) as follows:

- `on`: the cut levels for the image will be calculated and set according to the autocuts algorithm setting

- `override`: like `on`, except that once the cut levels are changed by the user manually it turns the setting to `off`

- `once`: like `on`, except that the setting is turned to `off` after the first image

- `off`: the cut levels are applied from the current viewer setting

(In the ref:`Reference Viewer`, this is set under the "Cut New" setting in the channel preferences.)

3.1.17 Reference Viewer Only

Command	Description
H	Raise **Header** tab (if Header plugin has been started).
Z	Raise **Zoom** tab (if Zoom plugin has been started).
D	Raise **Dialogs** tab.
C	Raise **Contents** tab.
less than (<)	Toggle collapse left pane.
greater than (>)	Toggle collapse right pane.
f	Toggle full screen.
F	Panoramic full screen.
m	Maximize window.
J	Cycle workspace type (tabs/mdi/stack/grid). Note that "mdi" type is not supported on all platforms.
k	Add a channel with a generic name.
Left, Right (arrow keys)	Previous/Next channel.
Up, Down (arrow keys)	Previous/Next image in channel.

Note: If there are one or more plugins active, additional mouse or keyboard bindings may be present. In general, the left mouse button is used to select, pick or move, and the right mouse button is used to draw a shape for the operation.

On the Mac, Ctrl + mouse button can also be used to draw or right-click. You can also invoke draw mode as described above in the section on modes.

USER'S MANUAL

NN

Ginga is a toolkit for building viewers for scientific data in Python, particularly astronomical data. It also includes a reference viewer for viewing FITS (Flexible Image Transport System) files.

The Ginga viewer is based on an image display widget that supports:

- Zooming and panning
- Color and intensity mapping
- A choice of several automatic cut levels algorithms, and
- Canvases for plotting scalable geometric forms.

In addition to the image display widget, the Ginga viewer provides a flexible plugin framework for extending the viewer with many different features.

A relatively complete set of standard plugins is provided for features that we expect from a modern viewer: panning and zooming windows, star catalog access, cuts, star pick/fwhm, and thumbnails.

4.1 Introduction

4.1.1 About

Ginga is a toolkit designed for building viewers for scientific image data in Python, visualizing 2D pixel data in numpy arrays. By default the Ginga toolkit can view standard RGB(A) type images as well as astronomical data such as contained in files based on the FITS (Flexible Image Transport System) file format. The viewer is easily extended to handle additional formats, so long as the data can be accessed as numpy arrays.

The Ginga toolkit is written and maintained by software engineers at the Subaru Telescope, National Astronomical Observatory of Japan, and the Space Telescope Science Institute. The code is released as open-source under a BSD license and maintained at http://ejeschke.github.io/ginga/

4.1.2 Features

The Ginga toolkit centers around an image display widget that supports zooming and panning, color and intensity mapping, a choice of several automatic cut levels algorithms, and canvases for plotting scalable geometric forms.

In addition to this widget, a general purpose reference FITS viewer is provided, based on a plugin framework. A relatively complete set of standard plugins are provided for features that we expect from a modern FITS viewer: panning and zooming windows, star catalog access, cuts, star pick/fwhm, thumbnails, etc.

4.2 Core Concepts

The Ginga reference viewer operation is organized around several basic concepts: *workspaces*, *channels*, *plugins*, and *modes*. Understanding these will greatly aid in using and modifying Ginga.

4.2.1 Workspaces

Ginga has a flexible workspace layout algorithm that allows customizing the appearance of the program. The majority of the Ginga interface is constructed as hierarchical series of horizontally or vertically-adjustable panels. Each panel usually terminates eventually into one or more *workspaces*.

Each workspace is implemented by a GUI toolkit container widget such as a notebook widget, where each item in the workspace is identified by a tab. However, workspaces can also take the form of a stack (like a tabbed widget but with no tabs showing), or a Multiple Document Interface (MDI) style container (subwindow desktop-style layout), or a grid layout.

Workspaces typically contain either a channel *viewer*, a *plugin* UI or another workspace. In its default configuration, Ginga starts up with a single row (horizontal) panel of three workspaces, as shown in the image below.
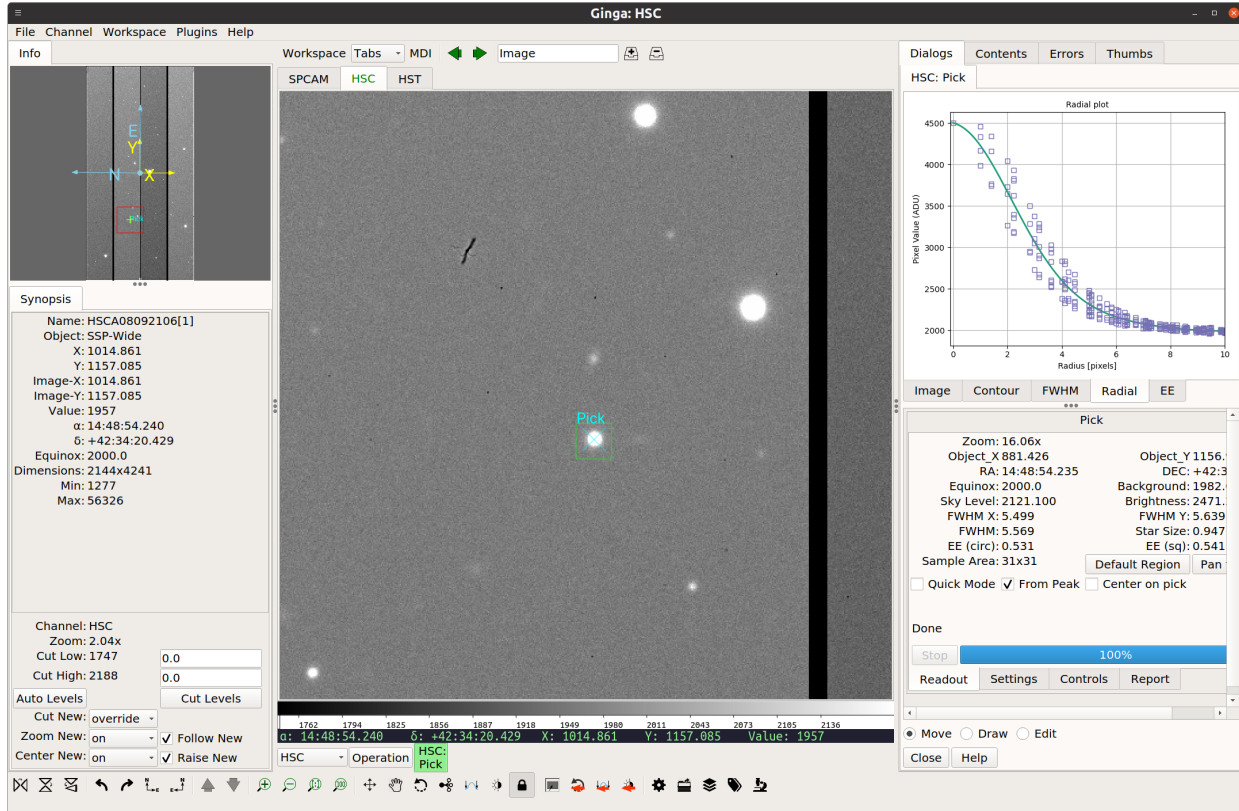
The panel is sandwiched vertically between a menu bar and a tool bar. The left workspace is further subdivided into an upper and lower, and there are also thin horizontal workspaces below the central workspace. The central workspace is mainly used for viewers, while the other workspaces hold the plugin UIs.

Some workspaces can be converted dynamically between the different types. If the workspace contains a *workspace toolbar*, the workspace type selector can be used to change the type:
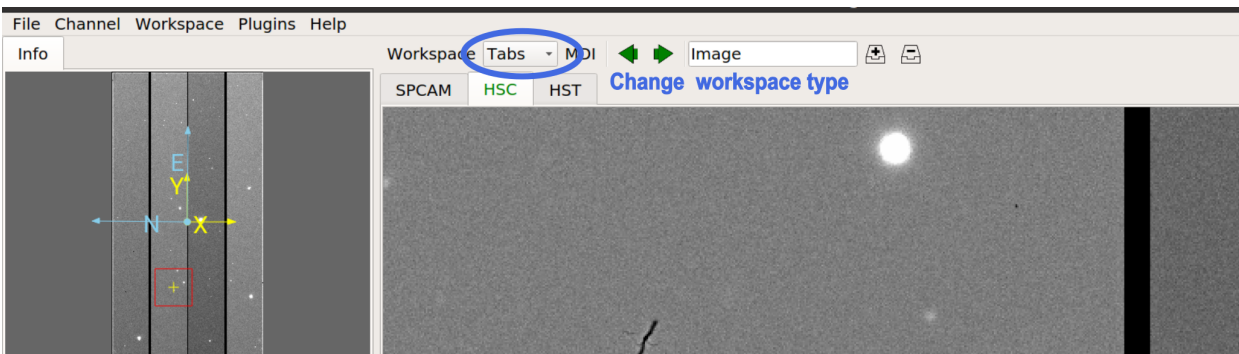
The main workspace configured as "Tabs" (tabbed notebook, the default):

The main workspace configured as "Grid":
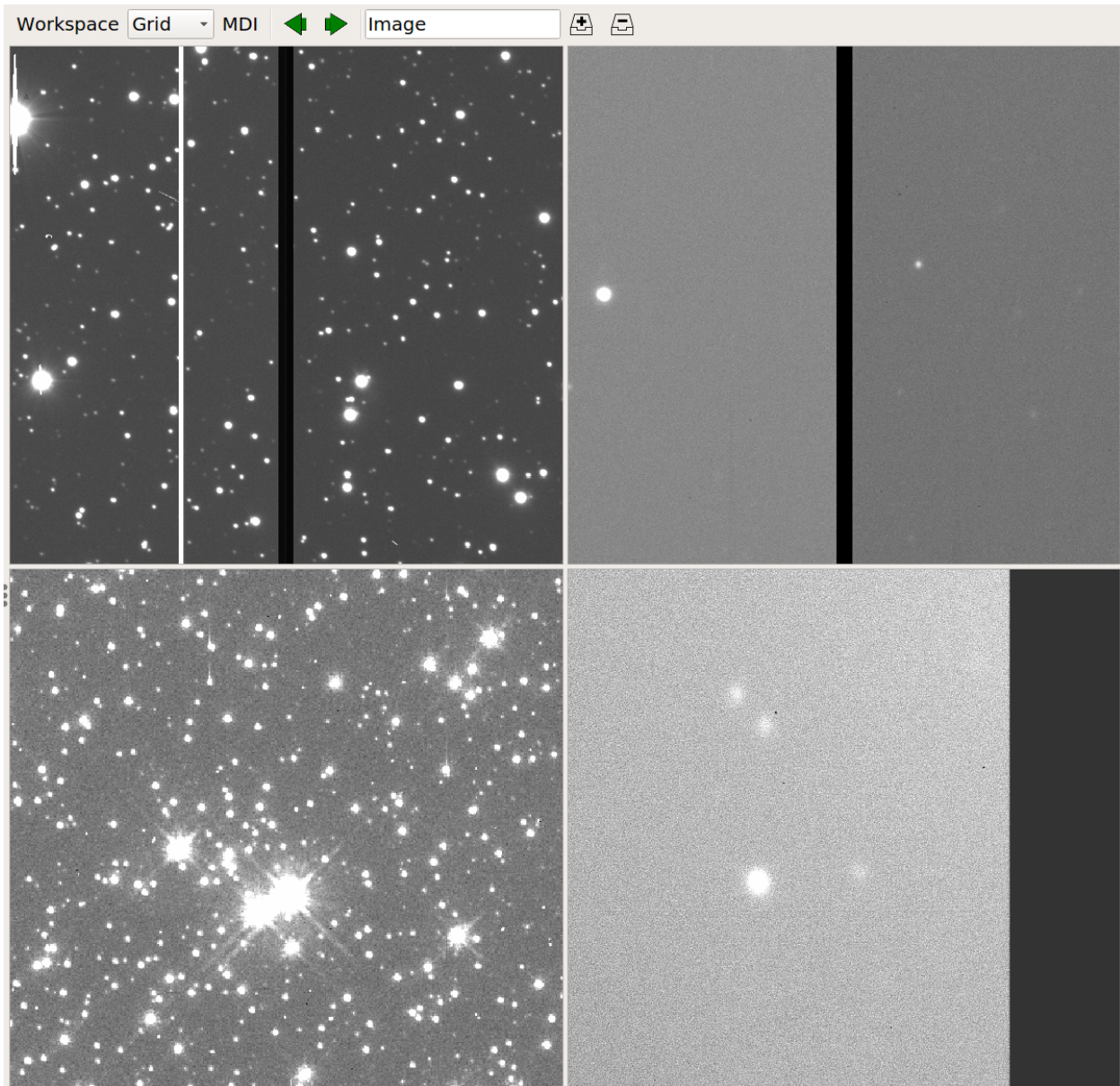
The main workspace configured as "Stack" (need to use workspace toolbar left/right arrows to switch between subwindows):
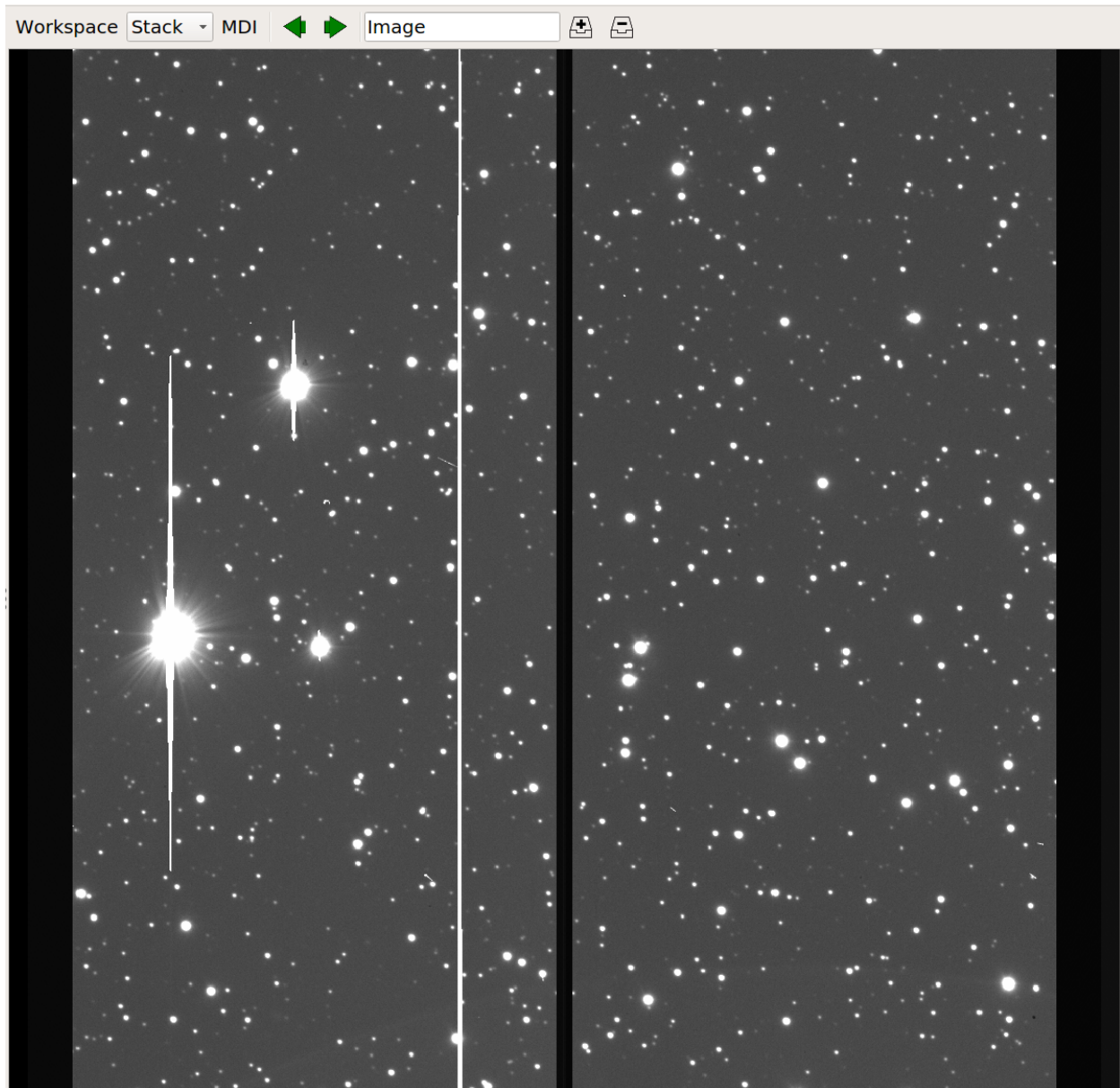
The main workspace configured as "MDI" (individual windows can be sized and moved around the workspace area):
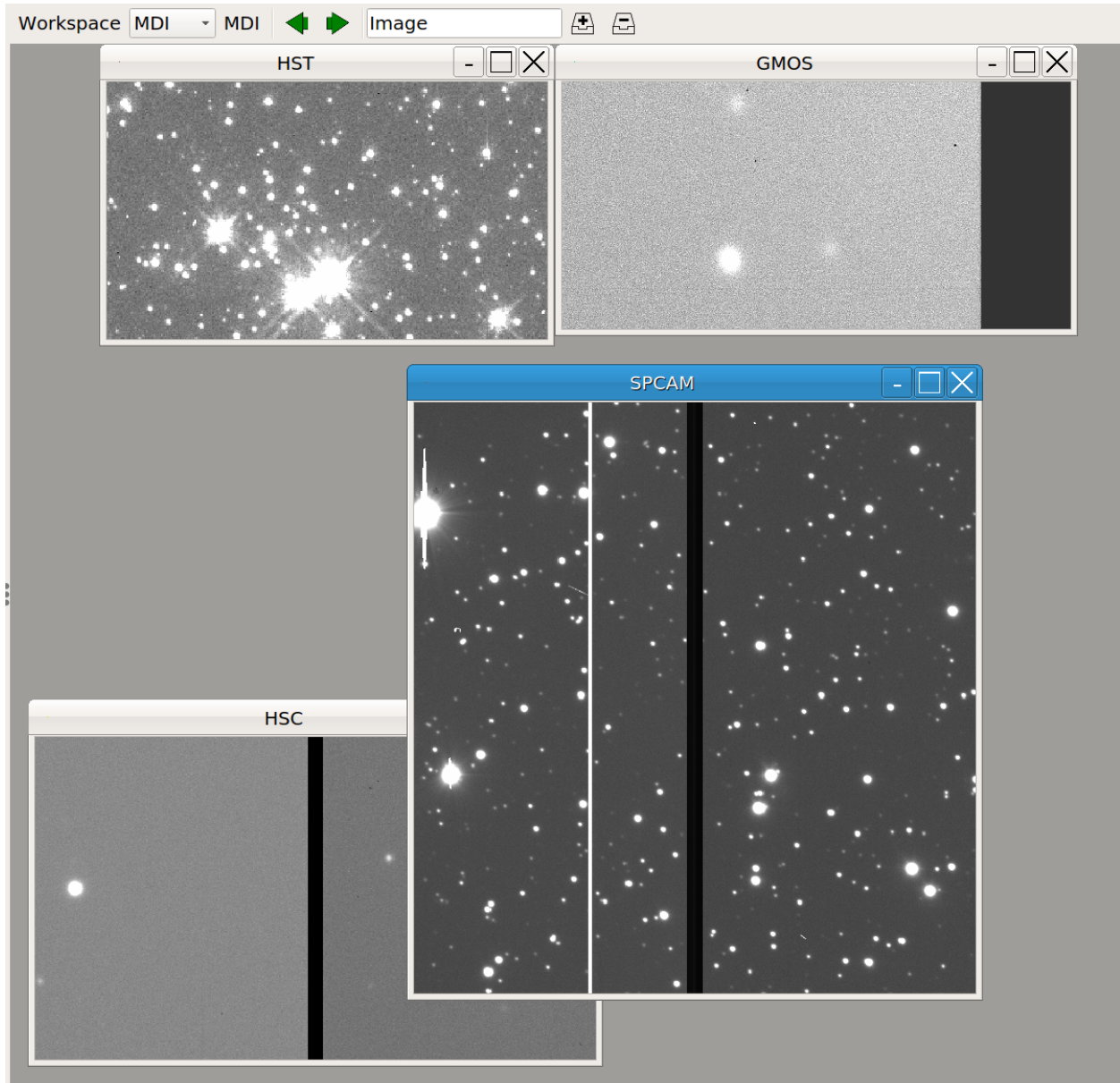
In the MDI configuration, the "MDI" menu in the workspace toolbar is enabled and can be used to tile or cascade the subwindows.

Workspaces can be nested. In the example shown below, we show a cutout of the main workspace (type "MDI") which has sub-workspaces: *ws1*, configured as type *Tabs* and has two channels, and *ws2*, configured as type "Grid" and containing four channels.
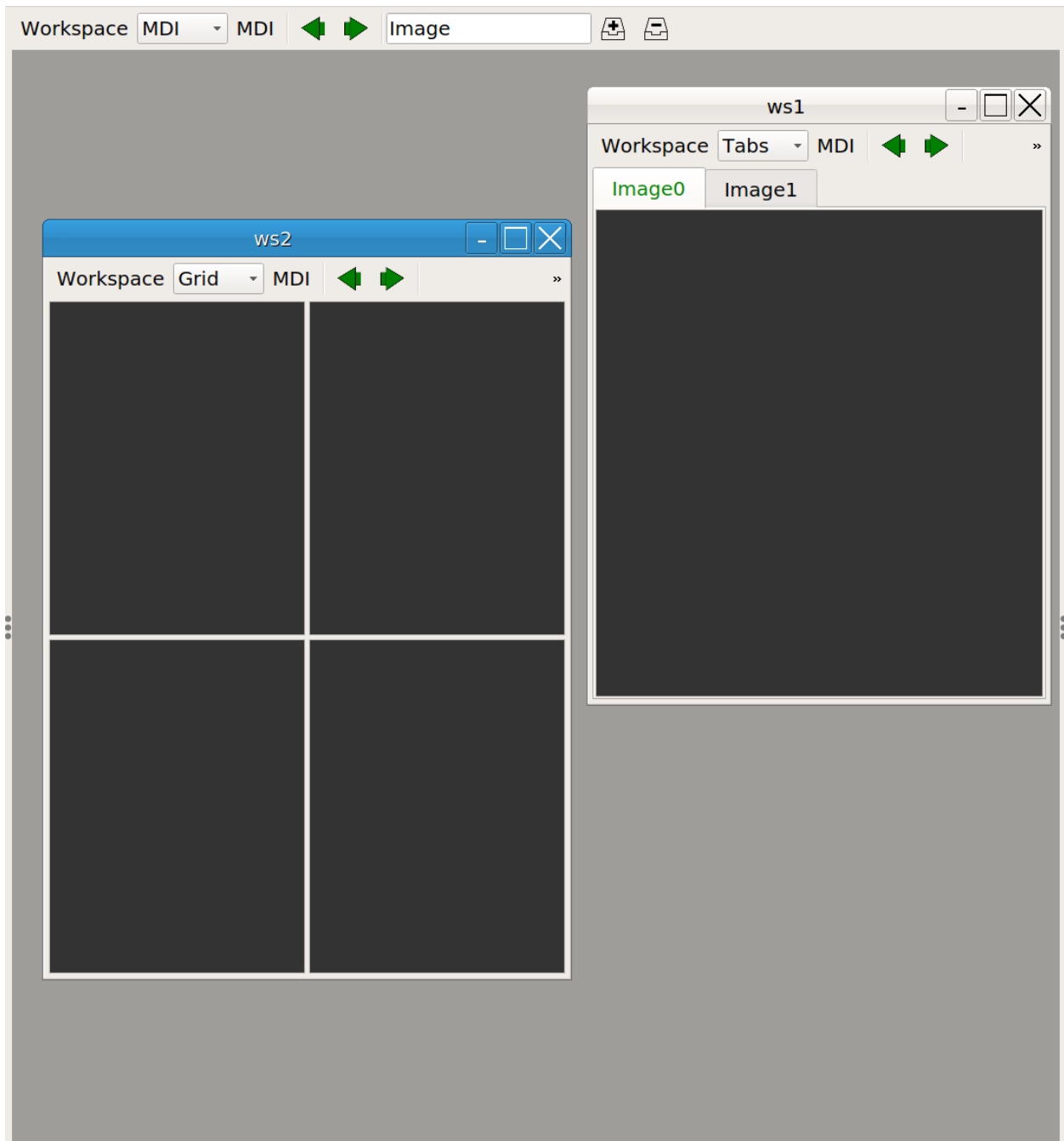
The initial layout of the workspaces is controlled by a table in the Ginga startup script (see *Customizing Ginga*). By changing this table the layout can be substantially altered.

4.2.2 Channels

Another core tenet of Ginga is that that data content is organized into *channels*. A channel can be thought of as simply a named category under which similar types of data might be organized. A few examples are:

- A channel for each type of instrument at a telescope

- A channel for each observation or calibration target

- Channels based on time or program or proposal identifier

If no channels are specified when Ginga starts up it simply creates a default channel named *Image*. New channels can be created using the *Channel/Add channel* menu item. Pressing the + button in the workspace menu also adds a new channel using the name specified in the text box just to the left of the button, or using a default prefix, if no name is specified. Clicking – removes the currently selected channel. Clicking the left or right arrow buttons changes which channel is selected in the workspace.

In the case where multiple channels are present, they are usually visually organized as tabs, a stack of windows, a grid, or an MDI interface within the workspace as described in the section above, depending on how the workspace is configured.
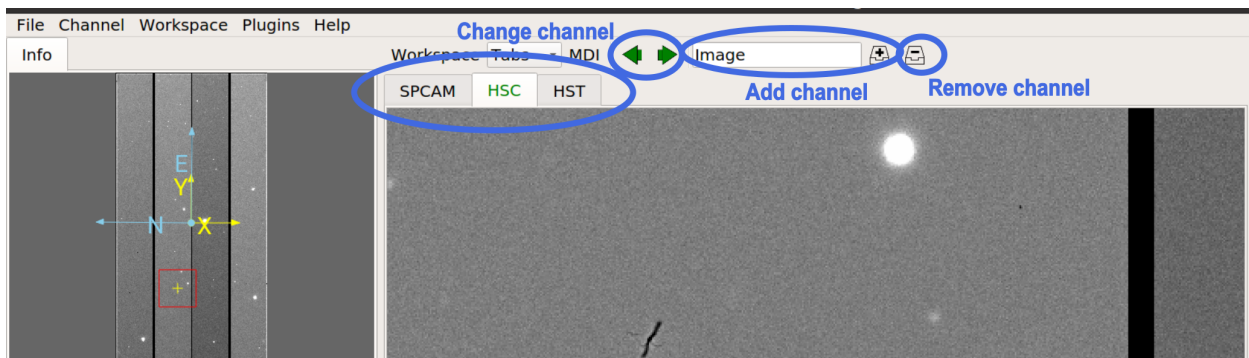
To change channels you simply click on the tab of the channel you want to view, or press the left or right arrow buttons in the workspace menu. There is also a channel selector in the plugin manager toolbar at the bottom of the center pane. You can change the channel by using the drop-down menu or by simply scrolling the mouse wheel on the control.
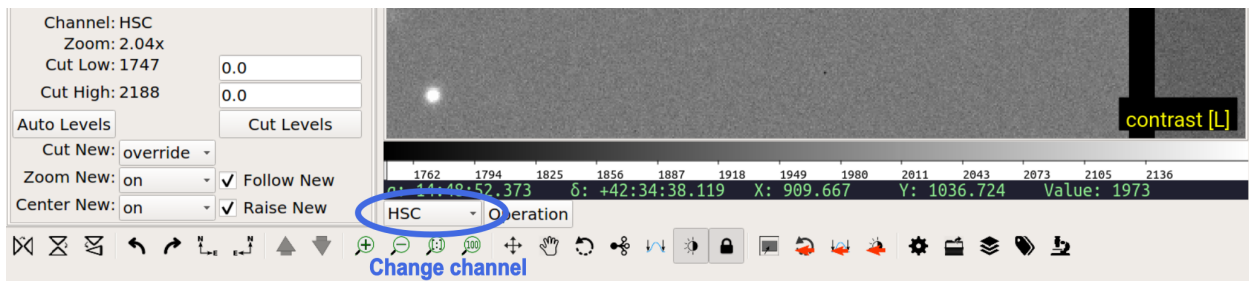
Channels occupy a flat namespace, i.e., there is no hierarchy in channel names.

Channel Viewers

A channel always has an image viewer associated with it, and may additionally have viewers for tables or other content. Only one viewer is active at a time per channel. The channel viewer is what you see in the window representing that channel. The viewer will display the content of the currently selected data item in that channel, and the appropriate viewer will be change according to the type of the currently selected item in the channel (e.g. the table viewer will be activated when a table is selected, the image viewer when an image is selected and so on).

In the Toolbar plugin, just above the status bar near the bottom of the window, clicking the up or down arrows moves between items within the selected channel. The ordering of the items in the channel is by default determined by the order in which they were added, but can be changed to alphanumeric sorted ordering if desired (this is a channel preference that can be set under the "General" category in the channel preferences).
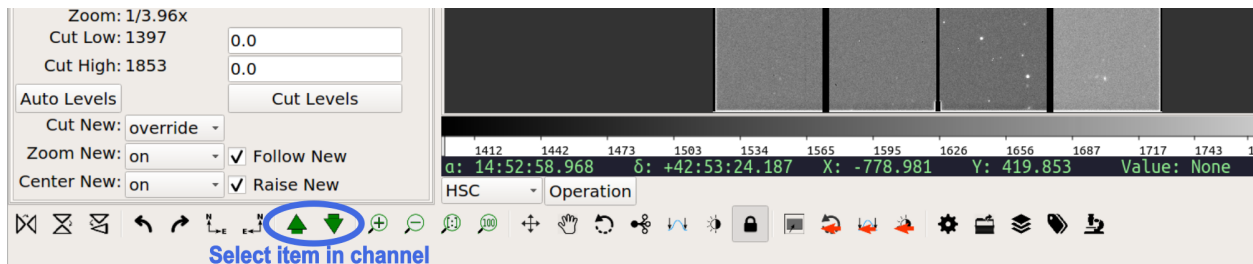
By default, images are loaded into the same channel you are currently viewing (unless your viewer has been customized to load images according to special rules).

Note: To keep items organized, simply change to the desired channel before opening a new item or drag the file to the desired channel viewer.

Channel Settings

Many preferences in Ginga are set on a per-channel basis. Some per-channel settings include:

- Color distribution algorithm
- Color map
- Intensity map
- Cut levels
- Auto cut levels algorithm
- Transforms (flip, swap)
- Rotation
- WCS display coordinates
- Zoom algorithm
- Scale
- Interpolation type
- Pan position

A new channel will generally inherit the settings for the generic *Image* channel until new preferences are defined and saved for that channel.

If you create a new channel and had previously saved preferences for a channel with that name, the new channel will adopt those preferences. Thus you can set up channels configured for certain telescopes or for types of data and easily reuse them in later sessions.

Another idea embodied in the channel concept is that the user should not have to manage memory usage too explicitly. Each channel has a setting that limits how many images it should keep in memory. If the number of images exceeds the limit then Ginga will remove older images and load them back in as needed without user intervention.

Note: Many channel settings can be set and saved using the "Preferences" plugin.

4.2.3 Plugins

Almost all functionality in Ginga is achieved through the use of a plugin architecture.

Plugins are quasi-independent Python modules that can optionally have a Graphical User Interface. If they do have a GUI, it can be loaded at program startup or be dynamically opened and closed during the duration of the viewer's execution.

Plugins can be *global*, in which case they don't have any particular affiliation with a channel and are generally invoked singularly, or *local* in which case they can be invoked in multiple instances—one per channel. Global plugins are often written in a way that they respond to the action of the user changing channels. As an example, the "Pan" plugin will change its image to match the image shown in the selected channel.

In this documentation we will also use the word *operation* to describe activating a plugin. For example, a "pick" operation would use the Pick plugin.

Plugins are written as encapsulated Python modules that are loaded dynamically when Ginga starts. There is an API for programming plugins (see *Developing with Ginga*).

The plugins are each described in more detail in *Plugins*.

For those plugins that do have a visible interface, the Ginga startup script can map them to certain workspaces. By manipulating this mapping (and manipulating the workspace layout) we can customize the reference viewer to achieve flexible layouts.

In the image at the top, the left workspace contains UIs for two global plugins: *Pan* (shown under the tab "Info") and *Info* (somewhat confusingly, shown under the tab "Synopsis"–the tab name for a plugin can be different from it's canonical name). The middle workspace holds all the viewing panes for each channel. The right workspace has the Dialogs, Thumbs, Contents and Error panes. The operation of these plugins is described in *Plugins*.

4.2.4 Modes

Ginga provides a number of default bindings for key and pointer actions. However, there are too many different actions to bind to a limited set of keys and pointer buttons. *Modes* allow us to overcome this limitation.

Modes are a mechanism that allow Ginga to accommodate many key and pointer bindings for a large number of possible operations.

Modes are set on a per-channel basis. A mode is activated by pressing a particular key combination when the focus is in the viewer, or by clicking an appropriate button in the Toolbar plugin. When the viewer is in a mode, the behavior is that some special key, pointer and scroll bindings will be activated and override the default ones. An adjacent viewer for a different channel may be in a different mode, or no mode.

Note: If a mode does not override a particular binding, the default binding will still be active, unless an active canvas being shown in the viewer has registered for the same binding.

4.3 General Operation

This chapter describes the general manipulations of images using Ginga.

For the most part these manipulations apply both to Ginga ImageView classes that can be embedded in a Python application, as well as to the reference viewer distributed with Ginga.

Note: In cases where we are referring to something that is only available in the reference viewer these will be prefixed by the notation [RV].

4.3.1 Keyboard and mouse operations

In this documentation we will use the following terms to describe the operations performed with the mouse:

- *Click* or *Left-click* means to click on an item with the left mouse button;
- *Drag* or *Left-drag* means to click, hold and drag with the left mouse button;
- *Scroll* means to scroll with the middle mouse wheel or a trackpad/touchpad;
- *Scroll-click* means to click with the middle mouse wheel/button;
- *Scroll-drag* means to click, hold and drag with the middle mouse wheel/button;
- *Right-click* means to click on an item with the right mouse button;
- *Right-drag* means to click, hold and drag with the right mouse button.

Mouse operations are also modified by the keyboard buttons *Shift*, and *Ctrl*.

Shift-click means to press *and hold* the Shift key while clicking with left mouse button. *Shift-right-click* is the same using the right mouse button, etc.

Some mouse-controlled operations in Ginga are initiated by a key stroke. In these cases the key is pressed and released (not held), and then the mouse is used to control the operation. Such operations are either terminated by releasing the mouse button (if the operation employs a drag), and clicking on the image or by pressing the Esc key (if not a drag operation).

Note: We describe the standard key and mouse bindings here. However these bindings can be changed completely by the user. For more information on changing the bindings, see Section *Binding Config File*.

4.3.2 Loading a FITS or RGB image file

There are several ways to load a file into Ginga:

- Ginga supports drag-and-drop in a typical desktop environment, so you can simply drag and drop files from a graphical file manager such as the Mac Finder or Linux Nautilus onto a Ginga viewing pane to load an image.

- [RV] Another way is to invoke the `FBrowser` plugin, which opens in the *Dialogs* tab. The plugin pane shows file and folder contents and allows navigation up and down the filesystem hierarchy by double-clicking on folder names. Simply navigate to the location of the file and double-click on the file name to load it, or drag it onto the image pane.

- [RV] Use the *Load Image* entry from the `File` menu on the main menu bar at the top of the window. This opens a standard file dialog popup window where you can navigate to the file you wish to load.

4.3.3 Zooming and panning

The display object used throughout most of the Ginga panels has built-in support for zooming and panning. The *Quick Reference* has the complete listing of default keyboard and mouse bindings.

For example:

- The scroll wheel of the mouse can be used to zoom in and out, along with the "+" and "-" keys.

- The backquote key will fit the image to the window.

- Digit keys (1, 2, etc.) will zoom in to the corresponding zoom level, while holding Shift and pressing a zoom key zooms out to the corresponding level.

When zoomed in, panning is enabled. Panning takes two forms:

1) *Proportional panning* or "drag panning" pans the image in direct proportion to the distance the mouse is moved. You can think of this as dragging the image canvas in the direction you want to move it under the window portal. To utilize a proportional pan, Ctrl-drag the canvas, or press Space followed by "q" to go into pan mode, and then drag the canvas.

2) *Free panning* allows scrolling around the entire image by mapping the entire image boundaries to the window boundaries. For example, moving the mouse to the upper right-hand corner of the window will pan to the upper right hand corner of the image, etc. You can think of this mode as moving the window portal around over the canvas. To initiate a free pan, press Space followed by "w" to enter "freepan" mode and then Scroll-drag to move around the window.

[RV] The `Pan` plugin (usually embedded under the *Info* tab) shows the outline of the current pan position as a rectangle on a small version of the whole image. Dragging this outline will also pan the image in the main window. You can also click anywhere in the Pan window to set the pan position, or right drag an outline to roughly specify the region to zoom and pan to together.

Pan position

Panning in Ginga is based on an (X, Y) coordinate known as the *pan position*. The pan position determines what Ginga will try to keep in the middle of the window as the image is zoomed.

When zoomed out, you can Shift-click on a particular point in the image (or press the "p" key while hovering over a spot), setting the pan position. Zooming afterward will keep the pan position in the center of the window. To reset the pan position to the center of the image, press 'c'.

Ginga has an auto zoom feature to automatically fit newly loaded images to the window, similar to what happens when the backquote key is pressed. See "Zoom Preferences" section in *Preferences* for details.

4.3.4 How Ginga maps an image to color

The process of mapping a monochrome science image to color in Ginga involves four steps, in order:

1) Applying the *cut levels*, which scales all values in the image to a specified range[1],

2) Applying a *color distribution algorithm*, which distributes values within that range to indexes into a color map table, and

3) Applying a *shift map*, which shifts and stretches or shrinks the values according to the user's contrast adjustment[2], and finally,

4) Applying an *intensity map* and *color map* to map the final output to RGB pixel values.

Setting cut levels

When visualizing pixel data with an arbitrary value range, the range is first scaled into a limited range based on the low and high *cut levels* defined in the view object. These cut levels can be set manually by the user or automatically based on an algorithm. This eliminates the effect of outlier pixel/flux values.

Manually setting cut levels

There are several ways to manually set the cut levels:

• Pressing Space followed by "s" key will put the viewer into "cuts" mode. Here you can invoke a dual (high and low) interactive cut levels. Click and drag the mouse horizontally in the window to interactively set the high level, and vertically to set the low level; and when you reach the desired levels, release the mouse button. Scrolling the mouse wheel in this mode will also change the low and high cut levels simultaneously–toward or away from each other, resulting in lower or higher contrast.

• [RV] The "Cut Low" and "Cut High" boxes in the Info plugin panel can be used. The current values are shown to the left; simply type a new value in the corresponding box and press Enter or click the "Cut Levels" button below. Cut values can also be set from the "Histogram" plugin.

Automatically setting cut levels

Ginga can algorithmically estimate and set the cut levels–called *auto (cut) levels*. To activate the auto levels:

• Press the ("a") key when the viewing widget has the focus.

• [RV] Click the "Auto Levels" button in the Info plugin panel.

[RV] The auto cut levels feature is controlled by several factors in the preferences, including the choice of algorithm and some parameters to the algorithm. See "Auto Cuts Preferences" section in *Preferences* for details.

Ginga can also automatically set the cut levels for new images displayed in the view. See "New Image Preferences" section in *Preferences* for details.

---

[1] Some image viewers or graphing programs use the term "limits" for what we call "cut levels".
[2] What some programs call a "contrast/bias" adjustment.

Setting the color distribution algorithm

Ginga supports a number of color scale distribution algorithms, including:

- "linear",
- "log",
- "power",
- "sqrt",
- "squared",
- "asinh",
- "sinh", and
- "histeq" (histogram equalization).

These can be sampled with the current color and intensity maps by pressing Space followed by "d" key to go into "dist" mode, and then scrolling the mouse, pressing the up/down keys, or the "b" and "n" keys.

Press Esc to exit the "dist" mode.

To reset to the default ("linear") map, press "D" (capital D).

[RV] The color scale distribution algorithms can also be set from the `Preferences` plugin, under the heading "Color Distribution".

Making contrast adjustments

The value range can be shifted and stretched or squeezed to alter the visibility and contrast of the image. This is sometimes called a "bias/contrast" adjustment in other viewers.

In most Ginga configurations the shift map adjustment is bound to the Ctrl-right drag combination (hold Ctrl down and right drag). Dragging left/right shifts the map, and up/down stretches or shrinks the map.

You can also press "t" to enter "contrast" mode, where you can then use a regular Left-drag.

Changing the color and intensity maps

The color and intensity maps control the final mapping of colors to the values in the image.

Intensity Maps

Intensity maps are available to produce a final permutation on the value range of the image before color is applied. The function of these largely overlaps the function of the color distribution algorithm, so *most users will typically use either one or the other, but not both*.

For example, the intensity map "log" essentially applies a log distribution to the range. If this has already been done with the color distribution "log", the effect is doubly applied.

Possible values for the intensity map are:

- "equa",
- "expo",
- "gamma",

- "jigsaw",
- "lasritt",
- "log",
- "neg",
- "neglog",
- "null", "ramp" and
- "stairs".

"ramp" is the default value.

While in "cmap" mode (described below), the "j" and "k" keys can be used to cycle through the intensity maps.

Color Maps

To change color maps from the keyboard shortcuts, press Space followed by "y" to go into "cmap" mode. While in "cmap" mode you can change color maps by scrolling the mouse, pressing the up/down keys, or the "b" and "n" keys.

While in "cmap" mode, pressing "I" (uppercase) will invert the current color map. Press Esc to exit cmap mode.

Note: Setting a new color map will cancel the color map inversion. Some color maps are available in both regular and inverted forms. If selecting an already inverted (aka "reversed") color map it is not necessary to explicitly invert it.

While many color maps are available built in, users can also define their own color maps or use matplotlib color maps, if the `matplotlib` package is installed.

[RV] The `ColorMapPicker` global plugin is useful you to visualize all of the colormaps and apply one to the currently active channel viewer.

4.3.5 Transforming the image view

Ginga provides several controls for transforming the image view. The image can be flipped in the X axis ("horizontally"), Y axis ("vertically"), have the X and Y axes swapped, or any combination thereof. These operations can be done by keyboard shortcuts:

- Press "[" to flip in X, "{" to restore.
- Press "]" to flip in Y, "}" to restore.
- Press "" to swap X and Y axes, "|" to restore.

The image can also be rotated in arbitrary amounts.

An interactive rotate operation can be initiated by pressing Space follwed by "r" in the image and then dragging the mouse horizontally left or right to set the angle. Press "R" (Shift+R) to restore the angle to 0 (unrotated).

Note: It is less computationally-intensive to rotate the image using the simple transforms (flip, swap) than by the rotation feature. Rotation may slow down some viewing operations.

[RV] The image can also be transformed in the channel *Preferences* (see "Transform Preferences") which has checkboxes for flip X, flip Y, swap XY and a box for rotation by degrees.

4.4 Modes

As discussed in the Concepts section on *Modes*, modes are a mechanism that allow Ginga to accommodate many key and pointer bindings for a large number of possible operations. Modes are associated with the basic Ginga viewer widget and so can be used even with the widget standalone in your own programs (i.e., apart from the Reference Viewer; see *Developing with Ginga* for details).

4.4.1 Invoking Modes

Modes are used to make bindings between keyboard, mouse, trackpad and gesture events and certain operations on the viewer widget. By invoking a mode, certain keystrokes and cursor bindings are enabled for certain operations within the viewer window in which the mode was entered. Typically, pressing the space key with the widget focus in the viewer window is used to enter the special "Meta" mode, in which you can then enter any one of the registered modes by following up pressing its mode activation key.

Only one mode (or no mode) can be active at a time in a particular viewer widget. Once a mode is activated, the *mode indicator* should turn on in one of the corners of the viewer window to indicate which mode is active:
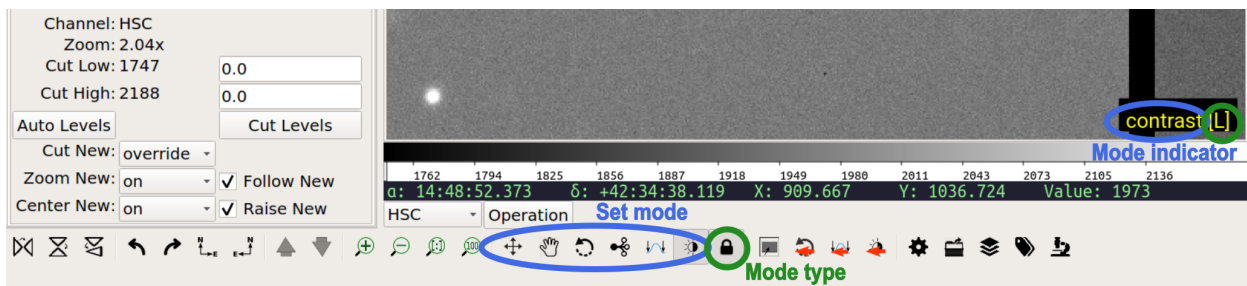
To exit a mode, press the Esc (escape) key.

Note: In the Reference Viewer, you can also enter and exit certain modes using the "Toolbar" plugin (see *Toolbar*).

4.4.2 Modal vs. Non-modal Operation

Ginga viewer widgets have a dual modal/non-modal (aka "modeless") operation. If no mode is active, then certain operations from a mode can still be invoked by default bindings that are declared for the modeless state. For example, by default, the scroll wheel will normally operate the zoom operation on the viewer even though that is officially an operation defined in the "Pan" mode. Furthermore, Ginga canvases can also register for keystroke and cursor bindings. The resolution order of handling a particular keyboard or cursor event is as follows:

1. if the viewer is in a mode, and there is a binding for the event in the mode

2. if there is a active canvas that has a binding for the event

3. if there is a binding for the event in the modeless state

At each stage of this event resolution, if the event is not handled by the event handler (in the previous stage), then the next stage of event handling is invoked.

Basically, this boils down to the following practical advice:

- A mode takes precedence; you can always count on the bindings in the mode doing what they do, no matter what canvases are active.

- If you are using an active canvas (like many Reference Viewer plugins do), you may need to exit out of a mode if bindings for the mode interfere (mask) necessary bindings for the canvas.

- If some default modeless binding doesn't seem to be working, it may be because an active canvas has registered for that binding. In such a case, you can invoke a mode to do the operation and then exit the mode to get back to working with the canvas, or close the plugin that has installed the active canvas.

4.4.3 Mode Types

The mode switching system has a *mode type* which can be set to one of:

- `oneshot`: The mode is exited by initiating and finishing a cursor drag, or when Esc is pressed (if no cursor drag is performed)

- `locked`: The mode is locked until the meta mode key is pressed again (or Esc)

By default most modes are activated in "locked" type.

Note: When the lock is active, it is signified by an additional "[L]" (locked) appearing in the mode indicator; oneshot mode is indicated by "[O]". In the figure above, you can see the mode indicator showing that the viewer is in "contrast" mode, with the mode type as "Lock". In the Reference Viewer, the same information can be seen in the Toolbar plugin, where the lock icon shows the state of the mode type (`engaged == 'Lock'`).

Standard Modes

These are the set of modes that come with Ginga. Those interested in writing their own custom mode should refer to *Developing Modes* in the developers manual.

Note: The standard modes and default bindings are summarized in the Quick Reference *Quick Reference*.

4.4.4 CMap

CMap Mode enables bindings that can adjust the color and intensity maps of an image in a Ginga image viewer.

Enter the mode by

- Space, then "y"

Exit the mode by

- Esc

Default bindings in mode

- Y : reset colormap to default (gray)

- r : restore color map (undo any color map inversion, rotation or stretch)

- I : invert current color map (preserving any rotation or stretch)

- up arrow : set previous colormap in list (preserves any rotation or stretch)

- down arrow : set next colormap in list (preserves any rotation or stretch)

- c : overlay a visible colormap on the image

- i : restore the default intensity map to "ramp", the default

- j, left arrow : set previous intensity map in list

- k, right arrow : set next intensity map in list

- scroll : choose color map from list

- Ctrl + scroll : choose intensity map from list

- left drag : rotate current color map

- right click : restore color map (same as "r")

4.4.5 Contrast

Contrast Mode enables bindings that can adjust the contrast of an image in a Ginga image viewer.

Enter the mode by

- Space, then "t"

Exit the mode by

- Esc

Default bindings in mode

- T : restore contrast to defaults

- left drag : adjust contrast * Interactive shift/stretch colormap (aka contrast and bias). * Moving left/right controls shift, up/down controls stretch. * Release button when satisfied with the contrast.

- right click : restore contrast to defaults

4.4.6 Cuts

Cuts Mode enables bindings that can adjust the low and high cut levels of an image in a Ginga image viewer.

Enter the mode by

- Space, then "s"

Exit the mode by

- Esc

Default bindings in mode

- l : set low cut level to the value of the pixel under the cursor
- h : set high cut level to the value of the pixel under the cursor
- S : set the the low and high cut levels to the min/max values in the image
- A : set the low and high cut levels to 0, 255; useful for standard RGB images, mostly
- a : perform an auto cut levels using the currently selected auto cuts algorithm and parameters
- b, up arrow : select the previous auto cuts algorithm in the list
- n, down arrow : select the next auto cuts algorithm in the list
- colon : toggle auto cuts for new images "on" or "off" in this viewer
- semicolon : set auto cuts for new images to "override" in this viewer
- scroll : adjust contrast by squeezing or stretching levels; one direction squeezes, the other stretches
- Ctrl + scroll : adjust micro contrast by squeezing or stretching levels; similar to scroll, but amount of stretch/squeeze is reduced
- Shift + scroll : change current auto cuts algorithm
- left drag : adjust levels by moving cursor; moving left/right adjusts high level, up/down adjusts low level
- Shift + left drag : adjust low level by moving cursor; moving left/right adjusts low level
- Ctrl + left drag : adjust high level by moving cursor; moving left/right adjusts high level
- right click : perform an auto levels (same as "a")

4.4.7 Dist

Dist Mode enables bindings that can adjust the color distribution of an image in a Ginga image viewer.

These algorithms are similar to "curves" type profiles: "linear", "log", "power", "sqrt", "squared", "asinh", "sinh", "histeq"

Enter the mode by

- Space, then "d"

Exit the mode by

- Esc

Default bindings in mode

- D : reset the color distribution algorithm to "linear"
- b, up arrow : select the previous distribution algorithm in the list
- n, down arrow : select the next distribution algorithm in the list
- scroll : select the color distribution algorithm by scrolling

4.4.8 Pan

Pan Mode enables bindings that can set the pan position (the position under the center pixel) and zoom level (scale) in a Ginga image viewer.

Enter the mode by

- Space, then "q"

Exit the mode by

- Esc

Default bindings in mode

- plus, equals : zoom in one zoom level
- minus, underscore : zoom out one zoom level
- 1-9,0 : zoom to level N (0 is 10)
- Shift + 1-9,0 : zoom to level -N (0 is -10)
- backquote : fit image to window size
- doublequote : toggle auto fit for new images "on" or "off" in this viewer
- singlequote : set auto fit for new images to "override" in this viewer
- p : pan to the position under the cursor
- c : pan to the center of the image
- z : save zoom level (scale)
- r : pan to cursor and zoom to saved scale level
- left/right/up/down arrow : pan left/right/up/down by a small percentage

- Shift + left/right/up/down arrow : pan left/right/up/down by a very small percentage

- pageup (pagedown) : pan up (down) by a large percentage of the screen

- home (end) : pan towards the top (bottom) of the image

- "?" : toggle auto center for new images "on" or "off" in this viewer

- "/" : set auto center for new images to "override" in this viewer

- scroll : zoom (scale) the image

- left drag : pan the view

- right drag : camera pan the view

4.4.9 FreePan

FreePan Mode enables bindings that can set the pan position (the center pixel) and zoom level (scale) in a Ginga image viewer.

It differs from Pan mode in the types of scrolling and panning controls.

Enter the mode by

- Space, then "w"

Exit the mode by

- Esc

Default bindings in mode

- middle drag : freely pan around the image

- left click : set pan under the cursor and zoom in

- right click : zoom out

- pan gesture : zoom in/out of the image

- Shift + pan gesture : zoom in/out of the image with origin set

4.4.10 Rotate

Rotate Mode enables bindings that can flip or swap the axes of an image, or rotate it, in a Ginga image viewer.

Enter the mode by

- Space, then "r"

Exit the mode by

- Esc

Default bindings in mode

- "[" : (toggle) flip the image in the X axis
- "{" : Restore the X axis
- "]" : (toggle) flip the image in the Y axis
- "}" : Restore the Y axis
- backslash : (toggle) swap the X and Y axes
- "|" : Restore the swapped axes to normal
- R : reset rotation to 0 deg (does not reset any flips or swaps)
- t : resets any flips or swaps
- period : rotate image incrementally by +90 deg
- comma : rotate image by incrementally -90 deg
- o : orient the image so that North points up and East points left
- O : orient the image so that North points up and East points right
- left drag : rotate the image interactively
- right click : reset the rotation to 0 deg (same as R)
- rotation gesture : rotate the image interactively

4.4.11 Naxis

Naxis Mode enables bindings that can move through the slices in an image stack in a Ginga image viewer.

Enter the mode by

- Space, then "n"

Exit the mode by

- Esc

Default bindings in mode

- scroll : select previous or next slice of current axis

- Ctrl + scroll : select previous or next axis as current axis

- left drag : select slice as a function of percentage of cursor/window width

- up/down arrow : select previous or next axis as current axis

4.5 Plugins

Ginga is written so that most of the functionality of the program is achieved through the use of plugins. This modular approach allows a large degree of flexibility and customization, as well as making overall design and maintenance of the program simpler.

Plugins are divided into two types: *global* and *local*. A global plugin has a single instance shared by all channels, while a local plugin creates a unique instance for each channel. If you switch channels, a global plugin will respond to the change by updating itself, while a local plugin will remain unchanged if the channel is switched, because its operation is specific to a given channel. (Ginga's concept of channels is discussed in *Channels*.)

This chapter describes the set of plugins that come with Ginga. Those interested in writing their own custom plugins should refer to *Writing a Global Plugin* or *Anatomy of a Local Ginga Plugin*.

4.5.1 Global plugins

Toolbar

`Toolbar` provides a set of convenience UI controls for common operations on viewers.

Plugin Type: Global

`Toolbar` is a global plugin. Only one instance can be opened.

Usage

Hovering over an icon on the toolbar should provide you with usage tool tip.

It is customizable using `~/.ginga/plugin_Toolbar.cfg`, where ~ is your HOME directory:

```
#
# Toolbar plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Toolbar.cfg"

# Accepted value: 'oneshot' or 'locked'
mode_type = 'oneshot'
```

Pan
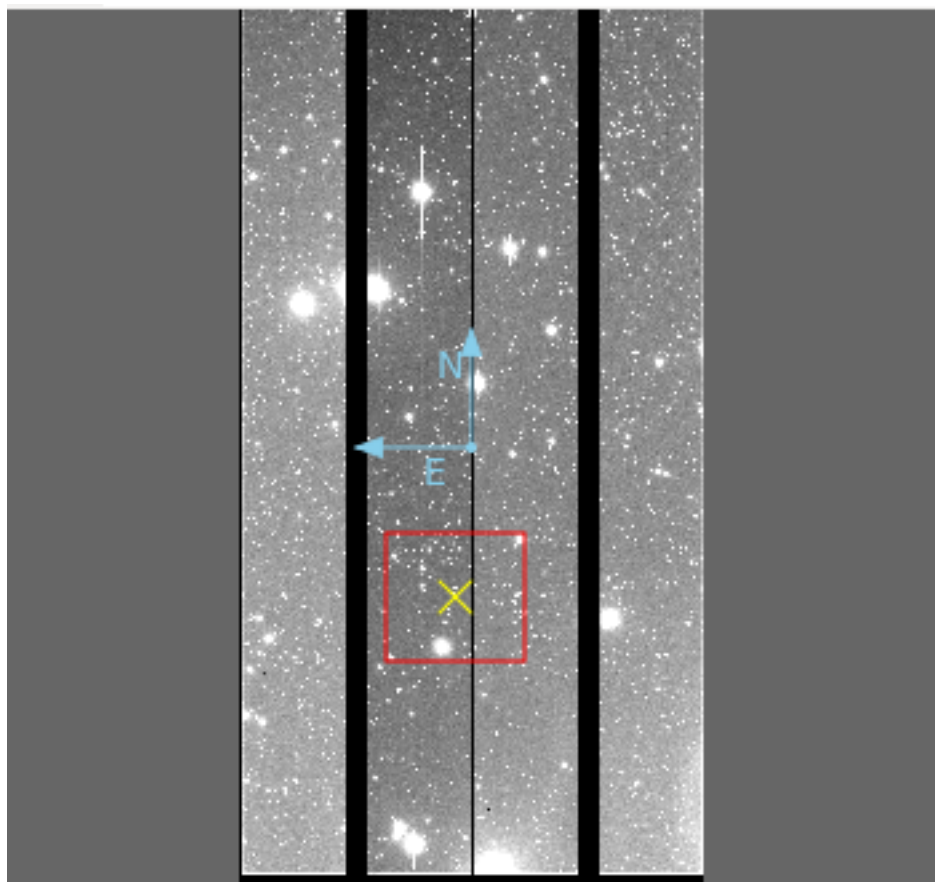
The Pan plugin provides a small panning image that gives an overall "birds-eye" view of the channel image that last had the focus. If the channel image is zoomed in 2X or greater, then the pan region is shown graphically in the Pan image by a rectangle.

Plugin Type: Global

Pan is a global plugin. Only one instance can be opened.

Usage

The channel image can be panned by clicking and/or dragging to place the rectangle. Using the right mouse button to drag a rectangle will force the channel image viewer to try to match the region (taking into account the differences in the aspect ratio between the drawn rectangle and the window dimensions). Scrolling in the Pan image will zoom the channel image.

The color/intensity map and cut levels of the Pan image are updated when they are changed in the corresponding channel image. The Pan image also displays the World Coordinate System (WCS) compass, if valid WCS metadata is present in the FITS HDU being viewed in the channel.

The Pan plugin usually appears as a sub-pane under the "Info" tab, next to the Info plugin.

This plugin is not usually configured to be closeable, but the user can make it so by setting the "closeable" setting to True in the configuration file–then Close and Help buttons will be added to the bottom of the UI.

It is customizable using ~/.ginga/plugin_Pan.cfg, where ~ is your HOME directory:

```
#
# Pan plugin preferences file
```

(continues on next page)

```
#
# Place this in file under ~/.ginga with the name "plugin_Pan.cfg"

# Share a canvas with the channel viewer
use_shared_canvas = False

# color of pan position marker
pan_position_color = 'yellow'

# color of pan rectangle
pan_rectangle_color = 'red'

# color of compass
compass_color = 'skyblue'

# rotate the pan image if the main image is rotated?
rotate_pan_image = True

# Add a close button to this plugin, so that it can be stopped
closeable = False
```

Info

Name: SUPA01118784[PRIMARY,1]
Object: M27
X: 504.773
Y: 1494.557
Value: 488
α: 19:59:08.977
δ: +22:32:17.112
Equinox: 2000.0
Dimensions: 2272x4273
Min: 137
Max: 65535

Zoom: 1/3.57x
Cut Low: 141
Cut High: 1086
Auto Levels   Cut Levels
Cut New: override
Zoom New: on
Center New: off

The `Info` plugin provides a pane of commonly useful metadata about the associated channel image. Common information includes some metadata header values, coordinates, dimensions of the image, minimum and maximum values, etc. As the cursor is moved around the image, the X, Y, Value, RA, and DEC values are updated to reflect the value under the cursor.

Plugin Type: Global

`Info` is a global plugin. Only one instance can be opened.

Usage

At the bottom of the `Info` interface the cut levels controls. Here the low and high cut levels are shown and can be adjusted. Pressing the "Auto Levels" button will recalculate cut levels based on the current auto cut levels algorithm and parameters defined in the channel preferences.

Below the "Auto Levels" button, the status of the settings for "Cut New", "Zoom New", and "Center New" are shown for the currently active channel. These indicate how new images that are added to the channel will be affected by auto cut levels, fitting to the window and panning to the center of the image.

The "Follow New" checkbox controls whether the viewer will automatically display new images added to the channel. The "Raise New" checkbox controls whether an image viewer window is raised when a new image is added. These two controls can be useful, for example, if an external program is adding images to the viewer, and you wish to prevent interruption of your work examining a particular image.

As a global plugin, `Info` responds to a change of focus to a new channel by displaying the metadata from the new channel. It typically appears under the "Synopsis" tab in the user interface.

This plugin is not usually configured to be closeable, but the user can make it so by setting the "closeable" setting to True in the configuration file–then Close and Help buttons will be added to the bottom of the UI.

Header

Keyword	Value
SIMPLE	True
BITPIX	16
NAXIS	2
NAXIS1	2272
NAXIS2	4273
EXTEND	False
BZERO	32768.0
BSCALE	1.0
BUNIT	ADU
BLANK	-32768
DATE-OBS	2009-08-22
UT	09:34:25.911
UT-STR	09:34:25.911
UT-END	09:35:55.010
HST	23:34:25.911
HST-STR	23:34:25.911
HST-END	23:35:55.010
LST	21:15:48.968
LST-STR	21:15:48.968
LST-END	21:17:18.311
MJD	55065.398914
TIMESYS	UTC
MJD-STR	55065.398914
MJD-END	55065.399945
ZD-STR	17.858
ZD-END	18.2
SECZ-STR	1.051
SECZ-END	1.053
AIRMASS	1.0526
AZIMUTH	282.679
ALTITUDE	72.142
PROP-ID	o99005
OBSERVER	Jeschke, Inagaki, Streeper,
FRAMEID	SUPA01118760
EXP-ID	SUPE01118760
DATASET	DS000
OBS-MOD	IMAG_N_VGW
OBS-ALOC	Observation
DATA-TYP	OBJECT
OBJECT	M27
RA	19:59:40.168

☐ Sortable

The `Header` plugin provides a listing of the metadata associated with the image.

Plugin Type: Global

`Header` is a global plugin. Only one instance can be opened.

Usage

The `Header` plugin shows the FITS keyword metadata from the image. Initially only the Primary HDU metadata is shown. However, in conjunction with the `MultiDim` plugin, the metadata for other HDUs will be shown. See `MultiDim` for details.

If the "Sortable" checkbox has been checked in the lower left of the UI, then clicking on a column header will sort the table by values in that column, which may be useful for quickly locating a particular keyword.

If the "Include primary header" checkbox toggles the inclusion of the primary HDU keywords or not. This option may be disabled if the image was created with an option not to save the primary header.

It is customizable using `~/.ginga/plugin_Header.cfg`, where ~ is your HOME directory:

```
#
# Header plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Header.cfg"

# Sort header
sortable = True

# Include primary header in table output
include_primary_header = False

# If True, color every other row in alternating shades to improve
# readability of long tables
color_alternate_rows = True

# Maximum number of rows that will turn off auto column resizing (for speed)
max_rows_for_col_resize = 5000
```

Zoom
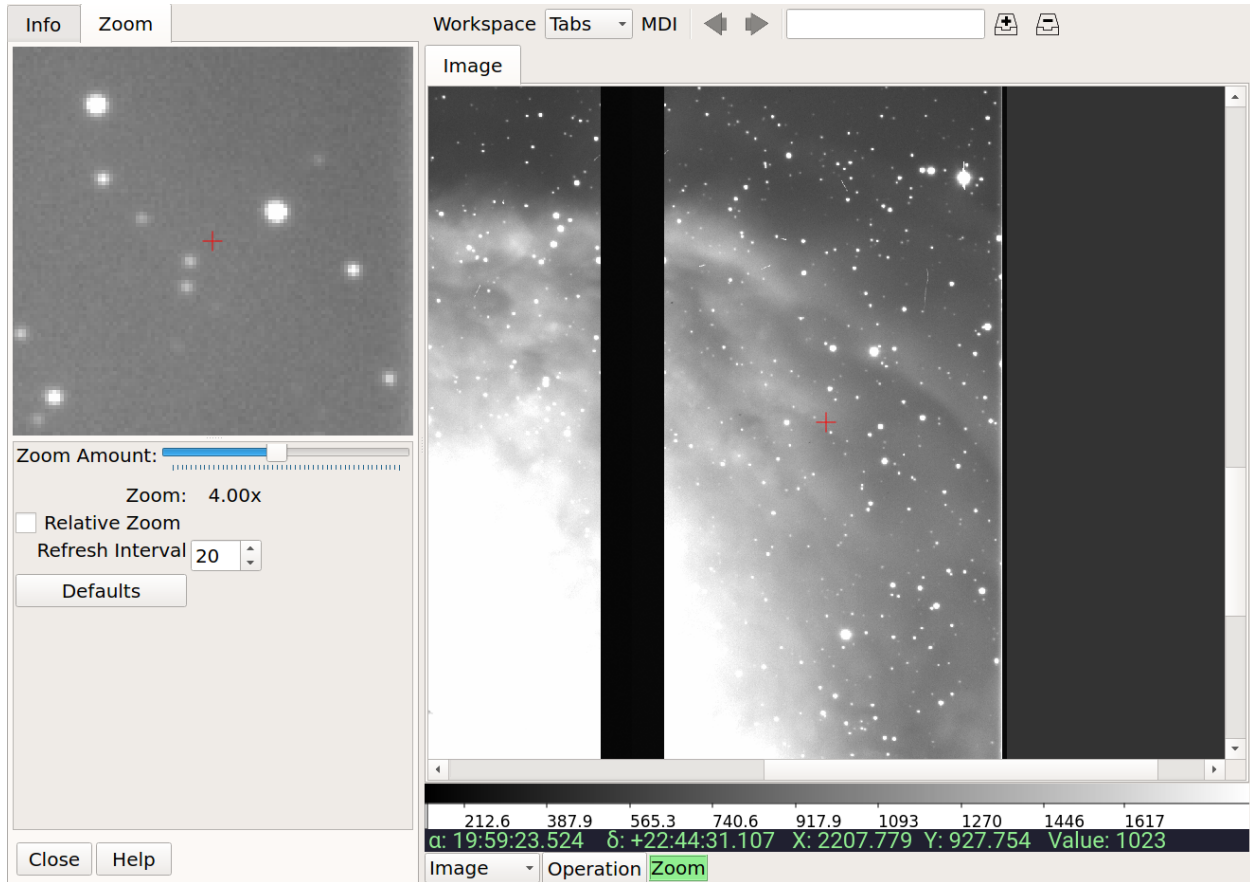
The Zoom plugin shows an enlarged image of a cutout region centered under the cursor position in the associated channel image. As the cursor is moved around the image, the zoom image updates to allow close inspection of the pixels or precise control in conjunction with other plugin operations.

Plugin Type: Global

Zoom is a global plugin. Only one instance can be opened.

Usage

The magnification of the zoom window can be changed by adjusting the "Zoom Amount" slider.

Two modes of operation are possible – absolute and relative zoom:

- In absolute mode, the zoom amount controls exactly the zoom level shown in the cutout; For example, the channel image may be zoomed into 10X, but the zoom image will only show a 3X image if the zoom amount is set to 3X.

- In relative mode, the zoom amount setting is interpreted as relative to the zoom setting of the channel image. If the zoom amount is set to 3X and the channel image is zoomed to 10X then the zoom image shown will be 13X (10X + 3X). Note that the zoom amount setting can be < 1, so a setting of 1/3X with a 3X zoom in the channel image will produce a 1X zoom image.

The "Refresh Interval" setting controls how quickly the Zoom plugin responds to the movement of the cursor in updating the zoom image. The value is specified in milliseconds.

Tip: Usually setting a small refresh interval *improves* the overall responsiveness of the zoom image, and the default value of 20 is a reasonable one. You can experiment with the value if the zoom image seems too jerky or out of sync

---

with the mouse movement in the channel image window.

The "Defaults" button restores the default settings of the controls.

It is customizable using ~/.ginga/plugin_Zoom.cfg, where ~ is your HOME directory:

```
#
# Zoom plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Zoom.cfg"

# default zoom level
zoom_amount = 3

# refresh interval (sec)
# NOTE: usually a small delay speeds things up
refresh_interval = 0.02
```

Thumbs
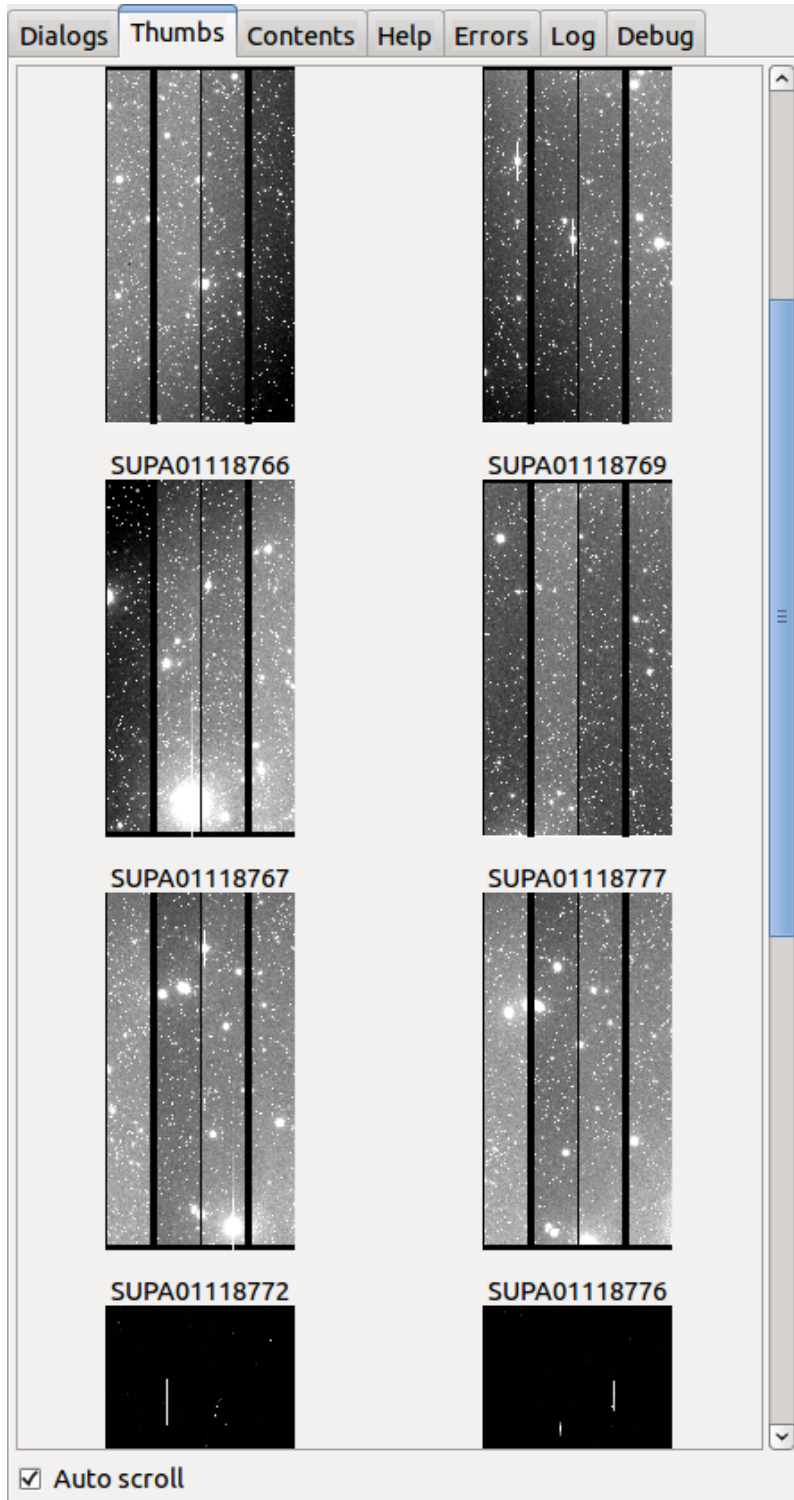
The Thumbs plugin provides a thumbnail index of all images viewed since the program was started.

Plugin Type: Global

Thumbs is a global plugin. Only one instance can be opened.

Usage

By default, `Thumbs` appear in cronological viewing history, with the newest images at the bottom and the oldest at the top. The sorting can be made alphanumeric by a setting in the "plugin_Thumbs.cfg" configuration file.

Clicking on a thumbnail navigates you directly to that image in the associated channel. Hovering the cursor over a thumbnail will show a tool tip that contains a couple of useful pieces of metadata from the image.

The "Auto Scroll" checkbox, if checked, will cause the `Thumbs` pan to scroll to the active image.

This plugin is not usually configured to be closeable, but the user can make it so by setting the "closeable" setting to True in the configuration file–then Close and Help buttons will be added to the bottom of the UI.

Excluding images from Thumbs

Note: This also controls the behavior of `Contents`.

Although the default behavior is for every image that is loaded into the reference viewer to show up in `Thumbs`, there may be cases where this is undesirable (e.g., when there are many images being loaded at a periodic rate by some automated process). In such cases there are two mechanisms for suppressing certain images from showing up in `Thumbs`:

- Assigning the "genthumb" setting to False in a channel's settings (for example from the `Preferences` plugin, under the "General" settings) will exclude the channel itself and any of its images.

- Setting the "nothumb" keyword in the metadata of an image wrapper (not the FITS header, but by e.g., `image.set(nothumb=True)`) will exclude that particular image from `Thumbs`, even if the "genthumb" setting is True for that channel.

It is customizable using `~/.ginga/plugin_Thumbs.cfg`, where ~ is your HOME directory:

```
#
# Thumbs plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Thumbs.cfg"

# If you revisit the same directories frequently
# caching thumbs saves a lot of time when they need to be regenerated
cache_thumbs = False

# cache location-- "local" puts them in a .thumbs subfolder, otherwise
# they are cached in ~/.ginga/thumbs
cache_location = 'local'

# Scroll the pane automatically when new thumbnails arrive
auto_scroll = True

# Keywords to extract and show if we mouse over the thumbnail
tt_keywords = ['OBJECT', 'FRAMEID', 'UT', 'DATE-OBS']

# Mandatory unique image identifier in tooltip
mouseover_name_key = 'NAME'

# How many seconds to wait after an image is altered to begin trying
# to rebuild a matching thumb.  Usually a few seconds is good in case
# there is ongoing adjustment of the image
rebuild_wait = 0.5
```

(continues on next page)

```
# Max length of thumb on the long side
thumb_length = 180

# Separation between thumbs in pixels
thumb_hsep = 15
thumb_vsep = 15

# Sort the thumbs alphabetically: 'alpha' or None
sort_order = None

# Thumbnail label length in num of characters (None = no limit)
label_length = 25

# Cut off long label ('left', 'right', or None)
label_cutoff = 'right'

# Option to highlight images that are displayed in channels.
# If set to True this option will only highlight the image that is in the
# channel with the keyboard focus
highlight_tracks_keyboard_focus = True

# Highlighted label colors
label_bg_color = 'lightgreen'
label_font_color = 'white'

label_font_size = 10

# Load visible thumbs in the background to replace placeholder icons
autoload_visible_thumbs = True

# Length of time to wait after scrolling to begin autoloading
autoload_interval = 1.0

# list of attributes to transfer from the channel viewer to the
# thumbnail generator if the channel has an image in it
transfer_attrs = ['transforms', 'cutlevels', 'rgbmap']

# Add a close button to this plugin, so that it can be stopped
closeable = False
```

Contents

The `Contents` plugin provides a table of contents-like interface for all the images viewed since the program was started. Unlike `Thumbs`, `Contents` is sorted by channel. The contents also shows some configurable metadata from the image.

Plugin Type: Global

`Contents` is a global plugin. Only one instance can be opened.

Usage

Click on a column heading to sort the table by that column; Click again to sort the other way.

Note: The columns and their values are drawn from the FITS header, if applicable. This can be customized by setting the "columns" parameter in the "plugin_Contents.cfg" settings file.

The active image in the currently focused channel will normally be highlighted. Double-click on an image will force that image to be shown in the associated channel. Single-click on any image to activate the buttons at the bottom of the UI:

- "Display": Make the image the active image.

- "Move": Move the image to another channel.

- "Copy": Copy the image to another channel.

- "Remove": Remove the image from the channel.

If "Move" or "Copy" is done on an image that has been modified in Ginga (which would have an entry under
`ChangeHistory`, if used), the modification history will be retained as well. Removing an image from a channel
destroys any unsaved changes.

This plugin is not usually configured to be closeable, but the user can make it so by setting the "closeable" setting to
True in the configuration file–then Close and Help buttons will be added to the bottom of the UI.

Excluding images from Contents

Note: This also controls the behavior of `Thumbs`.

Although the default behavior is for every image that is loaded into the reference viewer to show up in `Contents`,
there may be cases where this is undesirable (e.g., when there are many images being loaded at a periodic rate by
some automated process). In such cases there are two mechanisms for suppressing certain images from showing up in
`Contents`:

- Assigning the "genthumb" setting to False in a channel's settings (for example from the `Preferences` plugin,
  under the "General" settings) will exclude the channel itself and any of its images.

- Setting the "nothumb" keyword in the metadata of an image wrapper (not the FITS header, but by e.g., `image.`
  `set(nothumb=True)`) will exclude that particular image from `Contents`, even if the "genthumb" setting is True
  for that channel.

It is customizable using `~/.ginga/plugin_Contents.cfg`, where ~ is your HOME directory:

```
#
# Contents plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Contents.cfg"

# columns to show from metadata -- NAME and MODIFIED recommended
# format: [(col header, keyword1), ... ]
columns = [ ('Name', 'NAME'), ('Object', 'OBJECT'), ('Filter', 'FILTER01'), ('Date',
↪'DATE-OBS'), ('Time UT', 'UT'), ('Modified', 'MODIFIED')]

# If set to True, will always expand the tree in Contents when new entries are added
always_expand = True

# Option to highlight images that are displayed in channels.
# If set to True this option will only highlight the image that is in the
# channel with the keyboard focus
highlight_tracks_keyboard_focus = False

# If True, color every other row in alternating shades to improve
# readability of long tables
color_alternate_rows = True

# Highlighted row colors (in addition to bold text)
row_font_color = 'green'

# Maximum number of rows that will turn off auto column resizing (for speed)
max_rows_for_col_resize = 100
```

```
# Add a close button to this plugin, so that it can be stopped
closeable = False
```

Colorbar
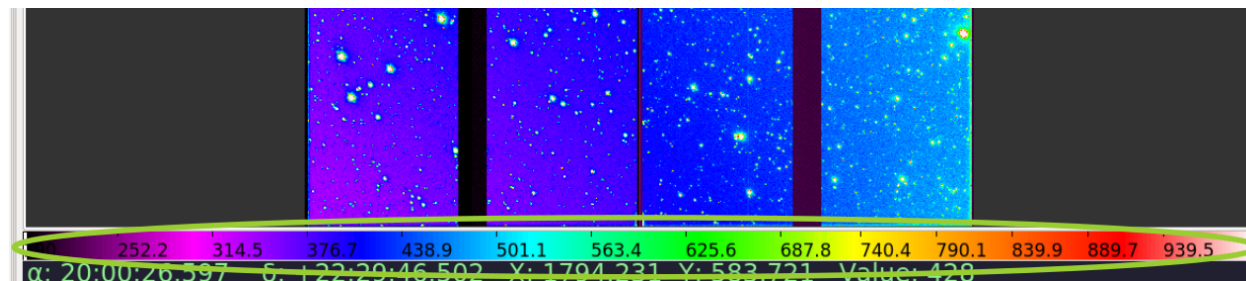
The `Colorbar` plugin shows a colorbar indicating the colormap applied to the image and showing the example values along the range.

Plugin Type: Global

`Colorbar` is a global plugin. Only one instance can be opened.

Usage

Clicking and dragging in the `Colorbar` window will shift the colormap left or right. Scrolling will stretch or shrink the colormap at the cursor position. Right-clicking will restore the colormap from any shift or stretch.

If the focus shifts to another channel, the colorbar will be updated to reflect that channel's colormap and value information.

It is customizable using `~/.ginga/plugin_Colorbar.cfg`, where ~ is your HOME directory:

```
#
# Colorbar plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Colorbar.cfg"

# Set colorbar height, if default is too big or little
cbar_height = 36

# size of font used in the color bar
fontsize = 10
```

Cursor
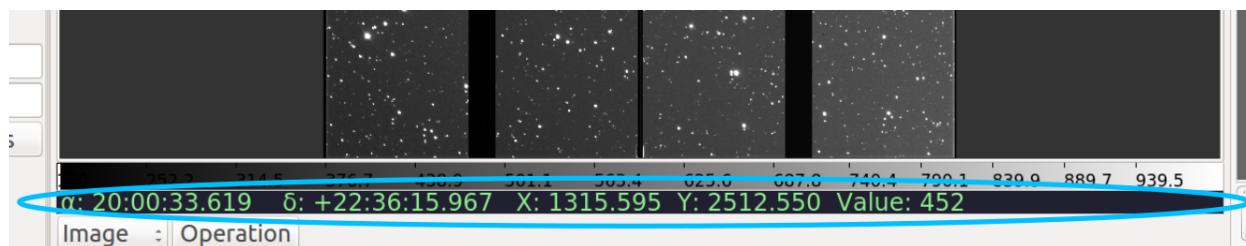
The `Cursor` plugin displays a summary line of text that changes as the user moves the cursor around an image. In the

---

standard reference viewer configuration, it appears as a line containing green text just below the `Colorbar` plugin.

Plugin Type: Global

`Cursor` is a global plugin. Only one instance can be opened.

Usage

`Cursor` simply tracks the cursor as it moves around an image and displays information about the pixel coordinates, WCS coordinates (if available) and the value of the pixel under the cursor.

There is no associated configuration GUI.

Note: Pixel coordinates are affected by the general setting "pixel_coords_offset" which can be set in the "general.cfg" configuration file for ginga. The default is value for this setting is 1.0, which means pixel coordinates are reported from an origin of 1, as per the FITS standard.

It is customizable using `~/.ginga/plugin_Cursor.cfg`, where ~ is your HOME directory:

```
#
# Cursor plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Cursor.cfg"

# Set to a particular font if you like one
font_name = 'fixed'

# Set to a point size if you have a preference
font_size = None
```

Operations

This plugin defines the GUI for managing local plugins, a.k.a., "operations".

Plugin Type: Global

`Operations` is a global plugin. Only one instance can be opened.

Usage

The `Operations` plugin acts as a visual interface to the reference viewer plugin manager. With this plugin, you can change the active channel, start, stop, or unfocus a local plugin on a channel, and see which local plugins are running.

Note: By replacing or subclassing this plugin, you can customize the way the reference viewer starts and manages operations.

It is customizable using `~/.ginga/plugin_Operations.cfg`, where ~ is your HOME directory:

```
#
# Operations plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Operations.cfg"

# Show the change channel drop-down control
show_channel_control = True
```

```
# Color to highlight focused plugins in the operations control tray
focuscolor = "lightgreen"

# Change to False to use combobox+button launch approach
use_popup_menu = True
```

WBrowser

Web browser plugin for Ginga.

Plugin Type: Global

`WBrowser` is a global plugin. Only one instance can be opened.

Usage

This global plugin is used to browse help pages for Ginga.

When a "Help" button is pressed from a plugin (e.g., `Pick`), Ginga will attempt to download an existing documentation build from *ReadTheDocs* for the matching version. If successful, plugin documentation from that download is displayed. If not successful or deliberately disabled in "plugin_WBrowser.cfg", Ginga will render the plugin's docstring locally.

It is customizable using `~/.ginga/plugin_WBrowser.cfg`, where ~ is your HOME directory:

```
#
# WBrowser plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_WBrowser.cfg"

# Set to True to force offline doc browsing, otherwise download from RTD
offline_doc_only = False
```

FBrowser (Open File)

This brings up *FBrowser* hybrid plugin as a global plugin.

ColorMapPicker
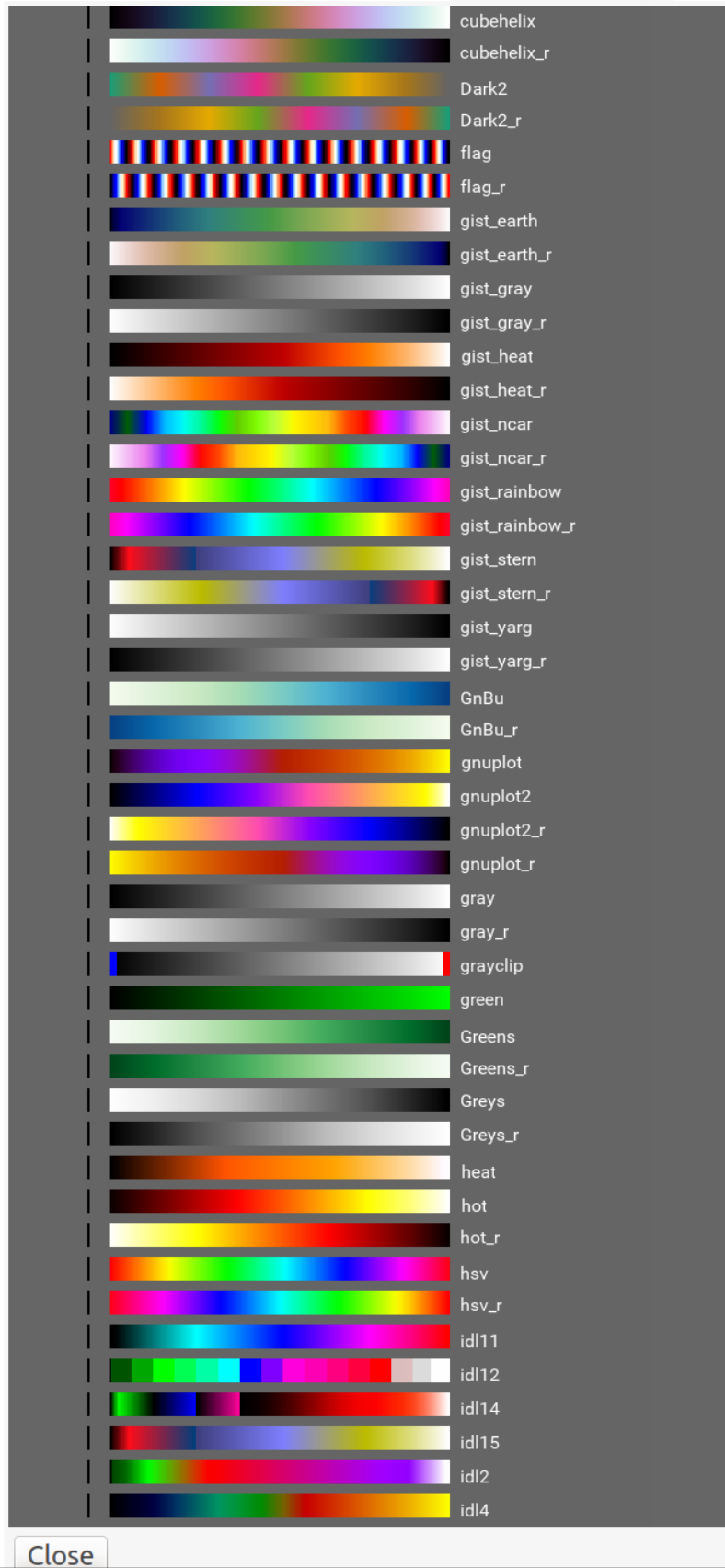
The `ColorMapPicker` plugin is used to graphically browse and select a colormap for a channel image viewer.

Plugin Type: Global or Local

`ColorMapPicker` is a hybrid global/local plugin, which means it can be invoked in either fashion. If invoked as a local plugin then it is associated with a channel, and an instance can be opened for each channel. It can also be opened as a global plugin.

Usage

Operation of the plugin is very simple: the colormaps are displayed in the form of colorbars and labels in the main view pane of the plugin. Click on any one of the bars to set the colormap of the associated channel (if invoked as a local plugin) or the currently active channel (if invoked as a global plugin).

You can scroll vertically or use the scroll bars to move through the colorbar samples.

Note: When the plugin starts for the first time, it will generate a bitmap RGB image of colorbars and labels corresponding to all the available colormaps. This can take a few seconds depending on the number of colormaps installed.

Colormaps are shown with the "ramp" intensity map applied.

It is customizable using `~/.ginga/plugin_ColorMapPicker.cfg`, where ~ is your HOME directory:

```
#
# ColorMapPicker plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_ColorMapPicker.cfg"

cbar_ht = 20
cbar_wd = 300
cbar_sep = 10
cbar_pan_accel = 1.0
```

Errors

The `Errors` plugin reports error messages on the viewer.

Plugin Type: Global

`Errors` is a global plugin. Only one instance can be opened.

Usage

When an error occurs in Ginga, its message may be reported here.

This plugin is not usually configured to be closeable, but the user can make it so by setting the "closeable" setting to True in the configuration file–then Close and Help buttons will be added to the bottom of the UI.

RC

The RC plugin implements a remote control interface for the Ginga viewer.

Plugin Type: Global

RC is a global plugin. Only one instance can be opened.

Usage

The RC (Remote Control) plugin provides a way to control Ginga remotely through the use of an XML-RPC interface. Start the plugin from the "Plugins" menu (invoke "Start RC") or launch ginga with the `--modules=RC` command line option to start it automatically.

By default, the plugin starts up with server running on port 9000 bound to the localhost interface – this allows connections only from the local host. If you want to change this, set the host and port in the "Set Addr" control and press `Enter` – you should see the address update in the "Addr:" display field.

Please note that the host part (before the colon) does not indicate *which* host you want to allow access from, but to which interface to bind. If you want to allow any host to connect, leave it blank (but include the colon and port number) to allow the server to bind on all interfaces. Press "Restart" to then restart the server at the new address.

Once the plugin is started, you can use the `ggrc` script (included when `ginga` is installed) to control Ginga. Take a look at the script if you want to see how to write your own programmatic interface.

Show example usage:

```
$ ggrc help
```

Show help for a specific Ginga method:

```
$ ggrc help ginga <method>
```

Show help for a specific channel method:

```
$ ggrc help channel <chname> <method>
```

Ginga (viewer shell) methods can be called like this:

```
$ ggrc ginga <method> <arg1> <arg2> ...
```

Per-channel methods can be called like this:

```
$ ggrc channel <chname> <method> <arg1> <arg2> ...
```

Calls can be made from a remote host by adding the options:

```
--host=<hostname> --port=9000
```

(In the plugin GUI, be sure to remove the "localhost" prefix from the "addr", but leave the colon and port.)

Examples

Create a new channel:

```
$ ggrc ginga add_channel FOO
```

Load a file:

```
$ ggrc ginga load_file /home/eric/testdata/SPCAM/SUPA01118797.fits
```

Load a file into a specific channel:

```
$ ggrc ginga load_file /home/eric/testdata/SPCAM/SUPA01118797.fits FOO
```

Cut levels:

```
$ ggrc channel FOO cut_levels 163 1300
```

Auto cut levels:

```
$ ggrc channel FOO auto_levels
```

Zoom to a specific level:

```
$ ggrc -- channel FOO zoom_to -7
```

(Note the use of `--` to allow us to pass a parameter beginning with `-`.)

Zoom to fit:

```
$ ggrc channel FOO zoom_fit
```

Transform (arguments are a boolean triplet: `flipx flipy swapxy`):

```
$ ggrc channel FOO transform 1 0 1
```

Rotate:

```
$ ggrc channel FOO rotate 37.5
```

Change colormap:

```
$ ggrc channel FOO set_color_map rainbow3
```

Change color distribution algorithm:

```
$ ggrc channel FOO set_color_algorithm log
```

Change intensity map:

```
$ ggrc channel FOO set_intensity_map neg
```

In some cases, you may need to resort to shell escapes to be able to pass certain characters to Ginga. For example, a leading dash character is usually interpreted as a program option. In order to pass a signed integer, you may need to do something like:

```
$ ggrc -- channel FOO zoom -7
```

Interfacing from within Python

It is also possible to control Ginga in RC mode from within Python. The following describes some of the functionality.

Connecting

First, launch Ginga and start the RC plugin. This can be done from the command line:

```
ginga --modules=RC
```

From within Python, connect with a `RemoteClient` object as follows:

```python
from ginga.util import grc
host = 'localhost'
port = 9000
viewer = grc.RemoteClient(host, port)
```

This viewer object is now linked to the Ginga using RC.

Load an Image

You can load an image from memory in a channel of your choosing. First, connect to a channel:

```
ch = viewer.channel('Image')
```

Then, load a Numpy image (i.e., any 2D `ndarray`):

```python
import numpy as np
img = np.random.rand(500, 500) * 10000.0
ch.load_np('Image_Name', img, 'fits', {})
```

The image will display in Ginga and can be manipulated as usual.

Overlay a Canvas Object

It is possible to add objects to the canvas in a given channel. First, connect:

```
canvas = viewer.canvas('Image')
```

This connects to the channel named "Image". You can clear the objects drawn in the canvas:

```
canvas.clear()
```

You can also add any basic canvas object. The key issue to keep in mind is that the objects input must pass through the XMLRC protocol. This means simple data types (`float`, `int`, `list`, or `str`); No arrays. Here is an example to plot a line through a series of points defined by two Numpy arrays:

```python
x = np.arange(100)
y = np.sqrt(x)
points = list(zip(x.tolist(), y.tolist()))
canvas.add('path', points, color='red')
```

This will draw a red line on the image.

WCSMatch
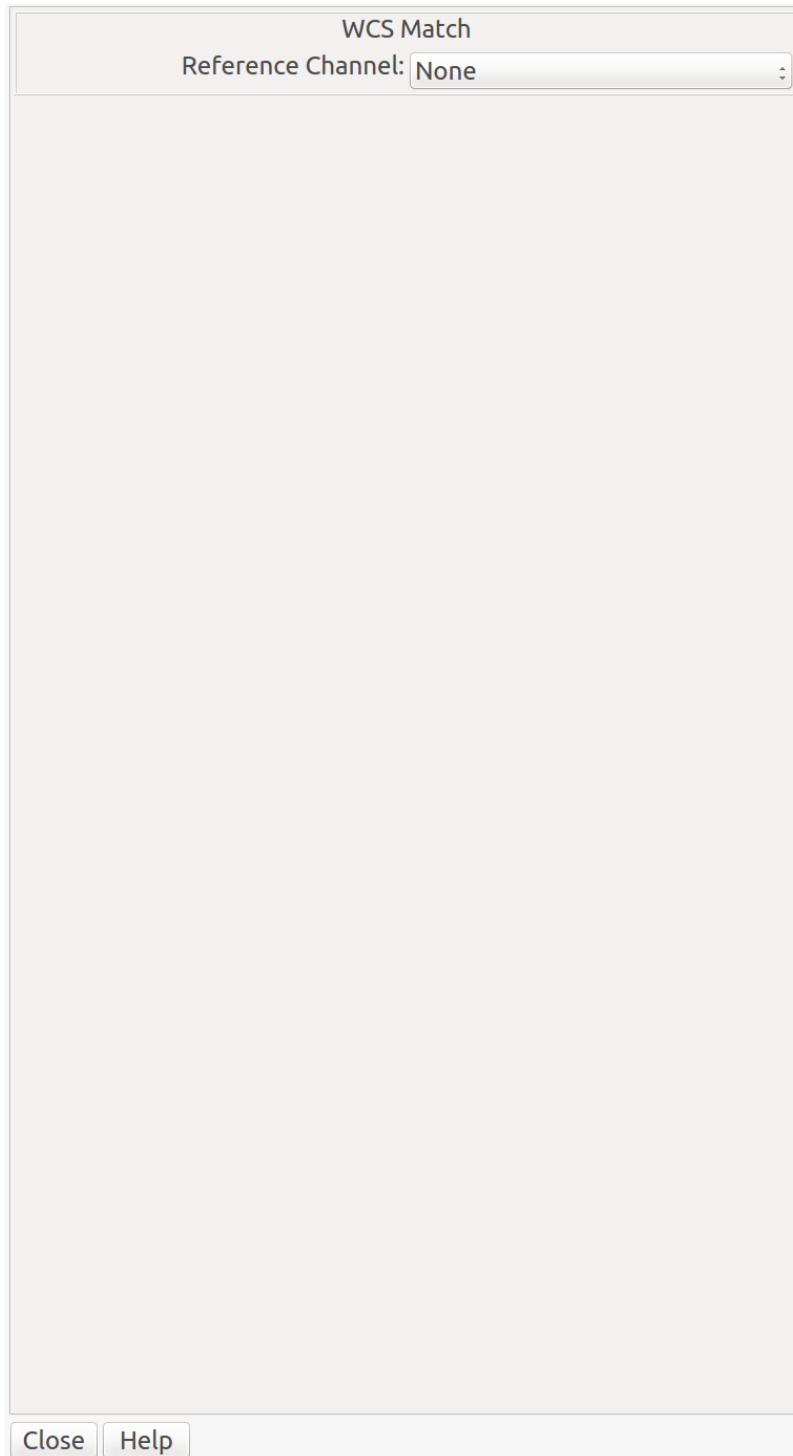
WCSMatch is a global
plugin for the Ginga image viewer that allows you to roughly align images with different scales and orientations using
the images' World Coordinate System (WCS) for viewing purposes.

Plugin Type: Global

WCSMatch is a global plugin. Only one instance can be opened.

Usage

To use, simply start the plugin, and from the plugin GUI select a channel from the drop-down menu labeled "Reference Channel". The image contained in that channel will be used as a reference for synchronizing the images in the other channels.

The channels will be synchronized in viewing (pan, scale (zoom), transforms (flips) and rotation. The checkboxes "Match Pan", "Match Scale", "Match Transforms" and "Match Rotation" can be checked or not to control which attributes are synchronized between channels.

To completely "unlock" the synchronization, simply select "None" from the "Reference Channel" drop-down menu.

Currently, there is no way to limit the channels that are affected by the plugin.

ChangeHistory
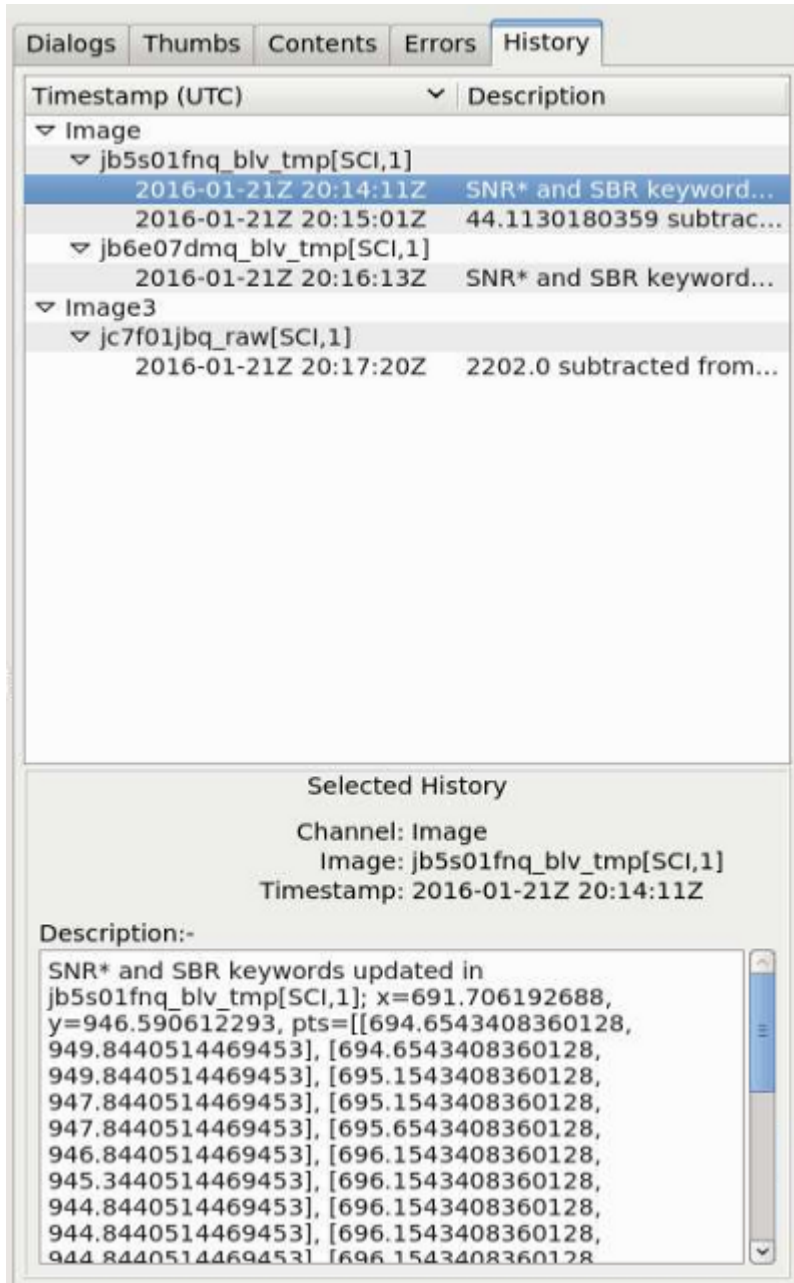
Keep track of buffer change history.

Plugin Type: Global

ChangeHistory is a global plugin. Only one instance can be opened.

This plugin is used to log any changes to data buffer. For example, a change log would appear here if a new image is added to a mosaic via the Mosaic plugin. Like Contents, the log is sorted by channel, and then by image name.

Usage

History should stay no matter what channel or image is active. New history can be added, but old history cannot be deleted, unless the image/channel itself is deleted.

The redo() method picks up an 'add-image-info' event and displays related metadata here. The metadata is

obtained as follows:

```
channel = self.fv.get_channel_info(chname)
iminfo = channel.get_image_info(imname)
timestamp = iminfo.time_modified
description = iminfo.reason_modified  # Optional
```

Both `'time_modified'` and `'reason_modified'` have to be explicitly set by the calling plugin in the same method that issues the `'add-image-info'` callback, like this:

```
# This changes the data buffer
image.set_data(new_data, ...)
# Add description for ChangeHistory
info = dict(time_modified=datetime.utcnow(),
           reason_modified='Data has changed')
self.fv.update_image_info(image, info)
```

It is customizable using ~/.ginga/plugin_ChangeHistory.cfg, where ~ is your HOME directory:

```
#
# ChangeHistory plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_ChangeHistory.cfg"

# If set to True, will always expand the tree in ChangeHistory when
# new entries are added
always_expand = True

# If set to True, rows will have alternate colors
color_alternate_rows = True

# Timestamp column width
ts_colwidth = 250
```

SAMP Control
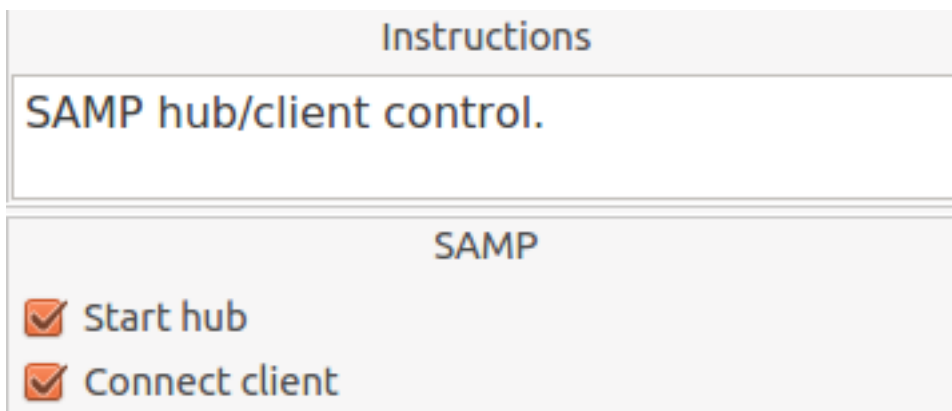
The SAMP plugin implements a SAMP interface for the Ginga reference viewer.

Note: To run this plugin, you need to install `astropy` that has the `samp` module.

Plugin Type: Global

`SAMP` is a global plugin. Only one instance can be opened.

Usage

Ginga includes a plugin for enabling SAMP (Simple Applications Messaging Protocol) support. With SAMP support, Ginga can be controlled and interoperate with other astronomical desktop applications.

The `SAMP` module is not started by default. To start it when Ginga starts, specify the command line option:

```
--modules=SAMP
```

Otherwise, start it using "Start SAMP" from the "Plugins" menu.

Currently, SAMP support is limited to `image.load.fits` messages, meaning that Ginga will load a FITS file if it receives one of these messages.

Ginga's `SAMP` plugin uses the `astropy.samp` module, so you will need to have `astropy` installed to use the plugin. By default, Ginga's `SAMP` plugin will attempt to start a SAMP hub if one is not found running.

It is customizable using `~/.ginga/plugin_SAMP.cfg`, where ~ is your HOME directory:

```
#
# SAMP plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_SAMP.cfg"

SAMP_channel = 'Image'

# Default location is set by Ginga's viewer.
#cache_location = '/my/cache/path/'

default_connect = True

start_hub = True
```

Log

```
Dialogs  Thumbs  Contents  Errors  Log
2017-04-10 17:39:32,341 | D | ImageView.py:1142 (redraw_now) | widget 'channel:Image' redraw (v
2017-04-10 17:39:32,341 | D | ImageView.py:1439 (_calc_bg_dimensions) | approx area covered is (
2017-04-10 17:39:32,341 | D | ImageView.py:1508 (apply_transforms) | reshape time 0.000 sec
2017-04-10 17:39:32,341 | D | ImageView.py:1527 (apply_transforms) | rotate time 0.000 sec, total
2017-04-10 17:39:32,341 | D | ImageView.py:1536 (apply_transforms) | ctr=169,13 off=180,180 ds
2017-04-10 17:39:32,341 | D | ImageView.py:1540 (apply_transforms) | win=339,27 coverage=349
2017-04-10 17:39:32,342 | D | ImageView.py:1390 (get_rgb_object) | times: total=0.0005
2017-04-10 17:39:32,342 | D | ImageViewQt.py:172 (render_image) | redraw pixmap=<PyQt5.QtGu
2017-04-10 17:39:32,342 | D | ImageViewQt.py:175 (render_image) | drawing to pixmap
2017-04-10 17:39:32,342 | D | ImageViewQt.py:148 (_render_offscreen) | data shape is 360x360x4
2017-04-10 17:39:32,348 | D | ImageView.py:1197 (check_cursor_location) | cursor location change
2017-04-10 17:39:32,348 | D | ImageViewQt.py:253 (update_image) | updating window from pixmap
2017-04-10 17:39:32,348 | D | ImageView.py:1142 (redraw_now) | widget 'colorbar' redraw (whence
2017-04-10 17:39:32,370 | D | wcsmod.py:778 (pixtocoords) | ra, dec = 300.023517, 22.760580
2017-04-10 17:39:32,379 | D | ImageView.py:1439 (_calc_bg_dimensions) | approx area covered is (
2017-04-10 17:39:32,381 | D | AutoCuts.py:73 (cut_levels) | loval=148.00 hival=1299.31
2017-04-10 17:39:32,382 | D | image.py:453 (draw_image) | shape of index is (434, 231)
2017-04-10 17:39:32,387 | D | ImageView.py:1508 (apply_transforms) | reshape time 0.000 sec
2017-04-10 17:39:32,388 | D | ImageView.py:1527 (apply_transforms) | rotate time 0.000 sec, total
2017-04-10 17:39:32,388 | D | ImageView.py:1536 (apply_transforms) | ctr=168,217 off=285,285 d
2017-04-10 17:39:32,388 | D | ImageView.py:1540 (apply_transforms) | win=337,435 coverage=45
2017-04-10 17:39:32,388 | D | ImageView.py:1390 (get_rgb_object) | times: total=0.0089
2017-04-10 17:39:32,388 | D | ImageViewQt.py:172 (render_image) | redraw pixmap=<PyQt5.QtGu
2017-04-10 17:39:32,388 | D | ImageViewQt.py:175 (render_image) | drawing to pixmap
2017-04-10 17:39:32,388 | D | ImageViewQt.py:148 (_render_offscreen) | data shape is 570x570x4
2017-04-10 17:39:32,390 | D | ImageViewQt.py:253 (update_image) | updating window from pixmap
2017-04-10 17:39:32,391 | D | ImageView.py:1142 (redraw_now) | widget 'panimage' redraw (when
2017-04-10 17:39:40,981 | I | Log.py:113 (set_loglevel_cb) | GUI log level changed to 'Info'
2017-04-10 17:39:42,261 | I | Log.py:113 (set_loglevel_cb) | GUI log level changed to 'Debug'
2017-04-10 17:39:43,660 | I | Log.py:113 (set_loglevel_cb) | GUI log level changed to 'Info'

Info                                              :  1000
☑ Auto scroll                                        Clear
Close  Help
```

See the logging output of the reference viewer.

Plugin Type: Global

Log is a global plugin. Only one instance can be opened.

Usage

The Log plugin builds a UI that includes a large scrolling text widget showing the active output of the logger. The latest output shows up at the bottom. This can be useful for troubleshooting problems.

There are four controls:

- The combo box on the lower left allows you to choose the level of logging desired. The four levels, in order of verbosity are: "debug", "info", "warn", and "error".

- The box with the number on the lower right allows you to set how many lines of input to keep in the display buffer (e.g., keep only the last 1000 lines).

- The checkbox "Auto scroll", if checked, will cause the large text widget to scroll to the end as new log messages are added. Uncheck this if you want to peruse the older messages and study them.

- The "Clear" button is used to clear the text widget, so that only new logging shows up.

Command

This plugin provides a command line interface to the reference viewer.

Note: The command line is for use *within* the plugin UI. If you are looking for a *remote* command line interface, please see the RC plugin.

Plugin Type: Global

`Command` is a global plugin. Only one instance can be opened.

Usage

Get a list of commands and parameters:

```
g> help
```

Execute a shell command:

```
g> !cmd arg arg ...
```

Notes

An especially powerful tool is to use the `reload_local` and `reload_global` commands to reload a plugin when you are developing that plugin. This avoids having to restart the reference viewer and laboriously reload data, etc. Simply close the plugin, execute the appropriate "reload" command (see the help!) and then start the plugin again.

Note: If you have modifed modules *other* than the plugin itself, these will not be reloaded by these commands.

SaveImage (Save File)
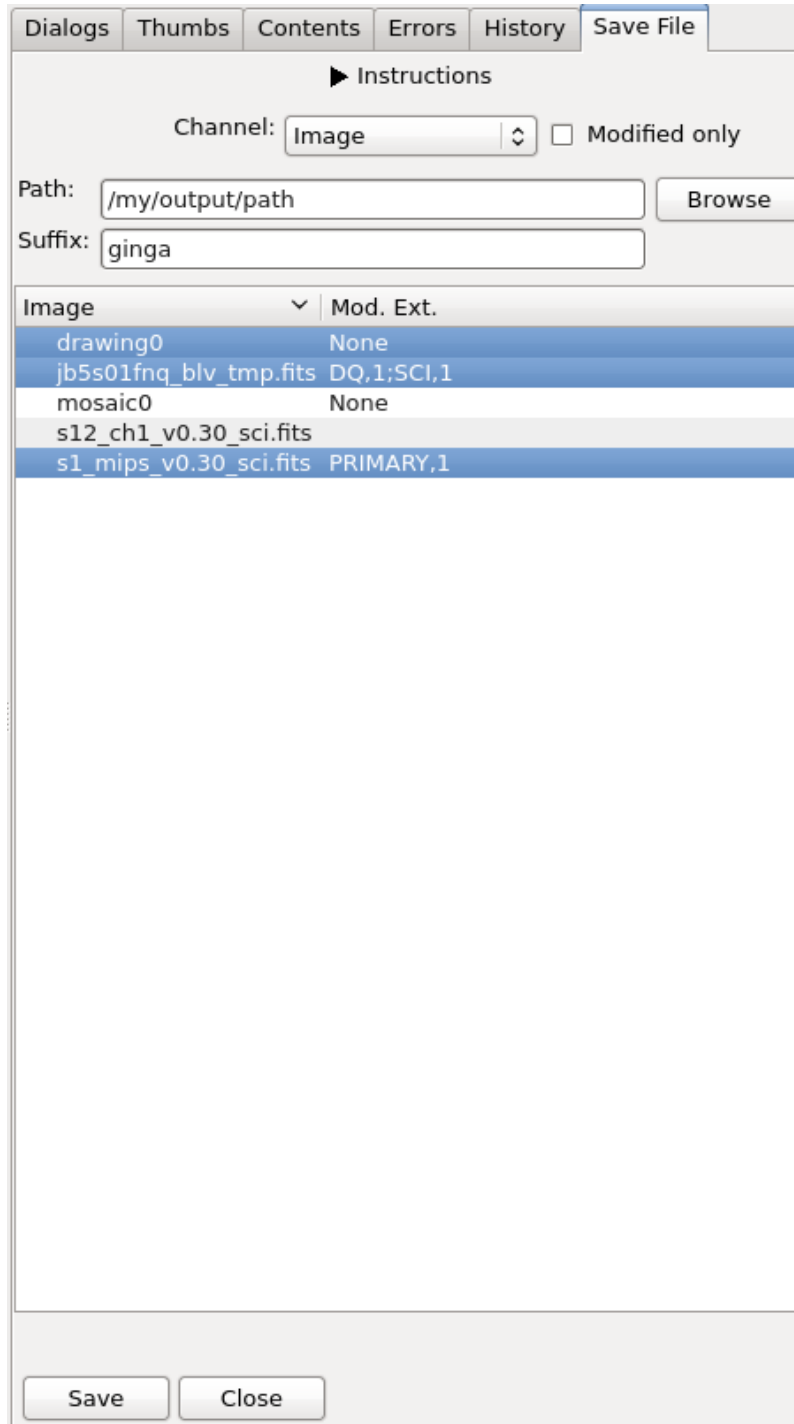
Save images to output files.

Plugin Type: Global

SaveImage is a global plugin. Only one instance can be opened.

Usage

This global plugin is used to save any changes made in Ginga back to output images. For example, a mosaic image that was created by the Mosaic plugin. Currently, only FITS images (single or multiple extensions) are supported.

Given the output directory (e.g., `/mypath/outputs/`), a suffix (e.g., `ginga`), an image channel (`Image`), and a selected image (e.g., `image1.fits`), the output file will be `/mypath/outputs/image1_ginga_Image.fits`. Inclusion of the channel name is optional and can be omitted using plugin configuration file, `plugin_SaveImage.cfg`. The modified extension(s) will have new header or data extracted from Ginga, while those not modified will remain untouched. Relevant change log entries from the `ChangeHistory` global plugin will be inserted into the history of its `PRIMARY` header.

Note: This plugin uses the module `astropy.io.fits` to write the output images, regardless of what is chosen for `FITSpkg` in the `general.cfg` configuration file.

It is customizable using `~/.ginga/plugin_SaveImage.cfg`, where `~` is your HOME directory:

```
#
# SaveImage plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_SaveImage.cfg"

# Default output parameters. Can also be changed in the GUI.
output_directory = '.'
output_suffix = 'ginga'

# Include channel name in the suffix.
# If False, only output_suffix is used regardless of channel.
include_chname = True

# Clobber existing output files or not
clobber = False

# Only list modified images from the channel
modified_only = True

# Maximum mosaic size to allow for writing out.
# This is useful to prevent super large mosaic from being written.
# Default is 10k x 10k
max_mosaic_size = 1e8

# Maximum number of rows that will turn off auto column resizing (for speed)
max_rows_for_col_resize = 5000
```

Downloads
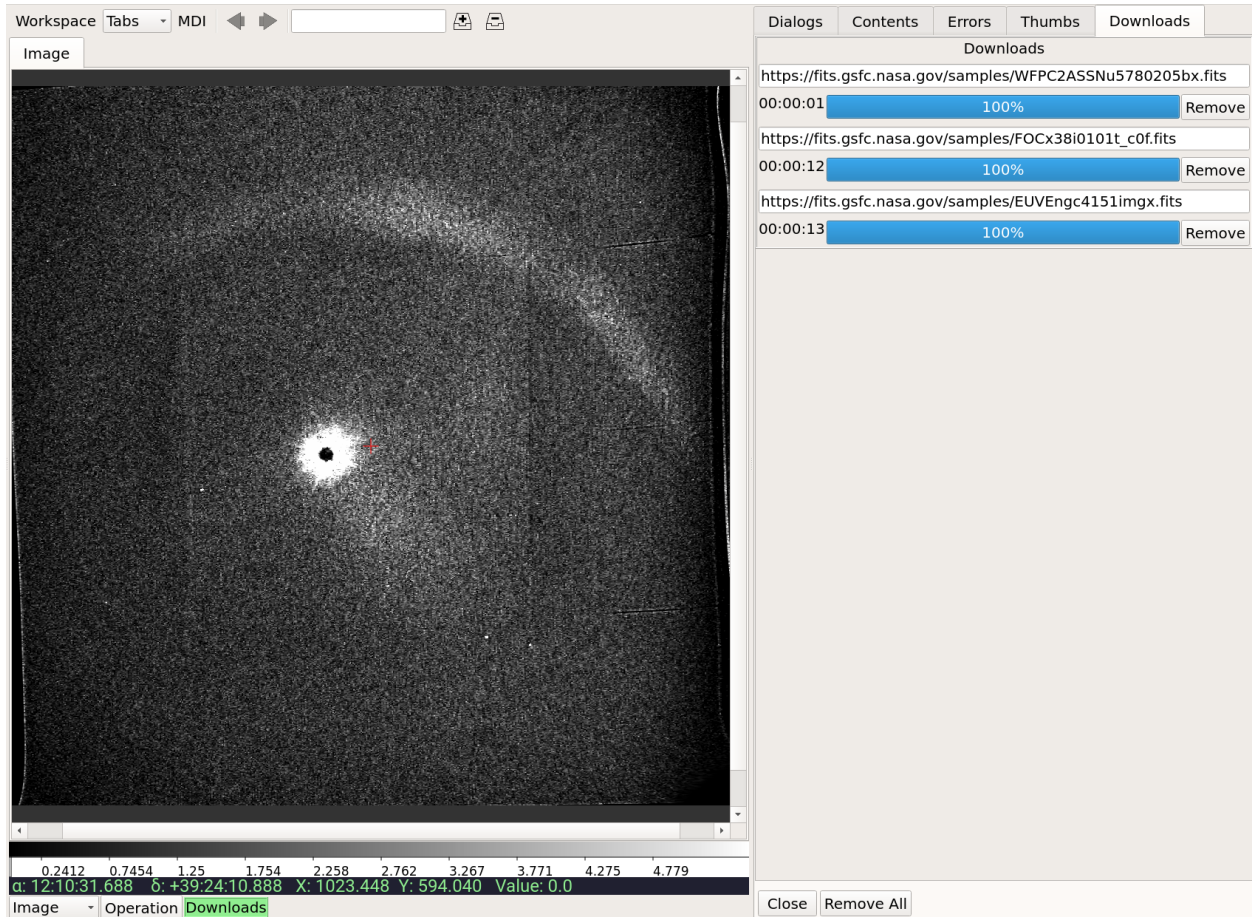
Downloads GUI for the Ginga reference viewer.

Plugin Type: Global

`Download` is a global plugin. Only one instance can be opened.

Usage

Open this plugin to monitor the progress of URI downloads. Start it using the "Plugins" or "Operations" menu, and selecting the "Downloads" plugin from under the "Util" category.

If you want to initiate a download, simply drag a URI into a channel image viewer or the `Thumbs` pane.

You can remove the information about a download at any time by clicking the "Clear" button for its entry. You can clear entries for all downloads by clicking the "Clear All" button at the bottom.

Currently, it is not possible to cancel a download in progress.

Settings

The `auto_clear_download` option, if set to `True`, will cause a download entry to be automatically deleted from the pane when the download completes. It does not remove any downloaded file(s).

The download folder can be user-defined by assigning a value to the "download_folder" setting in ~/.ginga/general.cfg. If unassigned, it defaults to a folder in the platform-specific default temp directory (as told by the Python 'tempfile' module).

4.5.2 Local plugins

An *operation* is the activation of a local plugin to perform some function. The plugin manager toolbar at the bottom of the center pane is the graphical way to start an operation.

Pick
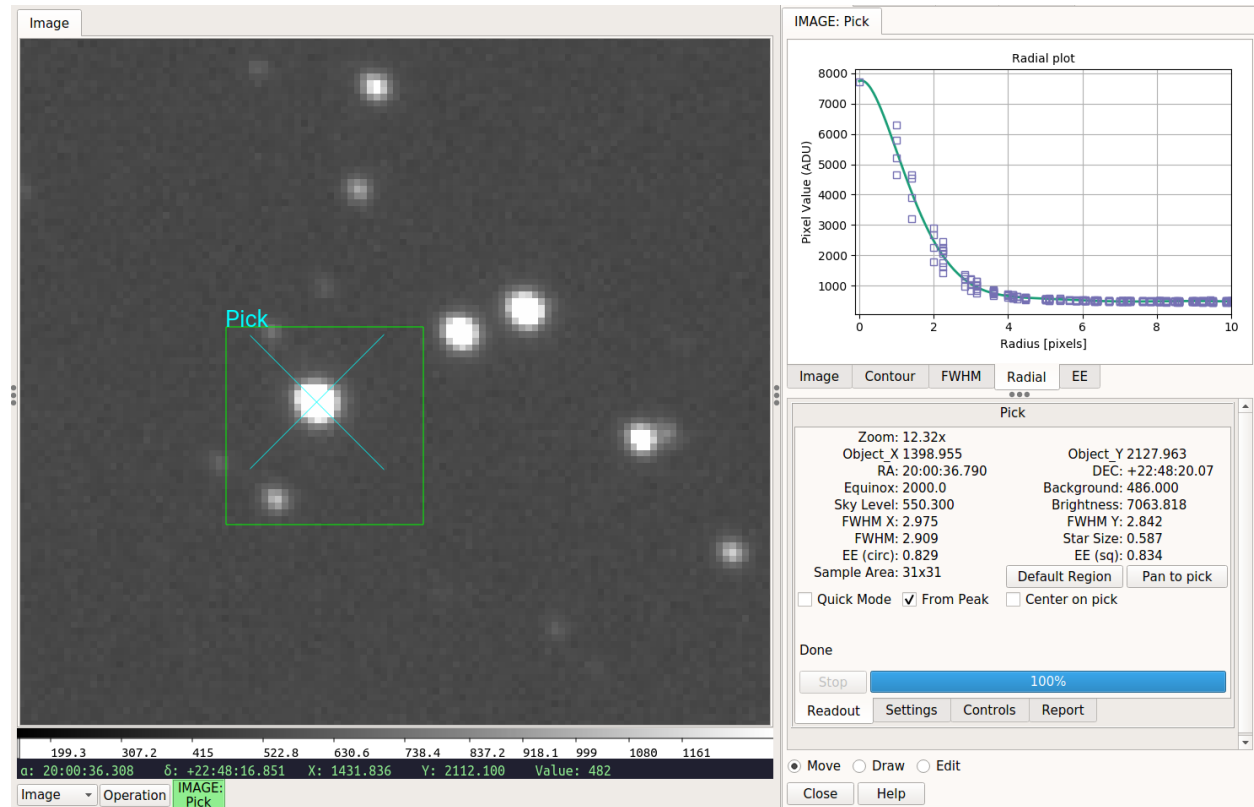
Perform quick astronomical stellar analysis.

Plugin Type: Local

`Pick` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

The `Pick` plugin is used to perform quick astronomical data quality analysis on stellar objects. It locates stellar candidates within a drawn box and picks the most likely candidate based on a set of search settings. The Full Width Half Max (FWHM) is reported on the candidate object, as well as its size based on the plate scale of the detector. Rough measurement of background, sky level and brightness is also done.

Defining the pick area

The default pick area is defined as a box of approximately 30x30 pixels that encloses the search area.

The move/draw/edit selector at the bottom of the plugin is used to determine what operation is being done to the pick area:

- If "move" is selected, then you can move the existing pick area by dragging it or clicking where you want the center of it placed. If there is no existing area, a default one will be created.

- If "draw" is selected, then you can draw a shape with the cursor to enclose and define a new pick area. The default shape is a box, but other shapes can be selected in the "Settings" tab.
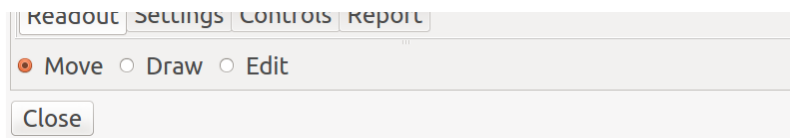
Fig. 1: "Move", "Draw", and "Edit" buttons.

- If "edit" is selected, then you can edit the pick area by dragging its control points, or moving it by dragging in the bounding box.

After the area is moved, drawn or edited, `Pick` will search the area for all peaks and evaluate the peaks based on the criteria in the "Settings" tab of the UI (see "The Settings Tab" below) and try to locate the best candidate matching the settings.

Note: the "Quick Mode" and "From Peak" checkboxes were removed in Ginga release v4.0.

If a candidate is found

The candidate will be marked with a point (usually an "X") in the channel viewer canvas, centered on the object as determined by the horizontal and vertical FWHM measurements.

The top set of tabs in the UI will be populated as follows:
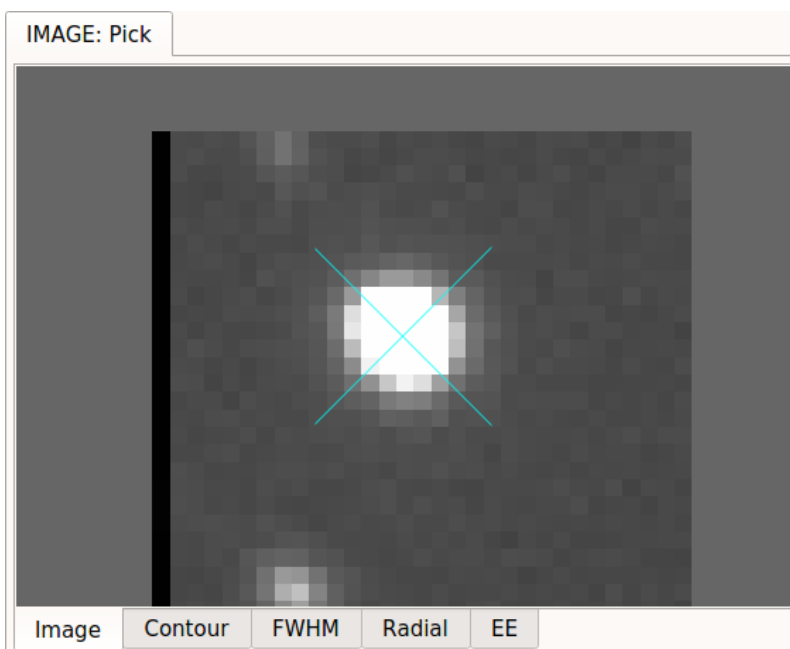
Fig. 2: "Image" tab of `Pick` area.

The "Image" tab will show the contents of the cutout area. The widget in this tab is a Ginga widget and so can be zoomed and panned with the usual keyboard and mouse bindings (e.g., scroll wheel). It will also be marked with a point centered on the object and additionally the pan position will be set to the found center.

The "Contour" tab will show a contour plot. This is a contour plot of the area immediately surrounding the candidate, and not usually encompassing the entire region of the pick area. You can use the scroll wheel to zoom the plot and a click of the scroll wheel (mouse button 2) to set the pan position in the plot.
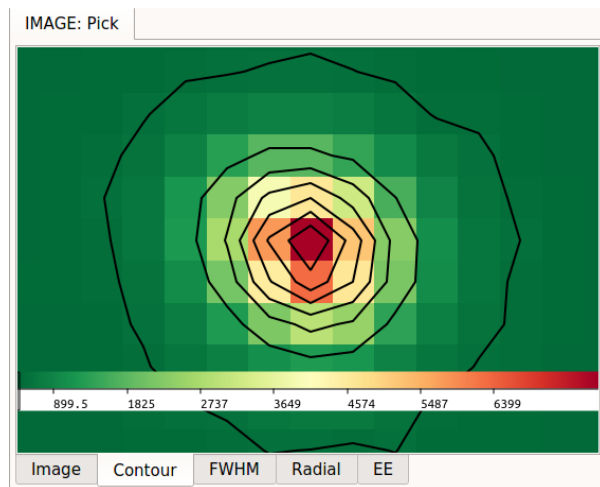
Fig. 3: "Contour" tab of `Pick` area.
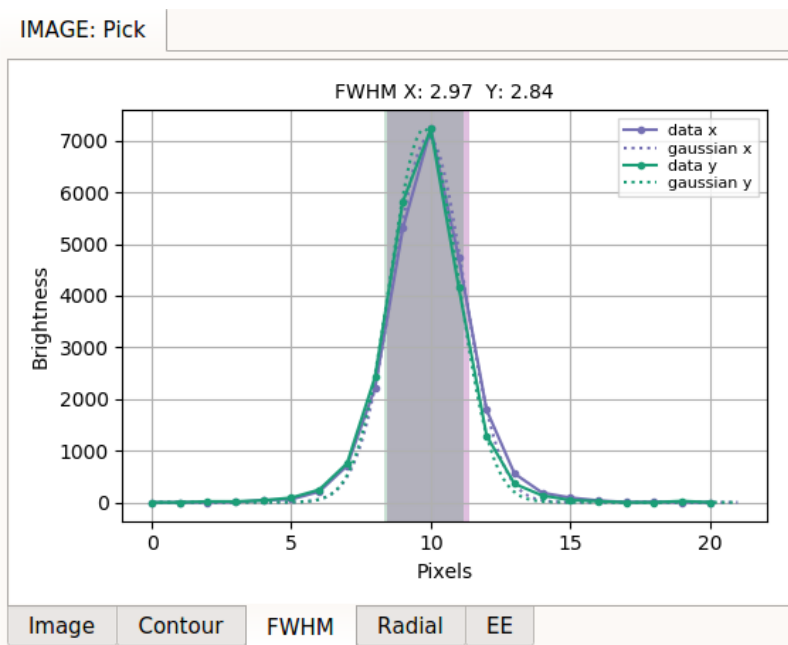
Fig. 4: "FWHM" tab of `Pick` area.

The "FWHM" tab will show a FWHM plot. The purple lines show measurements in the X direction and the green lines show measurements in the Y direction. The solid lines indicate actual pixel values and the dotted lines indicate the fitted 1D function. The shaded purple and green regions indicate the FWHM measurements for the respective axes.
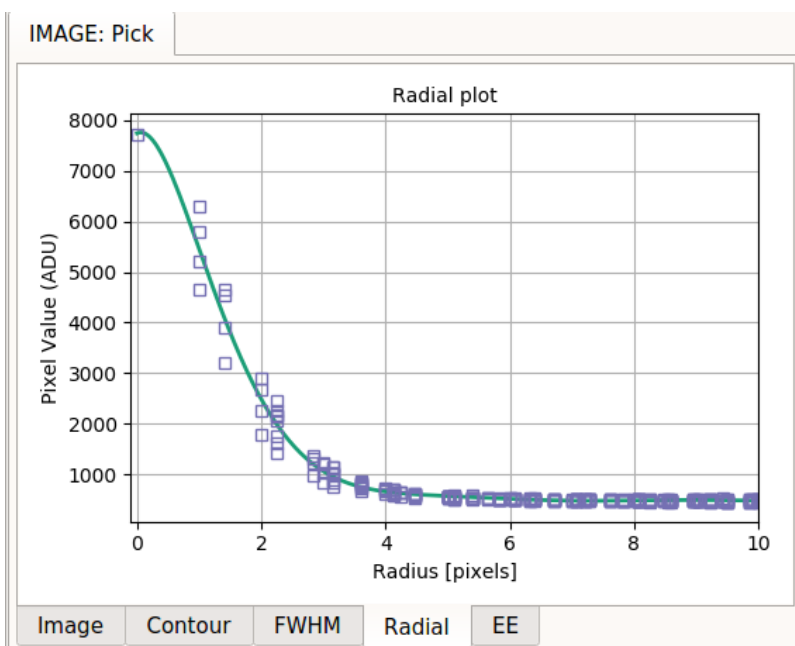
Fig. 5: "Radial" tab of `Pick` area.

The "Radial" tab contains a radial profile plot. Plotted points in purple are data values, and a line is fitted to the data.

The "EE" tab contains a plot of fractional encircled and ensquared energies (EE) in purple and green, respectively, for the chosen target. Simple background subtraction is done in a way that is consistent with FWHM calculations before EE values are measured. The sampling and total radii, shown as black dashed lines, can be set in the "Settings" tab; when these are changed, click "Redo Pick" to update the plot and measurements. The measured EE values at the given sampling radius are also displayed in the "Readout" tab. When reporting is requested, the EE values at the given sampling radius and the radius itself will be recorded under "Report" table, along with other information.

When "Show Candidates" is active, the candidates near the edges of the bounding box will not have EE values (set to 0).

The "Readout" tab will be populated with a summary of the measurements. There are two buttons and three check boxes in this tab:

- The "Default Region" button restores the pick region to the default shape and size.

- The "Pan to pick" button will pan the channel viewer to the located center.

- If "Center on pick" is checked, the shape will be recentered on the located center, if found (i.e., the shape "tracks" the pick).

The "Controls" tab has a couple of buttons that will work off of the measurements.

- The "Bg cut" button will set the low cut level of the channel viewer to the measured background level. A delta to this value can be applied by setting a value in the "Delta bg" box (press "Enter" to change the setting).

- The "Sky cut" button will set the low cut level of the channel viewer to the measured sky level. A delta to this value can be applied by setting a value in the "Delta sky" box (press "Enter" to change the setting).

- The "Bright cut" button will set the high cut level of the channel viewer to the measured sky+brightness levels. A delta to this value can be applied by setting a value in the "Delta bright" box (press "Enter" to change the

IMAGE: Pick

EE(circ)=0.829, EE(sq)=0.834

Image | Contour | FWHM | Radial | EE

Pick

Zoom: 12.32x
Object_X 1398.955   Object_Y 2127.963
RA: 20:00:36.790   DEC: +22:48:20.07
Equinox: 2000.0   Background: 486.000
Sky Level: 550.300   Brightness: 7063.818
FWHM X: 2.975   FWHM Y: 2.842
FWHM: 2.909   Star Size: 0.587
EE (circ): 0.829   EE (sq): 0.834
Sample Area: 31x31   Default Region | Pan to pick

☐ Quick Mode ☑ From Peak ☐ Center on pick
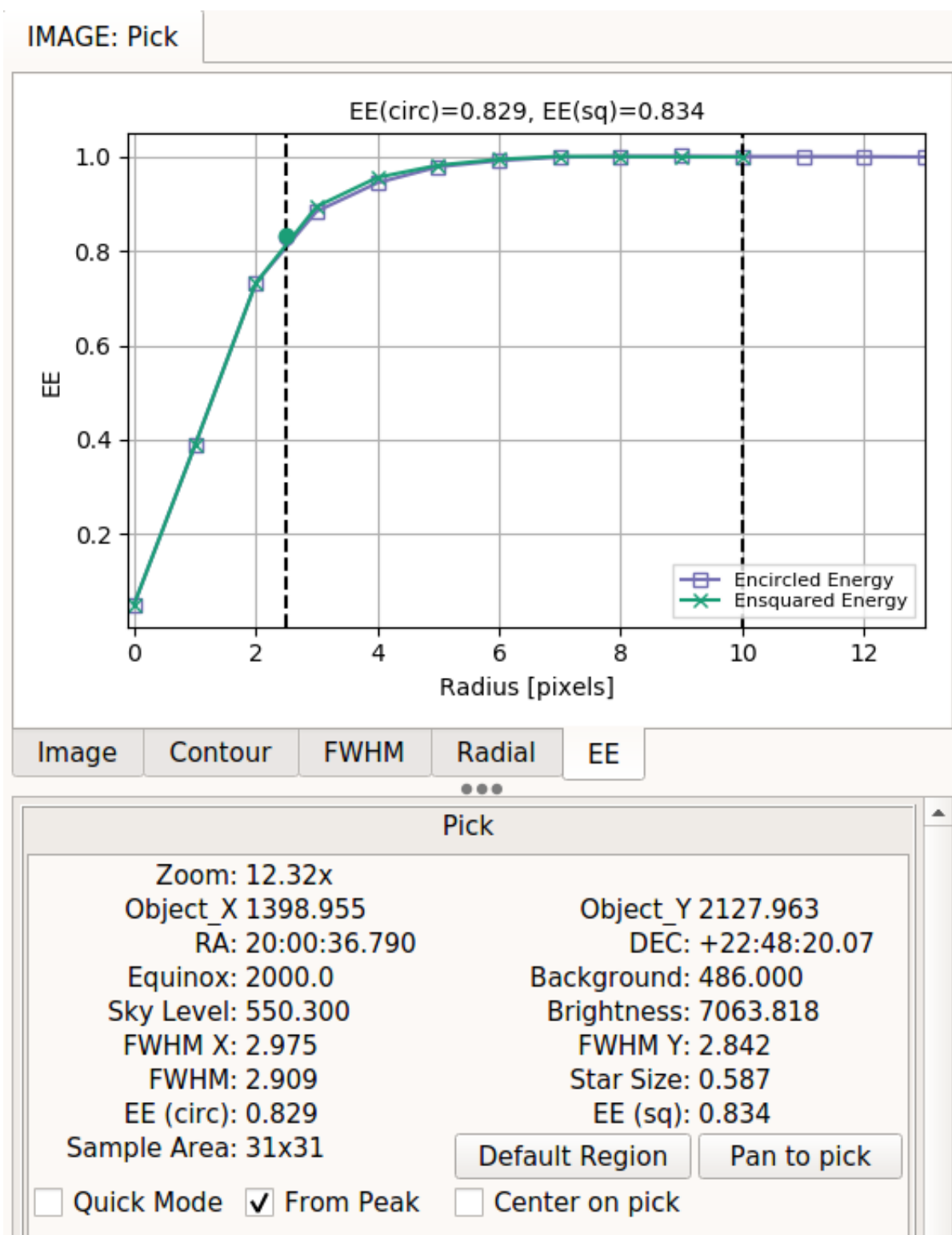
Fig. 6: "EE" tab of `Pick` area.
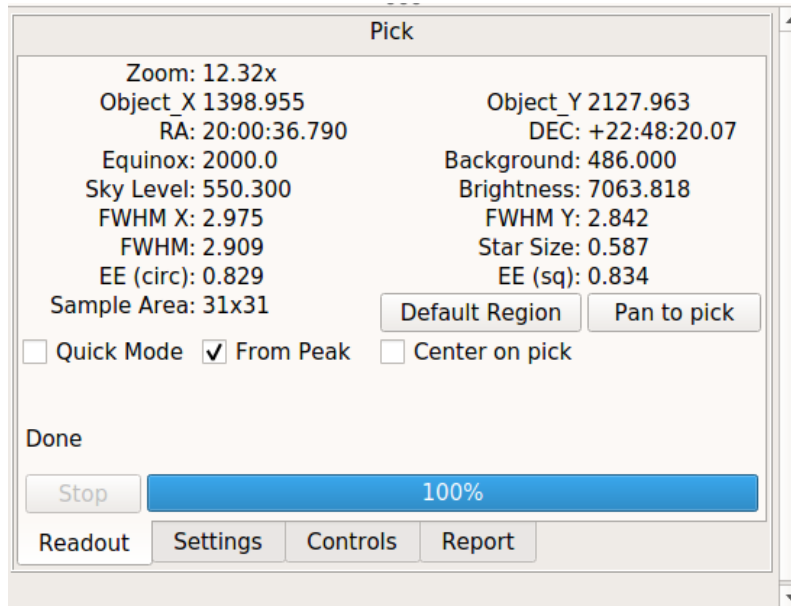
Fig. 7: "Readout" tab of `Pick` area.
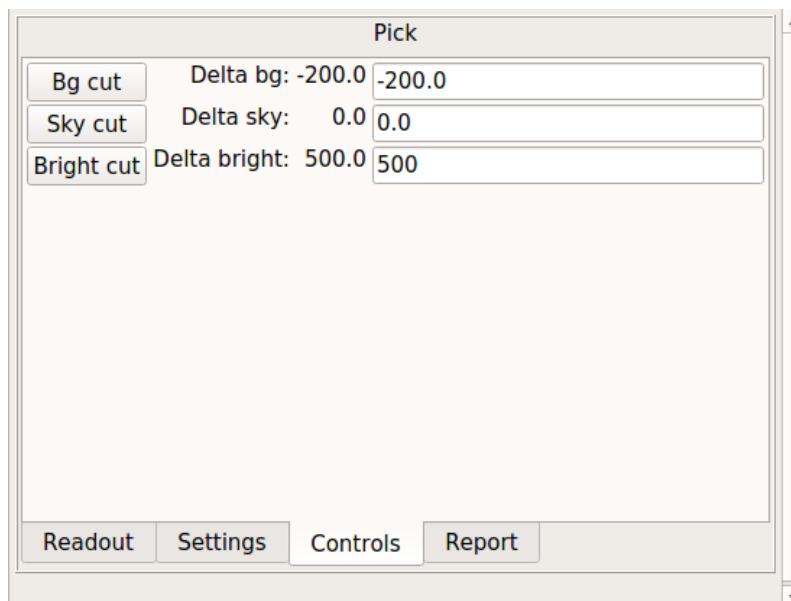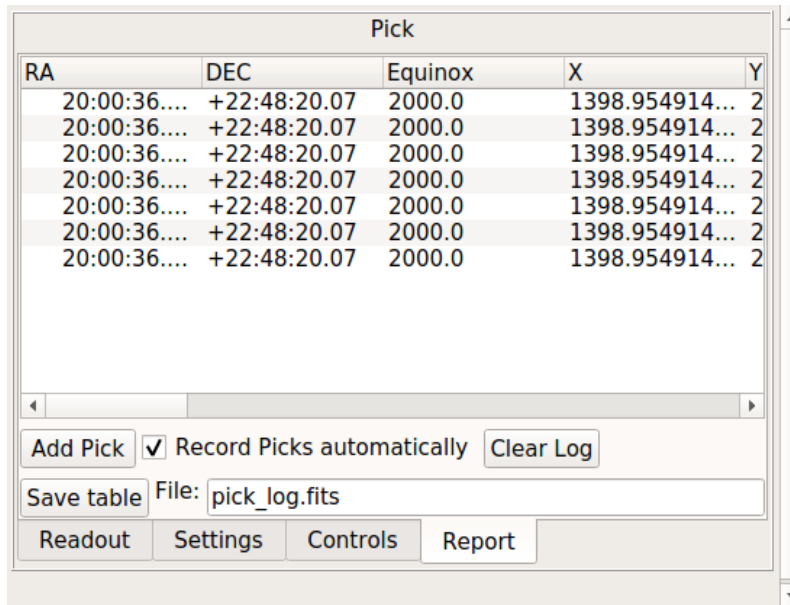
Fig. 8: "Controls" tab of `Pick` area.

setting).

Fig. 9: "Report" tab of `Pick` area.

The "Report" tab is used to record information about the measurements in tabular form.

By pressing the "Add Pick" button, the information about the most recent candidate is added to the table. If the "Record Picks automatically" checkbox is checked, then any candidates are added to the table automatically.

Note: If the "Show Candidates" checkbox in the "Settings" tab is checked, then *all* objects found in the region (according to the settings) will be added to the table instead of just the selected candidate.

You can clear the table at any time by pressing the "Clear Log" button. The log can be saved to a table by putting a valid path and filename in the "File:" box and pressing "Save table". File type is automatically determined by the given extension (e.g., ".fits" is FITS and ".txt" is plain text).

If no candidate is found

If no candidate can be found (based on the settings), then the pick area is marked with a red point centered on the pick area.

The image cutout will be taken from this central area and so the "Image" tab will still have content. It will also be marked with a central red "X".

The contour plot will still be produced from the cutout.

All the other plots will be cleared.

The Settings Tab

The "Settings" tab controls aspects of the search within the pick area:

- The "Show Candidates" checkbox controls whether all detected sources are marked or not (as shown in the figure below). Additionally, if checked, then all the found objects are added to the pick log table when using the "Report" controls.
- The "Draw type" parameter is used to choose the shape of the pick area to be drawn.
- The "Radius" parameter sets the radius to be used when finding and evaluating bright peaks in the image.
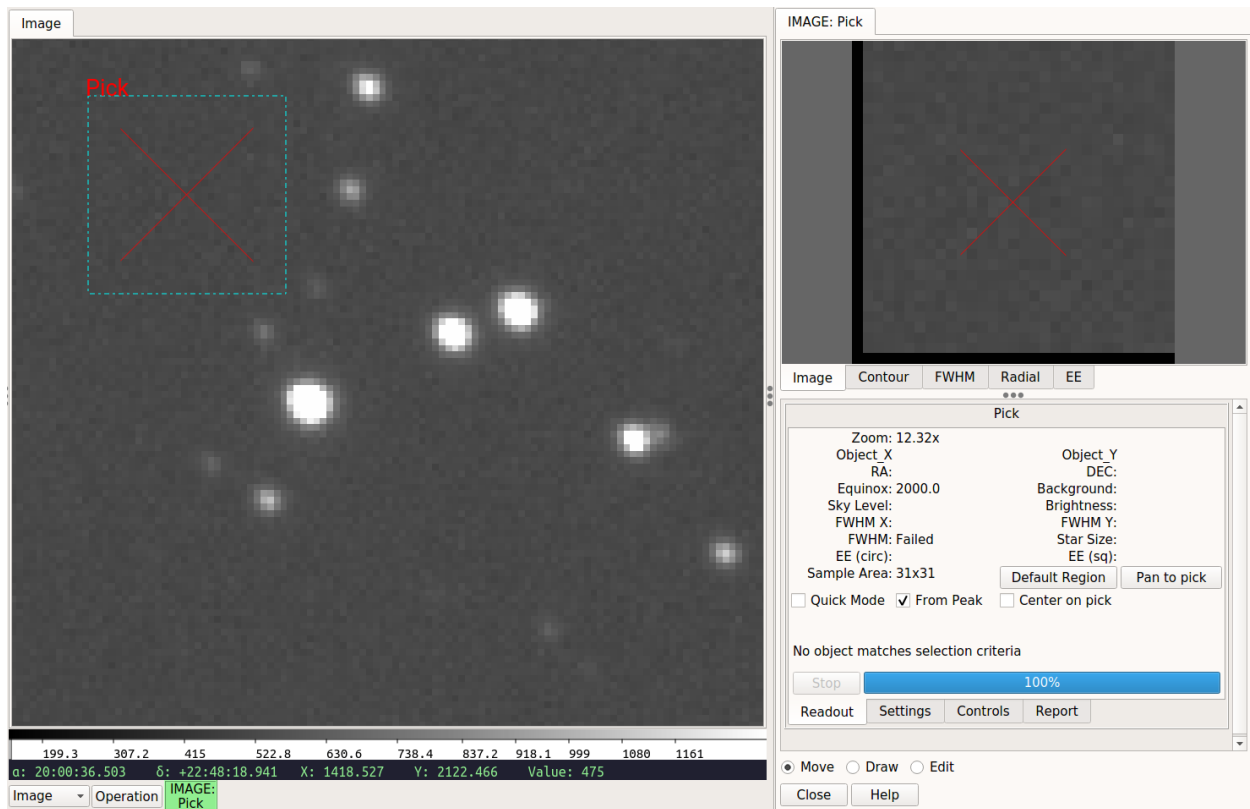
Fig. 10: Marker when no candidate found.
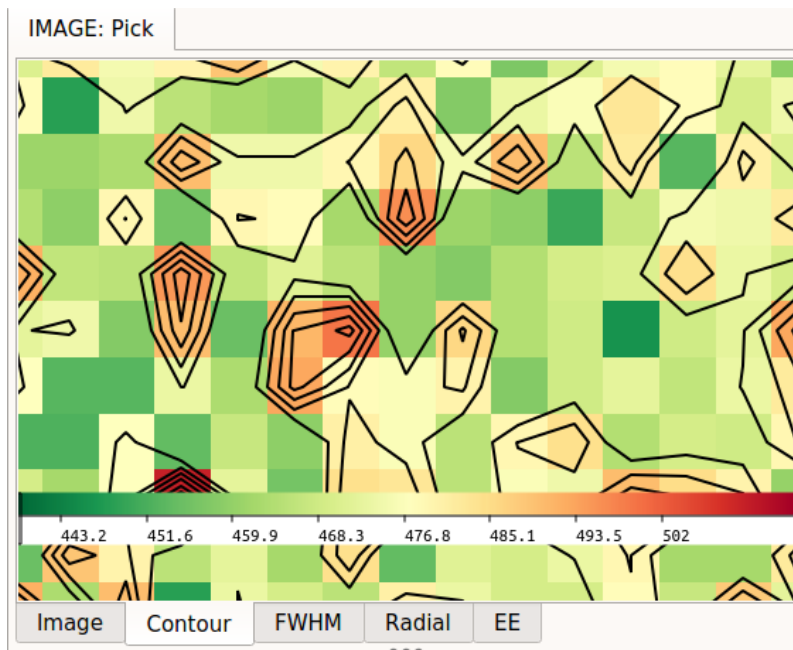
Fig. 11: Contour when no candidate found.
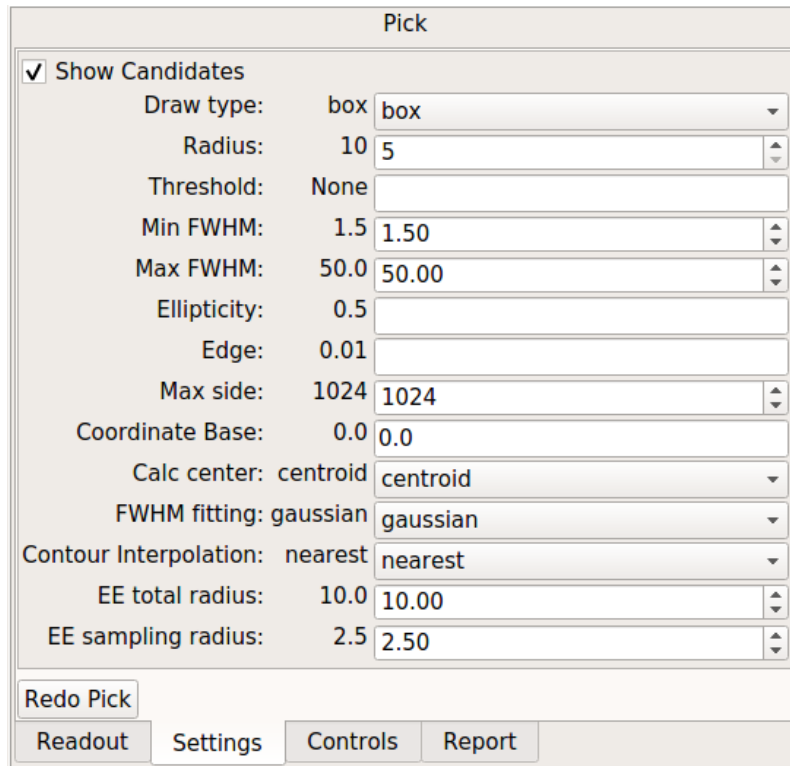
Fig. 12: "Settings" tab of `Pick` plugin.

- The "Threshold" parameter is used to set a threshold for peak finding; if set to "None", then a reasonable default value will be chosen.

- The "Min FWHM" and "Max FWHM" parameters can be used to eliminate certain sized objects from being candidates.

- The "Ellipticity" parameter is used to eliminate candidates based on their asymmetry in shape.

- The "Edge" parameter is used to eliminate candidates based on how close to the edge of the cutout they are. *NOTE: currently this works reliably only for non-rotated rectangular shapes.*

- The "Max side" parameter is used to limit the size of the bounding box that can be used in the pick shape. Larger sizes take longer to evaluate.

- The "Coordinate Base" parameter is an offset to apply to located sources. Set to "1" if you want sources pixel locations reported in a FITS-compliant manner and "0" if you prefer 0-based indexing.

- The "Calc center" parameter is used to determine whether the center is calculated from FWHM fitting ("fwhm") or centroiding ("centroid").

- The "FWHM fitting" parameter is used to determine which function is is used for FWHM fitting ("gaussian" or "moffat"). The option to use "lorentz" is also available if "calc_fwhm_lib" is set to "astropy" in `~/.ginga/plugin_Pick.cfg`.

- The "Contour Interpolation" parameter is used to set the interpolation method used in rendering the background image in the "Contour" plot.

- The "EE total radius" defines the radius (for encircled energy) and box half-width (for ensquared energy) in pixels where EE fraction is expected to be 1 (i.e., all the flux for a point-spread function is contained within).

- The "EE sampling radius" is the radius in pixel used to sample the measured EE curves for reporting.

The "Redo Pick" button will redo the search operation. It is convenient if you have changed some parameters and want to see the effect based on the current pick area without disturbing it.
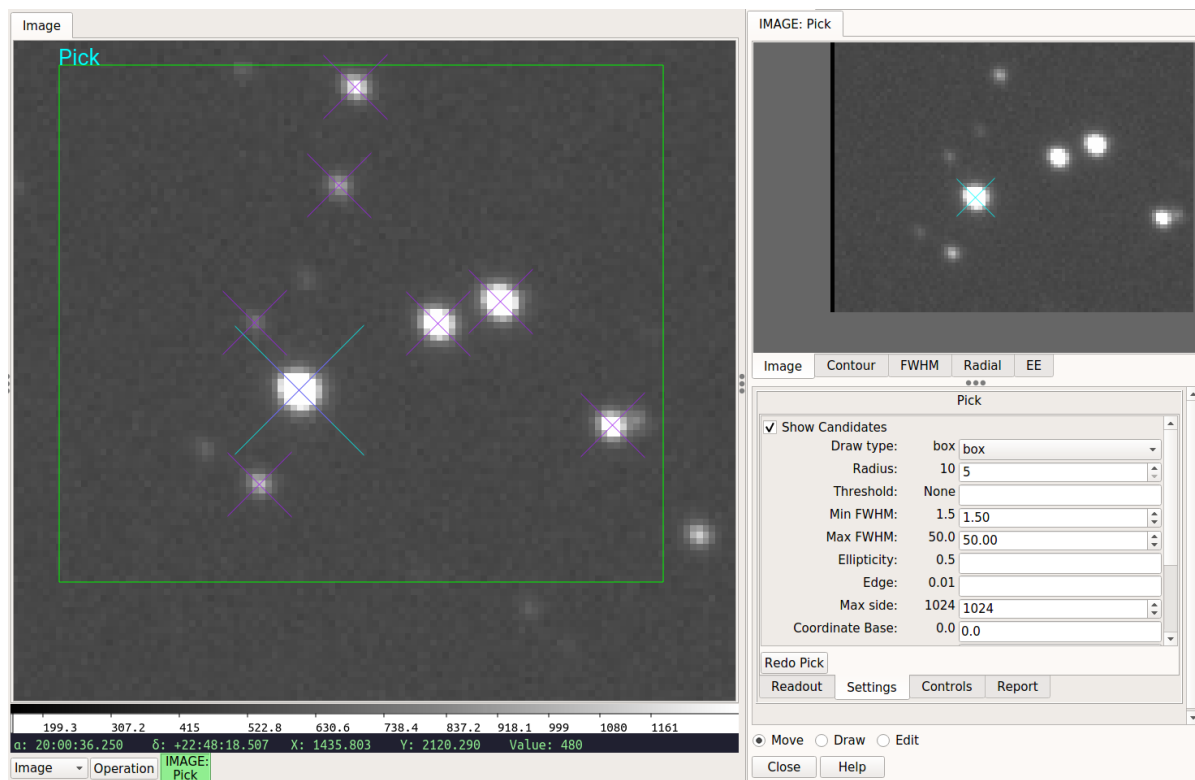
Fig. 13: The channel viewer when "Show Candidates" is checked.

User Configuration

It is customizable using `~/.ginga/plugin_Pick.cfg`, where ~ is your HOME directory:

```
#
# Pick plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Pick.cfg"

color_pick = 'green'
shape_pick = 'box'
color_candidate = 'purple'

# Offset to add to Pick results. Default is 1.0 for FITS like indexing,
# set to 0.0 here if you prefer numpy-like 0-based indexing
pixel_coords_offset = 0.0

# Maximum side for a pick region
max_side = 1024

# For image cutout viewer ("Image" tab)
# you can set autozoom and autocuts preferences
cutout_autozoom = 'override'
cutout_autocuts = 'off'
```

```
# For contour plot ("Contour" tab)
# widget type: let choose automatically or force 'ginga' or 'matplotlib'
# (choice of 'ginga' requires scikit-image to be installed)
contour_widget = 'choose'
# if ginga widget is chosen, you can set autozoom and autocuts preferences
contour_autozoom = 'override'
contour_autocuts = 'override'
num_contours = 8
# How big of a radius are we willing to consider from the center of the
# pick?  bigger numbers == slower
contour_size_min = 10
contour_size_limit = 70


# should the pick shape recenter on the found object center, if any?
# useful for "tracking" an object that is moving from image to image
center_on_pick = False


# Star candidate search parameters
radius = 10
# Set threshold to None to auto calculate it
threshold = None
# Minimum and maximum fwhm to be considered a candidate
min_fwhm = 1.5
max_fwhm = 50.0
# Minimum ellipticity to be considered a candidate
min_ellipse = 0.5
# Percentage from edge to be considered a candidate
edge_width = 0.01
# Graphically indicate all possible considered candidates
show_candidates = False


# Center of object is based on FWHM ("fwhm") or centroid ("centroid")
# calculation:
calc_center_alg = 'centroid'


# Library to use for FWHM fitting ("native" or "astropy")
calc_fwhm_lib = 'native'


# Fitting function to use for FWHM ("gaussian" or "moffat")
calc_fwhm_alg = 'gaussian'


# Defaults for delta cut levels (in Controls tab)
delta_sky = 0.0
delta_bright = 0.0


# Encircled and ensquared energy (EE) calculations:
# a. Radius (pixel) where EE fraction is expected to be 1.
ee_total_radius = 10.0
# b. Radius (pixel) to sample EE for reporting.
ee_sampling_radius = 2.5
```

```
# use a different color/intensity map than channel image?
pick_cmap_name = None
pick_imap_name = None

# For Reports tab
record_picks = True

# Set this to a file name, if None a filename will be automatically chosen
report_log_path = None
```

Ruler
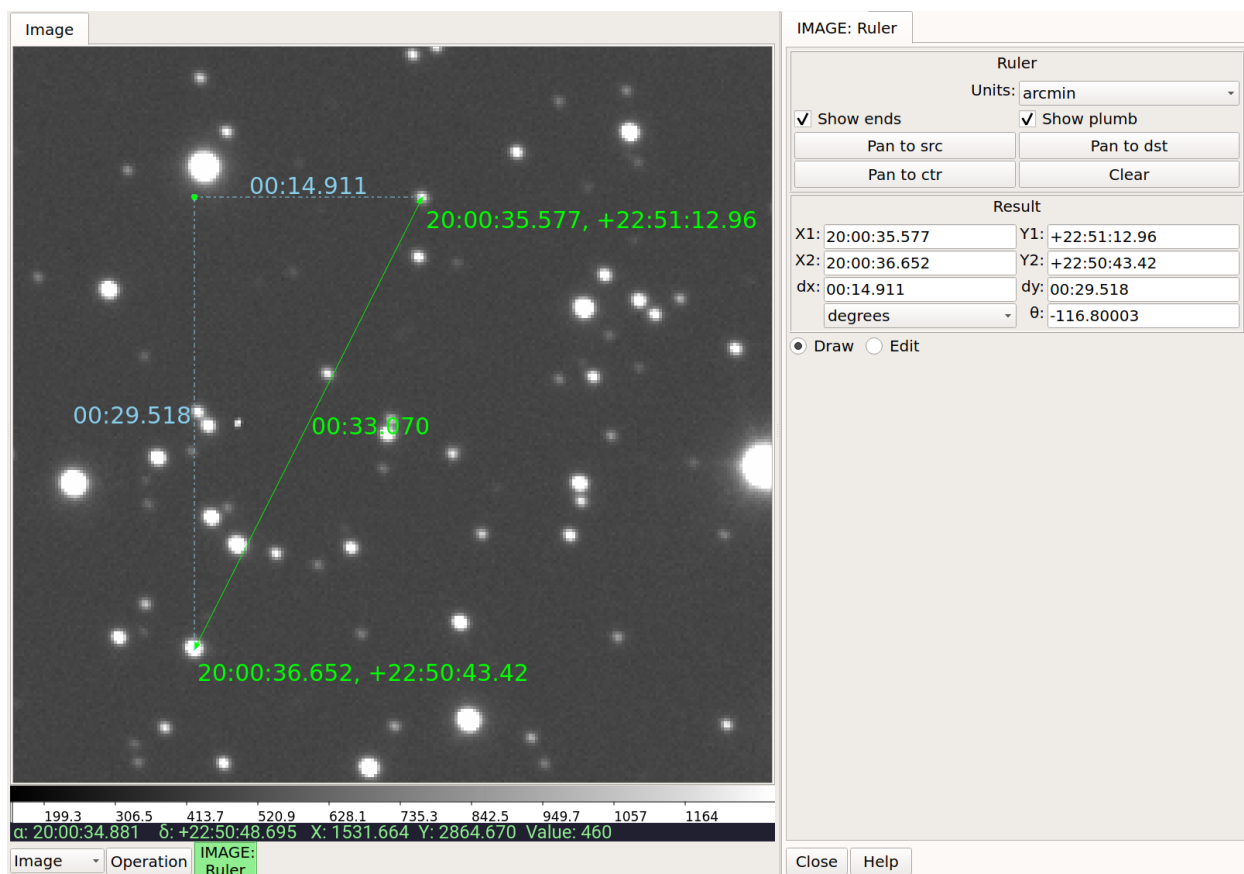
Ruler is a simple plugin designed to measure distances on an image.

Plugin Type: Local

Ruler is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

Ruler measures distance by calculating a spherical triangulation via WCS mapping of three points defined by a single line drawn on the image. By default, the distance is shown in arcminutes of sky, but using the "Units" control, it can be changed to show degrees or pixel distance instead.

Click and drag to establish a ruler between two points. When you finish the draw operation the ruler is established and the plugin UI will update to show detail about the line, including the endpoint positions and the angle of the line. The

units of the angle can be toggled between degrees and radians using the adjacent drop-down box.

To erase the old and make a new ruler, click and drag again. When another line is drawn, it replaces the first one. When the plugin is closed, the graphic overlay is removed. Should you want "sticky rulers", use the `Drawing` plugin (and choose "Ruler" as the drawing type).

Editing

To edit an existing ruler, click the radio button in the plugin UI labeled "Edit". If the ruler does not become selected immediately, click on the diagonal connecting the two points. This should establish a bounding box around the ruler and show its control points. Drag within the bounding box to move the ruler or click and drag the endpoints to edit the ruler. The ruler can also be scaled or rotated using those control points.

UI

The units shown for distance can be selected from the drop-down box in the UI. You have a choice of "arcmin", "degrees", or "pixels". The first two require a valid and working WCS in the image.

The endpoint values are shown in the UI, but can additionally be shown in the ruler graphic if the "Show ends" checkbox is toggled. Plumb lines will be shown if the "Show plumb" box is toggled.

Buttons

The "Pan to src" button will pan the main image to the origin of the line drawn, while "Pan to dst" will pan to the end. "Pan to ctr" sets the pan position to the center point of the line. These buttons may be useful for close up, zoomed-in work on the image. "Clear" clears the ruler from the image.

Tips Open the "Zoom" plugin to precisely see detail of the cursor area. The "Pick" plugin can also be used in conjunction with Ruler to identify the central point of an object, when aligning either end of the ruler.

MultiDim

plugin to navigate HDUs in a FITS file or planes in a 3D cube or higher dimension dataset.

Plugin Type: Local

`MultiDim` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

`MultiDim` is a plugin designed to handle data cubes and multi-HDU FITS files. If you have opened such an image in Ginga, starting this plugin will enable you to browse to other slices of the cube or view other HDUs.

For a data cube, you can save a slice as an image using the "Save Slice" button or create a movie using the "Save Movie" button by entering the "Start" and "End" slice indices. This feature requires `mencoder` to be installed.

For a FITS table, its data are read in using Astropy table. Column units are displayed right under the main header ("None" if no unit). For masked columns, masked values are replaced with pre-defined fill values.

Browsing HDUs

Use the HDU drop down list in the upper part of the UI to browse and select an HDU to open in the channel.

Navigating Cubes

Use the controls in the lower part of the UI to select the axis and to step through the planes in that axis.

User Configuration

It is customizable using `~/.ginga/plugin_MultiDim.cfg`, where ~ is your HOME directory:

```
#
# MultiDim plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_MultiDim.cfg"

# Sort option for HDU listing.
# Available attributes:
#   'index' -- Extension index
#   'name' -- Extension name
#   'extver' -- Extension version number
#   'htype' -- HDU type (PrimaryHDU, ImageHDU, TableHDU)
#   'dtype' -- Data type
# Example to sort by HDU name and extver:
#   sort_keys = ['name', 'extver']
# Default is to sort by index only:
sort_keys = ['index']

# Reverse for HDU listing?
sort_reverse = False
```

Cuts
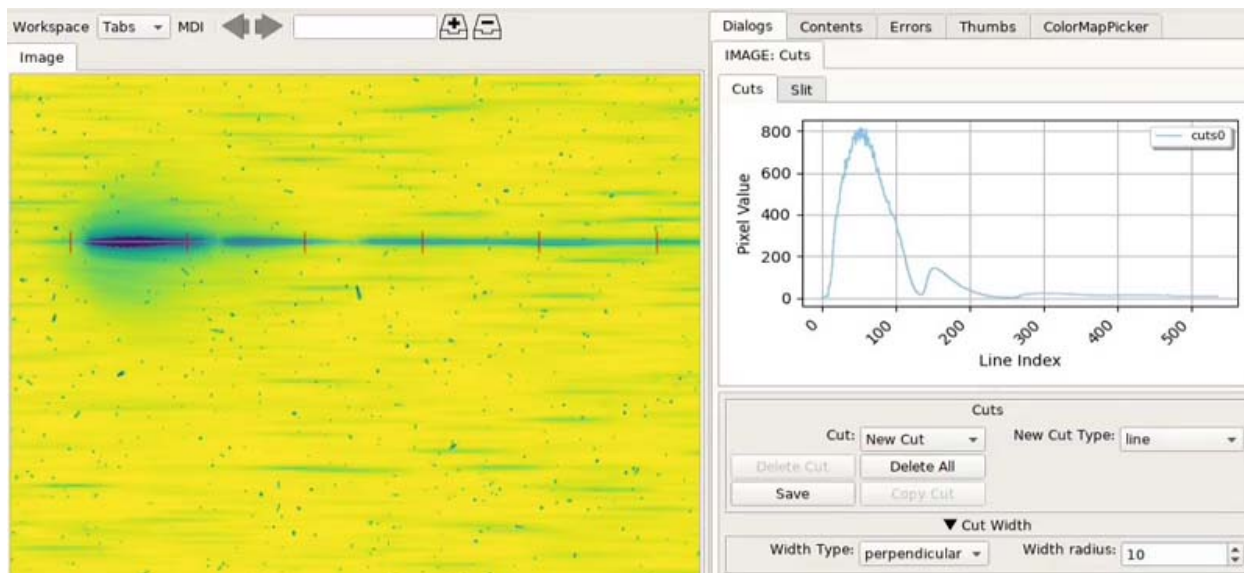
A plugin for generating a plot of the values along a line or path.

Plugin Type: Local

Cuts is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

Cuts plots a simple graph of pixel values vs. index for a line drawn through the image. Multiple cuts can be plotted.

There are four kinds of cuts available: line, path, freepath and beziercurve:

- The "line" cut is a straight line between two points.
- The "path" cut is drawn like an open polygon, with straight segments in-between.
- The "freepath" cut is like a path cut, but drawn using a free-form stroke following the cursor movement.
- The "beziercurve" path is a cubic Bezier curve.

If a new image is added to the channel while the plugin is active, it will update with the new calculated cuts on the new image.

If the "enable slit" setting is enabled, this plugin will also allow slit image functionality (for multidimensional images) via a "Slit" tab. In the tab UI, select one axis from the "Axes" list and draw a line. This will create a 2D image that assumes the first two axes are spatial and index the data along the selected axis. Much like Cuts, you can view the other slit images using the cut selection drop down box.

Drawing Cuts

The "New Cut Type" menu let you choose what kind of cut you are going to draw.

Choose "New Cut" from the "Cut" dropdown menu if you want to draw a new cut. Otherwise, if a particular named cut is selected then that will be replaced by any newly drawn cut.

While drawing a path or beziercurve cut, press 'v' to add a vertex, or 'z' to remove the last vertex added.

Keyboard Shortcuts

While hovering the cursor, press 'h' for a full horizontal cut and 'j' for a full vertical cut.

Deleting Cuts

To delete a cut, select its name from the "Cut" dropdown and click the "Delete" button. To delete all cuts, press "Delete All".

Editing Cuts

Using the edit canvas function, it is possible to add new vertices to an existing path and to move vertices around. Click the "Edit" radio button to put the canvas in edit mode. If a cut is not automatically selected, you can now select the line, path, or curve by clicking on it, which should enable the control points at the ends or vertices – you can drag these around. To add a new vertex to a path, hover the cursor carefully on the line where you want the new vertex and press 'v'. To get rid of a vertex, hover the cursor over it and press 'z'.

You will notice one extra control point for most objects, which has a center of a different color – this is a movement control point for moving the entire object around the image when in edit mode.

You can also select "Move" to just move a cut unchanged.

Changing Width of Cuts

The width of 'line' cuts can be changed using the "Width Type" menu:

- "none" indicates a cut of zero radius; i.e., only showing the pixel values along the line
- "x" will plot the sum of values along the X axis orthogonal to the cut.
- "y" will plot the sum of values along the Y axis orthogonal to the cut.
- "perpendicular" will plot the sum of values along an axis perpendicular to the cut.

The "Width radius" controls the width of the orthogonal summation by an amount on either side of the cut – 1 would be 3 pixels, 2 would be 5 pixels, etc.

Saving Cuts

Use the "Save" button to save the `Cuts` plot as as image and data as a Numpy compressed archive.

Copying Cuts

To copy a cut, select its name from the "Cut" dropdown and click the "Copy Cut" button. A new cut will be created from it. You can then manipulate the new cut independently.

User Configuration

It is customizable using `~/.ginga/plugin_Cuts.cfg`, where ~ is your HOME directory:

```
#
# Cuts plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Cuts.cfg"

# If set to True will always select a cut after drawing it
select_new_cut = True

# If set to True will automatically change to "move" mode after draw
draw_then_move = True

# If set to True will label cuts with a text annotation
label_cuts = True

# If set to True will add a legend to the cuts plot
show_cuts_legend = False

# If set to True will add Slit tab
```

```
enable_slit = False

# Default cut colors
colors = ['magenta', 'skyblue2', 'chartreuse2', 'cyan', 'pink', 'burlywood2', 'yellow3',
→'turquoise', 'coral1', 'mediumpurple2']

# If set to True, will update graph continuously as cursor is dragged
# around image
drag_update = False
```

Histogram
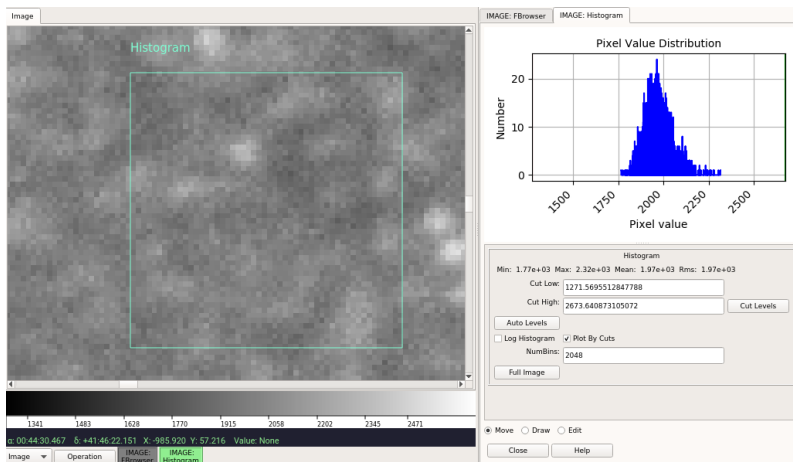
`Histogram` plots a histogram for a region drawn in the image, or for the entire image.

Plugin Type: Local

`Histogram` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

Click and drag to define a region within the image that will be used to calculate the histogram. To take the histogram of the full image, click the button in the UI labeled "Full Image".

Note: Depending on the size of the image, calculating the full histogram may take time.

If a new image is selected for the channel, the histogram plot will be recalculated based on the current parameters with the new data.

Unless disabled in the settings file for the histogram plugin, a line of simple statistics for the box is calculated and shown in a line below the plot.

UI Controls

Three radio buttons at the bottom of the UI are used to control the effects of the click/drag action:

- select "Move" to drag the region to a different location
- select "Draw" to draw a new region
- select "Edit" to edit the region

To make a log plot of the histogram, check the "Log Histogram" checkbox. To plot by the full range of values in the image instead of by the range within the cut values, uncheck the "Plot By Cuts" checkbox.

The "NumBins" parameter determines how many bins are used in calculating the histogram. Type a number in the box and press "Enter" to change the default value.

Cut Levels Convenience Controls

Because a histogram is useful feedback for setting the cut levels, controls are provided in the UI for setting the low and high cut levels in the image, as well as for performing an auto cut levels, according to the auto cut levels settings in the channel preferences.

You can set cut levels by clicking in the histogram plot:

- left click: set low cut
- middle click: reset (auto cut levels)
- right click: set high cut

In addition, you can dynamically adjust the gap between low and high cuts by scrolling the wheel in the plot (i.e. the "width" of the histogram plot curve). This has the effect of increasing or decreasing the contrast within the image. The amount that is changed for each wheel click is set by the plugin configuration file setting `scroll_pct`. The default is 10%.

User Configuration

It is customizable using ~/.ginga/plugin_Histogram.cfg, where ~ is your HOME directory:

```
#
# Histogram plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Histogram.cfg"

# Switch to "move" mode after selection
draw_then_move = True

# Number of bins for histogram
num_bins = 2048

# Histogram color
hist_color = 'aquamarine'

# Calculate extra statistics on box
show_stats = True

# Controls formatting (width) of statistics numbers
maxdigits = 7

# percentage to adjust cuts gap when scrolling in histogram
scroll_pct = 0.10
```

Crosshair
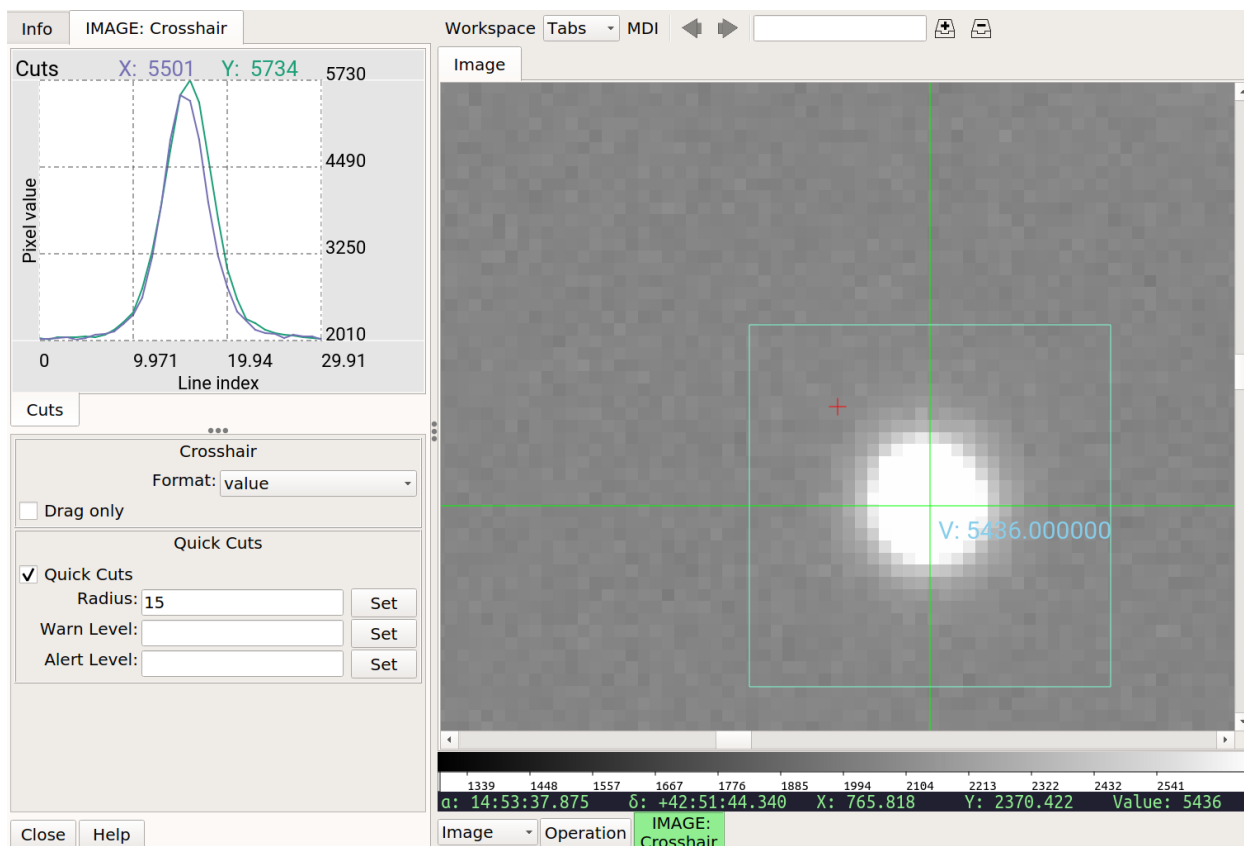
`Crosshair` is a simple plugin to draw crosshairs labeled with the position of the cross in pixels coordinates, WCS coordinates, or data value at the cross position.

Plugin Type: Local

`Crosshair` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

Select the appropriate type of output in the "Format" drop-down box in the UI: "xy" for pixel coordinates, "coords" for the WCS coordinates, and "value" for the value at the crosshair position.

If "Drag only" is checked, then the crosshair is only updated when the cursor is clicked or dragged in the window. If unchecked the crosshair is positioned by simply moving the cursor around the channel viewer window.

The "Cuts" tab contains a profile plot for the vertical and horizontal cuts represented by the visible box boundary present when "Quick Cuts" is checked. This plot is updated in real time as the crosshair is moved. When "Quick Cuts" is unchecked, the plot is not updated.

The size of the box is determined by the "radius" parameter.

The "Warn Level" control can be used to set a flux level above which a warning is indicated in the Cuts plot by a yellow line and the background turning yellow. The warning is triggered if any value along either the X or Y cut exceeds the warn level threshold.

The "Alert Level" control is similar, but represented by a red line and the background turning pink. The warning is triggered if any value along either the X or Y cut exceeds the alert level threshold. Alerts take precedence over warnings.

Both the "Warn" and "Alert" features can be turned off by simply setting a blank value. They are turned off by default.

The cuts plot is interactive, but it really only makes sense to use that if "Drag only" is checked. You can press 'x' or 'y' in the plot window to toggle on and off the autoaxis scaling feature for either axis, and scroll in the plot to zoom in the X axis (hold Ctrl down while scrolling to zoom the Y axis).

Crosshair provides a Pick plugin interaction feature: when the crosshair is over an object you can press 'r' in the channel viewer window to have the Pick plugin invoked on that particular location. If a Pick is not open already on that channel, it will be opened first.

User Configuration

It is customizable using ~/.ginga/plugin_Crosshair.cfg, where ~ is your HOME directory:

```
#
# Crosshair plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Crosshair.cfg"

# color of the crosshair
color = 'green'

# text color of crosshair
text_color = 'skyblue'

# box color indicating cut radius
box_color = 'aquamarine'

# cut plot line colors for X and Y
quick_h_cross_color = '#7570b3'
quick_v_cross_color = '#1b9e77'

# enable quick cuts plots by default
quick_cuts = False

# force drag only by default
drag_only = False

# set a warning level for the warning feature of the cuts plot
warn_level = None

# set an alery level for the alert feature of the cuts plot
alert_level = None

# set initial radius of the cuts box
cuts_radius = 15
```

Overlays
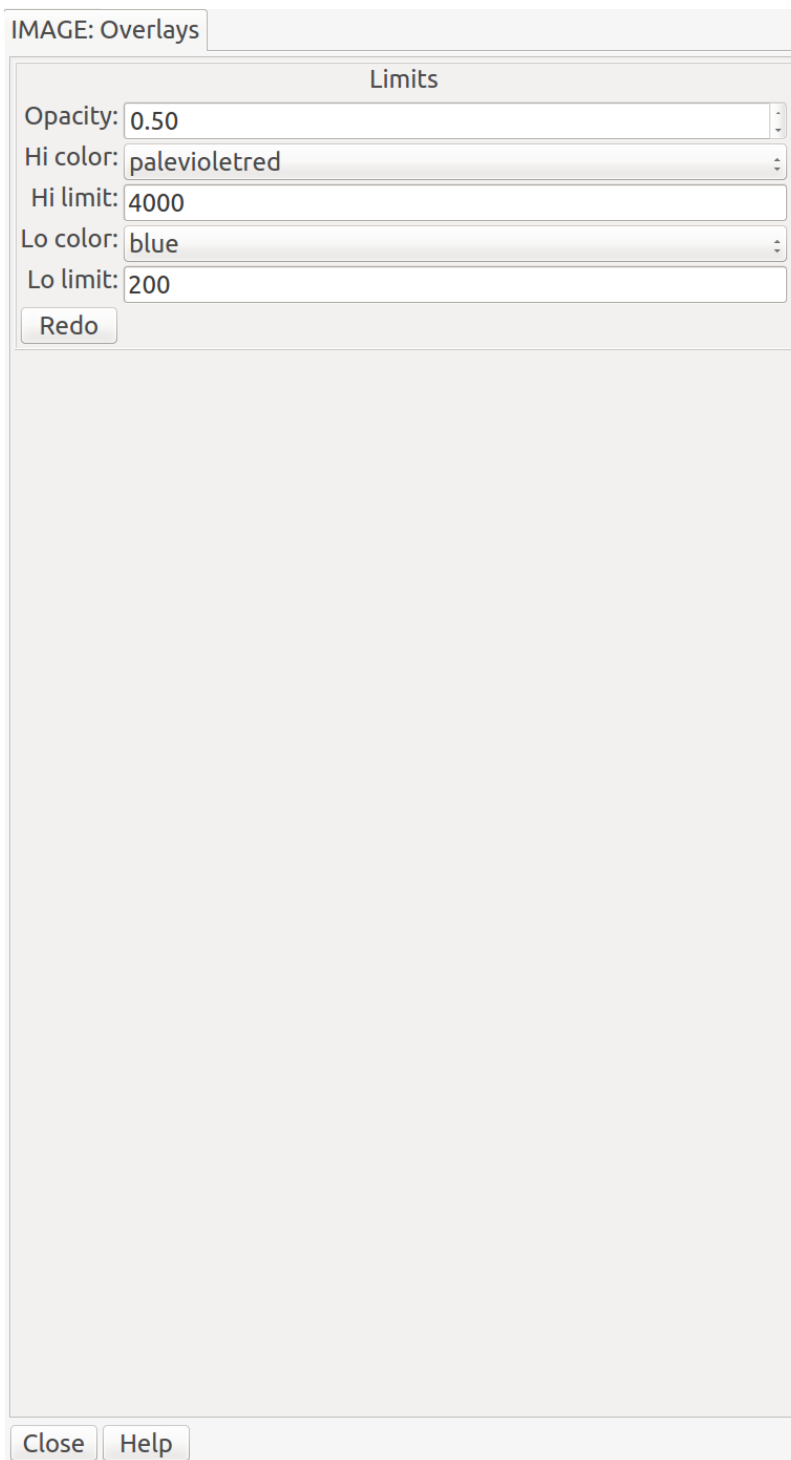
A plugin for generating color overlays representing under- and over-exposure in the loaded image.

Plugin Type: Local

Overlays is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

Choose colors from the drop-down menus for the low-limit and/or high-limit ("Lo color" and "Hi color", respectively). Specify the limits for low and high values in the limit boxes ("Lo limit" and "Hi limit", respectively). Set the opacity of the overlays with a value between 0 and 1 in the "Opacity" box. Finally, press the "Redo" button.

The color overlay should show areas below the low limit with a low color and the areas above the high limit in the high color. If you omit a limit (leave the box blank), that color won't be shown in the overlay.

If a new image is selected for the channel, the overlays image will be recalculated based on the current parameters with the new data.

WCSAxes
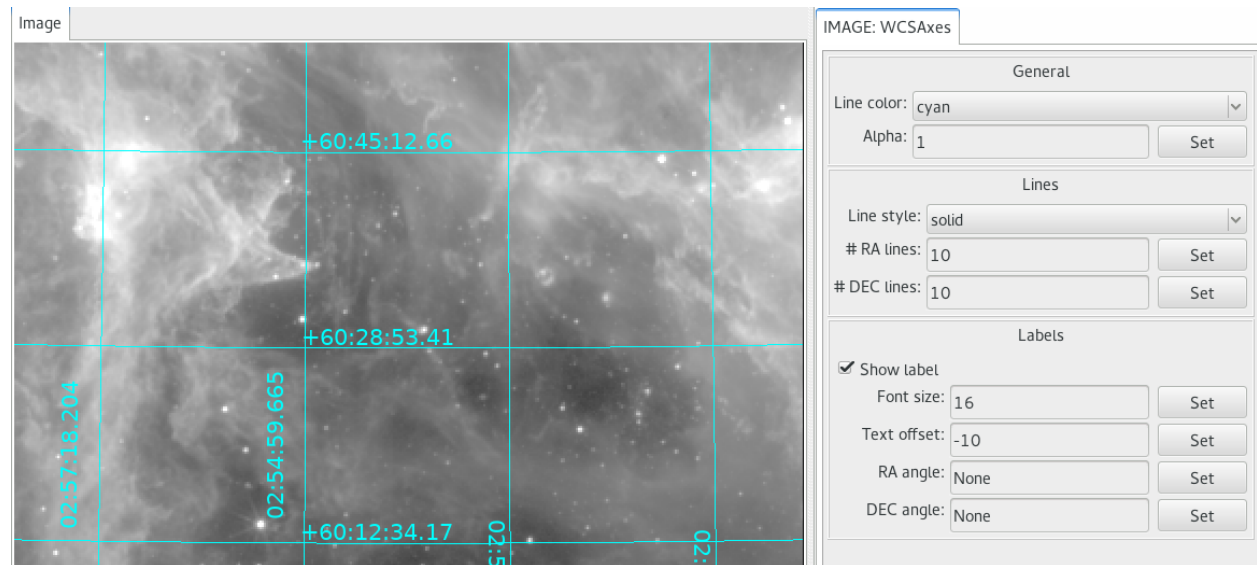
A plugin for generating WCS axes overlay in the loaded image.

Plugin Type: Local

`WCSAxes` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

As long as image as a valid WCS, WCS axes will be displayed. Use plugin GUI or configuration file to customize axes display.

It is customizable using `~/.ginga/plugin_WCSAxes.cfg`, where ~ is your HOME directory:

```
#
# WCSAxes plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_WCSAxes.cfg"

# Color, style, and width of grid lines
linecolor = 'cyan'
linestyle = 'solid'
linewidth = 1

# Alpha (opacity) of grid lines; 1=opaque, 0=transparent
alpha = 1
```

```
# Number of RA and DEC lines to draw
n_ra_lines = 10
n_dec_lines = 10

# Show RA and DEC values of grid lines
show_label = True

# Label font size
fontsize = 8

# Label offset from grid lines (pixels)
label_offset = 4
```

TVMark

Mark points from file (non-interative mode) on an image.

Plugin Type: Local

TVMark is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

This plugin allows non-interactive marking of points of interest by reading in a file containing a table with RA and DEC positions of those points. Any text or FITS table file that can be read by `astropy.table` is acceptable but user *must* define the column names correctly in the plugin configuration file (see below). An attempt will be made to convert RA and DEC values to degrees. If the unit conversion fails, they will be assumed to be in degrees already.

Alternately, if the file has columns containing the direct pixel locations, you can read these columns instead by unchecking the "Use RADEC" box. Again, the column names must be correctly defined in the plugin configuration file (see below). Pixel values can be 0- or 1-indexed (i.e., whether the first pixel is 0 or 1) and is configurable (see below). This is useful when you want to mark the physical pixels regardless of WCS (e.g., marking hot pixels on a detector). RA and DEC will still be displayed if the image has WCS information but they will not affect the markings.

To mark different groups (e.g., displaying galaxies as green circles and background as cyan crosses, as shown above):

1. Select green circle from the drop-down menus. Alternately, enter desired size or width.

2. Make sure "Use RADEC" box is checked, if applicable.

3. Using "Load Coords" button, load the file containing RA and DEC (or X and Y) positions for galaxies *only*.

4. Repeat Step 1 but now select cyan cross from the drop-down menus.

5. Repeat Step 2 but choose the file containing background positions *only*.

Selecting an entry (or multiple entries) from the table listing will highlight the marking(s) on the image. The highlight uses the same shape and color, but a slightly thicker line.

You can also highlight all the markings within a region both on the image and the table listing by drawing a rectangle on the image while this plugin is active.

Pressing the "Hide" button will hide the markings but does not clear the plugin's memory; That is, when you press "Show", the same markings will reappear on the same image. However, pressing "Forget" will clear the markings both from display and memory; That is, you will need to reload your file(s) to recreate the markings.

To redraw the same positions with different marking parameters, press "Forget" and repeat the steps above, as necessary. However, if you simply wish to change the line width (thickness), pressing "Hide" and then "Show" after you entered the new width value will suffice.

If images of very different pointings/dimensions are displayed in the same channel, markings that belong to one image but fall outside another will not appear in the latter.

To create a table that this plugin can read, one can use results from the `Pick` plugin, in addition to creating a table by hand, using `astropy.table`, etc.

Used together with `TVMask`, you can overlay both point sources and masked regions in Ginga.

It is customizable using `~/.ginga/plugin_TVMark.cfg`, where ~ is your HOME directory:

```
#
# TVMark plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_TVMark.cfg"

# Marking type -- 'circle' or 'cross'
marktype = 'circle'

# Marking color -- Any color name accepted by Ginga
markcolor = 'green'

# Marking size or radius
marksize = 5

# Marking line width (thickness)
markwidth = 1

# Specify whether pixel values are 0- or 1-indexed
```

(continues on next page)

```
pixelstart = 1

# True -- Use 'ra' and 'dec' columns to extract RA/DEC positions. This option
#          uses image WCS to convert to pixel locations.
# False -- Use 'x' and 'y' columns to extract pixel locations directly.
#          This does not use WCS.
use_radec = True

# Columns to load into table listing (case-sensitive).
# Whether RA/DEC or X/Y columns are used depend on associated GUI selection.
ra_colname = 'ra'
dec_colname = 'dec'
x_colname = 'x'
y_colname = 'y'
# Extra columns to display; e.g., ['colname1', 'colname2']
extra_columns = []
```

TVMask
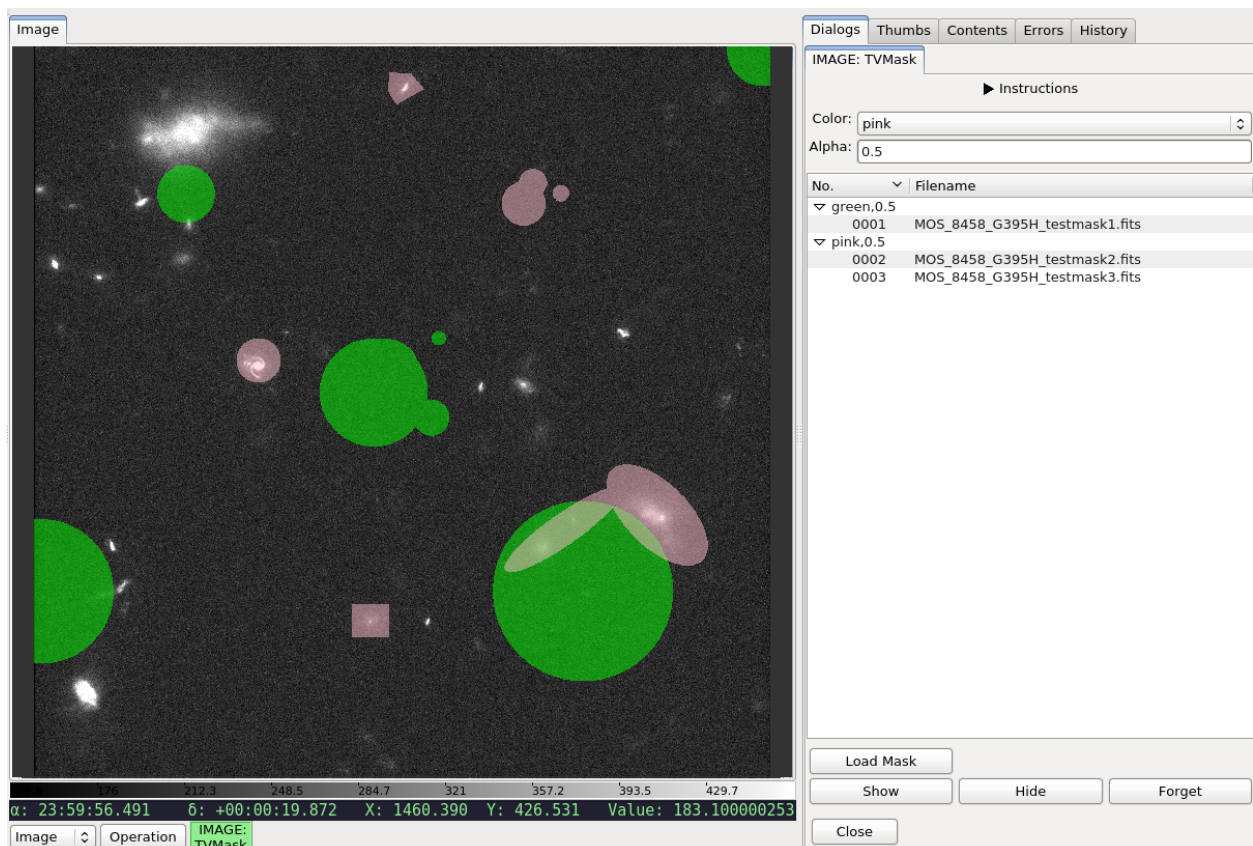
Display masks from file (non-interative mode) on an image.

Plugin Type: Local

TVMask is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

This plugin allows non-interactive display of mask by reading in a FITS file, where non-zero is assumed to be masked data.

To display different masks (e.g., some masked as green and some as pink, as shown above):

1. Select green from the drop-down menu. Alternately, enter desired alpha value.

2. Using "Load Mask" button, load the relevant FITS file.

3. Repeat (1) but now select pink from the drop-down menu.

4. Repeat (2) but choose another FITS file.

5. To display a third mask as pink too, repeat (4) without changing the drop-down menu.

Selecting an entry (or multiple entries) from the table listing will highlight the mask(s) on the image. The highlight uses a pre-defined color and alpha (customizable below).

You can also highlight all the masks within a region both on the image and the table listing by drawing a rectangle on the image while this plugin is active.

Pressing the "Hide" button will hide the masks but does not clear the plugin's memory; That is, when you press "Show", the same masks will reappear on the same image. However, pressing "Forget" will clear the masks both from display and memory; That is, you will need to reload your file(s) to recreate the masks.

To redraw the same masks with different color or alpha, press "Forget" and repeat the steps above, as necessary.

If images of very different pointings/dimensions are displayed in the same channel, masks that belong to one image but fall outside another will not appear in the latter.

To create a mask that this plugin can read, one can use results from the Drawing plugin (press "Create Mask" after drawing and save the mask using SaveImage), in addition to creating a FITS file by hand using astropy.io.fits, etc.

Used together with TVMark, you can overlay both point sources and masked regions in Ginga.

It is customizable using ~/.ginga/plugin_TVMask.cfg, where ~ is your HOME directory:

```
#
# TVMask plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_TVMask.cfg"

# Mask color -- Any color name accepted by Ginga
maskcolor = 'green'

# Mask alpha (transparency) -- 0=transparent, 1=opaque
maskalpha = 0.5

# Highlighted mask color and alpha
hlcolor = 'white'
hlalpha = 1.0
```

Blink

IMAGE: Blink

Blink

Interval: 1.0   Start Blink   Stop Blink

Max: 30.0   Min: 0.25
Mode: blink images in channel

Close   Help
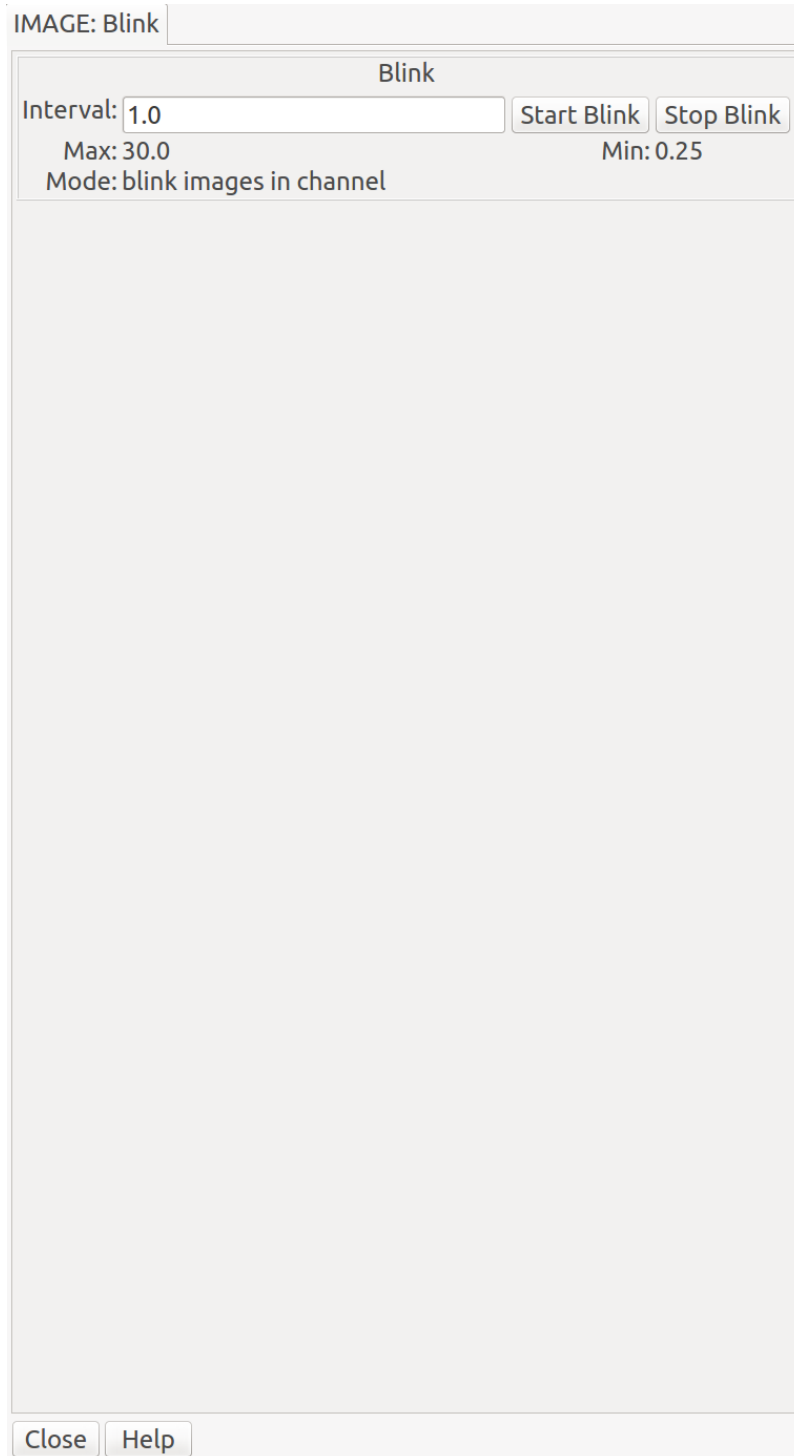
Blink switches through the images shown in a channel at a rate chosen by the user. Alternatively, it can switch between channels in the main workspace. In both cases, the primary purpose is to compare and contrast the images (within a channel, or across channels) visually within a short timescale – like blinking your eyes.

Plugin Type: Local or Global

Blink can be invoked either as a local plugin, in which case it cycles through the images in the channel, or as a global

plugin, in which case it cycles through the channels.

Local plugins are started from the "Operations" button, while global plugins are started from the "Plugins" menu.

Usage

Set the interval between image changes in terms of seconds in the box labeled "Interval". Then, press "Start Blink" to start the timed cycling, and "Stop Blink" to stop the cycling.

You can change the number in "Interval" and press Enter to dynamically change the cycle time while the cycle is running.

It is customizable using ~/.ginga/plugin_Blink.cfg, where ~ is your HOME directory:

```
#
# Blink plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Blink.cfg"

# Blink channels instead of images within a channel
# PLEASE NOTE: this option is DEPRECATED
blink_channels = False

# the maximum interval for a blink
interval_max = 30.0

# the minimum interval for a blink
interval_min = 0.25
```

LineProfile
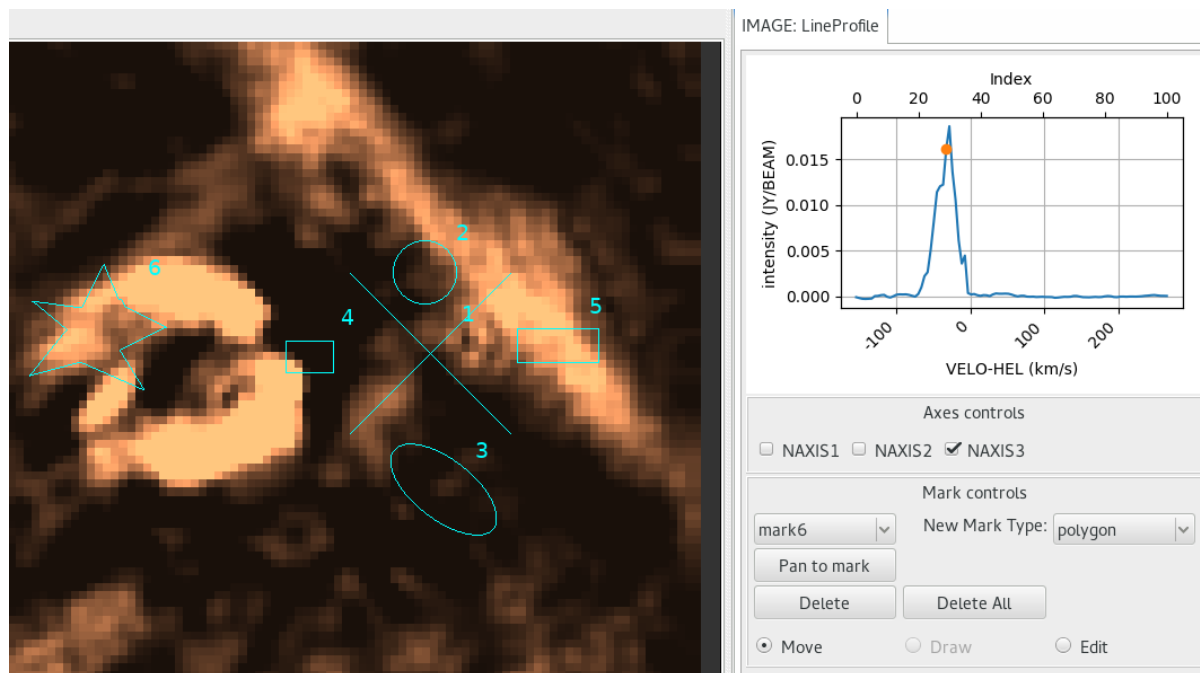
plugin to graph the pixel values along a straight line bisecting a cube.

Plugin Type: Local

`LineProfile` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

> **Warning:** There are no restrictions to what axes can be chosen. As such, the plot can be meaningless.

The `LineProfile` plugin is used for multidimensional (i.e., 3D or higher) images. It plots the values of the pixels at the current cursor position through the selected axis; or if a region is selected, it plots the mean in each frame. This can be used to create normal spectral line profiles. A marker is placed at the data point of the currently displayed frame.

Displayed X-axis is constructed using `CRVAL*`, `CDELT*`, `CRPIX*`, `CTYPE*`, and `CUNIT*` keywords from FITS header. If any of the keywords are unavailabled, the axis falls back to `NAXIS*` values instead.

Displayed Y-axis is constructed using `BTYPE` and `BUNIT`. If they are not available, it simply labels pixel values as "Signal".

To use this plugin:

1. Select an axis.

2. Pick a point or draw a region using the cursor.

3. Use `MultiDim` to change step values of axes, if applicable.

It is customizable using `~/.ginga/plugin_LineProfile.cfg`, where ~ is your HOME directory:

```
#
# LineProfile plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_LineProfile.cfg"

# Default mark type. This can be change in the GUI.
mark_type = 'point'

# Properties of a point mark type. Radius can be change in Edit mode.
mark_radius = 10
mark_style='cross'

# Mark color.
mark_color = 'cyan'
```

PixTable
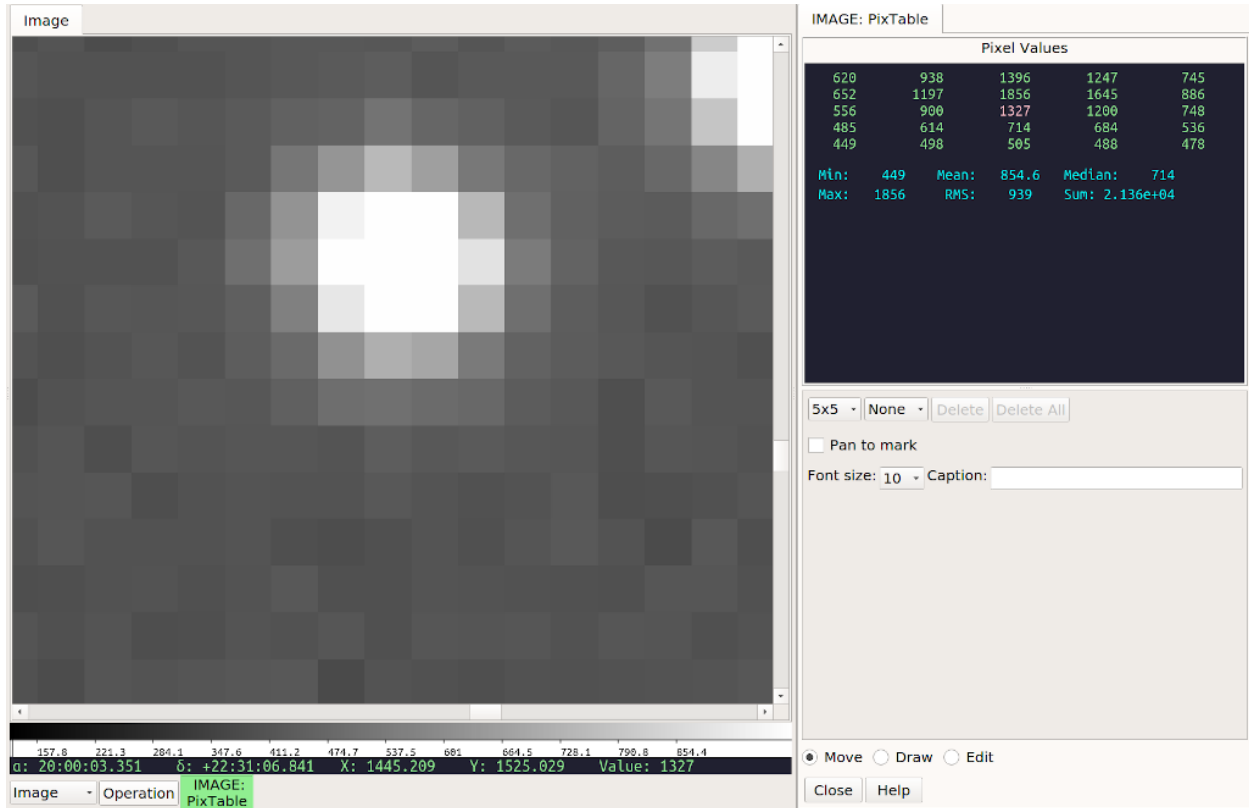
`PixTable` provides a way to check or monitor the pixel values in a region.

Plugin Type: Local

`PixTable` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Basic Use

In the most basic use, simply move the cursor around the channel viewer; an array of pixel values will appear in the "Pixel Values" display in the plugin UI. The center value is highlighted, and this corresponds to the value under the cursor.

You can choose a 3x3, 5x5, 7x7, or 9x9 grid from the left-most combobox control. It may help to adjust the "Font Size" control to prevent having the array values cut off on the sides. You can also enlarge the plugin workspace to see more of the table.

Note: The order of the value table shown will not necessarily match to the channel viewer if the images is flipped, transposed, or rotated.

Using Marks

When you set and select a mark, the pixel values will be shown surrounding the mark instead of the cursor. There can be any number of marks, and they are each noted with a numbered "X". Simply change the mark drop down control to select a different mark and see the values around it. The currently selected mark is shown with a different color than the others.

The marks will stay in position even if a new image is loaded and they will show the values for the new image. In this way you can monitor the area around a spot if the image is updating frequently.

If the "Pan to mark" checkbox is selected, then when you select a different mark from the mark control, the channel viewer will pan to that mark. This can be useful to inspect the same spots in several different images, especially when zoomed in tight to the image.

Note: If you change the mark control back to "None", then the pixel table will again update as you move the cursor around the viewer.

The "Caption" box can be used to set a text annotation that will be appended to the mark label when the next mark is created. This can be used to label a feature in the image, for example.

Deleting Marks

To delete a mark, select it in the mark control and then press the button marked "Delete". To delete all the marks, press the button marked "Delete All".

Moving Marks

When the "Move" radio button is checked, and a mark is selected, then clicking or dragging anywhere in the image will move the mark to that location and update the pixel table. If no mark is currently selected then a new one will be created and moved.

Drawing Marks

When the "Draw" radio button is checked, then clicking and dragging creates a new mark. The longer the draw, the bigger radius of the "X".

Editing Marks

When the "Edit" radio button is checked after a mark has been selected then you can drag the control points of the mark to increase the radius of the arms of the X or you can drag the bounding box to move the mark. If the editing control points are not shown, simply click on the center of a mark to enable them.

Special Keys

In "Move" mode the following keys are active: - "n" will place a new mark at the site of the cursor - "m" will move the current mark (if any) to the site of the cursor - "d" will delete the current mark (if any) - "j" will select the previous mark (if any) - "k" will select the next mark (if any)

User Configuration

It is customizable using `~/.ginga/plugin_PixTable.cfg`, where ~ is your HOME directory:

```
#
# PixTable plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_PixTable.cfg"

# Default font
font = 'fixed'

# Default font size
fontsize = 12

# default size for mark point radius
mark_radius = 10

# style of point to draw
mark_style = 'cross'
```

(continues on next page)

```
# color of non-selected marks
mark_color = 'purple'

# color of selected mark
select_color = 'cyan'

# whether to update the pixel table when moving a mark around
drag_update = True
```

Preferences

IMAGE: Preferences

Color Distribution

Algorithm: linear ▼

Dist Defaults

Color Mapping

Colormap: gray ▼

Intensity: ramp ▼

Color Defaults

Auto Cuts

Cut Low: 268.5

Cut High: 699.7

Cut Levels

Auto Method: zscale ▼   Auto Levels

contrast: 0.25

num_points: 1000

Transform

☐ Flip X   ☐ Flip Y   ☐ Swap XY

Rotate: 0.00000000 ▲▼

Restore

WCS

WCS Coords: icrs ▼

WCS Display: sexagesimal ▼

Zoom

Zoom Alg: Rate ▼

Zoom Rate: 1.61421356 ▲▼

Stretch XY: x ▼

Stretch Factor: 1.00000000 ▲▼

Scale X: 0.15585278964141247   Set

Scale Y: 0.15585278964141247   Set

Scale Min: None   Set

Scale Max: None   Set

Interpolation: basic ▼

Zoom Defaults

Panning

Pan X: 1136.5   ☐ WCS sexagesimal

Pan Y: 2137.0   Apply Pan

Pan Coord: data ▼

Center Image   ☐ Mark Center

▶ General

▶ Reset (Viewer)

▶ Remember (Image)

▼ New Images

Cut New: on ▼

Zoom New: off ▼

Center New: off ▼

☑ Follow New   ☑ Raise New

☑ Create thumbnail   ☐ Auto Orient

▶ ICC Profiles

Close   Help   Save Settings   Update Viewer Profile

4.5. Plugins

Make changes to channel settings graphically in the

Plugin Type: Local

`Preferences` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

The `Preferences` plugin sets the preferences *on a per-channel basis*. The preferences for a given channel are inherited from the "Image" channel until they are explicitly set and saved using this plugin.

If "Save Settings" is pressed, it will save the settings to the user's $HOME/.ginga folder (a "channel_NAME.cfg" file for each channel NAME) so that when a channel with the same name is created in future Ginga sessions it will obtain the same settings.

Color Distribution Preferences
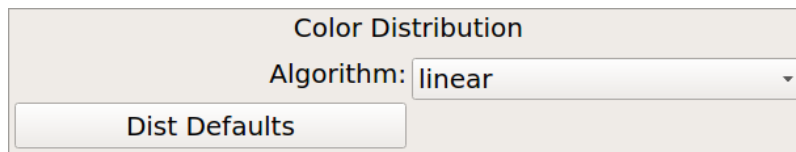
Fig. 14: "Color Distribution" preferences.

The "Color Distribution" preferences control the preferences used for the data value to color index conversion that occurs after cut levels are applied and just before final color mapping is performed. It concerns how the values between the low and high cut levels are distributed to the color and intensity mapping phase.

The "Algorithm" control is used to set the algorithm used for the mapping. Click the control to show the list, or simply scroll the mouse wheel while hovering the cursor over the control. There are eight algorithms available: linear, log, power, sqrt, squared, asinh, sinh, and histeq. The name of each algorithm is indicative of how the data is mapped to the colors in the color map. "linear" is the default.

Color Mapping Preferences
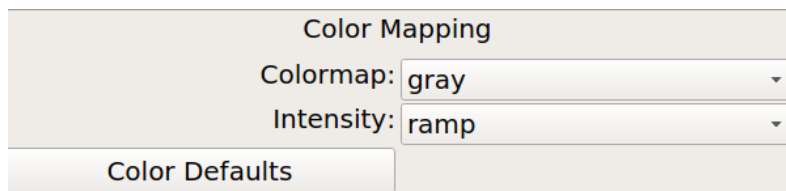
Fig. 15: "Color Mapping" preferences.

The "Color Mapping" preferences control the preferences used for the color map and intensity map, used during the final phase of the color mapping process. Together with the "Color Distribution" preferences, these control the mapping of data values into a 24-bpp RGB visual representation.

The "Colormap" control selects which color map should be loaded and used. Click the control to show the list, or simply scroll the mouse wheel while hovering the cursor over the control.

The "Intensity" control selects which intensity map should be used with the color map. The intensity map is applied just before the color map, and can be used to change the standard linear scale of values into an inverted scale, logarithmic, etc.

Ginga comes with a good selection of color maps, but should you want more, you can add custom ones or, if `matplotlib` is installed, you can load all the ones that it has. See "Customizing Ginga" for details.

Auto Cuts Preferences

Fig. 16: "Auto Cuts" preferences.

The "Auto Cuts" preferences control the calculation of cut levels for the view when the auto cut levels button or key is pressed, or when loading a new image with auto cuts enabled. You can also set the cut levels manually from here.

The "Cut Low" and "Cut High" fields can be used to manually specify lower and upper cut levels. Pressing "Cut Levels" will set the levels to these values manually. If a value is missing, it is assumed to default to the whatever the current value is.

Pressing "Auto Levels" will calculate the levels according to an algorithm. The "Auto Method" control is used to choose which auto cuts algorithm used: "minmax" (minimum maximum values), "median" (based on median filtering), "histogram" (based on an image histogram), "stddev" (based on the standard deviation of pixel values), or "zscale" (based on the ZSCALE algorithm popularized by IRAF). As the algorithm is changed, the boxes under it may also change to allow changes to parameters particular to each algorithm.

Transform Preferences

Fig. 17: "Transform" preferences.

The "Transform" preferences provide for transforming the view of the image by flipping the view in X or Y, swapping the X and Y axes, or rotating the image in arbitrary amounts.

The "Flip X" and "Flip Y" checkboxes cause the image view to be flipped in the corresponding axis.

The "Swap XY" checkbox causes the image view to be altered by swapping the X and Y axes. This can be combined with "Flip X" and "Flip Y" to rotate the image in 90 degree increments. These views will render more quickly than arbitrary rotations using the "Rotate" control.

The "Rotate" control will rotate the image view the specified amount. The value should be specified in degrees. "Rotate" can be specified in conjunction with flipping and swapping.

The "Restore" button will restore the view to the default view, which is unflipped, unswapped, and unrotated.

WCS Preferences

The "WCS" preferences control the display preferences for the World Coordinate System (WCS) calculations used to report the cursor position in the image.

The "WCS Coords" control is used to select the coordinate system in which to display the result.

Fig. 18: "WCS" preferences.

The "WCS Display" control is used to select a sexagesimal (`H:M:S`) readout or a decimal degrees readout.

Zoom Preferences

Fig. 19: "Zoom" preferences.

The "Zoom" preferences control Ginga's zooming/scaling behavior. Ginga supports two zoom algorithms, chosen using the "Zoom Alg" control:

- The "step" algorithm zooms the image inwards in discrete steps of 1X, 2X, 3X, etc. or outwards in steps of 1/2X, 1/3X, 1/4X, etc. This algorithm results in the least artifacts visually, but is a bit slower to zoom over wide ranges when using a scrolling motion because more "throw" is required to achieve a large zoom change (this is not the case if one uses of the shortcut zoom keys, such as the digit keys).

- The "rate" algorithm zooms the image by advancing the scaling at a rate defined by the value in the "Zoom Rate" box. This rate defaults to the square root of 2. Larger numbers cause larger changes in scale between zoom levels. If you like to zoom your images rapidly, at a small cost in image quality, you would likely want to choose this option.

Note that regardless of which method is chosen for the zoom algorithm, the zoom can be controlled by holding down `Ctrl` (coarse) or `Shift` (fine) while scrolling to constrain the zoom rate (assuming the default mouse bindings).

The "Stretch XY" control can be used to stretch one of the axes (X or Y) relative to the other. Select an axis with this control and roll the scroll wheel while hovering over the "Stretch Factor" control to stretch the pixels in the selected axis.

The "Scale X" and "Scale Y" controls offer direct access to the underlying scaling, bypassing the discrete zoom steps. Here, exact values can be typed to scale the image. Conversely, you will see these values change as the image is zoomed.

The "Scale Min" and "Scale Max" controls can be used to place a limit on how much the image can be scaled.

The "Interpolation" control allows you to choose how the image will be interpolated. Depending on which support packages are installed, the following choices can be made:

- "basic" is nearest-neighbor using a built in algorithm, this is always available, is reasonably fast, and is the default.

- "area"

- "bicubic"

- "lanczos"

- "linear"

- "nearest" is nearest-neighbor (using support package)

The "Zoom Defaults" button will restore the controls to the Ginga default values.

Pan Preferences

Fig. 20: "Pan" preferences.

The "Pan" preferences control Ginga's panning behavior.

The "Pan X" and "Pan Y" controls offer direct access to set the pan position in the image (the part of the image located at the center of the window) – you can see them change as you pan around the image. You can set these values and then press "Apply Pan" to pan to that exact position.

If the "Pan Coord" control is set to "data" then panning is controlled by data coordinates in the image; if set to "WCS" then the values shown in the "Pan X" and "Pan Y" controls will be WCS coordinates (assuming a valid WCS is in the image). In the latter case, the "WCS sexagesimal" control can be left unchecked to show/set the coordinates in degrees, or checked to show/set the values in standard sexigesimal notation.

The "Center Image" button sets the pan position to the center of the image, as calculated by halving the dimensions in X and Y.

The "Mark Center" check box, when checked, will cause Ginga to draw a small reticle in the center of the image. This is useful for knowing the pan position and for debugging.

General Preferences

Fig. 21: "General" preferences.
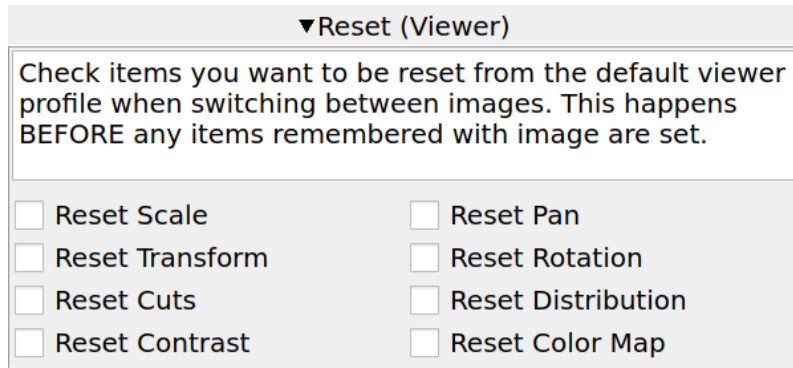
The "Num Images" setting specifies how many images can be retained in buffers in this channel before being ejected. A value of zero (0) means unlimited–images will never be ejected. If an image was loaded from some accessible storage and it is ejected, it will automatically be reloaded if the image is revisited by navigating the channel.

The "Sort Order" setting determines whether images are sorted in the channel alphabetically by name or by the time when they were loaded. This principally affects the order in which images are cycled when using the up/down "arrow"

keys or buttons, and not necessarily how they are displayed in plugins like "Contents" or "Thumbs" (which generally have their own setting preference for ordering).

The "Use scrollbars" check box controls whether the channel viewer will show scroll bars around the edge of the viewer frame to pan the image.

Reset (Viewer) Preferences

Fig. 22: "Reset" (Viewer) preferences.

Each channel viewer has a *viewer profile* that is initialized to the state of the viewer just after creation and the restoration of saved settings for that channel. When switching between images, the attributes of the viewer can be reset to this profile according to the checked boxes in this section. *If nothing is checked, nothing will be reset from the viewer profile*.

To use this feature, set your viewer preferences as you prefer and click the "Update Viewer Profile" button at the bottom of the plugin. Now check which items should be reset to those values between images. Finally, click the "Save Settings" button at the bottom if you want these settings to be persistent across Ginga restarts and set as the default user profile for this channel when you restart ginga and recreate this channel.

- "Reset Scale" will reset the zoom (scale) level to the viewer profile

- "Reset Pan" will reset the pan position to the viewer profile

- "Reset Transform" will reset any flip/swap transforms to the viewer profile

- "Reset Rotation" will reset any rotation to the viewer profile

- "Reset Cuts" will reset any cut levels to the viewer profile

- "Reset Distribution" will reset any color distribution to the viewer profile

- "Reset Contrast" will reset any contrast/bias to the viewer profile

- "Reset Color Map" will reset any color map settings to the viewer profile

---

Tip: If you use this feature you may also want to set "Remember (Image) Preferences" (see below).

Note: The complete order of adjustments is:

- any reset items from the default viewer profile, if any

- any remembered items from the image profile are applied, if any

- any auto adjustments (cuts/zoom/center) are applied, if they were not overridden by a remembered setting

---

Remember (Image) Preferences

Fig. 23: "Remember" (Image) preferences.

When an image is loaded, an *image profile* is created and attached to the image metadata in the channel. These profiles are continuously updated with viewer state as the image is manipulated. The "Remember" preferences control which attributes of these profiles are restored to the viewer state when the image is navigated (back) to in the channel:

- "Remember Scale" will restore the zoom (scale) level of the image

- "Remember Pan" will restore the pan position in the image

- "Remember Transform" will restore any flip or swap axes transforms

- "Remember Rotation" will restore any rotation of the image

- "Remember Cuts" will restore any cut levels for the image

- "Remember Distribution" will restore any color distribution (linear,log,etc)

- "Remember Contrast" will restore any contrast/bias adjustment

- "Remember Color Map" will restore any color map choices made

If nothing is checked, nothing will be restored from the image profile.

Note: These items will be set BEFORE any auto (cut/zoom/center new) adjustments are made. If a remembered item is set, it will override any auto adjustment setting for the channel.

Tip: If you use this feature you may also want to set "Reset (Viewer) Preferences" (see above).

An Example

As an example of using the Reset and Remember settings, suppose that you frequently use the contrast adjustment. You would like the contrast that you set with a particular image to be restored when that image is viewed again. However, when you view a new image, you would like the contrast to start out at some normal setting.

To accomplish this, manually reset the contrast to the desired default setting. Check "Reset Contrast" and then press "Update Viewer Profile". Finally, check "Remember Contrast". Click "Save Settings" to make the channel settings persistent.

New Image Preferences

The "New Images" preferences determine how Ginga reacts when a new image is loaded into the channel. *This includes when an older image is revisited by clicking on its thumbnail in the ``Thumbs`` plugin or double-clicking on it's name in the ``Contents`` plugin.*
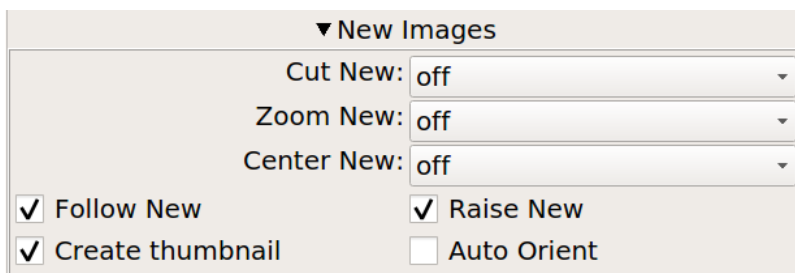
Fig. 24: "New Image" preferences.

The "Cut New" setting controls whether an automatic cut-level calculation should be performed on the new image, or whether the currently set cut levels should be applied. The possible settings are:

- "off": always use the currently set cut levels;

- "once": calculate a new cut levels for the first image visited, then turn "off";

- "override": calculate a new cut levels until the user overrides it by manually setting a cut levels, then turn "off"; or

- "on": calculate a new cut levels always.

Tip: The "override" setting is provided for the convenience of having automatic cut levels, while preventing a manually set cuts from being overridden when a new image is ingested. When typed in the image window, the semicolon key can be used to toggle the mode back to override (from "off"), while colon will set the preference to "on". The `Info` (tab: Synopsis) plugin shows the state of this setting.

The "Zoom New" setting controls whether visiting an image should set the zoom level to fit the image to the window. The possible settings are:

- "off": always use the currently set zoom levels;

- "once": fit the first image to the window, then turn to "off";

- "override": images are automatically fitted until the zoom level is changed manually, then the mode automatically changes to "off", or

- "on": the new image is always zoomed to fit.

Tip: The "override" setting is provided for the convenience of having an automatic zoom, while preventing a manually set zoom level from being overridden when a new image is ingested. When typed in the image window, the apostrophe (a.k.a. "single quote") key can be used to toggle the mode back to "override" (from "off"), while quote (a.k.a. double quote) will set the preference to "on". The `Info` (tab: Synopsis) plugin shows the state of this setting.

The "Center New" setting controls whether visiting an image should cause the pan position to be reset to the center of the image. The possible settings are:

- "off": leave the current pan position as is;

- "once": center the first image visited, then turn to "off";

- "override": images are automatically centered until the pan position is changed manually, then the mode automatically changes to "off", or

- "on": the new image is always centered.

The "Follow New" setting is used to control whether Ginga will change the display if a new image is loaded into the channel. If unchecked, the image is loaded (as seen, for example, by its appearance in the `Thumbs` tab), but the display will not change to the new image. This setting is useful in cases where new images are being loaded by some automated means into a channel and the user wishes to study the current image without being interrupted.

The "Raise New" setting controls whether Ginga will raise the tab of a channel when an image is loaded into that channel. If unchecked, then Ginga will not raise the tab when an image is loaded into that particular channel.

The "Create Thumbnail" setting controls whether Ginga will create a thumbnail for images loaded into that channel. In cases where many images are being loaded into a channel frequently (e.g., a low frequency video feed), it may be undesirable to create thumbnails for all of them.

The "Auto Orient" setting controls whether Ginga should attempt to orient images by default according to image metadata. This is currently only useful for RGB (e.g. JPEG) images that contain such metadata. It does not auto orient by WCS, at present.

ICC Profiles Preferences
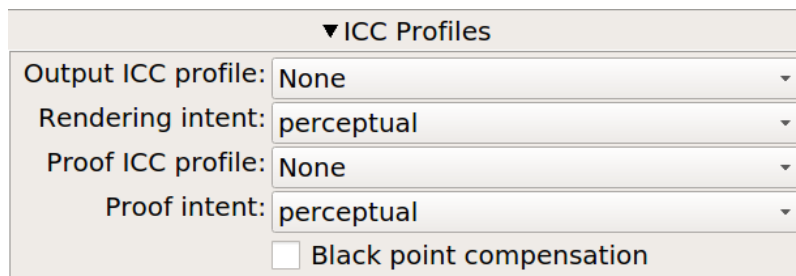
Fig. 25: "ICC Profiles" preferences.

Ginga can make use of ICC (color management) profiles in the rendering chain using the LittleCMS library.

Note: To make use of ICC profiles, create a "profiles" folder in the Ginga "home" (usually $HOME/.ginga) and put any necessary profiles there. A working profile should be set by adding a value for "icc_working_profile" in your $HOME/.ginga/general.cfg file– do not include any leading path, just the filename of an ICC file in the profiles folder. This will be used to convert any RGB files containing a profile to the working profile.

You can set the output profiles for any channel in this section of the Preferences plugin.

The "Output ICC profile" control selects which profile to use for the output rendering to the display. The choices are from your profile files in $HOME/.ginga/profiles. Normally this should be a display profile.
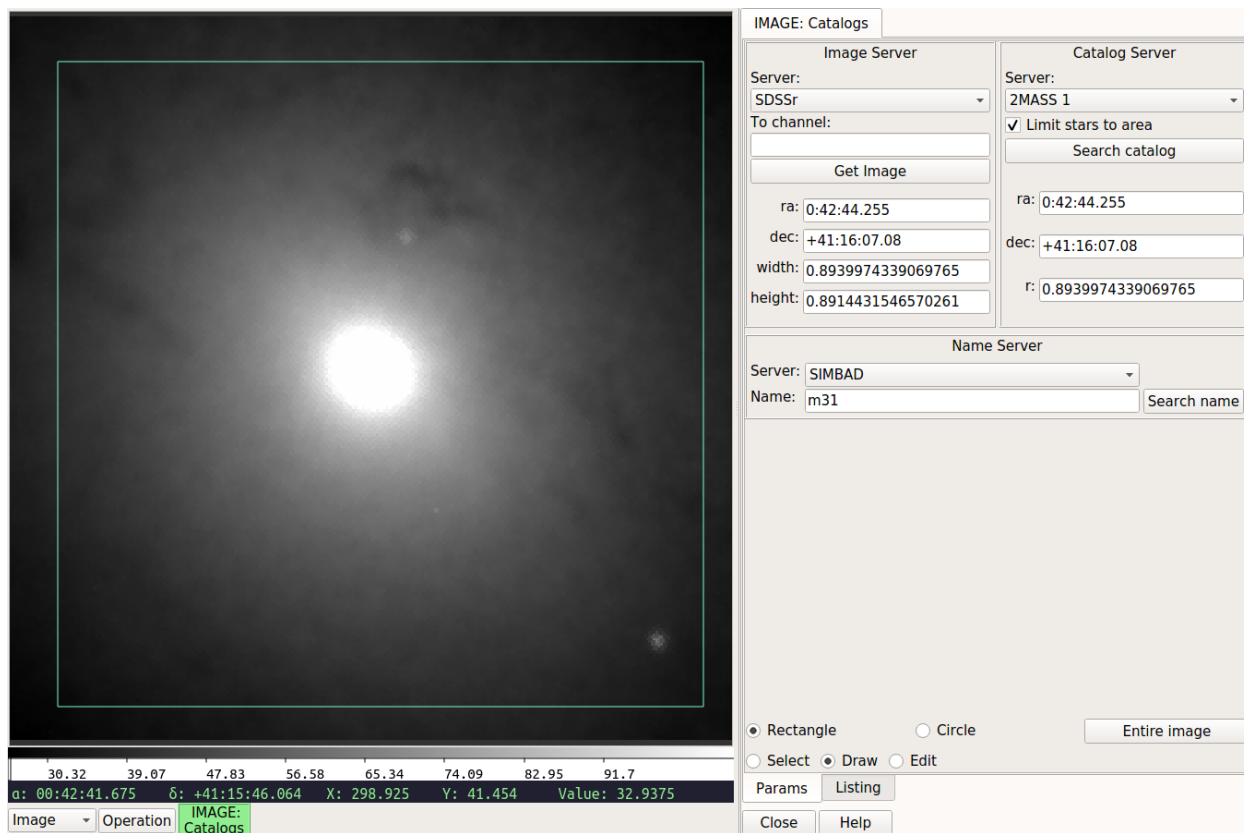
The "Rendering intent" control chooses the algorithm used to render the color in the ICC conversion process. The choices are:

- absolute_colorimetric
- perceptual
- relative_colorimetric
- saturation

The "Proof ICC profile" and "Proof intent" are similarly chosen for proofing.

The "Black point compensation" checkbox turns on or off this feature in the color conversion process. See the documentation for LittleCMS or ICC color management in general for details on these choices.

Catalogs

A plugin for plotting object locations from a catalog on an image.

Plugin Type: Local

`Catalogs` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Note: To use `Catalogs`, it is necessary to install the `astroquery` package.

Warning: Configuration of `Catalogs` via `ginga_config.py` technique in Ginga 3.2 or later is not officially supported and may not work as in previous releases. See the new user configuration instructions below.

Usage

Fetching an image

- By name resolver: Using the Name Server box, choose a server and type a name into the "Name" field. Press "Search name". If the name is resolved, the "ra" and "dec" fields in the Image Server box will be populated. Select a server, adjust width and/or height, and press "Get Image".

- By existing image in the channel: Draw a shape on the displayed image ("rectangle" or "circle" can be chosen at the bottom of the plugin GUI) and adjust search parameters as desired. When you are ready, press "Get Image" to perform the search.

Note: It can take some time for the image download to finish, depending on the size of the field, network conditions, etc. Normally, if the search or download fails then an error will be popped up in the Errors plugin.

If the image is downloaded successfully it should appear in the channel viewer.

Fetching and plotting objects from catalogs

To plot objects, the Catalogs plugin needs an image with a valid WCS loaded into the channel. You can either load your own image or fetch one from an image server as described in "Fetching an image" above.

Choosing a center:

- By name resolver: Using the Name Server box, choose a server and type a name into the "Name" field. Press "Search name". If the name is resolved, the "ra" and "dec" fields in the Catalog Server box will be populated. Select a server, adjust width and/or height, and press "Search catalog".

- By existing image in the channel: Draw a shape on the displayed image ("rectangle" or "circle" can be chosen at the bottom of the plugin GUI) and adjust search parameters as desired. When you are ready, press "Search catalog" to perform the search.

Note: It can take some time for the search result to finish, depending on the size of the field, network conditions, etc. Normally, if the search fails then an error will be popped up in the Errors plugin.

When search results are available, they will be displayed on the image and also listed in a table on the plugin GUI. You can click on either the table or the image to highlight selection.

User Configuration

It is customizable using ~/.ginga/plugin_Catalogs.cfg, where ~ is your HOME directory:

```
#
# Catalogs plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Catalogs.cfg"

draw_type = 'circle'

select_color = 'skyblue'

color_outline = 'aquamarine'

click_radius = 10


# NAME SOURCES
# Name resolvers for astronomical object names
#
# Format: list of dicts
# Each dict defines a source, and has the following fields:
#   shortname: str
#       the short name appearing in the control for selecting a source
#       in the plugin
#
#   fullname: str
#       the full name, should correspond *exactly* with the name required
#       by astroquery.vo_conesearch "catalog" parameter
#
#   type: str
#       should be "astroquery.names" for an astroquery.names function
#
```

```
name_sources = [
{'shortname': "SIMBAD", 'fullname': "SIMBAD",
'type': 'astroquery.names'},
{'shortname': "NED", 'fullname': "NED",
'type': 'astroquery.names'},
]


# CATALOG SOURCES
#
# Format: list of dicts
# Each dict defines a source, and has the following fields:
#   shortname: str
#       the short name appearing in the control for selecting a source
#       in the plugin
#
#   fullname: str
#       the full name, should correspond *exactly* with the name required
#       by astroquery.vo_conesearch "catalog" parameter
#
#   type: str
#       should be "astroquery.vo_conesearch" for an astroquery.vo_conesearch
#       function
#
#   mapping: dict
#       a nested dict providing the mapping for the return results to the GUI,
#       in terms of field name to Ginga table.
#       There must be keys for 'id', 'ra' and 'dec'. 'mag', if present, can be
#       a list of field names that define magnitudes of the elements in various
#       wavelengths.
#
catalog_sources = [
{'shortname': "GSC 2.3",
'fullname': "Guide Star Catalog 2.3 Cone Search 1",
'type': 'astroquery.vo_conesearch',
'mapping': {'id': 'objID', 'ra': 'ra', 'dec': 'dec', 'mag': ['Mag']}},
{'shortname': "USNO-A2.0 1",
'fullname': "The USNO-A2.0 Catalogue (Monet+ 1998) 1",
'type': 'astroquery.vo_conesearch',
'mapping': {'id': 'USNO-A2.0', 'ra': 'RAJ2000', 'dec': 'DEJ2000',
'mag': ['Bmag', 'Rmag']}},
{'shortname': "2MASS 1",
'fullname': "Two Micron All Sky Survey (2MASS) 1",
'type': 'astroquery.vo_conesearch',
'mapping': {'id': 'htmID', 'ra': 'ra', 'dec': 'dec',
'mag': ['h_m', 'j_m', 'k_m']}},
]


# IMAGE SOURCES
#
# Format: list of dicts
```

```
# Each dict defines a source, and has the following fields:
#   shortname: str
#       the string that should correspond *exactly* with the name required
#       by astroquery.skyview "survey" parameter for get_image_list()
#
#   fullname: str
#       the full name, mostly descriptive
#
#   type: str
#       should be "astroquery.image"
#
#   source: str
#       should be "skyview"
#
#
image_sources = [
{'shortname': "DSS",
'fullname': "Digital Sky Survey 1",
'type': 'astroquery.image',
'source': 'skyview'},
{'shortname': "DSS1 Blue",
'fullname': "Digital Sky Survey 1 Blue",
'type': 'astroquery.image',
'source': 'skyview'},
{'shortname': "DSS1 Red",
'fullname': "Digital Sky Survey 1 Red",
'type': 'astroquery.image',
'source': 'skyview'},
{'shortname': "DSS2 Red",
'fullname': "Digital Sky Survey 2 Red",
'type': 'astroquery.image',
'source': 'skyview'},
{'shortname': "DSS2 Blue",
'fullname': "Digital Sky Survey 2 Blue",
'type': 'astroquery.image',
'source': 'skyview'},
{'shortname': "DSS2 IR",
'fullname': "Digital Sky Survey 2 Infrared",
'type': 'astroquery.image',
'source': 'skyview'},
]
```

Mosaic
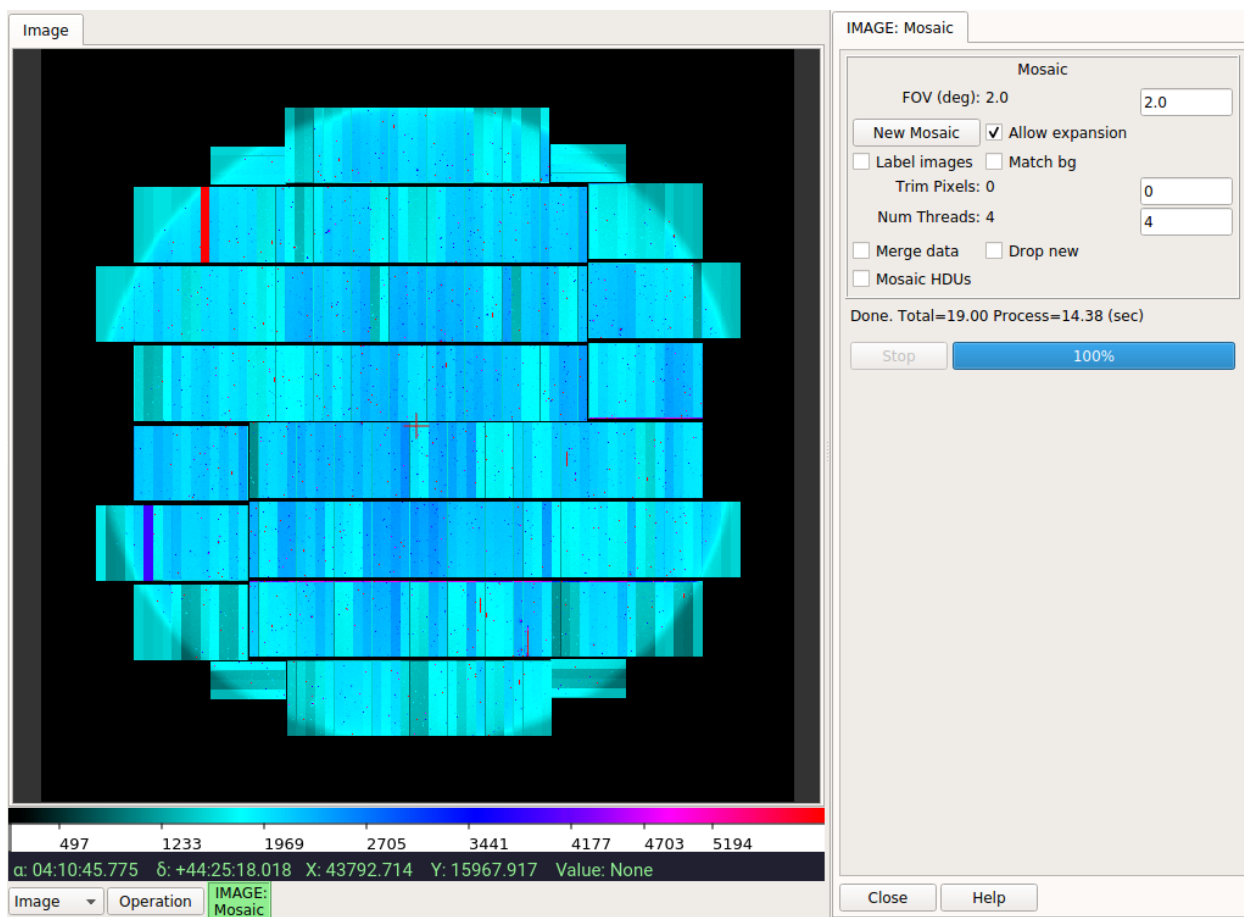
Plugin to create an image mosaic by constructing a composite image.

Plugin Type: Local

`Mosaic` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

> **Warning:** This can be very memory intensive.

This plugin is used to automatically build a mosaic image in the channel using images provided by the user (e.g., using `FBrowser`). The position of an image on the mosaic is determined by its WCS without distortion correction. This is meant as a quick-look tool, not a replacement for image drizzling that takes account of image distortion, etc. The mosaic only exists in memory but you can save it out to a FITS file using `SaveImage`.

When a mosaic falls out of memory, it is no longer accessible in Ginga. To avoid this, you must configure your session such that your Ginga data cache is sufficiently large (see "Customizing Ginga" in the manual).

To create a new mosaic, set the FOV and drag files onto the display window. Images must have a working WCS. The first image's WCS will be used to orient the other tiles.

Difference from `Collage` plugin

- Allocates a single large array to hold all the mosaic contents

- Slower to build, but can be quicker to manipulate large resultant images

- Can save the mosaic as a new data file

- Fills in values between tiles with a fill value (can be `NaN`)

It is customizable using `~/.ginga/plugin_Mosaic.cfg`, where ~ is your HOME directory:

```
#
# Mosaic plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Mosaic.cfg"

# annotate images with their names
annotate_images = False

# default FOV for new mosaics
fov_deg = 0.2

# Try to match backgrounds
match_bg = False

# Number of pixels to trim from edges
trim_px = 0

# Merge (coadd pixels) instead of overlapping tiles
merge = False

# Number of threads to devote to opening images
num_threads = 4

# dropping a new file or files starts a new mosaic
drop_creates_new_mosaic = False

# Set to True when you want to mosaic image HDUs in a file
mosaic_hdus = False

# Limit on skew between X and Y axis after warping tile acceptable for mosaic
skew_limit = 0.1

# Allow the mosaic image to expand if new tiles are added that
# aren't in the region
allow_expand = True

# When expanding an image, pad on a side by this many deg
expand_pad_deg = 0.01

# Maximum delta from center of image (in deg) beyond which new images
# are assumed to start a new mosaic. Set to None if you never want this
# bahavior
max_center_deg_delta = 2.0

# Allow mosaic images to create thumbnail entries
make_thumbs = False
```

```
# Reuse existing mosaic for new mosaic (faster)
reuse_image = False
```

Collage
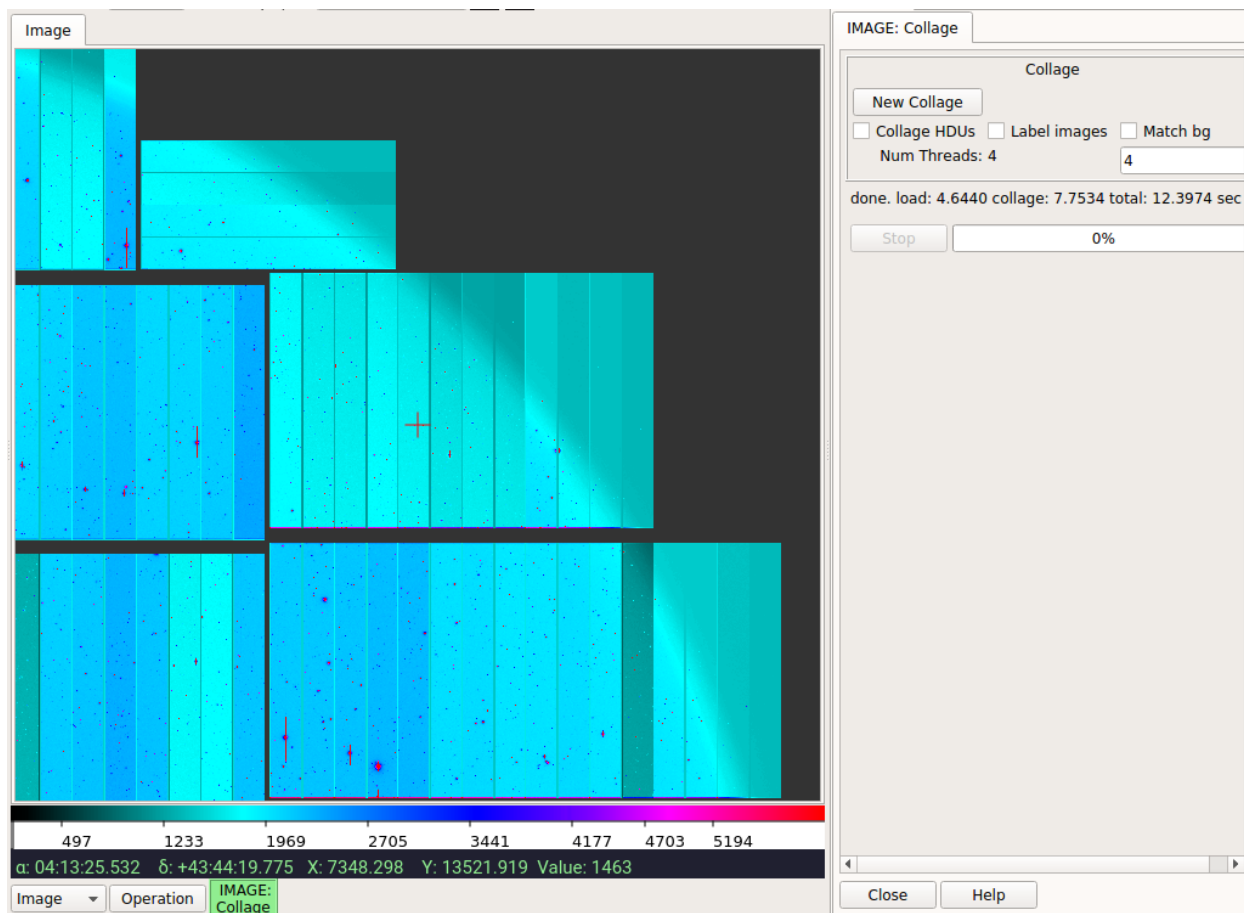
Plugin to create an image mosaic via the collage method.

Plugin Type: Local

`Collage` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

This plugin is used to automatically create a mosaic collage in the channel viewer using images provided by the user. The position of an image on the collage is determined by its WCS without distortion correction. This is meant as a quick-look tool, not a replacement for image drizzling that takes account of image distortion, etc.

The collage only exists as a plot on the Ginga canvas. No new single image is actually built (if you want that, see the "Mosaic" plugin). Some plugins that expect to operate on single images may not work correctly with a collage.

To create a new collage, click the "New Collage" button and drag files onto the display window (e.g. files can be dragged from the `FBrowser` plugin). Images must have a working WCS. The first image processed will be loaded and its WCS will be used to orient the other tiles. You can add new images to an existing collage simply by dragging addtional files.

Controls

The "Method" control is used to choose a method for mosaicing the images in the collage. It has two values: 'simple' and 'warp':

- 'simple' will attempt to rotate and flip the images according to the WCS. It is a fast method, at the expense of accuracy. It will not handle distortions near the edge of the field that should skew the image.

- 'warp' will use the WCS to completely move each pixel in the image according to the reference image's WCS. This may leave empty pixels in the image that are filled in by sampling from surrounding pixels. This will be slower than the simple method, and the time increases linearly with the size of the images.

Check the "Collage HDUs" button to have `Collage` attempt to plot all the image HDUs in a dragged file instead of just the first found one.

Check "Label Images" to have the plugin draw the name of each image over each plotted tile.

If "Match bg" is checked, the background of each tile is adjusted relative to the median of the first tile plotted (a kind of rough smoothing).

The "Num Threads" box assigns how many threads will be used from the thread pool to load the data. Using several threads will usually speed up loading of many files.

Difference from `Mosaic` plugin

- Doesn't allocate a large array to hold all the mosaic contents

- No need to specify output FOV or worry about it

- Can be quicker to show result (depends a bit on constituent images)

- Some plugins will not work correctly with a collage, or will be slower

- Cannot save the collage as a data file (although you can use "ScreenShot")

It is customizable using `~/.ginga/plugin_Collage.cfg`, where ~ is your HOME directory:

```
#
# Collage plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Collage.cfg"

# Set to True when you want to collage image HDUs in a file
collage_hdus = False

# annotate images with their names
annotate_images = False

# Try to match backgrounds
match_bg = False

# Number of threads to devote to opening images
num_threads = 4
```

Drawing

A plugin for drawing canvas forms (overlaid graphics).

Plugin Type: Local

`Drawing` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

This plugin can be used to draw many different shapes on the image display. When it is in "draw" mode, select a shape from the drop-down menu, adjust the shape's parameters (if needed), and draw on the image by using left mouse button. You can choose to draw in pixel or WCS space.

To move or edit an existing shape, set the plugin on "edit" or "move" mode, respectively.

To save the drawn shape(s) as mask image, click the "Create Mask" button and you will see a new mask image created in Ginga. Then, use `SaveImage` plugin to save it out as single-extension FITS. Note that the mask will take the size of the displayed image. Therefore, to create masks for different image dimensions, you need to repeat the steps multiple times.

Shapes drawn on the canvas can be loaded and/or saved in astropy-regions (compatible with DS9 regions) format. To use that you need to have installed the astropy-regions package. Simply draw objects on the canvas, with coords as "data" (pixel) or "wcs". Note that not all Ginga canvas objects can be converted to regions shapes and some attributes may not be saved, may be ignored or may cause errors trying to load the regions shapes in other software.

FBrowser

A plugin for browsing the local filesystem and loading files.

Plugin Type: Global or Local

`FBrowser` is a hybrid global/local plugin, which means it can be invoked in either fashion. If invoked as a local plugin then it is associated with a channel, and an instance can be opened for each channel. It can also be opened as a global plugin.

Usage

Navigate the directory tree until you come to the location files you want to load. You can double click a file to load it into the associated channel, or drag a file into a channel viewer window to load it into any channel viewer.

Multiple files can be selected by holding down `Ctrl` (`Command` on Mac), or `Shift`-clicking to select a contiguous range of files.

You may also enter full path to the desired image(s) in the text box such as `/my/path/to/image.fits`, `/my/path/to/image.fits[ext]`, or `/my/path/to/image*.fits[extname,*]`.

Because it is a local plugin, `FBrowser` will remember its last directory if closed and then restarted.

It is customizable using `~/.ginga/plugin_FBrowser.cfg`, where ~ is your HOME directory:

```
#
# FBrowser plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_FBrowser.cfg"

# Set to a specific directory to choose a starting point for file exploration.
# If None is given, it defaults to your HOME.
home_path = None
```

(continues on next page)

```
# This controls whether the plugin scans the FITS headers to create the
# listing (slow for large numbers of files)
scan_fits_headers = False

# If the number of files in the listing is greater than this, don't do
# a scan on the headers
scan_limit = 100

# if scan_fits_headers is True, then the keywords provides a map between
# attributes and FITS header keywords to fetch from the header
keywords = [('Object', 'OBJECT'), ('Date', 'DATE-OBS'), ('Time UT', 'UT')]

# columns lists the column headers and attributes to show in the listing.
# If you want to include FITS keywords, be sure to include the attributes
# defined in the keywords preference (i.e., 'Time UT', 'Object')
columns = [('Type', 'icon'), ('Name', 'name'), ('Size', 'st_size_str'), ('Mode', 'st_
↪mode_oct'), ('Last Changed', 'st_mtime_str')]

# If True, color every other row in alternating shades to improve
# readability of long tables
color_alternate_rows = True

# Maximum number of rows that will turn off auto column resizing (for speed)
max_rows_for_col_resize = 5000
```

ColorMapPicker

This brings up *ColorMapPicker* hybrid plugin as a local plugin.

Compose

A plugin for composing RGB images from constituent monochrome images.

Plugin Type: Local

Compose is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

Start the Compose plugin from the "Operation->RGB" (below) or "Plugins->RGB" (above) menu. The tab should show up under the "Dialogs" tab in the viewer to the right as "IMAGE:Compose".

1. Select the kind of composition you want to make from the "Compose Type" drop down: "RGB" for composing three monochrome images into a color image, "Alpha" to compose a series of images as layers with different alpha values for each layer.

2. Press "New Image" to start composing a new image.

For RGB composition

1. Drag your three constituent images that will make up the R, G, and B planes to the "Preview" window – drag them in the order R (red), G (green), and B (blue). Alternatively, you can load the images into the channel viewer one by one and after each one pressing "Insert from Channel" (similarly, do these in the order of R, G, and B).

In the plugin GUI, the R, G, and B images should show up as three slider controls in the "Layers" area of the plugin, and the Preview should show a low resolution version of how the composite image looks with the sliders set.
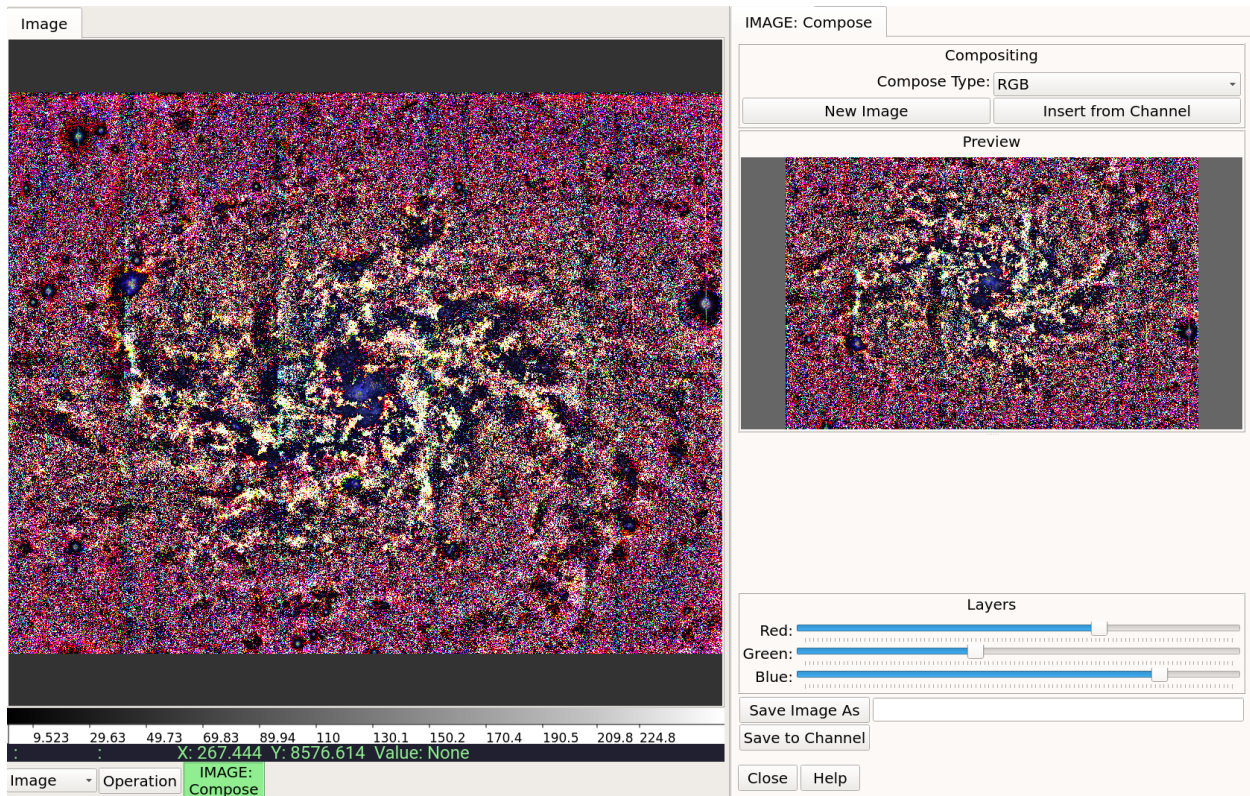
Fig. 26: Composing an RGB Image.

2. Play with the alpha levels of each layer using the sliders in the `Compose` plugin; as you adjust a slider the preview image should update.

3. When you see something you like, you can save it to a file using the "Save As" button (use "jpeg" or "png" as the file extension), or insert it into the channel using the "Save to Channel" button.

For Alpha composition

For Alpha-type composition the images are just combined in the order shown in the stack, with Layer 0 being the bottom layer, and successive layers stacked on top. Each layer's alpha level is adjustible by a slider in the same manner as discussed above.

1. Drag your N constituent images that will make up the layers to the "Preview" window, or load the images into the channel viewer one by one and after each one pressing "Insert from Channel" (the first image will be at the bottom of the stack–layer 0).

2. Play with the alpha levels of each layer using the sliders in the `Compose` plugin; as you adjust a slider the preview image should update.

3. When you see something you like, you can save it to a file using the "Save As" button (use "fits" as the file extension), or insert it into the channel using the "Save to Channel" button.

General Notes

- The preview window is just a ginga widget, so all the usual bindings apply; you can set color maps, cut levels, etc. with the mouse and key bindings.
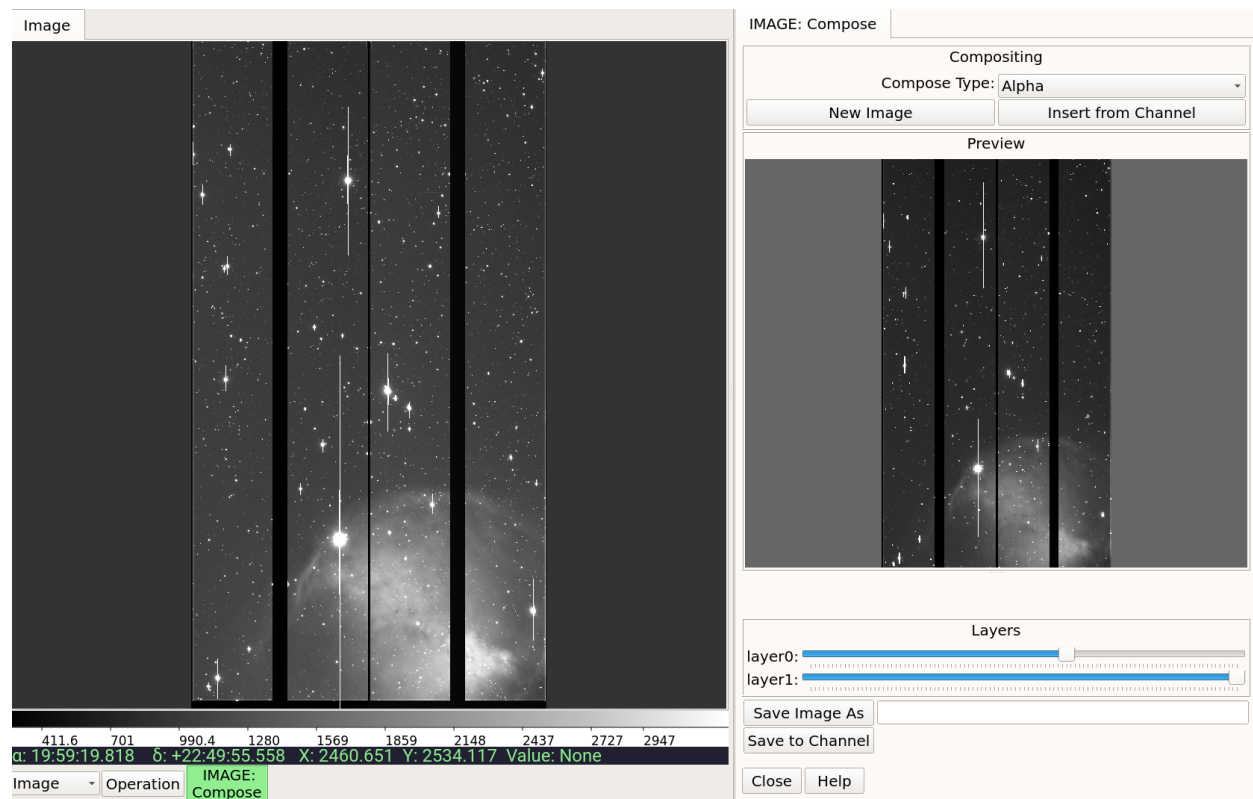
Fig. 27: Alpha-composing an image.

PlotTable
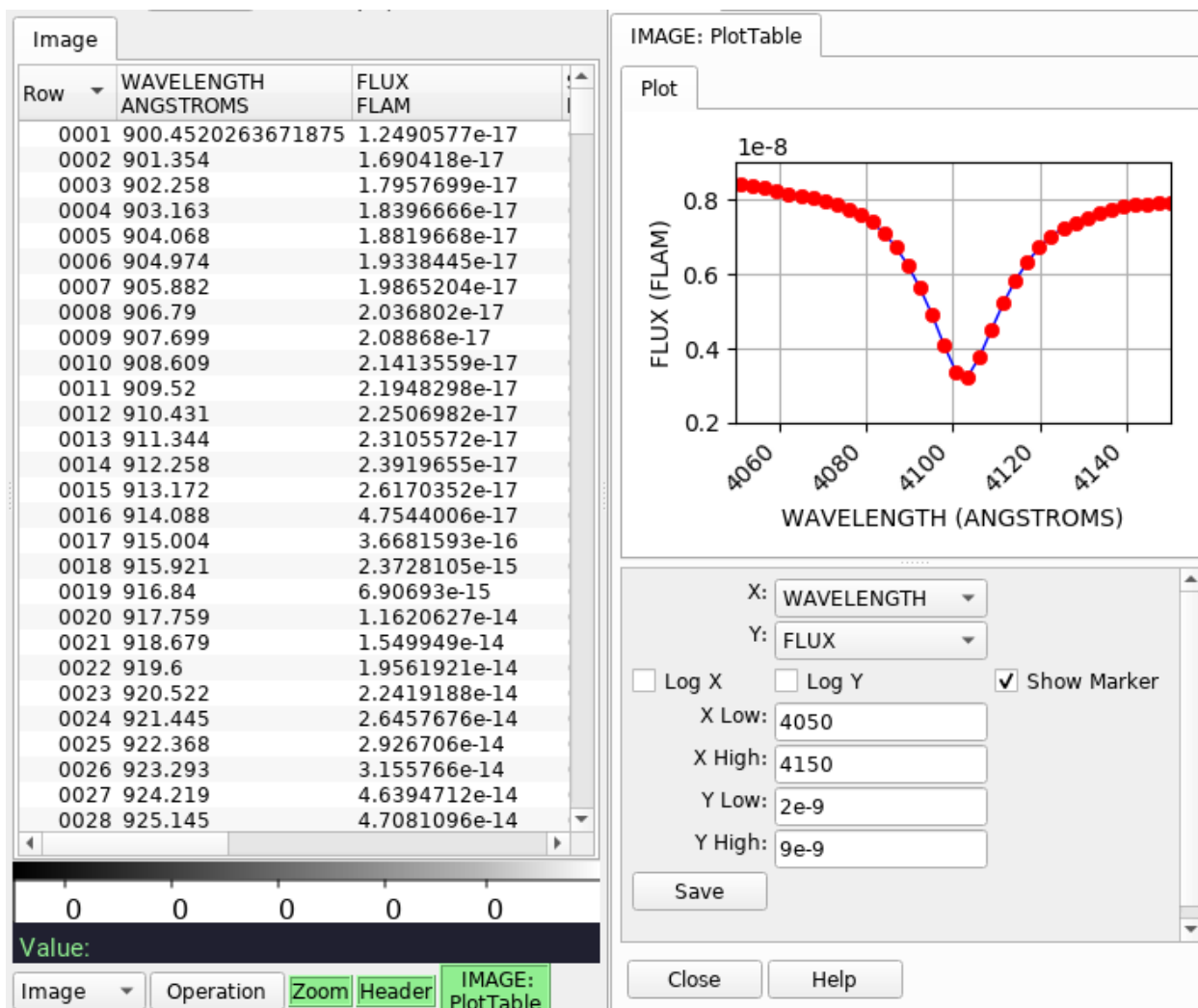
A plugin to display basic plot for any two selected columns in a table.

Plugin Type: Local

`PlotTable` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

`PlotTable` is a plugin designed to plot any two selected columns for a given FITS table HDU (can be accessed via `MultiDim`). For masked columns, masked data is not shown (even if only one of the `(X, Y)` pair is masked). It is meant as a way to quickly look at table data and not for detailed scientific analysis.

It is customizable using `~/.ginga/plugin_PlotTable.cfg`, where `~` is your HOME directory:

```
#
# PlotTable plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_PlotTable.cfg"

# matplotlib options for plotted line
linewidth = 1
```

```
linestyle = '-'
linecolor = 'blue'

# matplotlib options for markers
markersize = 6
markerwidth = 0.5
markerstyle = 'o'
markercolor = 'red'

# Show markers (can also be set in GUI)
show_marker = True

# Table column numbers to plot (can also be set in GUI).
# 0 = row indices, not the first table column
x_index = 1
y_index = 2

# Output file suffix (as accepted by matplotlib)
file_suffix = '.png'
```

Pipeline

Simple data processing pipeline plugin for Ginga.

Plugin Type: Local

`Pipeline` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Usage

It is customizable using `~/.ginga/plugin_Pipeline.cfg`, where ~ is your HOME directory:

```
#
# Pipeline plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Pipeline.cfg"

num_threads = 4
```

ScreenShot
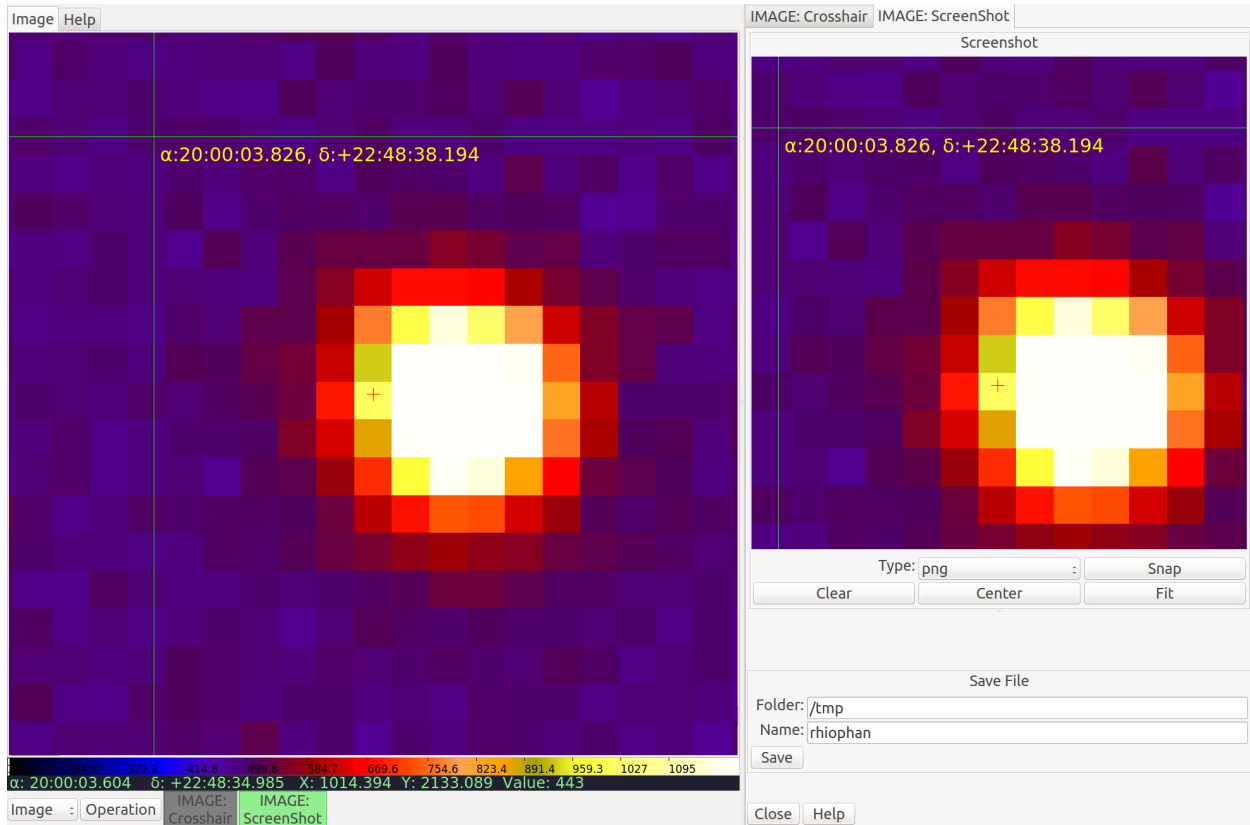
Capture PNG or JPEG images of the channel viewer image.

Usage

1. Select the RGB graphics type for the snap from the "Type" combo box.

2. Press "Snap" when you have the channel image the way you want to capture it.

A copy of the RGB image will be loaded into the `ScreenShot` viewer. You can pan and zoom within the `ScreenShot` viewer like a normal Ginga viewer to examine detail (e.g., see the magnified difference between JPEG and PNG formats).

3. Repeat (1) and (2) until you have the image you want.

4. Enter a valid path for a new file into the "Folder" text box.

5. Enter a valid name for a new file into the "Name" text box. There is no need to add the file extension; it will be added, if needed.

6. Press the "Save" button. The file will be saved where you specified.

Notes

- PNG offers less artifacts for overlaid graphics, but files are larger than JPEG.

- The "Center" button will center the snap image; "Fit" will set the zoom to fit it to the window; and "Clear" will clear the image. Press "Full" to zoom to 100% pixels (1:1 scaling).

- The "Screen size" checkbox (checked by default) will save the image at exactly the size of the channel viewer window. To save at a different size, uncheck this box, and set the size via the "Width" and "Height" boxes.

- The "Lock aspect" feature only works if "Screen size" is unchecked; if enabled, then changing width or height will alter the other parameter in order to maintain the aspect ratio shown in the "Aspect" box.

AutoLoad

is a simple plugin to monitor a folder for new files and automatically load them into a channel when they appear.

Plugin Type: Local

`AutoLoad` is a local plugin, which means it is associated with a channel. An instance can be opened for each channel.

Note: You need to install the Python "watchdog" package to use this plugin.

Usage

- To set up a folder to be monitored, type a path of a folder (directory) in the "Watched folder" field and press ENTER or click "Set".

- If you need to distinguish between files that will be added to this folder, you may type a Python regular expression in the "Regex" box and click "Set". Only files with names matching the pattern will be considered. Note that the regex is for the filename only; not any part of the folder path.

- If you ever want to pause the auto loading, you can check the box marked "Pause"; this will stop any auto loading. Note that if you subsequently uncheck the box, files that arrived in the intervening period will not be loaded.

Note: Monitoring folders that reside on network drives may or may not work.

User Configuration

It is customizable using `~/.ginga/plugin_AutoLoad.cfg`, where ~ is your HOME directory:

```
#
# AutoLoad plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_AutoLoad.cfg"

# None or the name of a folder to watch
watch_folder = None

# None or a Python regex to check the filename.  If matching, will
# try to auto load it (example: "^.+\.fits$")
filename_regex = None

# Start off with auto-loading paused
start_paused = False
```

4.6 Customizing Ginga

One of the primary guiding concepts behind the Ginga project is to provide convenient ways to build custom viewers. The reference viewer embodies this concept through the use of a flexible layout engine and the use of plugins to implement all the major user interface features. By modifying or replacing the layout and adding, subclassing or removing plugins you can completely change the look, feel and operation of the reference viewer.

This chapter explains how you can customize the Ginga reference viewer in various ways, as a user or a developer.

4.6.1 Configuration Options

Ginga examines a configuration directory on startup to check for any configuration files or customization of the default behavior.

Note: The configuration area is determined first by examining the command line option `--basedir`. If that is not set, then the environment variable `GINGA_HOME` is checked. If that is not set, then `$HOME/.ginga` (Mac OS X, Linux) or `$HOMEDRIVE:$HOMEPATH\\.ginga` (Windows) will be used.

Examples of the types of configuration files with comments describing the effects of the parameters can be found in `.../ginga/examples/configs`.

The config files that end in `.cfg` use a stripped down Pythonic format consisting of comments, blank lines and `keyword = value` pairs, where values are specified using Python literal syntax.

General Config Files

There is general top-level configuration file `general.cfg` in the configuration area. You can find an example in the examples area described above.

Binding Config File

One configuration file that many users will be interested in is the one controlling how keyboard and mouse/touch bindings are assigned. This is handled by the configuration file `bindings.cfg`. Several examples are stored in `.../ginga/examples/bindings`, including an example for users familiar with the ds9 mouse controls, and an example for users using a touchpad without a mouse (pinch zoom and scroll panning). Simply copy the appropriate file to your Ginga settings area as `bindings.cfg`.

Plugin Config Files

Many of the plugins have their own configuration file, with preferences that are only changed via that file. You can copy an example configuration file to your Ginga settings area and change the settings to your liking.

Here is an example of a plugin configuration file for the `Ruler` plugin:

```
#
# Ruler plugin preferences file
#
# Place this in file under ~/.ginga with the name "plugin_Ruler.cfg"

# Show plumb lines or not
show_plumb = True

# Show end values or not (end values are shown in the UI boxes,
# regardless)
show_ends = True

# Color for a drawn ruler
rule_color = 'green'

# Color used when drawing
```

(continues on next page)

```
draw_color = 'cyan'

# Units of distance.  Choices are 'arcmin', 'degrees' or 'pixels';
# the first two require a working WCS in the image.
units = 'arcmin'

# The unit for showing the angle of the ruler.
# Choices are 'degrees' or 'radians'
angle_unit = 'degrees'
```

Usually it is sufficient to simply close the plugin and open it again to pick up any settings changes, but some changes may require a viewer restart to take effect.

Channel Config Files

Channels also use configuration files to store many different settings for the channel viewer windows. When a channel is created, the reference viewer looks to see if there is a configuration file for that channel in the configuration area; if so, the settings therein are used to configure it. If not, the settings for the generic startup channel "Image" are used to configure the new channel. The "Preferences" plugin can be used to set many of the channel settings. If you set these for the "Image" channel and use the "Save" button, other channels will inherit them. You can also manually copy the example file from `.../ginga/examples/configs/channel_Image.cfg` to your configuration area and edit it if you prefer.

4.6.2 Customizing the Layout

Ginga has a flexible table-driven layout scheme for dynamically creating workspaces and mapping the available plugins to workspaces. This layout can be specified with a Python structure (`layout`) in the configuration area. If there is no file initially, Ginga will use the built in default layout. Ginga will will update its window size, position and some layout information to the layout file when the program is closed, creating a new custom layout. Upon a subsequent startup Ginga will attempt to restore the window to the saved configuration.

Note: The name of the layout file is set in the general configuration file (`general.cfg`) as the value for `layout_file`. Set it to "layout.json" if you prefer to use the JSON format or "layout" if you prefer to use the Python format (the default).

Note: If you don't want Ginga to remember your changes to the window size or position, you can add the option `save_layout = False` to your general configuration file. Ginga will still read the layout from the file (if it exists–otherwise it will use the default, built-in layout), but will not update it when closing.

Note: Invoking the program with the `--norestore` option prevents it from reading the saved layout file, and forces use of the internal default layout table. This may be needed in some cases when the layout changes in an incompatible way between when the program was last stopped and when it was started again.

Format of the Layout Table

The table consists of a list containing nested lists. Each list represents a container or a non-container endpoint, and has the following format:

```
[type  config-dict  optional-content0 ... optional-contentN]
```

The first item in a list indicates the type of the container or object to be constructed. The following types are available:

- `seq`: defines a sequence of top-level windows to be created

- `hpanel`: a horizontal panel of containers, with handles to size them

- `vpanel`: a vertical panel of containers, with handles to size them

- `hbox`: a horizontal panel of containers of fixed size

- `vbox`: a vertical panel of containers of fixed size

- `ws`: a workspace that allows a plugin or a channel viewer to be loaded into it. A workspace can be configured in four ways: as a tabbed notebook (`wstype="tabs"`), as a stack (`wstype="stack"`), as an MDI (Multiple Document Interface, `wstype="mdi"`) or a grid (`wstype="grid"`).

In every case the second item in the sublist is a dictionary that provides some optional parameters that modify the characteristics of the widget. If there is no need to override the default parameters the dictionary can simply be empty. All types of containers honor the following parameters in this `dict`:

- `width`: can specify a desired width in pixels for the container.

- `height`: can specify a desired height in pixels for the container.

- `name`: specifies a mapping of a name to the created container widget. The name is important especially for workspaces, as they may be referred to as an output destination when registering plugins.

The optional third and following items in a list are specifications for nested content. The format for nested content depends on the type of the container:

- `seq`, `hpanel` and `vpanel` types expect the nested content items to be lists, as described above.

- `hbox` and `vbox` content items can be lists (as described above) or `dict` s. A `vbox` `dict` should have a `row` value and optionally a `stretch` value; an `hbox` `dict` should have a `col` value and optionally a `stretch` value. The `row` and `col` values should be lists as described above.

- The `ws` (workspace) type takes one optional content item, which should be a sublist containing 2-item lists (or tuples) with the format `(name, content)`, where `content` is a list as described above. The `name` is used to identify each content item in the way appropriate for the workspace type, whether that is a notebook tab, MDI window titlebar, etc.

Here is the standard layout (Python format), as an example:

```python
['seq',
 {},
 ['vbox',
  {'height': 700, 'name': 'top', 'width': 1400},
  {'row': ['hbox', {'name': 'menu'}], 'stretch': 0},
  {'row': ['hpanel',
           {'name': 'hpnl'},
           ['ws',
            {'group': 2,
             'height': -1,
             'name': 'left',
```

```
                'width': 300,
                'wstype': 'tabs'},
              [('Info',
                ['vpanel',
                 {},
                 ['ws',
                  {'group': 3,
                   'height': 250,
                   'name': 'uleft',
                   'wstype': 'stack'}],
                 ['ws',
                  {'group': 3,
                   'height': 330,
                   'name': 'lleft',
                   'wstype': 'tabs'}])])]],
            ['vbox',
             {'name': 'main', 'width': 600},
             {'row': ['ws',
                      {'default': True,
                       'group': 1,
                       'name': 'channels',
                       'use_toolbar': True,
                       'wstype': 'tabs'}],
              'stretch': 1},
             {'row': ['ws', {'group': 99, 'name': 'cbar', 'wstype': 'stack'}],
              'stretch': 0},
             {'row': ['ws', {'group': 99, 'name': 'readout', 'wstype': 'stack'}],
              'stretch': 0},
             {'row': ['ws',
                      {'group': 99, 'name': 'operations', 'wstype': 'stack'}],
              'stretch': 0}],
            ['ws',
             {'group': 2,
              'height': -1,
              'name': 'right',
              'width': 400,
              'wstype': 'tabs'},
             [('Dialogs',
               ['ws', {'group': 2, 'name': 'dialogs', 'wstype': 'tabs'}])]]],
     'stretch': 1},
    {'row': ['ws',
             {'group': 2, 'height': 40, 'name': 'toolbar', 'wstype': 'stack'}],
     'stretch': 0},
    {'row': ['hbox', {'name': 'status'}], 'stretch': 0}]]
```

In the above example, we define a top-level window consisting of a vbox (named "top") with 4 layers: a hbox ("menu"), hpanel ("hpnl"), a workspace ("toolbar") and another hbox ("status"). The main horizontal panel ("hpnl") has three containers: a workspace ("left"), a vbox ("main") and a workspace ("right"). The "left" workspace is pre-populated with an "Info" tab containing a vertical panel of two workspaces: "uleft" and "lleft" (upper and lower left). The "right" workspace is pre-populated with a "Dialogs" tab containing an empty workspace. The "main" vbox is configured with four rows of workspaces: "channels", "cbar", "readout" and "operations".

Note: The workspace that has as a configuration option `default:   True` (in this example, "channels") will be used as the default workspace where new channels should be created.

4.6.3 Customizing the set of plugins

In the configuration directory, the presence of a file `plugins.json` will override the built-in configuration of plugins. The file format is a JSON array containing JSON objects, each of which configures a plugin. Example:

```
[
    {
        "__bunch__": true,
        "module": "Info",
        "tab": "Synopsis",
        "workspace": "lleft",
        "start": true,
        "hidden": true,
        "category": "System",
        "menu": "Info [G]",
        "ptype": "global"
    },
    ...
]
```

The keys for each object are defined as follows:

- `__bunch__`: should be present and set to `true` to force deserialization to a `Bunch`.

- `module`: The name of the module in the `$PYTHONPATH` containing the plugin.

- `class`: if present, indicates the name of the class within the module that denotes the plugin (if not present the class is assumed to be named identically to the module).

- `tab`: the name that the plugin should appear as when opened in a workspace (usually as a tab, but it depends on the type of workspace). Often the same name as the class, but can be different. If not present, defaults to the class or module name of the plugin.

- `workspace`: the name of the workspace defined in the layout file (or default layout) where the plugin should be started (see section below on workspace customization).

- `start`: `true` if the module is of the global type and should be started at program startup. Defaults to `false`.

- `hidden`: `true` if the plugin should be hidden from the "Operation" and "Plugins" menus. Often paired with `hidden` being `true` for plugins that are considered to be a necessary part of continuous operation from program startup to shutdown. Defaults to `false`.

- `category`: an arbitrary organizational name under which plugins are organized in the `Operation` and `Plugins` menus.

- `menu`: a name for how the plugin should appear under the category in the menu structure. The convention is to append "[G]" if it is a global plugin.

- `ptype`: either "local" or "global", depending on whether the plugin is a local or global one.

- `optray`: to prevent a control icon from appearing in the `Operations` plugin manager tray specify `false`. The default for non-hidden plugins is `true` and for hidden plugins `false`.

- `enabled`: `false` to disable the plugin from being loaded.

See the standard set of plugins in `.../ginga/examples/layouts/standard/plugins.json`

Custom plugin directory

If there is a `plugins` directory in the configuration area, it is added to the `PYTHONPATH` for the purpose of loading plugins. You can put plugin modules in this directory, and then add entries to the `plugins.json` file described above to add new, custom plugins.

4.6.4 Customizing the Reference Viewer (with Python) During Initialization

For the ultimate flexibility, the reference viewer can be customized during viewer initialization using a Python module called `ginga_config`, which can be anywhere in the user's Python import path, including in the Ginga configuration folder described above.

Specifically, this file will be imported and two methods will be run if defined: `pre_gui_config(ginga_shell)` and `post_gui_config(ginga_shell)`. The parameter to each function is the main viewer shell. These functions can be used to define a different viewer layout, add or remove plugins, add menu entries, add custom image or star catalogs, etc. We will refer back to these functions in the sections below.

Workspace configuration

You can create a layout table (as described above in "Customizing the Workspace") as a Python data structure, and then replace the default layout table in the `pre_gui_config()` method described above:

```
my_layout = [
              ...
            ]

def pre_gui_config(ginga_shell):
    ...

    ginga_shell.set_layout(my_layout)
```

If done in the `pre_gui_config()` method (as shown) the new layout will be the one that is used when the GUI is constructed. See the default layout in `~ginga.rv.main` as an example.

Start Plugins and Create Channels

You can create channels and start plugins using the `post_gui_config()` method.

A plugin can be started automatically in `post_gui_config()` using the `start_global_plugin()` or `start_local_plugin()` methods, as appropriate:

```
def post_gui_config(ginga_shell):
    # Auto start global plugins
    ginga_shell.start_global_plugin('Zoom')
    ginga_shell.start_global_plugin('Header')

    # Auto start local plugin
    ginga_shell.add_channel('Image')
    ginga_shell.start_local_plugin('Image', 'Histogram', None)
```

Alternately, you can also start plugins via the command line interface using `--plugins` and `-modules` for local and global plugins, respectively. To load multiple plugins at once, use a comma-separated list. For example:

```
ginga --plugins=MyLocalPlugin,Imexam --modules=MyGlobalPlugin
```

Adding Plugins

A plugin can be added to the reference viewer in `pre_gui_config()` using the `add_plugin()` method with a specification ("spec") for the plugin:

```python
from ginga.misc.Bunch import Bunch

def pre_gui_config(ginga_shell):
    ...

    spec = Bunch(module='DQCheck', klass='DQCheck', workspace='dialogs',
                 category='Utils', ptype='local')
    ginga_shell.add_plugin(spec)
```

The above call would try to load a local plugin called "DQCheck" from a module called "DQCheck". When invoked from the Operations menu it would occupy a spot in the "dialogs" workspace (see layout discussion above).

Disabling Plugins

Both local and global plugins can be disabled (thus, not shown in the reference viewer) using the `--disable-plugins` option in the command line interface. To remove multiple plugins at once, use a comma-separated list. For example:

```
ginga --disable-plugins=Zoom,Compose
```

Alternately, plugins can also be disabled via `general.cfg` configuration file. For example:

```
disable_plugins = "Zoom,Compose"
```

Finally, if you are using a custom "plugins.json" file as described above, you can simply set the `enabled` attribute to `False` in the JSON object for that plugin in the file.

Some plugins, like `Operations`, when disabled, may result in inconvenient GUI experience.

4.6.5 Making a Custom Startup Script

You can make a custom startup script to make the same reference viewer configuration available without relying on a custom set of startup files or the `ginga_config` module. To do this we make use of the *main* module:

```python
import sys
from argparse import ArgumentParser

from ginga.rv.main import ReferenceViewer

# define your custom layout
my_layout = [ ... ]
```

```python
# define your custom plugin list
plugins = [ ... ]

if __name__ == "__main__":
    viewer = ReferenceViewer(layout=my_layout)
    # add plugins
    for spec in plugins:
        viewer.add_plugin(spec)

    argprs = ArgumentParser(description="Run my custom viewer.")
    viewer.add_default_options(argprs)
    (options, args) = argprs.parse_known_args(sys_argv[1:])

    viewer.main(options, args)
```

DEVELOPER'S MANUAL

This section provides some resources for developers interested in using Ginga as a visualization solution for their own software.

5.1 Image Data Wrappers

The image viewer can load data in a number of formats, but all formats are wrapped with a class that corresponds to the *model* part of the model-view-controller design used by Ginga. These wrappers make the data accessible in a common interface for the image viewer. The most common wrappers are `AstroImage` and `RGBImage`, for single band (i.e. "monchromatic") and multi-band (i.e. "RGB") data, respectively.

5.1.1 AstroImage

An `AstroImage` combines image data with metadata (including keywords) and optionally, world coordinate information.

Data can be loaded in a number of ways. For the following examples, assume that we created a wrapper object via:

```
>>> from ginga.AstroImage import AstroImage
>>> img = AstroImage()
```

Note: Ginga provides extensive logging throughout the code, so if you are using a Python logger you can pass it to the constructor to have it log extra information about errors when methods on the image object are being used. Assuming you had a logger configured as `logger` you would pass it like so:

```
>>> img = AstroImage(logger=logger)
```

Once you have an object, you can load data directly contained in a `numpy.ndarray`:

```
>>> import numpy as np
>>> data = np.random.randint(0, 10000, (2000, 3000), dtype=np.uint)
>>> img.load_data(data)
```

Note: if you want to provide metadata (e.g. a separate set of FITS-type keywords) you can add it:

```
>>> img.update_keywords(kw_dict)
```

From an `astropy.io.fits.HDU`:

```
>>> from astropy.io import fits
>>> with fits.open("/path/to/image.fits") as fits_f:
>>>     img.load_hdu(fits_f[0])
```

From an `astropy.nddata.NDData` (or subclass, like `CCDData`):

```
>>> img.load_nddata(ndd_obj)
```

Files are best loaded from the appropriate file format loader module. For a FITS file:

```
>>> from ginga.util import io_fits
>>> img = io_fits.load_file("/path/to/image.fits")
```

Or, e.g. to choose a particular HDU:

```
>>> from ginga.util import io_fits
>>> img = io_fits.load_file("/path/to/image.fits[SCI]")
```

5.1.2 RGBImage

The `RGBImage` class is used to store conventional type 3 or 4-band RGB images.

```
>>> from ginga.RGBImage import RGBImage
>>> img = RGBImage()
```

Note: RGBImage constructor also supports the `logger` keyword parameter described above:

```
>>> img = RGBImage(logger=logger)
```

RGB images support the `load_data` method (note the shape and type of the array):

```
>>> data = np.random.randint(0, 256, (1000, 1000, 3), dtype=np.uint8)
>>> img.load_data(data)
```

Files can also be loaded from standard RGB formats (PNG, JPEG, etc) using the `io_rgb` module:

```
>>> from ginga.util import io_rgb
>>> img = io_rgb.load_file("/path/to/image.jpg")
```

5.2 Ginga Image Viewer Operations

This chapter describes the operations supported by the basic Ginga image viewer and how to access them programatically.

5.2.1 Manipulating the view programatically

The sections below describe how to manipulate the view programatically. For all of these API examples we assume the viewer object is contained in variable v. One way to create one:

```
>>> from ginga import toolkit
>>> toolkit.use('qt5')
>>> from ginga.gw import Viewers
>>> from ginga.misc import log
>>> logger = log.get_logger("viewer1", log_stderr=True, level=40)
>>> v = Viewers.CanvasView(logger=logger)
```

Many of the API calls simply set a value in the viewer's settings object; if so, we note the associated keyword for that setting.

5.2.2 Loading the Viewer

The image viewer can load data in a number of formats, but all formats are wrapped with a class that corresponds to the *model* part of the model-view-controller design used by Ginga. For more information on creating an image wrapper object, see *Image Data Wrappers*.

Once you have successfully loaded an image wrapper, you can set it into the image viewer:

```
>>> v.set_image(img)
```

5.2.3 Scaling

In Ginge lexicon, "scaling" is setting a precise scale factor for pixels. For example a scale factor of 2.0 means that two pixels are shown in the viewer for every one pixel of data.

Ginga allows different scale factors for the X and Y axes.

Set a precise scale for X and Y:

```
>>> v.set_scale((0.5, 0.5))
```

Note: Corresponds to the viewer setting scale.

Get the scale for X and Y:

```
>>> v.get_scale_xy()
(0.5, 0.5)
```

If the scale factors are always set to the same value and adjusted equally in tandem, then you can obtain the general scale value as:

```
>>> v.get_scale_max()
0.5
```

Scaling Limits

Ginga places a limit on the scale when the scale would result in an aberrant viewing condition. For example, it will cease to increase the scale when the pixel size approaches the window size.

Ginga has a feature for setting arbitrary hard limits on the minimum and maximum scale.

To get the scale limits (returns minimum and maximum scale):

```
>>> v.get_scale_limits()
(1e-05, 10000.0)
```

To set the scale limits:

```
>>> v.set_scale_limits(min_scale, max_scale)
```

The limits apply to all dimensions. A value of `None` (the default) means that no artificial limit will be imposed.

5.2.4 Zooming

Although Ginga uses scale factors to accomplish image scaling, it also supports the concept of zooming via *levels*. It does this by mapping a zoom level to a scale factor according to the *zoom algorithm* (see following section) in use.

Zoom algorithms assume that a 1:1 scale is always zoom level 0. Providing a negative zoom level zooms "out" (smaller scale) while a positive value zooms "in" (larger scale).

Set a zoom level:

```
>>> v.zoom_to(4)
```

Zoom in one level:

```
>>> v.zoom_in()
```

Zoom out three levels:

```
>>> v.zoom_out(3)
```

Zoom so that the loaded image fits the window:

```
>>> v.zoom_fit()
```

Get the current zoom level corresponding to the current scale:

```
>>> v.get_zoom()
-8.544
```

Note: Zoom levels need not be integers, and there should be an invertable mapping between scale and zoom level defined by the zoom algorithm.

Zoom Algorithms

There are two main zoom algorithms, "step" and "rate", for determining zoom levels and corresponding scales. The step algorithm just zooms in integer multiples of the pixels: 1, 2, 3, 4, … or 1/2, 1/3, 1/4, etc. The rate algorithm simply applies a multiplier to the current scale to arrive at the new scale. This can be more appropriate to achieve a faster (or slower) rate of zoom than stepping.

Get the zoom algorithm in use:

```
>>> v.get_zoom_algorithm()
'step'
```

Set the zoom algorithm:

```
>>> v.set_zoom_algorithm('rate')
```

Note: Corresponds to the viewer setting `zoom_algorithm`.

Get the zoom rate:

```
>>> v.get_zoomrate()
1.4142135623730951
```

Set the zoom rate:

```
>>> v.set_zoomrate(1.1)
```

Note: Corresponds to the viewer setting `zoom_rate`. The zoom rate should be defined as a value greater than 1. This value is ignored when the "step" algorithm is in use.

5.2.5 Panning

The "pan position" defines the point on which the viewer should be centered. Normally this is specified in data coordinates, but it can also be specified in world coordinates if a valid WCS is available in an image that is loaded.

Get the pan position in data coordinates (the default):

```
>>> v.get_pan()
(1136.0, 2136.5)
```

Set the pan position:

```
>>> v.set_pan((500.0, 1500.0))
```

Get the pan position in world coordinates (in this case, degrees):

```
>>> v.get_pan(coord='wcs')
(300.16929984148425, 22.80602873666544)
```

Set the pan position in world coordinates:

```
>>> v.set_pan((298.21, 24.6), coord='wcs')
```

Note: Corresponds to the viewer settings `pan` and `pan_coord`.

Pan to 25% of the X axis and 75% of the Y axis:

```
>>> v.panset_pct(0.25, 0.75)
```

Center the image (i.e., pan to center):

```
>>> v.center_image()
```

Get the coordinates in the actual data corresponding to the area shown in the display for the current zoom level and pan:

```
>>> v.get_pan_rect()
array([[ 886. , 1886.5],
       [ 886. , 2386.5],
       [1386. , 2386.5],
       [1386. , 1886.5]])
```

The values are returned as corners lower-left, upper-left, upper-right, lower-right.

5.2.6 Transforms

The Ginga viewer provides three quick transforms in addition to rotation (described below). These are flipping in the X axis, flipping in the Y axis, and swapping axes. These three transforms can be set in a single call, with three booleans, in the order just listed.

Flip the view in the X dimension:

```
>>> v.transform(True, False, False)
```

Flip the view in the Y dimension:

```
>>> v.transform(False, True, False)
```

Flip the view in the X dimension and swap the X and Y axes:

```
>>> v.transform(True, False, True)
```

Get the existing transforms:

```
>>> v.get_transforms()
(True, False, True)
```

Note: Corresponds to the viewer settings `flip_x`, `flip_y` and `swap_xy`.

Attempt to orient the viewer according to the image loaded (may set transforms to accomplish this):

```
>>> v.auto_orient()
```

5.2.7 Rotation

The Ginga viewer can also rotate the image. Values are specified in degrees.

Rotate the view 45 degrees:

```
>>> v.rotate(45.0)
```

Get current rotation:

```
>>> v.get_rotation()
45.0
```

Note: Corresponds to the viewer setting `rot_deg`.

5.2.8 Cut Levels

The cut levels are Ginga's lexicon for the low and high values used to establish the mapping from data values to the minimum and maximum pixel luminance values in the viewer. Values in the data below the low cut value will be driven to the bottom luminance value and values above the high cut value will be driven to the top luminance value. The values in between are scaled to the range between these values.

Setting cut levels on the viewer:

```
>>> v.cut_levels(lo_val, hi_val)
```

Get current cut levels:

```
>>> v.get_cut_levels()
(440.6118816303353, 606.8032622632695)
```

Note: Corresponds to the viewer setting `cuts`.

Auto cut levels

Calculating and applying an auto cut levels (aka "auto levels"), using the current algorithm setting and parameters:

```
>>> v.auto_levels()
```

Find out what automatic cut levels algorithms are available:

```
>>> v.get_autocut_methods()
('minmax', 'median', 'histogram', 'stddev', 'zscale')
```

Set viewer to use a specific algorithm, with parameters:

```
>>> v.set_autocut_params('histogram', pct=0.90)
```

Note: Every auto cuts algorithm is encapsulated into an auto cuts class. The parameters can vary according to the parameters of the algorithm and are passed as keyword parameters to this call.

Retrieve the current autocuts object:

```
>>> v.autocuts
<ginga.AutoCuts.Histogram at 0x7fb404a19dd8>
```

Explicitly set the autocuts object directly:

```
>>> from ginga.AutoCuts import ZScale
>>> ac = ZScale(v.get_logger(), contrast=0.4)
>>> v.set_autocuts(ac)
```

Note: Unless you are using a custom autocuts class it is generally easier to just use the set_autocut_params() method.

5.2.9 Color Distribution

The color distribution algorithm distributes the values in the image *after* the cut levels to the colors defined by the color map. This normally describes a mapping curve such as linear, logarithmic, etc.

Find out what color distribution algorithms are available:

```
>>> v.get_color_algorithms()
['linear', 'log', 'power', 'sqrt', 'squared', 'asinh', 'sinh', 'histeq']
```

Set a specific color distribution algorithm:

```
>>> v.set_color_algorithm('log')
```

Find out which one is being used:

```
>>> v.get_settings().get('color_algorithm')
'log'
```

5.2.10 Color Map

Find out what color maps are available:

```
>>> from ginga import cmap
>>> cmap.get_names()
['Accent',
 'Accent_r',
 'afmhot',
 'afmhot_r',
 ...
'YlOrBr_r',
```

(continues on next page)

```
'YlOrRd',
'YlOrRd_r']
```

Set a color map:

```
>>> v.set_color_map('YlOrBr_r')
```

Find out which one is being used:

```
>>> v.get_settings().get('color_map')
'YlOrBr_r'
```

Enable matplotlib color maps to be available (if `matplotlib` is installed):

```
>>> cmap.add_matplotlib_cmaps()
```

Invert the color map:

```
>>> v.invert_cmap()
```

Shift the color map by `pct` percent:

```
>>> v.shift_cmap(0.2)
```

Stretch/shrink and shift color map:

```
>>> v.scale_and_shift_cmap(scale_pct, shift_pct)
```

5.2.11 Intensity Map

Note: Intensity maps are a feature that largely duplicates the functionality of color distributions (see above). It performs a remapping of the colors after the color mapping phase has been applied. We suggest that most users will want to leave the default setting of 'ramp', which is leaves the color map as is.

Find out what intensity maps are available:

```
>>> from ginga import imap
>>> imap.get_names()
['equa',
 'expo',
 'gamma',
 'jigsaw',
 'lasritt',
 'log',
 'neg',
 'neglog',
 'null',
 'ramp',
 'stairs',
 'ultrasmooth']
```

Set an intensity map:

```
>>> v.set_intensity_map('lasritt')
```

Find out which one is being used:

```
>>> v.get_settings().get('intensity_map')
'lasritt'
```

5.2.12 Auto configuration

Ginga has some settings that control whether certain initializations are performed when a new image is set in the viewer.

Enable auto orientation of new image (see `auto_orient()` under "Transforms"):

```
>>> v.enable_auto_orient(True)
```

Note: Corresponds to the viewer setting `auto_orient`.

Enable auto centering (pan to center of new image):

```
>>> v.enable_autocenter('on')
```

Note: Corresponds to the viewer setting `autocenter`.

Enable auto cuts (calculate and set cut levels of new image):

```
>>> v.enable_autocuts('on')
```

Note: Corresponds to the viewer setting `autocuts`.

Enable auto zoom (scale to fit new image to window):

```
>>> v.enable_autozoom('on')
```

Note: Corresponds to the viewer setting `autozoom`.

The autocenter, autocuts, and autozoom settings allow the following values:

- 'on': apply to every new image
- 'once': apply to the first image set only, then turn 'off'
- 'override': apply to each image until the user overrides manually, then turn 'off', and
- 'off': never apply

5.2.13 Miscellaneous Operations

Set the background color of the viewer:

```
>>> v.set_bg(0.2, 0.2, 0.2)
```

Note: Corresponds to the viewer setting `color_bg`.

Set the foreground color of the viewer (used for some text overlays):

```
>>> v.set_fg(0.8, 0.9, 0.7)
```

Note: Corresponds to the viewer setting `color_fg`.

Put a message onscreen for 2 seconds:

```
>>> v.onscreen_message("Hello, world!", delay=2.0)
```

Get the last position of the cursor in data coordinates:

```
>>> v.get_last_data_xy()
(782.4466094067262, 2136.5)
```

Whether the viewer widget should take focus when the cursor enters the window:

```
>>> v.set_enter_focus(True)
```

Note: Corresponds to the viewer setting `enter_focus`.

Get the bounding box of viewer extents (returns lower-left and upper-right corners of the bounding box):

```
>>> v.get_limits()
((0.0, 0.0), (2272.0, 4273.0))
```

Note: Normally the limits are defined by an image that is loaded, if any. But they can also be overridden, as shown below.

Set explicit limits for the viewer:

```
>>> v.set_limits(((250.0, 250.0), (2500.0, 2500.0)))
```

Note: Corresponds to the viewer setting `limits`.

5.3 Ginga Canvas Graphics

This chapter describes the basic architecture of Ginga's canvas-viewer-renderer model, and describes how to do graphics operations on canvases.

5.3.1 Canvases and Canvas Objects

Ginga's canvas is based on the `DrawingCanvas` class. On the canvas can be placed a number of different kinds of *canvas objects*, including many geometric shapes. The set of canvas objects includes:

- `Text`: a piece of text having a single point coordinate.
- `Polygon`: a closed polygon defined by N points.
- `Path`: an open polygon defined by N points.
- `Box`: a rectangular shape defined by a single center point, two radii and a rotation angle.
- `Ellipse`: an elliptical shape defined by a single center point, two radii and a rotation angle.
- `Triangle`: an equilateral triangular shape defined by a single center point, two radii and a rotation angle.
- `Circle`: a circular shape defined by a center point and a radius.
- `Point`: a marker for a point defined by a single point and a radius for the "arms".
- `Rectangle` – a rectangular shape defined by two points.
- `Line` – a line defined by two points.
- `RightTriangle` – a right triangle defined by two points.
- `Compass` – a compass defined by a point and a radius.
- `Ruler` – a ruler defined by two points.
- `Crosshair` – a crosshair defined by one point.
- `Annulus` – an annulus defined by one point, a radius and a width.
- `Annulus2R` – an annulus defined by one point, two radii and two widths.
- `Image` – a raster image anchored by a point.
- `NormImage` – a subclass of `Image`, with rendering done with the aid of a colormap, a color distribution algorithm (linear, log, etc), and defined low and high cut levels.
- `CompoundObject`: a compound object combining a series of other canvas objects.
- `Canvas`: a transparent subcanvas on which items can be placed.
- `DrawingCanvas`: Like a `Canvas`, but also can support manual drawing operations initiated in a viewer to create shapes on itself.
- `ColorBar`: a bar with a color range and ticks and value markers to help indicate the mapping of color to the value range of the data.
- `ModeIndicator`: a small rectangular overlay with text indicating that the user has entered a special keyboard/mouse mode.

All canvas objects are subclasses of `CanvasObjectBase` and may also contain mixin classes that define common attributes or behavior. For example, `Line`, `Ruler` and `RightTriangle` are all subclasses of the mixin class `TwoPointMixin`.

Note: In most general canvas systems you can layer objects in any order. In Ginga there is an optimization of canvas redrawing that merges image bitmaps before updating other kinds of canvas objects. This means that while images can be stacked in any order, effectively you cannot have other objects appear *underneath* image objects. For most uses of the viewer this is not a big limitation.

5.3.2 Viewers

All Ginga viewers are subclasses of `ImageViewBase`. These objects implement a viewport onto a `DrawingCanvas` object. Each viewer contains a handle to a canvas and provides a particular view onto that canvas defined by:

- dimensions of their viewport (i.e. the height and width of the native widget's window into which the viewer is rendering),
- scale in X and Y dimensions,
- a *pan position* linking the center of the viewport to a canvas coordinate,
- a transform consisting of possible flips in X, Y axes and/or swapping of X/Y axes, and
- a rotation.

Two different `ImageView`-based viewers can share the same canvas handle, providing different views into the same canvas. Another typical arrangement for sharing is where each viewer has a private canvas, and on each private canvas is placed a shared transparent subcanvas, an arrangement which allows each viewer to have a mix of private and shared canvas objects. Another common idiom is to layer multiple `DrawingCanvas` objects to more easily manage multiple collections of overlaid graphics.

The various subclasses of `ImageView` are designed to render into a different widget set's "native" canvas using a `CanvasRenderer` customized for that target.

5.3.3 Using Canvases

The recommended way of using canvases is to create your own `DrawingCanvas` and add (or remove) it to/from the existing (default) viewer canvas.

Creating a Ginga Canvas

Assuming we have created a viewer (`view`):

```
v_canvas = view.get_canvas()
DrawingCanvas = v_canvas.get_draw_class('drawingcanvas')
mycanvas = DrawingCanvas()
v_canvas.add(mycanvas)
```

You can create several different canvases and add them or remove them as needed from the default viewer canvas. The items added to individual canvases stay on those canvases, allowing a good deal of control in managing canvas overlays on top of images, which appear under those canvases.

Enabling User Interaction on a Canvas

To enable user interaction on a canvas, use the following methods on it:

```
mycanvas.set_surface(view)        # associate the canvas with a viewer
mycanvas.ui_set_active(True)      # enable user interaction on this canvas
```

User Drawing on a Canvas

To enable user drawing on the canvas, enable user interaction as described above, then use the following methods:

```
mycanvas.enable_draw(True)        # enable user drawing on this canvas
mycanvas.set_draw_mode('draw')

# without this call, you can only draw with the right mouse button
# using the default user interface bindings
mycanvas.register_for_cursor_drawing(view)
```

If you want to get a callback after something has been drawn:

```
# the callback function gets the canvas and the tag of the drawn
# object as parameters
#
def draw_cb(canvas, tag):
    obj = canvas.get_object_by_tag(tag)
    # do something with ``obj``
    ...

mycanvas.add_callback('draw-event', draw_cb)
```

Set Drawing Parameters

To set the drawing parameters (what will be drawn by the user):

```
mycanvas.set_drawtype('box', color='red')
```

To see the kinds of objects that can be drawn on a Ginga canvas, refer to the section above on "Canvases and Canvas Objects". With the `set_drawtype` call, most drawing types are specified in all lower case with no spaces (e.g. "righttriangle"). Various object attributes (line and fill, etc) are set by keyword parameters:

```
mycanvas.set_drawtype('polygon', color='lightblue', linewidth=2,
                      fill=True, fillcolor='yellow', fillalpha=0.4)
```

Editing Objects on a Canvas

DrawingCanvases have a built in editor that can handle basic editing of drawn (or programatically) added items.

To enable user editing on a canvas, add the following calls in the setup of the canvas:

```python
mycanvas.enable_edit(True)       # enable user editing on this canvas
```

To set the mode on a canvas from drawing to editing:

```python
mycanvas.set_draw_mode('edit')
```

If you want to get a callback after an object has been edited on a canvas:

```python
# the callback function gets the canvas and the object reference
# of the edited object as parameters
#
def edit_cb(canvas, obj):
    # do something with ``obj``
    ...

mycanvas.add_callback('edit-event', edit_cb)
```

It is also possible to set a direct edit callback on the object itself. Assuming we have a handle to an object (`obj`) that has been added to a canvas (drawn or added programatically):

```python
# the callback function gets the object reference of the edited
# object as a parameter
#
def obj_edit_cb(obj):
    # do something with ``obj``
    print("object of type '{}' has been edited".format(obj.kind))

obj.add_callback('edited', obj_edit_cb)
```

"Pick" Callbacks

There are a group of actions under the umbrella term of "pick callbacks" that can be registered for objects on a `DrawingCanvas`.

To set the canvas mode from "draw" or "edit" to "pick":

```python
mycanvas.set_draw_mode('pick')
```

NOTE: Canvas objects are not "pickable" by default. To make an object "pickable", set it's "pickable" attribute to `True`. This can be done before or after it has been drawn or placed on a canvas:

```python
obj.pickable = True
obj.add_callback('pick-down', pick_cb, 'down')
obj.add_callback('pick-up', pick_cb, 'up')
obj.add_callback('pick-move', pick_cb, 'move')
obj.add_callback('pick-hover', pick_cb, 'hover')
obj.add_callback('pick-enter', pick_cb, 'enter')
obj.add_callback('pick-leave', pick_cb, 'leave')
obj.add_callback('pick-key', pick_cb, 'key')
```

From the above example you can see all the possible callbacks for "pick". In setting up the callback, we append a "pick type" string to the callback signature so that we can easily distinguish the pick action in the callback (you could also just define different callback functions):

```python
# callback parameters are: the object, the canvas, the event, a
# point (in data coordinates) and the pick "type"

def pick_cb(obj, canvas, event, pt, ptype):
    print("pick event '%s' with obj %s at (%.2f, %.2f)" % (
        ptype, obj.kind, pt[0], pt[1]))
    return True
```

The pick type (`ptype` in the above example) will be one of:

- "enter": cursor entered the area of the object,

- "hover": cursor is hovering over the object,

- "leave": cursor as exited the area of the object,

- "down": cursor was pressed down inside the object,

- "move": cursor is being moved while pressed,

- "up": cursor was released,

- "key": a key was pressed while the cursor was inside the object

5.3.4 Support for Astropy regions

Ginga provides a module for plotting Astropy `regions` shapes on canvases. To use this, import the `ginga.util.ap_regions` module and use one of the three module functions `astropy_region_to_ginga_canvas_object`, `add_region`, or `ginga_canvas_object_to_astropy_region`.

`astropy_region_to_ginga_canvas_object` takes a `regions` shape and returns a Ginga canvas object that most closely implements the shape. The object returned can be used like any Ginga canvas object: it can be used in a compound object, added to a canvas, etc. Assuming you have a viewer `v` and an Astropy region `r`:

```python
from ginga.util import ap_region
obj = ap_region.astropy_region_to_ginga_canvas_object(r)
canvas = v.get_canvas()
canvas.add(obj)
```

`add_region` is a convenience method for both converting an object and adding it to a canvas.

```python
ap_region.add_region(canvas, r)
```

`ginga_canvas_object_to_astropy_region` provides the reverse transformation, taking a Ginga canvas object and converting it to the closest representation as an Astropy region.

```python
r = ap_region.ginga_canvas_object_to_astropy_region(obj)
```

5.4 Plots

Note: Ginga's graph plotting features are new, and the API should be considered somewhat experimental. In the future, some elements of `PlotAide` might be merged into existing components of the viewer or action bindings classes.

Ginga has some basic facilities for graphing interactive XY line plots. Ginga plots may be useful in cases where good interactive performance in a line plot is desirable (e.g. plotting an XY line repeatedly at speed), or being able to zoom in and out of X and Y axes independently, flip axes, swap axes or pan axes independently. There is some special support for plotting and examining time-series data.

Graphs are shown in a Ginga viewer and managed mostly by a separate `PlotAide` object, which manipulates the viewer and its main canvas. The plot aide is like a coordinator or intermediary between the components of the plot. It initializes the viewer for special user plot interaction and provides some coordination between XY plots and other "plot decor" like plot title and key, axis titles, axis grids and markers, etc.

5.4.1 Plot objects

Plot objects are available by importing classes from `ginga.canvas.types.plots`. These classes are divided into "plot decor": plot title and key area (`PlotTitle`), X axis title, grid and markers (`XAxis`), Y axis title grid and markers (`YAxis`), plot background (`PlotBG`), etc. `XYPlot` objects (or subclasses thereof) form the individual plot lines within the graph. In typical use, the plot aide provides a convenience method (`setup_standard_frame`) that can automatically add and configure the plot decor, so that the user then simply needs to instantiate `XYPlot` objects and add them to the plot via the plot aide's `add_plot` method.

Assuming that we have created and configured a `CanvasView` viewer (`v`) with an associated `ScrolledView` widget (`sw`), we can setup an empty graph (refer to *Using the Basic Ginga Viewer Object in Python Programs* for a review of creating a viewer and associated scroll widget):

```python
from ginga.plot.PlotAide import PlotAide

aide = PlotAide(v)
aide.configure_scrollbars(sw)

aide.setup_standard_frame(title="Functions", x_title="x", y_title="f(x)")
```

To plot, simply call `plot` or `plot_xy` (depending on whether your data points are together or in separate arrays) on each `XYPlot` object that is present in the graph. Then call `update_plots()` on the plot aide to update the viewer.

Let's add two plots to this interactive graph:

```python
from ginga.canvas.types.plots import XYPlot
import numpy as np

sin_plot = XYPlot(name='Sin', color='red', linewidth=2.0)
aide.add_plot(sin_plot)

cos_plot = XYPlot(name='Cos', color='blue', linewidth=2.0)
aide.add_plot(cos_plot)
```

Finally, let's assign some data to the plots and update the viewer:

```
x_arr = np.linspace(-10.0, 10.0, 100)
y_arr = np.sin(x_arr)
sin_plot.plot_xy(x_arr, y_arr)

y_arr = np.cos(x_arr)
cos_plot.plot_xy(x_arr, y_arr)

aide.update_plots()
```
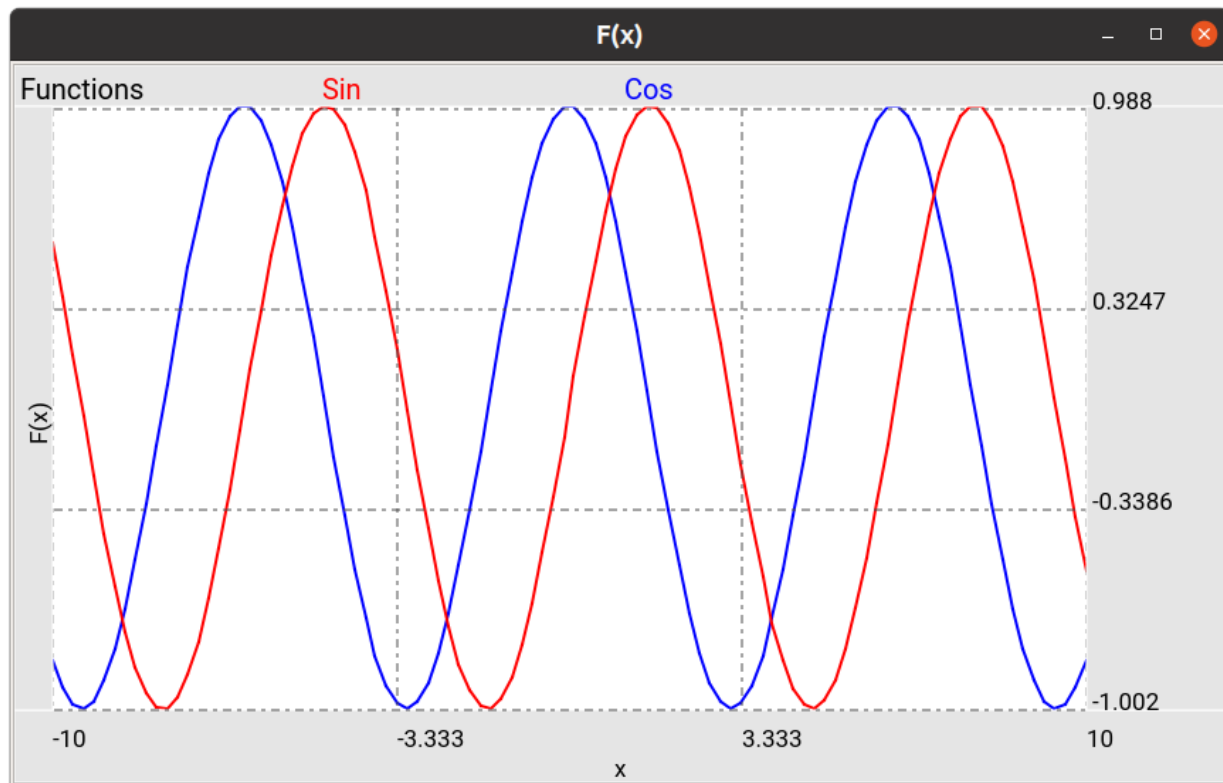

Fig. 1: The plotted Sine and Cosine curves.

Support for autoscaling axes

The plot aide has two settings that control auto adjustment of the viewer's scale and pan positions when the view is updated (via `update_plots`). The settings are intrinsic to the plot aide. The values can be toggled by keystroke commands within the viewer, or programatically by assigning the appropriate settings on the plot aide:

```
aide.settings.set(autoaxis_x='off', autoaxis_y='vis')
```

autoaxis_x

The setting for `autoaxis_x` controls how the viewer will handle the X dimension as far as panning and scaling automatically when the view is updated. The settings are:

- `off`: the viewer makes no pan or scale adjustments to X

- `pan`: the viewer pans so that the values at the end of the plot are visible; this is useful for live time-series plots, for example

- `on`: the viewer scales and pans so that the full X plot can be fit to the plot area shown in the viewer

The default value in the plot aide is `on`.

autoaxis_y

The setting for `autoaxis_y` controls how the viewer will handle the Y dimension as far as panning and scaling automatically when the view is updated. The settings are:

- `off`: the viewer makes no pan or scale adjustments to Y

- `vis`: the viewer scales Y so that the Y values corresponding to the X values visible in the plot will fill the Y dimension of the plot area

- `on`: the viewer scales and pans so that the full Y range of the data (visible or not) could be shown in the plot area of the viewer

The default value in the plot aide is `on`.

5.4.2 Interactive Viewer Operations on Graphs

For default bindings for interactive graph operations, see the "Plot" mode in the Ginga Quick Reference (*Plot mode*). The plot aide will initialize the viewer into Plot mode, so that it is continually ready for user interaction with the graph.

Zooming Graphs

Zooming on graphs is handled independently in the X and Y axes. Mouse or touchpad scrolling is usually used to zoom the X axes. Note that zooming will normally change any autoaxis setting to `off` since you are overriding the setting. See the Quick Reference link above for the cursor and key commands in plot mode for zooming and changing the autoaxis settings interactively.

Panning Graphs

Panning graphs is usually accomplished via scroll bars. When used with a properly configured scroll widget (as shown in the example above), the scrollbar aligned with the X axis becomes visible when `autoaxis_x` becomes `off`. Similarly, the scrollbar aligned with the Y axis becomes visible when `autoaxis_y` becomes `off`. The scroll bars can then be used independently to pan the graph in either axis.

Flipping and swapping

You can flip the X or Y axis and also swap axes, if it makes sense to do so. The usual key bindings for these can be found in the Ginga quick reference under the Transform commands (*Transform commands*).

5.4.3 Time Series Plots

Time-series plots are plots in which time is plotted on the X axis. Ginga has some special support for these in the module `ginga.plot.time_series`. There are classes for `XTimeAxis`, `TimePlotTitle`, `TimePlotBG` that can be used in place of the normal plot decor, and an `XYDataSource` that can be used to efficiently keep track of a large fixed array of (x, y) points, from which a `XYPlot` can be conveniently updated. When using custom plot decor like this, you need to add it manually via the plot aide's `add_plot_decor` method, instead of using `setup_standard_frame`:

```python
import ginga.plot.time_series as tsp
from ginga.canvas.types.plots import YAxis

# our plot
aide = PlotAide(viewer)
aide.settings.set(autoaxis_x='pan', autoaxis_y='vis')

bg = tsp.TimePlotBG(warn_y=70.0, alert_y=80.0, linewidth=2)
aide.add_plot_decor(bg)

title = tsp.TimePlotTitle(title="Humidity (%)")
aide.add_plot_decor(title)

x_axis = tsp.XTimeAxis(num_labels=4)
aide.add_plot_decor(x_axis)

y_axis = YAxis(num_labels=4)
aide.add_plot_decor(y_axis)
```

The `TimePlotBG` class has support for a `warning` and an `alert`. These are set when the current (last or right-most) Y value exceeds the `warn_y` or `alert_y` values. In the above example, the warning value is set to 70.0 and the alert value is set to 80.0. The warning and alert levels, if set, are shown by yellow and red lines in the plot background. Additionally, if the current Y value exceeds the warning level then the background turns yellow, as shown in the example application below; if it exceeds the alert level then the background turns pink (alert takes precedence over warning). If either `warn_y` or `alert_y` values are not passed, or set to `None`, then there will be no warning or alert lines or background color change in the plot.

For more detail on time series plots, see the example "plot_time_series.py" under the "examples/gw" folder.

5.5 Developing Modes

Modes are associated with the basic Ginga viewer widget and are used to make bindings between keyboard, mouse, trackpad and gesture events and certain operations on the viewer. There are lots of operations possible, but only a limited number of key and cursor events; by invoking a mode, certain keystrokes and cursor operations are enabled. These same keystrokes and cursor events might do something different in another mode. You can read about the standard modes and the default bindings that come with Ginga in *Modes* "concepts", "about" *Modes* and in the *Quick Reference*.
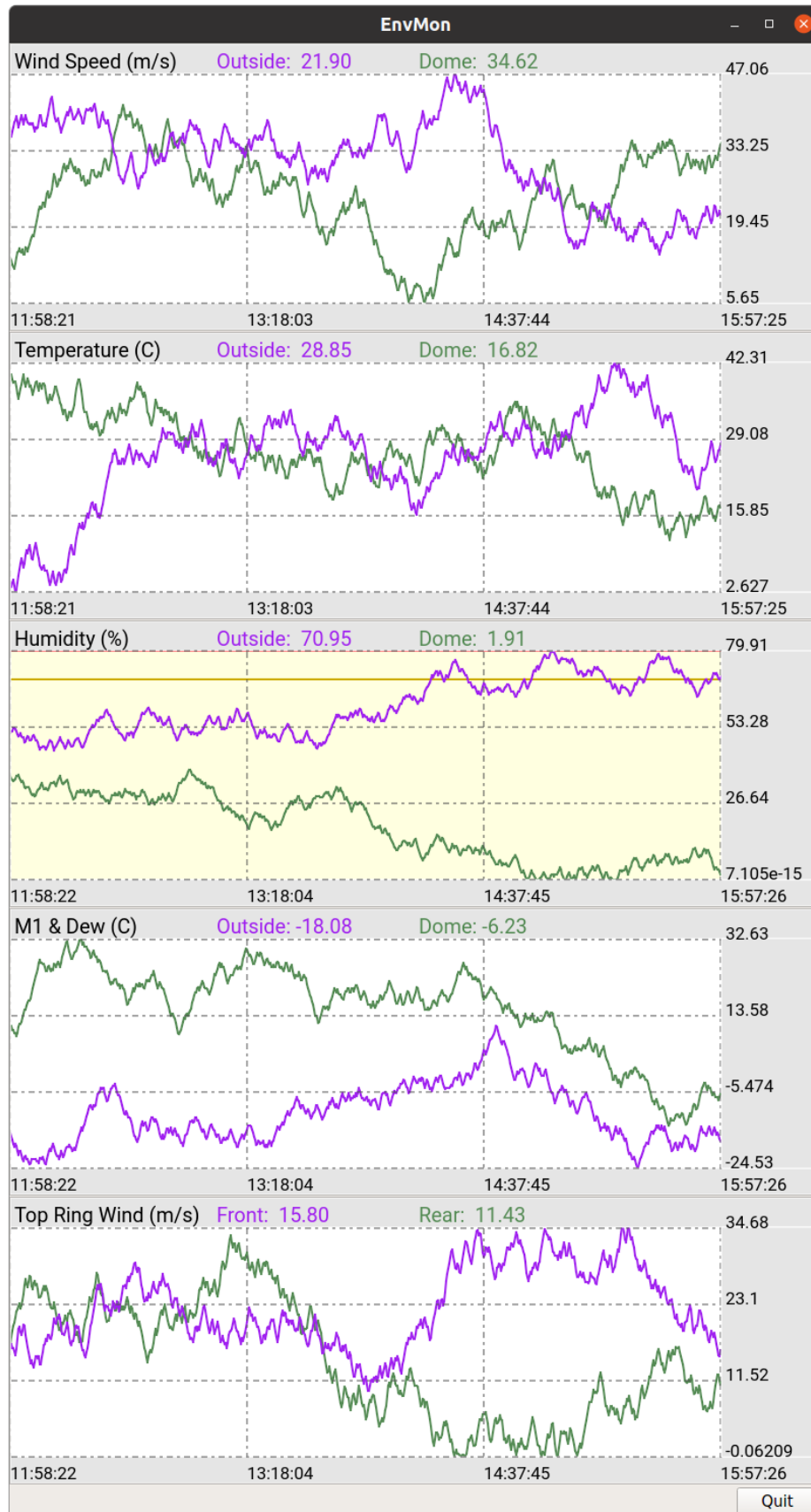
Fig. 2: An example of time series plots with fake environment data. Each plot contains 86400 seconds (24 hours) of data points, and can be zoomed and panned interactively using the methods described in the Quick Reference link above.

5.5.1 Writing a mode

If you are familiar with the Reference Viewer you may know about its plugin architecture. You can think of modes as mini-plugins that don't know about channels or other Reference Viewer entities, but only know about the Ginga viewer widget with which they are associated. The mode functions manipulate the viewer according to key and cursor bindings that they register.

Methods of a mode

A mode is a subclass derived from `Mode`. As far as the class structure goes, the following methods are required and explained below.

__init__

The constructor is called when the mode is first instantiated for a viewer; it should at a minimum call the superclass, and define an attribute named `actions` that is assigned a dictionary of bindings.

start

`start()` is called when the mode is activated by the user. It should do any initialization necessary since the constructor or the last call to `stop()`. For many cases, this may be nothing, since the purpose of most modes is simply to enable the new bindings for the duration of the mode.

stop

`stop()` is called when a mode is deactivated. It does any cleanup necessary and puts the mode in a state where it will be ready for a future call to `start()`.

__str__

The `__str__` method should be set to return a unique, lower-case string that is used to identify the mode and indicate the mode in the viewer mode indicator.

Other methods

All other methods are typically event callback bindings for cursor and key actions that are being handled by the mode, or helper functions used by those callbacks.

5.5.2 Bindings DSL and Event Callbacks

Ginga actions for binding cursor and key events is specified as a kind of Domain Specific Language that is compatible with the format for Ginga settings files. This is to allow the user to customize the bindings completely by providing a `bindings.cfg` file in their `$HOME/.ginga` folder. In the modes, this takes the form of a dictionary defined in the constructor and assigned to the attribute `actions`.

The key of each element of the dictionary usually matches the name of a method defined in the mode, and the value is a list of triggers (specified as strings) that should invoke the method. There are conventions that must be followed for both the name and the triggers.

Method names

For handling events, the (method) name must start with one of the following prefixes, which indicate the type of binding that will be made and the event handling:

- kp_ : a key press and release
- ms_ : a cursor (mouse, trackpad, etc) action: button down, button up, move
- sc_ : a scrolling (mouse, trackpad) action
- pi_ : a pinch gesture action
- pa_ : a pan gesture action

Note: Gestures are not supported equally on all platforms and toolkits. For example, under Qt on Mac OSX, a pan gesture is supported using the trackpad, but on Linux, that same gesture on a trackpad is handled as a scrolling action.

It is typical (but not essential) to have the next part of the method name match the mode in which it is implemented. Then, the suffix makes the purpose of the callback known.

Triggers

Triggers are spelled out as a string of the form: `<mode>+<modifier>+<action>` where either of the `<mode>` and `<modifier>` are optional (if omitted, then the preceding plus sign is also omitted).

`<mode>` is simply the name of the mode for which the binding should be active. This will match the name returned by the `__str__` method in the class implementation. If `<mode>` is omitted, the binding is assigned to the "modeless" operation, which means that it can be activated if no mode is currently activated and the event is not handled by some active canvas. This is mechanism by which "default" actions are handled by a mode for certain events even when no mode is currently active.

`<modifier>` stands for a keyboard modifier key. The usual defined ones are "shift" and "ctrl". An asterisk can be used as a wildcard in this position to indicate that the event should be bound for any combination of modifiers or lack of a modifier.

The `<action>` describes what is happening in combination with the `<mode>` and `<modifier>` to trigger the event. It is a key symbol, the name of a mouse button (usually "left", "middle", or "right"), "scroll" for the scroll action, and "pinch" or "pan" for the two gestures, respectively.

Examples

Assume that these are part of a `dict` being defined, or in a user's `bindings.cfg`.

kp_pan_page_up=['pan+*+page_up']

> The method that will be called is `kp_pan_page_up()`. The action that will trigger this is being in the "pan" mode, pressing any or no combinations of modifier keys with the key "page_up".

sc_zoom=['scroll']

> The method is `sc_zoom()`. It will be called when scrolling happens and the scrolling is not handled by any mode or an active canvas.

kp_zoom_fit=['backquote', 'pan+backquote']

> The method is `kp_zoom_fit()` and it will be called if the backquote key is pressed while in "pan" mode, and also any other time backquote is pressed and a mode or an active canvas does not handle it.

ms_rotate=['rotate+left']

> The method is `ms_rotate()` and it will be called when in the "rotate" mode and the left mouse button or trackpad is pressed, moved while pressed (a drag motion), and when released.

Event handler method signatures

Keyboard and cursor events both have the same callback method signature:

```
def kp_handler(self, viewer, event, data_x, data_y)
def ms_handler(self, viewer, event, data_x, data_y)
```

These are instance methods, as evidenced by the presence of `self`. The other parameters in the callback are:

- `viewer` : the viewer in which the action happened
- `event` : the event which was caught by the trigger
- `data_x`, `data_y` : the X/Y data coordinates where the cursor was when the event happened (this is also available in the `event`)

Note: The `data_x` and `data_y` parameters are for backward compatibility. It is recommended *not* to use them as they may be removed from the callback in a future version. Instead, use the values found in the `event` object.

Scroll, pinch, and pan events have a slightly different method signature:

```
def sc_handler(self, viewer, event)
def pi_handler(self, viewer, event)
def pa_handler(self, viewer, event)
```

These just receive the `viewer` and the `event` which precipitated the callback.

Note: To see what attributes are available in each event, see the `KeyEvent`, `PointEvent`, `ScrollEvent`, `PanEvent`, and `PinchEvent` in the *Reference/API* (look under `ginga.Bindings`).

5.6 Developing with Ginga

- modindex

Developers interested in using Ginga in their project will probably follow one of two logical development paths:

- using Ginga toolkit classes in a program of their own design, or

- starting with the full-featured reference viewer that comes with Ginga and customizing it for some special purpose, typically by modifying one of the plugins or writing a new plugin.

The first approach is probably best for when the developer has a custom application in mind, needs a minimal but powerful viewer or wants to develop an entirely new full-featured viewer. Developers interested in this direction should head over to the chapter on the viewer object (see *Using the Basic Ginga Viewer Object in Python Programs*).

The second approach is probably best for end users or developers that are mostly satisfied with the reference viewer as a general purpose tool and want to add some specific enhancements or functionality. Because the reference viewer is based on a flexible plugin architecture this is fairly straightforward to do.

5.6.1 Writing plugins for the reference viewer

The philosophy behind the design of the reference viewer distributed with the Ginga is that it is simply a flexible layout shell for instantiating instances of the Ginga view widgets described in the earlier section. All of the other important pieces of a modern FITS viewer–a panning widget, information panels, zoom widget, analysis panes–are implemented as plugins: encapsulated modules that interface with the viewing shell using a standardized API. This makes it easy to customize and to add, change or remove functionality in a very modular, flexible way.

The Ginga viewer divides the application window GUI into containers that hold either viewing widgets or plugins. The view widgets are called "channels" in the viewer nomenclature, and are a means of organizing images in the viewer, functioning much like "frames" in other viewers. A channel has a name and maintains its own history of images that have cycled through it. The user can create new channels as needed. For example, they might use different channels for different kinds of images: camera vs. spectrograph, or channels organized by CCD, or by target, or raw data vs. quick look, etc. In the default layout, the channel tabs are in the large middle pane, while the plugins occupy the left and right panes. Other layouts are possible, by simply changing a table used in the startup script.

Ginga distinguishes between two types of plugin: *global* and *local*. Global plugins are used where the functionality is generally enabled during the entire session with the viewer and where the plugin is active no matter which channel is currently under interaction with the user. Examples of global plugins include a panning view (a small, bird's-eye view of the image that shows a panning rectangle and allows graphical positioning of the pan region), a zoomed view (that shows an enlarged cutout of the area currently under the cursor), informational displays about world coordinates, FITS headers, thumbnails, etc. Figure *Two global plugins: Pan (top) and Info (bottom), shown sharing a tab.* shows an example of two global plugins occupying a notebook tab.

Local plugins are used for modal operations with images in specific channels. For example, the Pick plugin is used to perform stellar evaluation of objects, finding the center of the object and giving informational readings of the exact celestial coordinates, image quality, etc. The Pick plugin is only visible while the user has it open, and does not capture the mouse actions unless the channel it is operating on is selected. Thus one can have two different Pick operations going on concurrently on two different channels, for example, or a Pick operation in a camera channel, and a Cuts (line cuts) operation on a spectrograph channel. Figure *The Pick local plugin, shown occupying a tab.* shows an example of the Pick local plugin occupying a notebook tab.

Info	Header	Zoom

Name: SUPA0111876(
Object: M27
X: 444.413
Y: 1647.590
Value: 479.0
α: 19:58:47.959
δ: +22:46:55.43
Equinox: 2000.0
Dimensions: 2272x4273
Min: 170.0
Max: 65535.0
Zoom: 1/2.21x
Cut Low: 244.95
Cut High: 1046.85

Auto Levels   Cut Levels

Cut New: off
Zoom New: on

Preferences
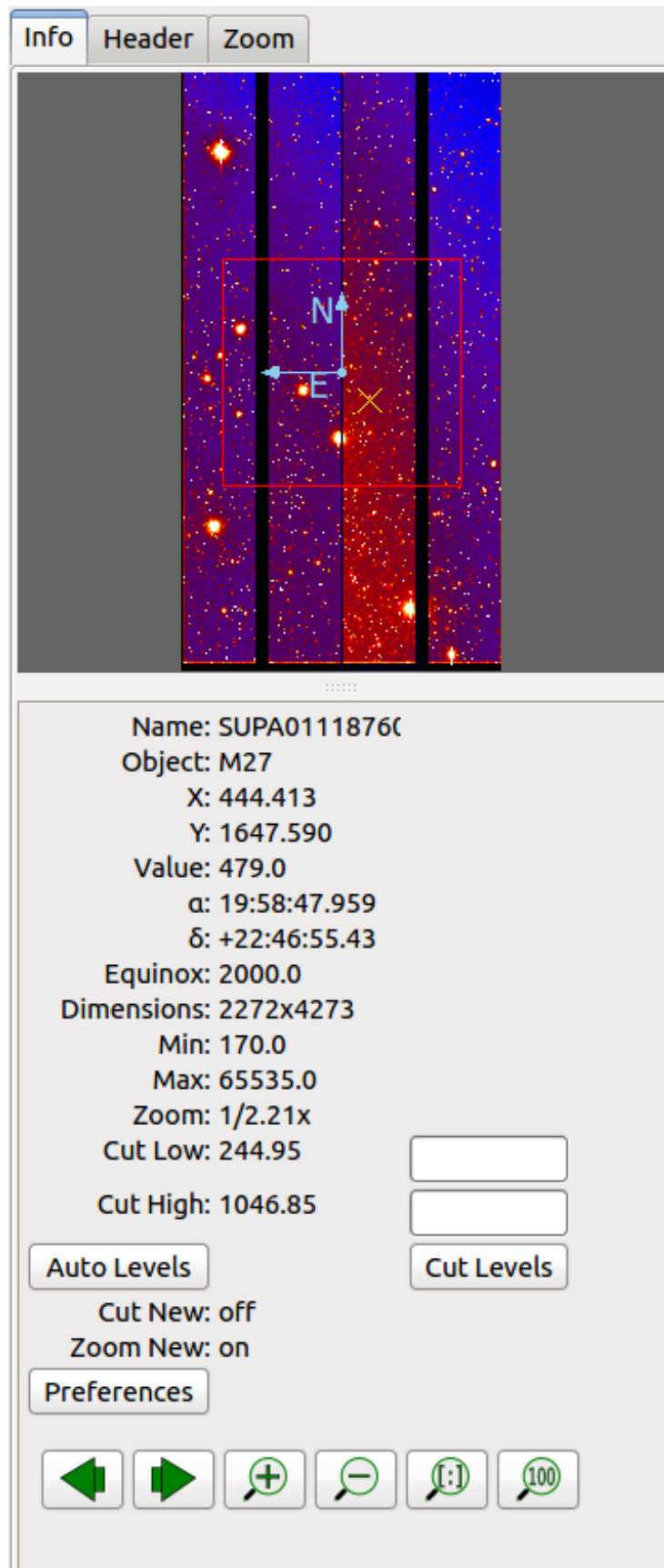
Fig. 3: Two global plugins: Pan (top) and Info (bottom), shown sharing a tab.
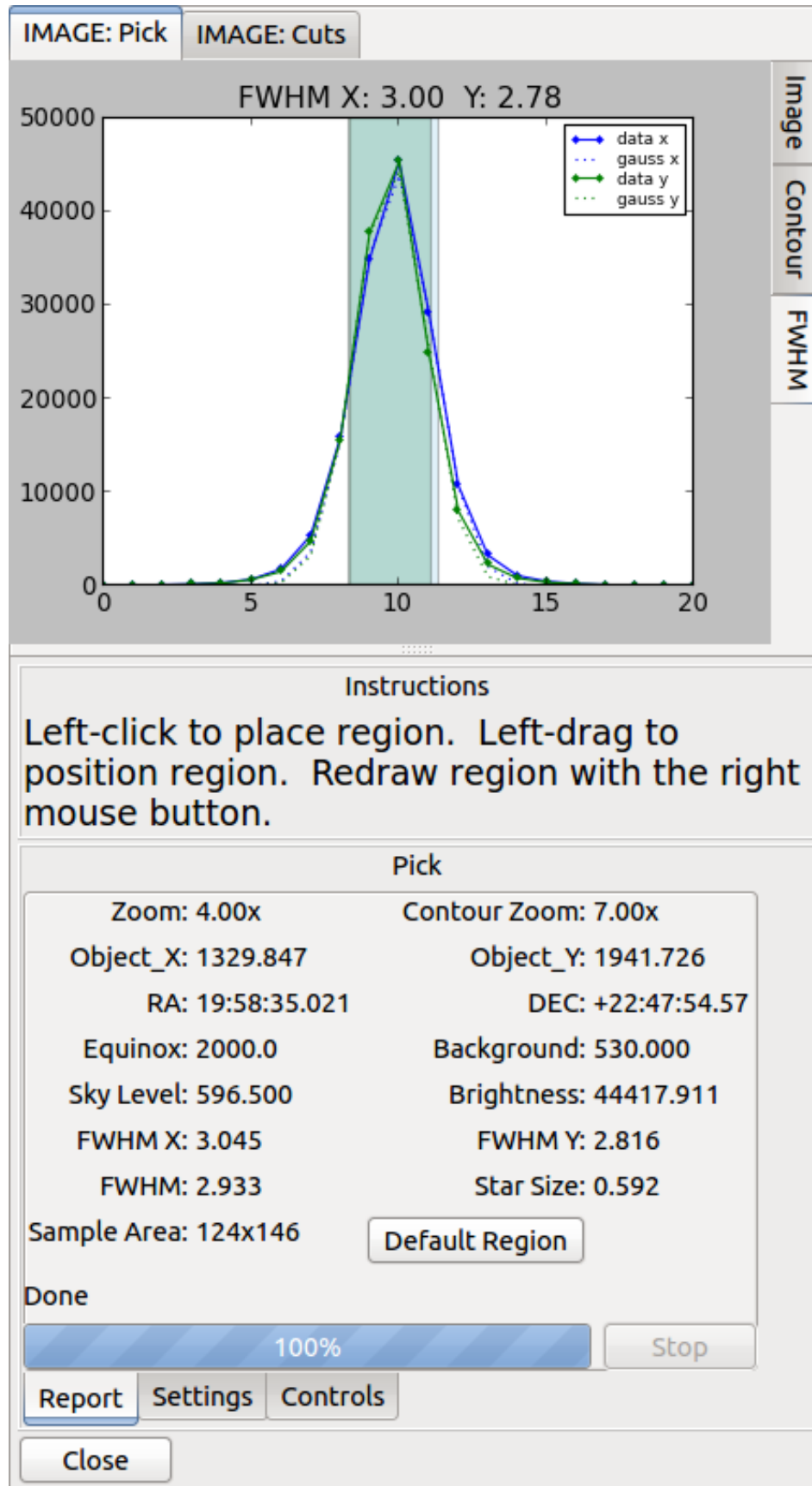
Fig. 4: The Pick local plugin, shown occupying a tab.

Anatomy of a Local Ginga Plugin

Let's take a look at a local plugin to understand the API for interfacing to the Ginga shell. In Listing 2, we show a stub for a local plugin.

```python
from ginga import GingaPlugin

class MyPlugin(GingaPlugin.LocalPlugin):

    def __init__(self, fv, fitsimage):
        super(MyPlugin, self).__init__(fv, fitsimage)

    def build_gui(self, container):
        pass

    def start(self):
        pass

    def stop(self):
        pass

    def pause(self):
        pass

    def resume(self):
        pass

    def redo(self):
        pass

    def __str__(self):
        return 'myplugin'
```

A little more fleshed out example: MyLocalPlugin

This is a skeleton for a local plugin. It is also good example of something that actually runs and can be copied as a template for a local plugin. This plugin is distributed with the Ginga package and can be loaded and invoked from a terminal:

```
ginga --plugins=MyLocalPlugin --loglevel=20 --log=/tmp/ginga.log
```

The plugin will be accessible via the "Operation" button in the Plugin Manager bar.

```python
from ginga import GingaPlugin
from ginga.misc import Widgets

# import any other modules you want here--it's a python world!

class MyLocalPlugin(GingaPlugin.LocalPlugin):

    def __init__(self, fv, fitsimage):
        """
        This method is called when the plugin is loaded for the  first
```

(continues on next page)

```
        time.  ``fv`` is a reference to the Ginga (reference viewer) shell
        and ``fitsimage`` is a reference to the specific viewer
        object associated with the channel on which the plugin is being
        invoked.
        You need to call the superclass initializer and then do any local
        initialization.
        """
        super(MyLocalPlugin, self).__init__(fv, fitsimage)

        # your local state and initialization code goes here

    def build_gui(self, container):
        """
        This method is called when the plugin is invoked.  It builds the
        GUI used by the plugin into the widget layout passed as
        ``container``.
        This method may be called many times as the plugin is opened and
        closed for modal operations.  The method may be omitted if there
        is no GUI for the plugin.

        This specific example uses the GUI widget set agnostic wrappers
        to build the GUI, but you can also just as easily use explicit
        toolkit calls here if you only want to support one widget set.
        """
        top = Widgets.VBox()
        top.set_border_width(4)

        # this is a little trick for making plugins that work either in
        # a vertical or horizontal orientation.  It returns a box container,
        # a scroll widget and an orientation ('vertical', 'horizontal')
        vbox, sw, orientation = Widgets.get_oriented_box(container)
        vbox.set_border_width(4)
        vbox.set_spacing(2)

        # Take a text widget to show some instructions
        self.msgFont = self.fv.getFont("sansFont", 12)
        tw = Widgets.TextArea(wrap=True, editable=False)
        tw.set_font(self.msgFont)
        self.tw = tw

        # Frame for instructions and add the text widget with another
        # blank widget to stretch as needed to fill emp
        fr = Widgets.Frame("Instructions")
        vbox2 = Widgets.VBox()
        vbox2.add_widget(tw)
        vbox2.add_widget(Widgets.Label(''), stretch=1)
        fr.set_widget(vbox2)
        vbox.add_widget(fr, stretch=0)

        # Add a spacer to stretch the rest of the way to the end of the
        # plugin space
        spacer = Widgets.Label('')
```

```python
        vbox.add_widget(spacer, stretch=1)

        # scroll bars will allow lots of content to be accessed
        top.add_widget(sw, stretch=1)

        # A button box that is always visible at the bottom
        btns = Widgets.HBox()
        btns.set_spacing(3)

        # Add a close button for the convenience of the user
        btn = Widgets.Button("Close")
        btn.add_callback('activated', lambda w: self.close())
        btns.add_widget(btn, stretch=0)
        btns.add_widget(Widgets.Label(''), stretch=1)
        top.add_widget(btns, stretch=0)

        # Add our GUI to the container
        container.add_widget(top, stretch=1)
        # NOTE: if you are building a GUI using a specific widget toolkit
        # (e.g. Qt) GUI calls, you need to extract the widget or layout
        # from the non-toolkit specific container wrapper and call on that
        # to pack your widget, e.g.:
        #cw = container.get_widget()
        #cw.addWidget(widget, stretch=1)

    def close(self):
        """
        Example close method.  You can use this method and attach it as a
        callback to a button that you place in your GUI to close the plugin
        as a convenience to the user.
        """
        chname = self.fv.get_channel_name(self.fitsimage)
        self.fv.stop_local_plugin(chname, str(self))
        return True

    def start(self):
        """
        This method is called just after ``build_gui()`` when the plugin
        is invoked.  This method may be called many times as the plugin is
        opened and closed for modal operations.  This method may be omitted
        in many cases.
        """
        self.tw.set_text("""This plugin doesn't do anything interesting.""")
        self.resume()

    def pause(self):
        """
        This method is called when the plugin loses focus.
        It should take any actions necessary to stop handling user
        interaction events that were initiated in ``start()`` or
        ``resume()``.
        This method may be called many times as the plugin is focused
```

```python
        or defocused.  It may be omitted if there is no user event handling
        to disable.
        """
        pass

    def resume(self):
        """
        This method is called when the plugin gets focus.
        It should take any actions necessary to start handling user
        interaction events for the operations that it does.
        This method may be called many times as the plugin is focused or
        defocused.  The method may be omitted if there is no user event
        handling to enable.
        """
        pass

    def stop(self):
        """
        This method is called when the plugin is stopped.
        It should perform any special clean up necessary to terminate
        the operation.  The GUI will be destroyed by the plugin manager
        so there is no need for the stop method to do that.
        This method may be called many  times as the plugin is opened and
        closed for modal operations, and may be omitted if there is no
        special cleanup required when stopping.
        """
        pass

    def redo(self):
        """
        This method is called when the plugin is active and a new
        image is loaded into the associated channel.  It can optionally
        redo the current operation on the new image.  This method may be
        called many times as new images are loaded while the plugin is
        active.  This method may be omitted.
        """
        pass

    def __str__(self):
        """
        This method should be provided and should return the lower case
        name of the plugin.
        """
        return 'mylocalplugin'
```

The instance variables "fv" and "fitsimage" will be assigned by the superclass initializer to self.fv and self.fitsimage–these are the reference viewer "shell" and the ginga display object respectively. To interact with the viewer you will be calling methods on one or both of these objects.

The "fitsimage" object is the ginga image viewer object associated with the channel. You can get a good idea of how to programmatically manipulate this viewer here *Ginga Image Viewer Operations* and by browsing the source of the many plugins distributed with Ginga. Most of them are not very long or complex. Also, a plugin can include any Python packages or modules that it wants and programming one is essentially similar to writing any other Python program. We

---

suggest picking a plugin that looks or does something similar to what you are interested in, copying it, and modifying it to fit your needs.

Launching and Debugging Your Plugin

The easiest way to start out is to create a plugins directory under your ginga configuration area. In a terminal:

```
mkdir $HOME/.ginga/plugins
```

Put your plugin in there (a good one to start with is to modify the MyLocalPlugin example that comes with Ginga):

```
cd .../ginga/examples/reference-viewer
cp MyLocalPlugin.py $HOME/.ginga/plugins/MyPlugin.py
```

To load it when the reference viewer starts (and add some logging to stderr as well as to a file):

```
ginga --plugins=MyPlugin --loglevel=20 --stderr --log=/tmp/ginga.log
```

To start the plugin from within the reference viewer, use the Plugin Manager bar just below the color and readout bars. Use the "Operation" menu to select your plugin and it should be launched in the right panel.

If you don't see the name of your plugin in the Operation menu, then there was probably an error trying to load it. Examine the log and search for the name of your plugin–you should find some error message associated with it.

If you select your plugin from the menu, but it doesn't launch a GUI, there may be a problem or error in the plugin file. Again, examine the log and search for the name of your plugin–you should find some error message associated with it. It may help for you to add some debugging messages to your plugin (either using self.logger.debug(". . .") or simple print statements to stdout) to gauge the progress of building the gui and plugin starting.

If the plugin launches, but encounters an error building the GUI, it should show some error messages (and probably a stack trace) in placeholders in the right panel in the container where it tried to build the GUI or possibly under the Errors tab.

Note: Ginga has a feature for quickly reloading plugins to facilitate rapid debugging cycles. If it is not already running, start the "Command Line" plugin from the "Plugins->Debug" menu in the menu bar. If your plugin launched (but has some error), make sure you have closed your plugin by right clicking (or Control + click on Mac touchpad) on the small box representing your plugin in the Plugin Manager bar and selecting "Stop". In the Command Line plugin, there is a small box labeled "Type command here:". Use the command "reload_local <plugin_name>"–this will reload the python module representing your plugin and you should be able to immediately restart it using the Plugin Manager bar as described above (if the plugin is of the global plugin variety, use the command "reload_global" instead).

If you have edited third party modules that are included in the plugin, this will not be enough to pick up those changes.

A more complex example: The `Ruler` Plugin

Finally, in Listing 3 we show a completed plugin for `Ruler`. The purpose of this plugin to draw triangulation (distance measurement) rulers on the image. For reference, you may want to refer to the ruler shown in *The Ruler local plugin GUI, shown occupying a tab.*.

```
#
# Ruler.py -- Ruler plugin for Ginga reference viewer
#
from ginga import GingaPlugin
```
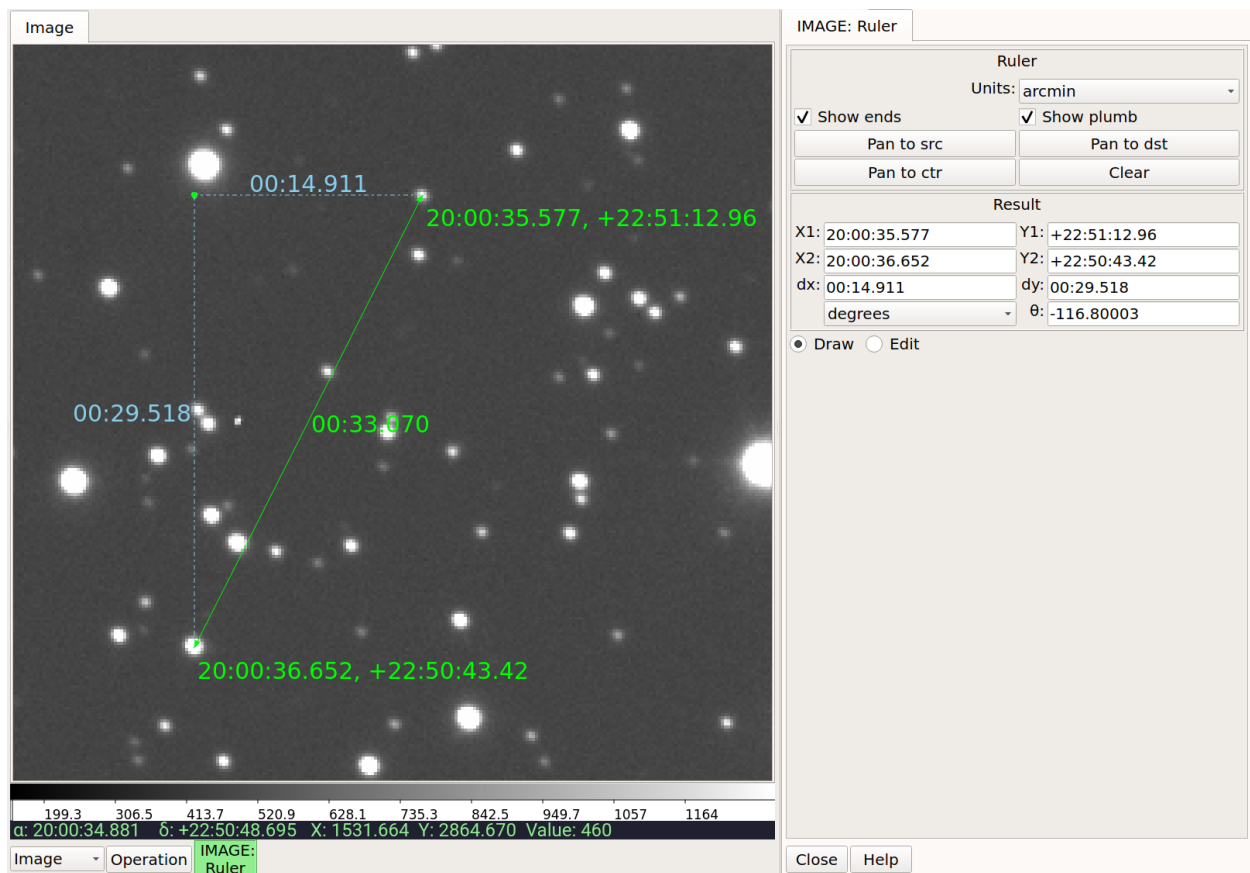
Fig. 5: The `Ruler` local plugin GUI, shown occupying a tab.

```python
from ginga.gw import Widgets

class Ruler(GingaPlugin.LocalPlugin):

    def __init__(self, fv, fitsimage):
        # superclass defines some variables for us, like logger
        super(Ruler, self).__init__(fv, fitsimage)

        self.rulecolor = 'green'
        self.layertag = 'ruler-canvas'
        self.ruletag = None

        self.dc = fv.get_draw_classes()
        canvas = self.dc.DrawingCanvas()
        canvas.enable_draw(True)
        canvas.enable_edit(True)
        canvas.set_drawtype('ruler', color='cyan')
        canvas.set_callback('draw-event', self.wcsruler)
        canvas.set_callback('draw-down', self.clear)
        canvas.set_callback('edit-event', self.edit_cb)
        canvas.set_draw_mode('draw')
        canvas.set_surface(self.fitsimage)
        canvas.register_for_cursor_drawing(self.fitsimage)
        canvas.name = 'Ruler-canvas'
        self.canvas = canvas

        self.w = None
        self.unittypes = ('arcmin', 'degrees', 'pixels')
        self.units = 'arcmin'

    def build_gui(self, container):
        top = Widgets.VBox()
        top.set_border_width(4)

        vbox, sw, orientation = Widgets.get_oriented_box(container)
        vbox.set_border_width(4)
        vbox.set_spacing(2)

        self.msgFont = self.fv.get_font("sansFont", 12)
        tw = Widgets.TextArea(wrap=True, editable=False)
        tw.set_font(self.msgFont)
        self.tw = tw

        fr = Widgets.Expander("Instructions")
        fr.set_widget(tw)
        vbox.add_widget(fr, stretch=0)

        fr = Widgets.Frame("Ruler")

        captions = (('Units:', 'label', 'Units', 'combobox'),
                    )
        w, b = Widgets.build_info(captions, orientation=orientation)
```

```python
        self.w = b

        combobox = b.units
        for name in self.unittypes:
            combobox.append_text(name)
        index = self.unittypes.index(self.units)
        combobox.set_index(index)
        combobox.add_callback('activated', lambda w, idx: self.set_units())

        fr.set_widget(w)
        vbox.add_widget(fr, stretch=0)

        mode = self.canvas.get_draw_mode()
        hbox = Widgets.HBox()
        btn1 = Widgets.RadioButton("Draw")
        btn1.set_state(mode == 'draw')
        btn1.add_callback('activated', lambda w, val: self.set_mode_cb('draw', val))
        btn1.set_tooltip("Choose this to draw a ruler")
        self.w.btn_draw = btn1
        hbox.add_widget(btn1)

        btn2 = Widgets.RadioButton("Edit", group=btn1)
        btn2.set_state(mode == 'edit')
        btn2.add_callback('activated', lambda w, val: self.set_mode_cb('edit', val))
        btn2.set_tooltip("Choose this to edit a ruler")
        self.w.btn_edit = btn2
        hbox.add_widget(btn2)

        hbox.add_widget(Widgets.Label(''), stretch=1)
        vbox.add_widget(hbox, stretch=0)

        spacer = Widgets.Label('')
        vbox.add_widget(spacer, stretch=1)

        top.add_widget(sw, stretch=1)

        btns = Widgets.HBox()
        btns.set_spacing(3)

        btn = Widgets.Button("Close")
        btn.add_callback('activated', lambda w: self.close())
        btns.add_widget(btn, stretch=0)
        btns.add_widget(Widgets.Label(''), stretch=1)
        top.add_widget(btns, stretch=0)

        container.add_widget(top, stretch=1)

    def set_units(self):
        index = self.w.units.get_index()
        units = self.unittypes[index]
        self.canvas.set_drawtype('ruler', color='cyan', units=units)
```

```python
        if self.ruletag is not None:
            obj = self.canvas.get_object_by_tag(self.ruletag)
            if obj.kind == 'ruler':
                obj.units = units
                self.canvas.redraw(whence=3)
        return True

    def close(self):
        chname = self.fv.get_channel_name(self.fitsimage)
        self.fv.stop_local_plugin(chname, str(self))
        return True

    def instructions(self):
        self.tw.set_text("""Draw (or redraw) a line with the cursor.

Display the Zoom tab at the same time to precisely see detail while drawing.""")

    def start(self):
        self.instructions()
        # start ruler drawing operation
        p_canvas = self.fitsimage.get_canvas()
        if not p_canvas.has_object(self.canvas):
            p_canvas.add(self.canvas, tag=self.layertag)

        self.canvas.delete_all_objects()
        self.resume()

    def pause(self):
        self.canvas.ui_set_active(False)

    def resume(self):
        self.canvas.ui_set_active(True)
        self.fv.show_status("Draw a ruler with the right mouse button")

    def stop(self):
        # remove the canvas from the image
        p_canvas = self.fitsimage.get_canvas()
        try:
            p_canvas.delete_object_by_tag(self.layertag)
        except:
            pass
        self.canvas.ui_set_active(False)
        self.fv.show_status("")

    def redo(self):
        obj = self.canvas.get_object_by_tag(self.ruletag)
        if obj.kind != 'ruler':
            return True
        # redraw updates ruler measurements
        self.canvas.redraw(whence=3)

    def clear(self, canvas, button, data_x, data_y):
```

```
        self.canvas.delete_all_objects()
        self.ruletag = None
        return False

    def wcsruler(self, surface, tag):
        obj = self.canvas.get_object_by_tag(tag)
        if obj.kind != 'ruler':
            return True
        # remove the old ruler
        try:
            self.canvas.delete_object_by_tag(self.ruletag)
        except:
            pass

        # change some characteristics of the drawn image and
        # save as the new ruler
        self.ruletag = tag
        obj.color = self.rulecolor
        obj.cap = 'ball'
        self.canvas.redraw(whence=3)

    def edit_cb(self, canvas, obj):
        self.redo()
        return True

    def edit_select_ruler(self):
        if self.ruletag is not None:
            obj = self.canvas.get_object_by_tag(self.ruletag)
            self.canvas.edit_select(obj)
        else:
            self.canvas.clear_selected()
        self.canvas.update_canvas()

    def set_mode_cb(self, mode, tf):
        if tf:
            self.canvas.set_draw_mode(mode)
            if mode == 'edit':
                self.edit_select_ruler()
        return True

    def __str__(self):
        return 'ruler'
```

#END

This plugin shows a standard design pattern typical to local plugins. Often one is wanting to draw or plot something on top of the image below. The `CanvasView` widget used by Ginga allows this to be done very cleanly and conveniently by adding a `DrawingCanvas` object to the image and drawing on that. Canvases can be layered on top of each other in a manner analogous to "layers" in an image editing program. Since each local plugin maintains it's own canvas, it is very easy to encapsulate the logic for drawing on and dealing with the objects associated with that plugin. We use this technique in the Ruler plugin. When the plugin is loaded (refer to `__init__()` method), it creates a canvas, enables drawing on it, sets the draw type and registers a callback for drawing events. When `start()` is called it adds that canvas to the widget. When `stop()` is called it removes the canvas from the widget (but does not destroy the canvas).

`pause()` disables user interaction on the canvas and `resume()` reenables that interaction. `redo()` simply redraws the ruler with new measurements taken from any new image that may have been loaded. In the `__init__()` method you will notice a `set_surface()` call that associates this canvas with a `ImageView`-based widget—this is the key for the canvas to utilize WCS information for correct plotting. All the other methods shown are support methods for doing the ruler drawing operation and interacting with the plugin GUI.

Writing a Global Plugin

The last example was focused on writing a local plugin. Global plugins employ a nearly identical API to that shown in Listing 2, except that the constructor does not take a `fitsimage` parameter. `pause()` and `resume()` can safely be omitted. Like local plugins, `build_gui()` can be omitted if there is no GUI associated with the plugin.

A template: MyGlobalPlugin

This is a skeleton for a global plugin, and serves as a decent example of something that can be copied as a template for a global plugin. This plugin is distributed with the Ginga package and can be loaded and invoked from a terminal:

```
ginga --modules=MyGlobalPlugin --loglevel=20 --log=/tmp/ginga.log
```

The plugin will be started at program startup and can be seen in the "MyGlobalPlugin" tab in the right panel. Watch the status message as you create new channels, delete channels or load images into channels.

```python
from ginga import GingaPlugin
from ginga.misc import Widgets

# import any other modules you want here--it's a python world!

class MyGlobalPlugin(GingaPlugin.GlobalPlugin):

    def __init__(self, fv):
        """
        This method is called when the plugin is loaded for the  first
        time.  ``fv`` is a reference to the Ginga (reference viewer) shell.

        You need to call the superclass initializer and then do any local
        initialization.
        """
        super(MyGlobalPlugin, self).__init__(fv)

        # Your initialization here

        # Create some variables to keep track of what is happening
        # with which channel
        self.active = None

        # Subscribe to some interesting callbacks that will inform us
        # of channel events.  You may not need these depending on what
        # your plugin does
        fv.set_callback('add-channel', self.add_channel)
        fv.set_callback('delete-channel', self.delete_channel)
        fv.set_callback('active-image', self.focus_cb)
```

(continues on next page)

```python
def build_gui(self, container):
    """
    This method is called when the plugin is invoked.  It builds the
    GUI used by the plugin into the widget layout passed as
    ``container``.
    This method could be called several times if the plugin is opened
    and closed.  The method may be omitted if there is no GUI for the
    plugin.

    This specific example uses the GUI widget set agnostic wrappers
    to build the GUI, but you can also just as easily use explicit
    toolkit calls here if you only want to support one widget set.
    """
    top = Widgets.VBox()
    top.set_border_width(4)

    # this is a little trick for making plugins that work either in
    # a vertical or horizontal orientation.  It returns a box container,
    # a scroll widget and an orientation ('vertical', 'horizontal')
    vbox, sw, orientation = Widgets.get_oriented_box(container)
    vbox.set_border_width(4)
    vbox.set_spacing(2)

    # Take a text widget to show some instructions
    self.msgFont = self.fv.getFont("sansFont", 12)
    tw = Widgets.TextArea(wrap=True, editable=False)
    tw.set_font(self.msgFont)
    self.tw = tw

    # Frame for instructions and add the text widget with another
    # blank widget to stretch as needed to fill emp
    fr = Widgets.Frame("Status")
    vbox2 = Widgets.VBox()
    vbox2.add_widget(tw)
    vbox2.add_widget(Widgets.Label(''), stretch=1)
    fr.set_widget(vbox2)
    vbox.add_widget(fr, stretch=0)

    # Add a spacer to stretch the rest of the way to the end of the
    # plugin space
    spacer = Widgets.Label('')
    vbox.add_widget(spacer, stretch=1)

    # scroll bars will allow lots of content to be accessed
    top.add_widget(sw, stretch=1)

    # A button box that is always visible at the bottom
    btns = Widgets.HBox()
    btns.set_spacing(3)

    # Add a close button for the convenience of the user
    btn = Widgets.Button("Close")
```

```python
        btn.add_callback('activated', lambda w: self.close())
        btns.add_widget(btn, stretch=0)
        btns.add_widget(Widgets.Label(''), stretch=1)
        top.add_widget(btns, stretch=0)

        # Add our GUI to the container
        container.add_widget(top, stretch=1)
        # NOTE: if you are building a GUI using a specific widget toolkit
        # (e.g. Qt) GUI calls, you need to extract the widget or layout
        # from the non-toolkit specific container wrapper and call on that
        # to pack your widget, e.g.:
        #cw = container.get_widget()
        #cw.addWidget(widget, stretch=1)

    def get_channel_info(self, fitsimage):
        chname = self.fv.get_channelName(fitsimage)
        chinfo = self.fv.get_channelInfo(chname)
        return chinfo

    def set_info(self, text):
        self.tw.set_text(text)

    # CALLBACKS

    def add_channel(self, viewer, chinfo):
        """
        Callback from the reference viewer shell when a channel is added.
        """
        self.set_info("Channel '%s' has been added" % (
                chinfo.name))
        # Register for new image callbacks on this channel's canvas
        fitsimage = chinfo.fitsimage
        fitsimage.set_callback('image-set', self.new_image_cb)

    def delete_channel(self, viewer, chinfo):
        """
        Callback from the reference viewer shell when a channel is deleted.
        """
        self.set_info("Channel '%s' has been deleted" % (
                chinfo.name))
        return True

    def focus_cb(self, viewer, fitsimage):
        """
        Callback from the reference viewer shell when the focus changes
        between channels.
        """
        chinfo = self.get_channel_info(fitsimage)
        chname = chinfo.name

        if self.active != chname:
            # focus has shifted to a different channel than our idea
```

```python
        # of the active one
        self.active = chname
        self.set_info("Focus is now in channel '%s'" % (
            self.active))
    return True

def new_image_cb(self, fitsimage, image):
    """
    Callback from the reference viewer shell when a new image has
    been added to a channel.
    """
    chinfo = self.get_channel_info(fitsimage)
    chname = chinfo.name

    # Only update our GUI if the activity is in the focused
    # channel
    if self.active == chname:
        imname = image.get('name', 'NONAME')
        self.set_info("A new image '%s' has been added to channel %s" % (
            imname, chname))
    return True

def start(self):
    """
    This method is called just after ``build_gui()`` when the plugin
    is invoked.  This method could be called more than once if the
    plugin is opened and closed.  This method may be omitted
    in many cases.
    """
    pass

def stop(self):
    """
    This method is called when the plugin is stopped.
    It should perform any special clean up necessary to terminate
    the operation.  This method could be called more than once if
    the plugin is opened and closed, and may be omitted if there is no
    special cleanup required when stopping.
    """
    pass

def close(self):
    self.fv.stop_global_plugin(str(self))
    return True

def __str__(self):
    """
    This method should be provided and should return the lower case
    name of the plugin.
    """
    return 'myglobalplugin'
```

Writing Separately Installable Plugins

If you want to distribute your plugin(s) as a separately installable package and have Ginga discover them when it starts up, you can use the Ginga Plugin Template to write your own package that installs plugins.

You can include as many plugins in your package as you want. You write your plugins in exactly the same way as described above, and they can be either global or local. For details, clone the repo at the link above and follow the directions in the README.

5.7 Using the Basic Ginga Viewer Object in Python Programs

- modindex

The core design principle of the Ginga project is to make it possible to easily build powerful image viewers in Python with many possible GUI toolkits.

This chapter is for developers who want to use only the Ginga rendering class in a program of their own design (not customizing the reference viewer).

5.7.1 Using the basic rendering class in new programs

Ginga basically follows the Model-View-Controller (MVC) design pattern, that is described in more detail in the chapter on internals (see *Ginga Internals*). The "view" classes are rooted in the base class `ImageView`. Ginga supports backends for different widget sets through various subclasses of this class.

Typically, a developer picks a GUI toolkit that has a supported backend (Gtk 3, Qt 5/6, Tk, matplotlib, HTML5 canvas) and writes a GUI program using that widget set with the typical Python toolkit bindings and API. Where they want a image view pane they instantiate the appropriate subclass of `ImageView` (usually a `CanvasView`), and using the `get_widget()` call extract the native widget and insert it into the GUI layout. A reference should also be kept to the view object, as this is typically what you will be calling methods on to control the viewer (see *Ginga Image Viewer Operations*).

Ginga does not create any additional GUI components beyond the image pane itself, however it does provide a standard set of keyboard and mouse bindings on the host widget that can be enabled, disabled or changed. The user interface bindings are configurable via a pluggable `Bindings` class which constitutes the "controller" part of the MVC design. There are a plethora of callbacks that can be registered, allowing the user to create their own custom user interface for manipulating the view. Of course, the developer can add many different GUI widgets from the selected toolkit to supplement or replace these built in controls.

Listing 1 shows a code listing for a simple graphical FITS viewer built using the subclass `CanvasView` from the module `ImageViewQt` (screenshot in Figure *A simple, "bare bones" FITS viewer written in Qt.*) written in around 100 or so lines of Python. It creates a window containing an image view and two buttons. This example will open FITS files dragged and dropped on the image window or via a dialog popped up when clicking the "Open File" button.

```python
#! /usr/bin/env python
#
# example1_qt.py -- Simple FITS viewer using the Ginga toolkit
#                    and Qt widgets.
#
import sys

from ginga.misc import log
from ginga.qtw.QtHelp import QtGui, QtCore
from ginga.qtw.ImageViewQt import CanvasView, ScrolledView
```

(continues on next page)
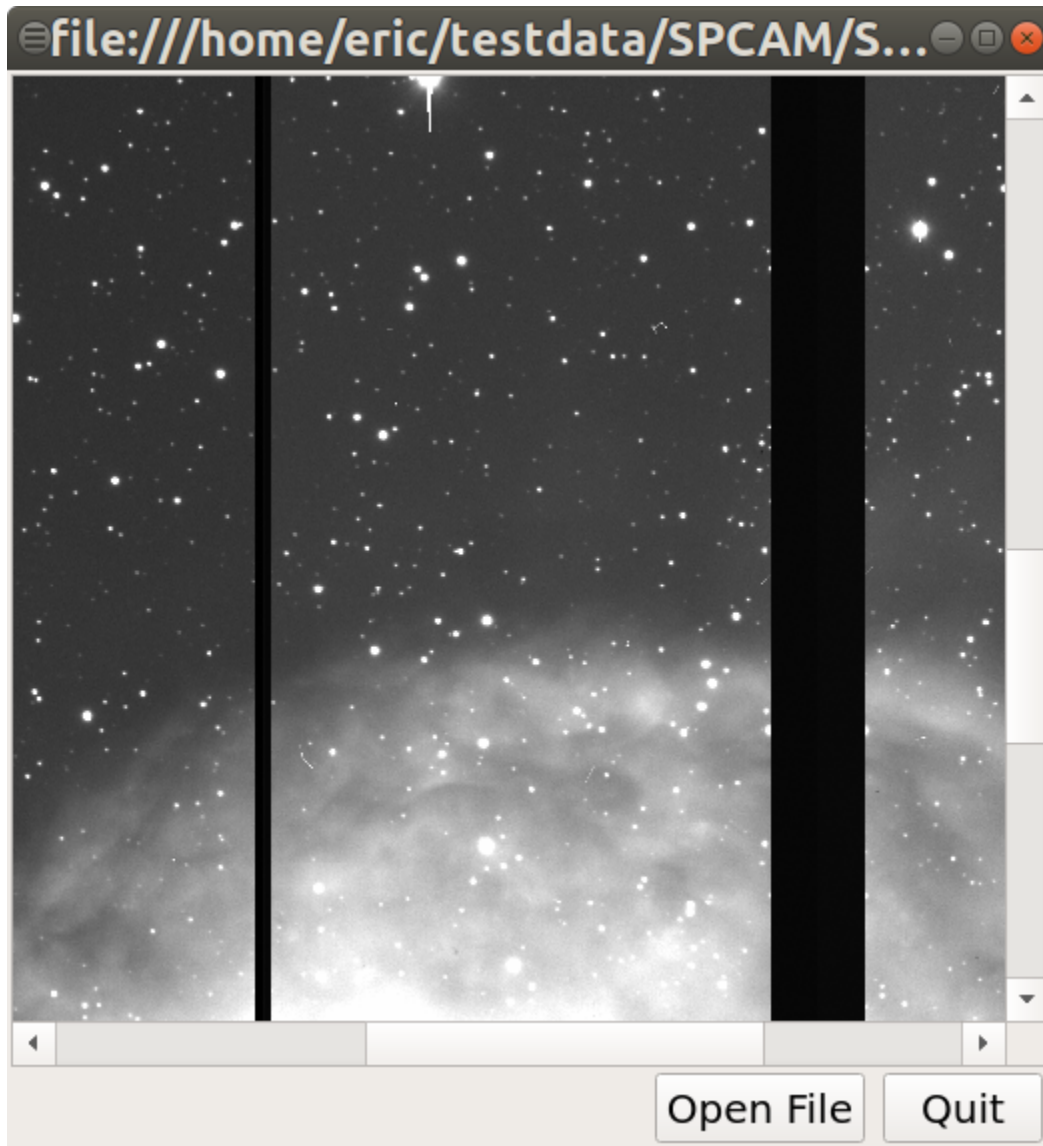
Fig. 6: A simple, "bare bones" FITS viewer written in Qt.

```python
from ginga.util.loader import load_data


class FitsViewer(QtGui.QMainWindow):

    def __init__(self, logger):
        super(FitsViewer, self).__init__()
        self.logger = logger

        # create the ginga viewer and configure it
        fi = CanvasView(self.logger, render='widget')
        fi.enable_autocuts('on')
        fi.set_autocut_params('zscale')
        fi.enable_autozoom('on')
        fi.set_callback('drag-drop', self.drop_file)
        fi.set_bg(0.2, 0.2, 0.2)
        fi.ui_set_active(True)
        self.fitsimage = fi

        # enable some user interaction
        bd = fi.get_bindings()
        bd.enable_all(True)

        w = fi.get_widget()
        w.resize(512, 512)

        # add scrollbar interface around this viewer
        sw = ScrolledView(fi)

        vbox = QtGui.QVBoxLayout()
        vbox.setContentsMargins(QtCore.QMargins(2, 2, 2, 2))
        vbox.setSpacing(1)
        vbox.addWidget(sw, stretch=1)

        hbox = QtGui.QHBoxLayout()
        hbox.setContentsMargins(QtCore.QMargins(4, 2, 4, 2))

        wopen = QtGui.QPushButton("Open File")
        wopen.clicked.connect(self.open_file)
        wquit = QtGui.QPushButton("Quit")
        wquit.clicked.connect(self.quit)

        hbox.addStretch(1)
        for w in (wopen, wquit):
            hbox.addWidget(w, stretch=0)

        hw = QtGui.QWidget()
        hw.setLayout(hbox)
        vbox.addWidget(hw, stretch=0)

        vw = QtGui.QWidget()
        self.setCentralWidget(vw)
```

```python
        vw.setLayout(vbox)

    def load_file(self, filepath):
        image = load_data(filepath, logger=self.logger)
        self.fitsimage.set_image(image)
        self.setWindowTitle(filepath)

    def open_file(self):
        res = QtGui.QFileDialog.getOpenFileName(self, "Open FITS file",
                                                ".", "FITS files (*.fits)")
        if isinstance(res, tuple):
            fileName = res[0]
        else:
            fileName = str(res)
        if len(fileName) != 0:
            self.load_file(fileName)

    def drop_file(self, fitsimage, paths):
        fileName = paths[0]
        self.load_file(fileName)

    def quit(self, *args):
        self.logger.info("Attempting to shut down the application...")
        self.deleteLater()


def main(options, args):

    app = QtGui.QApplication(sys.argv)

    # ginga needs a logger.
    # If you don't want to log anything you can create a null logger by
    # using null=True in this call instead of log_stderr=True
    logger = log.get_logger("example1", log_stderr=True, level=40)

    w = FitsViewer(logger)
    w.resize(524, 540)
    w.show()
    app.setActiveWindow(w)
    w.raise_()
    w.activateWindow()

    if len(args) > 0:
        w.load_file(args[0])

    app.exec_()


if __name__ == '__main__':
    main(None, sys.argv[1:])
```

Looking at the constructor for this particular viewer, you can see where we create a `CanvasView` object. On this object we enable automatic cut levels (using the 'zscale' algorithm), configure it to auto zoom the image to fit the window and

set a callback function for files dropped on the window. We extract the user-interface bindings with `get_bindings()`, and on this object enable standard user interactive controls for all the possible key and mouse operations. We then extract the platform-specific widget (Qt-based, in this case) using `get_widget()` and pack it into a Qt container along with a couple of buttons to complete the viewer.

Scanning down the code a bit, we can see that whether by dragging and dropping or via the click to open, we ultimately call the `load_file()` method to get the data into the viewer. `load_file()` creates an `AstroImage` object (the "model" part of our MVC design), which is then passed to the viewer via the `set_image()` method. `AstroImage` objects have methods for ingesting data via a file path, an `astropy.io.fits` HDU or a bare `Numpy` data array. For a reference on the model, see here:ref:`_ch-image-data-wrappers`.

Many of these sorts of examples for all supported backends are contained in the `examples` directory in the source distribution.

For a list of many methods provided by the viewer object see this reference *Ginga Image Viewer Operations*. You can also click on the module index link at the top of this chapter and then click on the link for `ImageViewBase`.

Graphics plotting with Ginga

A `CanvasView` actually pairs a view with a canvas object (in particular a `DrawingCanvas` object). You can get more detail about canvases and the objects they support (see *Ginga Canvas Graphics*). A variety of graphical shapes are available, and plotted objects scale, transform and rotate seamlessly with the viewer.

Rendering into Matplotlib Figures

Ginga can also render directly into a Matplotlib Figure, which opens up possibilities for overplotting beyond the limited capabilities of the Ginga canvas items. See the examples under "examples/matplotlib" for ideas, particularly "example4_mpl.py".

Rendering into HTML5 canvases

Ginga can render onto HTML5 canvases displayed in a web browser. This opens up interesting possibilities for server-based remote viewing tools. See the examples under "examples/pg", particularly "example2_pg.py".

Writing widget toolkit independent code

You can write code that allows the widget set to be abstracted by Ginga's widget wrappers. This is the same technique used to allow the reference viewer to switch between supported toolkits using the "-t" command line option. Currently only Qt (4/5), Gtk (3), and HTML5 (to a more limited degree) are supported, and there are some limitations compared to developing using a native toolkit directly. Nevertheless, the ability to target different platforms just by changing a command line option is a very interesting proposition.

See the examples under "examples/gw", particularly "example2.py".
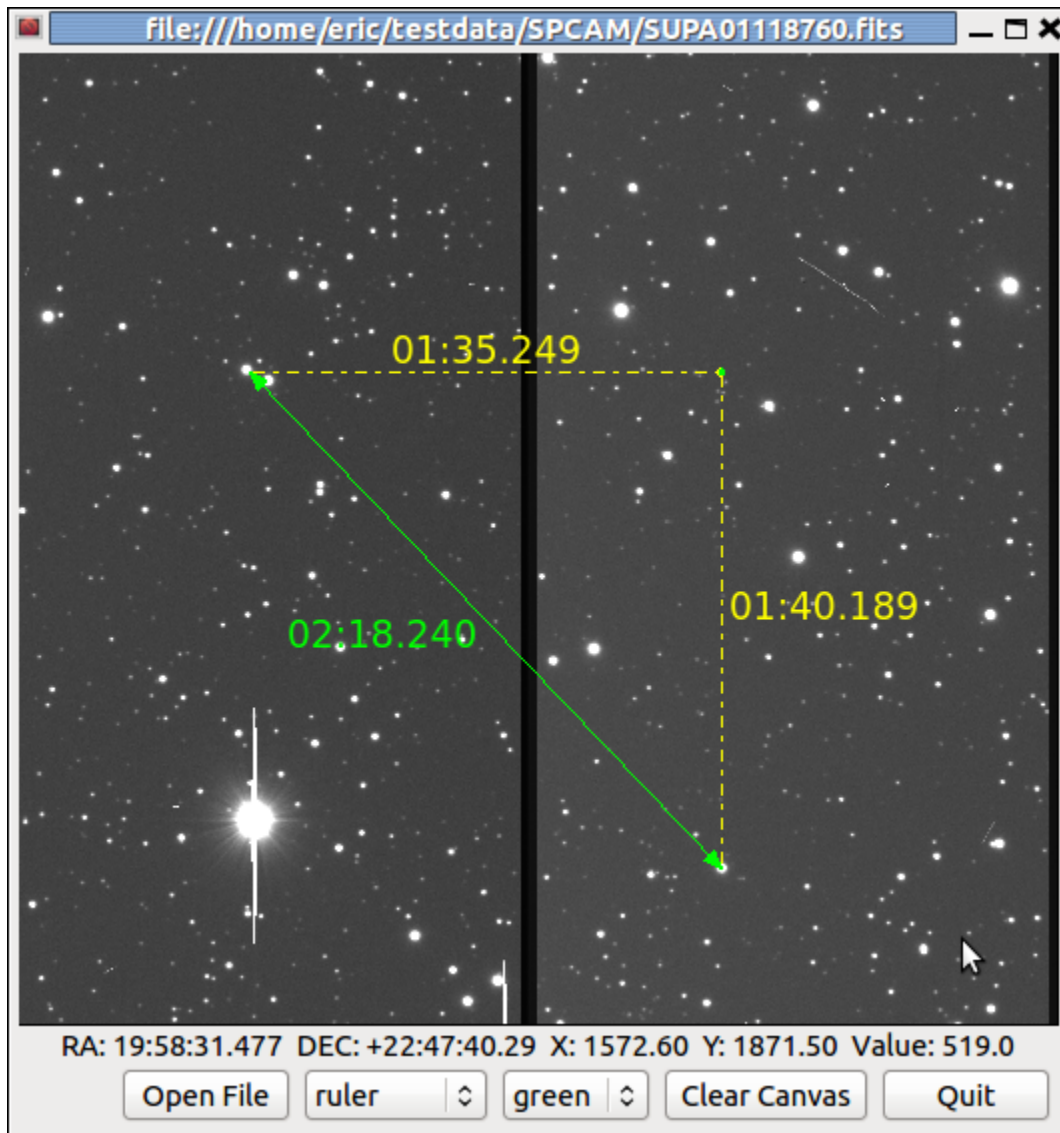
Fig. 7: An example of a `CanvasView` widget with graphical overlay.

5.8 Using Ginga with Jupyter

Ginga has the following modules available to support visualization in Jupyter Notebook or Lab:

- `ginga.web.jupyterw` uses ipywidgets; while
- `ginga.web.pgw` uses HTML5.

Example ipynb files are provided to illustrate basic usage of these modules.

5.9 Ginga Internals

This chapter explains the secret inner workings of Ginga and its classes so that you can subclass them and use them in your own applications.

5.9.1 Introduction

Ginga uses a version of the Model-View-Controller design pattern. The MVC pattern spells out a division of responsibilities and encapsulation where the Model provides various ways to access and interface to the data, the View provides ways to display the data and the Controller provides the methods and user interface hooks for controlling the view.

The Model
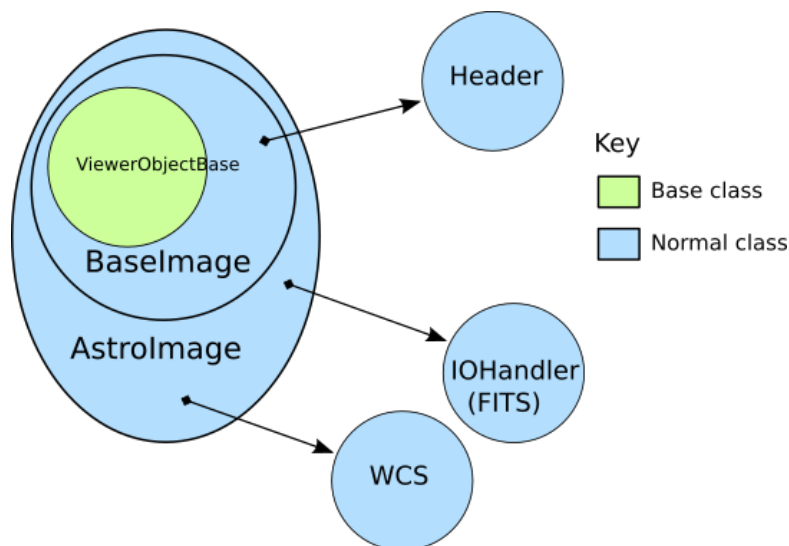
Fig. 8: Hierarchy of Ginga `AstroImage` class

The Model classes are rooted in the base class `BaseImage`. The basic interface to the data is expected to be a Numpy-like array object that is obtained via the `get_data()` method on the model. It also provides methods for obtaining scaled, cutouts and transformed views of the data, and methods for getting and setting key-value like metadata.

There are two subclasses defined on BaseImage: `RGBImage` and `AstroImage`. RGBImage is used for displaying 3 channel RGB type images such as JPEG, TIFF, PNG, etc. AstroImage is the subclass used to represent astronomical images and its organization is shown in Figure *Hierarchy of Ginga AstroImage class*. It has two delegate objects devoted to handling World Coordinate System transformations and file IO.

New models can be created, subclassing from BaseImage or AstroImage. As long as the model duck types like a BaseImage it can be loaded into a view object with the set_image() method. AstroImage provides a few convenience methods for accessing WCS information from the attached "wcs" attribute.

The View
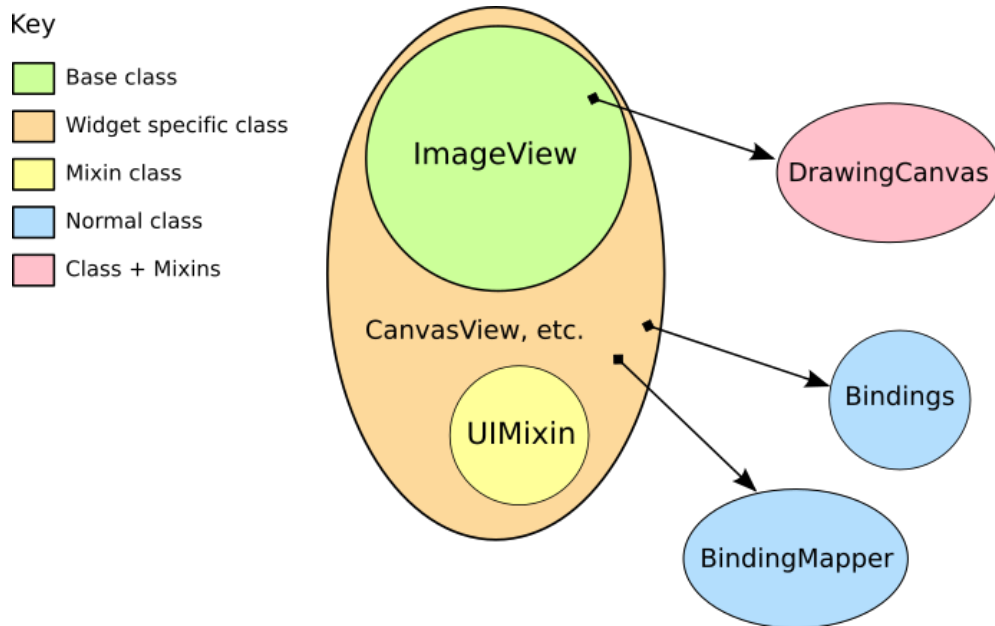
Fig. 9: Class structure of Ginga basic widget viewer

Figure *Class structure of Ginga basic widget viewer* shows the class inheritance of the CanvasView class, which is the prototypical viewer class to use in a program. The viewer is rooted in the base class ImageViewBase, which contains the settings that control the view, such as scale (zoom), pan position, rotation, transformation, etc along with a large number of methods to manipulate the viewer. Ginga supports "backends" for different widget sets (Qt, Gtk, Tk, etc.) through various subclasses of this base class, which provide an native window or canvas widget that can be painted with the resulting RGB[A] image produced by a renderer. A CanvasView viewer can be created for any supported back end.

Every viewer has a dedicated renderer as a delegate object. Renderers are also arranged in a hierarchical class structure. The base renderer class is RenderBase, which specifies an abstract base class that should be implemented to render a Ginga canvas onto a back end-specific viewer.

The Controller

The control interface is a combination of methods on the view object and a pluggable Bindings class which handles the mapping of user input events such as mouse, gesture and keystrokes into methods in the viewer. There are many callback functions that can be registered, allowing the user to create their own custom user interface for manipulating the view.

CanvasView connects various user interface events (mouse/cursor, keystrokes, etc.) with methods in the BindingMapper and Bindings delegate objects to implement most of the user event handling logic. With this layered class construction combined with appropriate delegate objects, it is possible to minimize the widget specific code and reuse a large amount of code across widget sets and platforms. This architecture makes it a fairly simple process to port the basic Ginga functionality to a new widget set. All that is required is that the new widget set have some kind

of native widget that supports painting an RGB image (like a canvas or image widget) and a way to register for user interaction events on that widget.

5.9.2 Graphics on Ginga
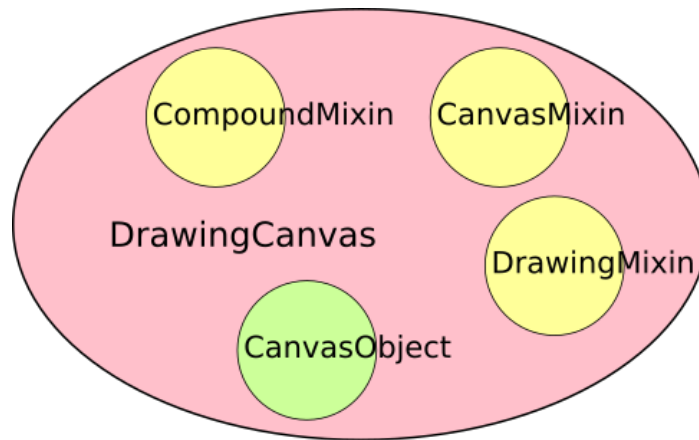
Fig. 10: Class structure of Ginga `DrawingCanvas` class.

Ginga's graphics are all rendered from objects placed on a Ginga canvas, including images. A Ginga canvas is a bit different from other types of canvases used in other graphics programs. For one thing, it has no inherent color or scale in any type of unit; it acts as a container for other graphics objects that are stacked in a particular order. A canvas itself is an object that can be placed on a canvas and so it is quite straightforward to have canvases nested in canvases or several canvases stacked together on one canvas, etc. The type of canvas that you will see used most frequently (primarily for its flexibility) is the `DrawingCanvas`, so named because it not only allows all the typical objects to be placed on it, but it also has methods that allow the user to draw or edit objects interactively on it. The relationship of a viewer to a canvas is that the viewer displays a canvas with a certain scale, rotation, transformations, color-mapping, pan position, etc. A canvas might be shared with another viewer which has different settings for those things.

All objects that can be drawn by Ginga (e.g. placed in a canvas) are decended from the `CanvasObjectBase` type, and made by using subclasses or composing with mixin classes to derive new object types. We will use the general term "Ginga canvas objects" to describe these various entities. In Figure *Class structure of Ginga DrawingCanvas class.* we can see that a `DrawingCanvas` is a composite of a `UIMixin` (user-interface mixin), a `DrawingMixin` and a `Canvas`. A `Canvas` in turn is a composite of a `CanvasMixin` and a `CompoundObject`. A `CompoundObject` is a composite of a `CompoundMixin` and a `CanvasObjectBase`.

Other Ginga canvas objects have a simpler pedigree. For example, a `Box` is a composite of a `OnePointTwoRadiusMixin` and a `CanvasObjectBase`–so is an `Ellipse`. The use of these mixin classes allows common functionality and attributes to be shared where the similarities allow.

For more information on canvases and canvas objects, refer to Chapter:ref:`_ch-canvas_graphics`.

5.9.3 Miscellaneous Topics

I want to use my own World Coordinate System!

No problem. Ginga encapsulates the WCS behind a pluggable object used in the AstroImage class. Your WCS should implement this abstract class:

```python
def MyWCS(object):
    def __init__(self, logger):
        self.logger = logger

    def get_keyword(self, key):
        return self.header[key]

    def get_keywords(self, *args):
        return [self.header[key] for key in args]

    def load_header(self, header, fobj=None):
        pass

    def pixtoradec(self, idxs, coords='data'):
        # calculate ra_deg, dec_deg
        return (ra_deg, dec_deg)

    def radectopix(self, ra_deg, dec_deg, coords='data', naxispath=None):
        # calculate x, y
        return (x, y)

    def pixtosystem(self, idxs, system=None, coords='data'):
        return (deg1, deg2)

    def datapt_to_wcspt(self, datapt, coords='data', naxispath=None):
        return [[ra_deg_0, dec_deg_0], [ra_deg_1, dec_deg_1], ...,
                [ra_deg_n, dec_deg_n]]

    def wcspt_to_datapt(self, wcspt, coords='data', naxispath=None):
        return [[x0, y0], [x1, y1], ..., [xn, yn]]
```

To use your WCS with Ginga create your images like this:

```python
from ginga.AstroImage import AstroImage
AstroImage.set_wcsClass(MyWCS)
...

image = AstroImage()
...
view.set_image(image)
```

or you can override the WCS on a case-by-case basis:

```python
from ginga.AstroImage import AstroImage
...

image = AstroImage(wcsclass=MyWCS)
```

```
...
view.set_image(image)
```

You could also subclass AstroImage or BaseImage and implement your own WCS handling. There are certain methods in AstroImage used for graphics plotting and plugins, however, so these would need to be supported if you expect the same functionality.

I want to use my own file storage format, not FITS!

First of all, you can always create an `AstroImage` and assign its components for wcs and data explicitly. Assuming you have your data loaded into an `numpy` array named `data`:

```python
from ginga import AstroImage
...

image = AstroImage()
image.set_data(data)
```

To create a valid WCS for this image, you can set the header in the image (this assumes `header` is a valid mapping of keywords to values):

```python
image.update_keywords(header)
```

An `AstroImage` can then be loaded into a viewer object with `set_dataobj()`. If you need a custom WCS see the notes in Section *I want to use my own World Coordinate System!*. If, however, you want to add a new type of custom loader into Ginga's file loading framework, you can do so using the following instructions.

Adding a new kind of file opener

Ginga's general file loading facility breaks the loading down into two phases: first, the file is identified by its `magic` signature (requires the optional Python module `python-magic` be installed) or MIME type. Once the general category of file is known, methods in the specific I/O module devoted to that type are called to load the file data.

The `ginga.util.loader` module is used to register file openers. An opener is a class that understand how to load data objects from a particular kind of file format.

For implementing your own special opener, take a look at the `BaseIOHandler` class in `ginga.util.io.io_base`. This is the base class for all I/O openers for Ginga. Subclass this class, and implement all of the methods that raise `NotImplementedError` and optionally implement any other methods marked with the comment "subclass should override as needed". You can study the `io_fits` and `io_rgb` modules to see how these methods are implemented for specific formats. Here is an example opener class for HDF5 standard image files:

```python
# This is open-source software licensed under a BSD license.
# Please see the file LICENSE.txt for details.
"""
Module wrapper for loading HDF5 files.
"""
import re
from collections import OrderedDict
import numpy as np

try:
```

```python
    import h5py  # noqa
    have_h5py = True
except ImportError:
    have_h5py = False

from ginga.util import iohelper
from ginga.util.io import io_base

__all__ = ['have_h5py', 'load_file', 'HDF5FileHandler']


def load_file(filepath, idx=None, logger=None, **kwargs):
    """
    Load an object from an H5PY file.
    See :func:`ginga.util.loader` for more info.

    """
    opener = HDF5FileHandler(logger)
    with opener.open_file(filepath):
        return opener.load_idx(idx, **kwargs)


class HDF5FileHandler(io_base.BaseIOHandler):
    """For loading HDF5 image files.
    """

    name = 'h5py'
    mimetypes = ['application/x-hdf']

    @classmethod
    def check_availability(cls):
        if not have_h5py:
            raise ValueError("Install 'h5py' to use this opener")

    def __init__(self, logger):
        if not have_h5py:
            raise ValueError(
                "Need 'h5py' module installed to use this file handler")

        super().__init__(logger)
        self.kind = 'hdf5'

        self._f = None

    def get_indexes(self):
        return self._f.keys()

    def get_header(self, idx):
        items = [(key, val.decode() if isinstance(val, bytes) else val)
                 for key, val in self._f[idx].attrs.items()]
        return OrderedDict(items)
```

```python
    def get_idx_type(self, idx):
        header = self.get_header(idx)
        if header.get('CLASS', None) in ['IMAGE']:
            return 'image'

        # TODO: is there a table spec for HDF5?

        return None

    def load_idx(self, idx, **kwargs):

        if idx is None:
            idx = self.find_first_good_idx()

        typ = self.get_idx_type(idx)
        if typ == 'image':
            from ginga import AstroImage, RGBImage

            header = self.get_header(idx)
            data_np = np.copy(self._f[idx][()])

            if 'PALETTE' in header:
                p_idx = header['PALETTE']
                p_data = self._f[p_idx][()]
                data_np = p_data[data_np]
                image = RGBImage.RGBImage(logger=self.logger)
            else:
                image = AstroImage.AstroImage(logger=self.logger)
                image.update_keywords(header)

            image.set_data(data_np)

            name = iohelper.name_image_from_path(self._path, idx=idx)
            image.set(path=self._path, name=name, idx=idx,
                      image_loader=load_file)

            return image

        raise ValueError("I don't know how to read dataset '{}'".format(idx))

    def open_file(self, filespec, **kwargs):
        # open the HDF5 file and make a full inventory of what is
        # available to load
        info = iohelper.get_fileinfo(filespec)
        if not info.ondisk:
            raise ValueError("File does not appear to be on disk: %s" % (
                info.url))

        self._path = info.filepath

        self.logger.debug("Loading file '%s' ..." % (self._path))
        self._f = h5py.File(self._path, 'r', **kwargs)
```

```python
        return self

    def close(self):
        _f = self._f
        self._f = None
        self._path = None
        _f.close()

    def __len__(self):
        if self._f is None:
            return 0
        return len(self._f)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()
        return False

    def load_idx_cont(self, idx_spec, loader_cont_fn, **kwargs):
        if len(self) == 0:
            raise ValueError("Please call open_file() first!")

        idx_lst = self.get_matching_indexes(idx_spec)
        for idx in idx_lst:
            try:
                dst_obj = self.load_idx(idx, **kwargs)

                # call continuation function
                loader_cont_fn(dst_obj)

            except Exception as e:
                self.logger.error("Error loading index '%s': %s" % (idx, str(e)))

    def find_first_good_idx(self):

        for idx in self.get_indexes():

            # rule out Datasets we can't deal with
            typ = self.get_idx_type(idx)
            if typ not in ('image', 'table'):
                continue

            # Looks good, let's try it
            return idx

        return None

    def get_matching_indexes(self, idx_spec):
        """
        Parameters
```

```
        ----------
        idx_spec : str
            A string in the form of a pair of brackets enclosing some
            index expression matching Datasets in the file

        Returns
        -------
        result : list
            A list of indexes that can be used to access each Dataset
            matching the pattern
        """
        # if index is missing, assume to open the first Dataset we know how
        # to do something with
        if idx_spec is None or idx_spec == '':
            idx = self.find_first_good_idx()
            return [idx]

        match = re.match(r'^\[(.+)\]$', idx_spec)
        if not match:
            return []

        name = match.group(1).strip()
        if re.match(r'^\d+$', name):
            # index just names a single dataset by number
            # Assume this means by order in the list
            return [int(name)]

        # find all datasets matching the name
        # TODO: could do some kind of regular expression matching
        idx_lst = []
        idx = 0
        for d_name in self.get_indexes():
            if name == '*' or name == d_name:
                idx_lst.append(d_name)

        return idx_lst
```

Once you have created your opener class (e.g. `HDF5FileHandler`), you can register it by:

```
from ginga.util import loader
import io_hdf5
loader.add_opener(io_hdf5.HDF5FileHandler, ['application/x-hdf'])
```

If you want to use this with the Ginga reference viewer, a good place to register the opener is in your `ginga_config.py` as discussed in Section *Customizing the Layout* of the Reference Viewer Manual. The best place is probably by implementing `pre_gui_config` and registering it as shown above in that function. Once your loader is registered, you will be able to drag and drop files and use the reference viewers regular loading facilities to load your data.

Changes to Ginga API in v4.0.0

Prior to Ginga v4.0.0, it was possible to use a *combination* viewer and canvas–a viewer object that acts also like a ginga canvas. These were accessible via the `ImageViewCanvas*` classes.

In Ginga v4.0.0 these "dual entity" classes have been removed, to simplify the code and clearly delineate the use of each kind of object: a viewer shows the contents of a canvas for some backend, whereas a canvas contains the items to be viewed (and can be shared by viewers).

If you have legacy code that is making canvas API calls on the viewer, you simply need to use the `get_canvas()` method on the viewer to get the canvas object and then make the canvas API call on that.

Porting Ginga to a New Widget Set

[*TBD*]

OPTIMIZING PERFORMANCE

There are several ways to optimize the performance of certain aspects of Ginga's operation.

6.1 OpenCv Acceleration

Ginga includes support for OpenCv accelerated operations (e.g. rotation and rescaling). *This support is used by default if the package is installed.*

To enable OpenCv support, install the python `opencv` module (you can find it here).

6.2 OpenGl Acceleration

Ginga includes support for OpenGL rendering with Qt or Gtk back ends. To use this with the Reference Viewer, simply append the command line option –opengl. This can be particularly useful with high resolution displays.

Note that certain aspects of normal rendering for Ginga canvas objects are unavailable or different with OpenGL:

- Inability to specify `linestyle` parameter (lines are always solid)
- Inability to specify `linewidth` parameter (always defaults to 1)

FAQS

This section contains some frequently asked questions (FAQs) regarding Ginga usage.

7.1 Platforms

7.1.1 Does Ginga run on Mac/Windows/Linux/XYZ?

Ginga is written entirely in the Python programming language, and uses only supporting Python packages. As long as a platform supports Python and the necessary packages, it can run some version of Ginga. On recent Linux, Mac and Windows versions, all of these packages are available.

7.1.2 Does Ginga work with Python 3?

Yes, but only Python 3.7 or later. Just install with Python 3. Of course, you need all the supporting modules for Python 3 (NumPy, SciPy, Qt 5, etc.).

7.1.3 Does Ginga work with Python 2?

No. If you absolutely have to use Python 2, please downgrade to Ginga 2.x series.

7.2 Toolkits

7.2.1 What GUI toolkit does Ginga use?

It depends what exactly you want to run. Ginga is both a toolkit for building viewers and also includes a "reference viewer". The example programs currently support Qt, GTK, Tk, matplotlib and web browser via HTML5 canvas. Some other toolkits are being worked on and may be partially supported.

The full reference viewer currently supports Qt (PyQt5, PyQt6, PySide2, PySide6) and Gtk (ver 3). The difference is explained here, in Section *Developing with Ginga*.

7.3 Control Bindings

7.3.1 Can I get DS9-like user interface mappings?

Save the file called bindings.cfg.ds9 and drop it in your `$HOME/.ginga` folder as "bindings.cfg". Then restart Ginga.

7.3.2 Can I customize the user interface mappings?

Yes. There is more information in the *Binding Config File* section.

7.3.3 Where can I find a quick reference of the bindings?

See Section *Quick Reference*.

7.4 Miscellaneous

7.4.1 Does Ginga work with SAMP?

Yes. See Section *SAMP Control*.

7.4.2 Is it possible to control Ginga remotely?

Yes. See Section *RC*.

7.4.3 When are you going to add the XYZ feature that DS9 has?

Maybe never. The Ginga package design goal was never to replace DS9, but to provide a full featured Python FITS widget that we could use to build directly in Python. This is clearly seen if you look at the example programs in `examples/*/example*.py`. The idea was to make it easy for someone to build any kind of custom viewer by having a full-featured widget to build on.

That said, we did write a reference viewer because we needed something with many of the convenience features of a modern FITS viewer. DS9 is almost the size of a small OS, however, and I'm not sure it is wise to try to match it feature for feature. Instead, since Ginga is plugin-based, you can write plugins to give you the features you need. DS9 is a "everything including kitchen sink" kind of viewer, whereas ginga reference viewer is more like a "take what you need from the pantry and whip it up" type viewer.

Please send a pull request!

7.4.4 Can I get Ginga reference viewer to save its size and position?

Yes. Add the line `"save_layout = True"` to your `~/.ginga/general.cfg` file.

If the file does not exist, create it, or copy the one from `ginga/examples/configs/general.cfg`.

7.5 World Coordinate System

7.5.1 What library are you using for WCS?

We are lucky to have several possible choices for a Python WCS package compatible with Ginga: AstLib, Kapteyn, Starlink, and Astropy WCS.

Kapteyn and Astropy wrap Mark Calabretta's "WCSLIB", AstLib wraps Jessica Mink's "wcstools", and I'm not sure what Starlink uses. Note that AstLib and Starlink require Astropy to be installed in order to create a WCS object from a FITS header.

To force the use of a particular one add this to your "general.cfg" in `$HOME/.ginga`:

```
WCSpkg = 'package'
```

Replace `'package'` with one of `'Astropy'`, `'Kapteyn'`, `'Starlink'`, `'astlib'`, or `'choose'`. If you pick `'choose'`, Ginga will try to pick one for you.

7.5.2 How easy is it for Ginga to support a custom WCS?

Pretty easy. See Section *I want to use my own World Coordinate System!*.

7.6 I/O and File Formats

7.6.1 What library are you using for FITS I/O?

There are two possible choices for a Python FITS file reading package compatible with Ginga: Astropy FITS and fitsio. Both are originally based on the CFITSIO library (although Astropy's version uses very little of it any more, while `fitsio` is still tracking the current version).

To force the use of a particular one add this to your "general.cfg" in `$HOME/.ginga`:

> FITSpkg = 'package'

Replace `'package'` with one of `'Astropy'`, `'fitsio'`, or `'choose'`. If you pick `'choose'`, Ginga will try to pick one for you.

7.6.2 How easy is it for Ginga to support a new file formats besides FITS?

Pretty easy. See Section *I want to use my own file storage format, not FITS!*.

7.7 Problems Displaying Images

Nothing changes in the image when I change settings under "Preferences".

Note: The Preferences plugin sets the preferences on a *per-channel* basis. Make sure the channel you are looking at has the same name as the prefix for the preferences. For example: "Image" and "Image: Preferences" or "Image1" and "Image1: Preferences".

The preferences for a given channel are copied from the default "Image" channel until they are explicitly set and saved using this plugin. So if you want preferences that follow around from channel to channel, save them as preferences for "Image" and any new channels created will get those as well, unless you have saved different ones under those channel names.

Nothing changes in the image when I change the "Auto Cuts" settings under Preferences. I've checked that I'm adjusting preferences for the same channel that I'm viewing.

Note: What is the setting for "Cut New" under the New Images section in Preferences for this channel?

If that setting is "Off" then you have elected not to have Ginga apply Auto Levels when an image is loaded in that channel. Press 'a' in the image window to force an auto cut levels and it will use the new settings.

No image shows in the display, and I get an error in the terminal about histogram and keyword "density".

Note: You need a newer version of NumPy.

I recommend getting NumPy v1.14 or later.

REFERENCE/API

8.1 ginga.canvas.CanvasMixin Module

8.1.1 Classes

CanvasMixin()	A CanvasMixin is combined with the CompoundMixin to make a tag-addressible canvas-like interface.

CanvasMixin

class ginga.canvas.CanvasMixin.**CanvasMixin**

> Bases: object

> A CanvasMixin is combined with the CompoundMixin to make a tag-addressible canvas-like interface. This mixin should precede the CompoundMixin in the inheritance (and so, method resolution) order.

Methods Summary

add(obj[, tag, tagpfx, belowThis, redraw])	
delete_all_objects([redraw])	
delete_object(obj[, redraw])	
delete_object_by_tag(tag[, redraw])	
delete_objects(objects[, redraw])	
delete_objects_by_tag(tags[, redraw])	
get_object_by_tag(tag)	
get_objects_by_tag_pfx(tagpfx)	
get_tags()	
get_tags_by_tag_pfx(tagpfx)	
has_tag(tag)	
lookup_object_tag(obj)	
lower_object_by_tag(tag[, belowThis, redraw])	
raise_object_by_tag(tag[, aboveThis, redraw])	
redraw([whence])	
subcanvas_updated_cb(canvas, whence)	This is a notification that a subcanvas (a canvas contained in our canvas) has been modified.
update_canvas([whence])	

Methods Documentation

add(*obj*, *tag=None*, *tagpfx=None*, *belowThis=None*, *redraw=True*)

delete_all_objects(*redraw=True*)

delete_object(*obj*, *redraw=True*)

delete_object_by_tag(*tag*, *redraw=True*)

delete_objects(*objects*, *redraw=True*)

delete_objects_by_tag(*tags*, *redraw=True*)

get_object_by_tag(*tag*)

> **get_objects_by_tag_pfx**(*tagpfx*)
>
> **get_tags**()
>
> **get_tags_by_tag_pfx**(*tagpfx*)
>
> **has_tag**(*tag*)
>
> **lookup_object_tag**(*obj*)
>
> **lower_object_by_tag**(*tag*, *belowThis=None*, *redraw=True*)
>
> **raise_object_by_tag**(*tag*, *aboveThis=None*, *redraw=True*)
>
> **redraw**(*whence=3*)
>
> **subcanvas_updated_cb**(*canvas*, *whence*)
>
>> This is a notification that a subcanvas (a canvas contained in our canvas) has been modified. We in turn signal that our canvas has been modified.
>
> **update_canvas**(*whence=3*)

8.2 ginga.canvas.CanvasObject Module

8.2.1 Functions

get_canvas_type(name)
get_canvas_types()
register_canvas_type(name, klass)
register_canvas_types(klass_dict)

get_canvas_type

ginga.canvas.CanvasObject.**get_canvas_type**(*name*)

get_canvas_types

ginga.canvas.CanvasObject.**get_canvas_types**()

register_canvas_type

ginga.canvas.CanvasObject.**register_canvas_type**(*name*, *klass*)

register_canvas_types

ginga.canvas.CanvasObject.**register_canvas_types**(*klass_dict*)

8.2.2 Classes

CanvasObjectBase(**kwdargs)	This is the abstract base class for a CanvasObject.

CanvasObjectBase

class ginga.canvas.CanvasObject.**CanvasObjectBase**(**kwdargs*)

    Bases: `Callbacks`

    This is the abstract base class for a CanvasObject. A CanvasObject is an item that can be placed on a Ginga canvas.

    This class defines common methods used by all such objects.

Methods Summary

calc_dual_scale_from_pt(pt, detail)	
calc_radius(viewer, p1, p2)	
calc_rotation_from_pt(pt, detail)	
calc_scale_from_pt(pt, detail)	
calc_vertexes(start_cx, start_cy, end_cx, end_cy)	
canvascoords(viewer, data_x, data_y)	
contains(x, y)	For backward compatibility.
contains_arr(x_arr, y_arr)	For backward compatibility.
contains_pt(pt)	
contains_pts(points)	
convert_mapper(tomap)	Converts our object from using one coordinate map to another.
copy([share])	
draw_arrowhead(cr, x1, y1, x2, y2)	

Table 1 – continued from previous page

draw_caps(cr, cap, points[, radius])	
draw_edit(cr, viewer)	
get_bbox([points])	Get bounding box of this object.
get_center_pt()	Return the geometric average of points as data_points.
get_cpoints(viewer[, points, no_rotate])	
get_data(*args)	
get_data_points([points])	Points returned are in data coordinates.
get_llur()	
get_move_scale_rotate_pts(viewer)	Returns 3 edit control points for editing this object: a move point, a scale point and a rotate point.
get_num_points()	
get_point_by_index(i)	
get_points()	Get the set of points that is used to draw the object.
get_pt(viewer, points, pt[, canvas_radius])	Takes an array of points `points` and a target point `pt`.
get_reference_pt()	
initialize(canvas, viewer, logger)	
is_compound()	
move_delta(xoff, yoff)	For backward compatibility.
move_delta_pt(off_pt)	
move_to(xdst, ydst)	For backward compatibility.
move_to_pt(dst_pt)	
point_within_line(points, p_start, p_stop, ...)	
point_within_radius(points, pt, canvas_radius)	Points `points` and point `pt` are in data coordinates.
rerotate_by_deg(thetas, detail)	
rescale_by(scale_x, scale_y, detail)	For backward compatibility.
rescale_by_factors(factors, detail)	
rotate(theta_deg[, xoff, yoff])	For backward compatibility.
rotate_by(theta_deg)	For backward compatibility.
rotate_by_deg(thetas)	
rotate_deg(thetas, offset)	
scale_by(scale_x, scale_y)	For backward compatibility.
scale_by_factors(factors)	

continues on next page

Table 1 – continued from previous page

scale_font(viewer)	
select_contains(viewer, x, y)	For backward compatibility.
select_contains_pt(viewer, pt)	
set_data(**kwdargs)	
set_data_points(points)	Input `points` must be in data coordinates, will be converted to the coordinate space of the object and stored.
set_point_by_index(i, pt)	
setup_edit(detail)	subclass should override as necessary.
swapxy(x1, y1, x2, y2)	
sync_state()	This method called when changes are made to the parameters.
use_coordmap(mapobj)	
within_line(viewer, points, p_start, p_stop, ...)	Points `points` and line endpoints `p_start`, `p_stop` are in data coordinates.
within_radius(viewer, points, pt, canvas_radius)	Points `points` and point `pt` are in data coordinates.

Methods Documentation

calc_dual_scale_from_pt(*pt*, *detail*)

calc_radius(*viewer*, *p1*, *p2*)

calc_rotation_from_pt(*pt*, *detail*)

calc_scale_from_pt(*pt*, *detail*)

calc_vertexes(*start_cx*, *start_cy*, *end_cx*, *end_cy*, *arrow_length=10*, *arrow_degrees=0.35*)

canvascoords(*viewer*, *data_x*, *data_y*)

contains(*x*, *y*)

> For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use contains_pt() instead.

contains_arr(*x_arr*, *y_arr*)

> For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use contains_pts() instead.

contains_pt(*pt*)

contains_pts(*points*)

convert_mapper(*tomap*)

> Converts our object from using one coordinate map to another.

> NOTE: In some cases this only approximately preserves the equivalent point values when transforming between coordinate spaces.

copy(*share=[]*)

draw_arrowhead(*cr*, *x1*, *y1*, *x2*, *y2*)

draw_caps(*cr*, *cap*, *points*, *radius=None*)

draw_edit(*cr*, *viewer*)

get_bbox(*points=None*)

    Get bounding box of this object.

        Returns

            (p1, p2, p3, p4): a 4-tuple of the points in data coordinates,
            beginning with the lower-left and proceeding counter-clockwise.

get_center_pt()

    Return the geometric average of points as data_points.

get_cpoints(*viewer*, *points=None*, *no_rotate=False*)

get_data(**args*)

get_data_points(*points=None*)

    Points returned are in data coordinates.

get_llur()

get_move_scale_rotate_pts(*viewer*)

    Returns 3 edit control points for editing this object: a move point, a scale point and a rotate point. These points are all in data coordinates.

get_num_points()

get_point_by_index(*i*)

get_points()

    Get the set of points that is used to draw the object.

    Points are returned in *data* coordinates.

get_pt(*viewer*, *points*, *pt*, *canvas_radius=None*)

    Takes an array of points `points` and a target point `pt`. Returns the first index of the point that is within the radius of the target point. If none of the points are within the radius, returns None.

get_reference_pt()

initialize(*canvas*, *viewer*, *logger*)

is_compound()

move_delta(*xoff*, *yoff*)

    For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use move_delta_pt instead.

move_delta_pt(*off_pt*)

move_to(*xdst*, *ydst*)

    For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use move_to_pt() instead.

move_to_pt(*dst_pt*)

point_within_line(*points*, *p_start*, *p_stop*, *canvas_radius*)

point_within_radius(*points*, *pt*, *canvas_radius*, *scales=(1.0, 1.0)*)

>   Points `points` and point `pt` are in data coordinates. Return True for points within the circle defined by a center at point `pt` and within canvas_radius.

rerotate_by_deg(*thetas*, *detail*)

rescale_by(*scale_x*, *scale_y*, *detail*)

>   For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use rescale_by_factors() instead.

rescale_by_factors(*factors*, *detail*)

rotate(*theta_deg*, *xoff=0*, *yoff=0*)

>   For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use rotate_deg() instead.

rotate_by(*theta_deg*)

>   For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use rotate_by_deg() instead.

rotate_by_deg(*thetas*)

rotate_deg(*thetas*, *offset*)

scale_by(*scale_x*, *scale_y*)

>   For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use scale_by_factors() instead.

scale_by_factors(*factors*)

scale_font(*viewer*)

select_contains(*viewer*, *x*, *y*)

>   For backward compatibility. TO BE DEPRECATED–DO NOT USE. Use select_contains_pt() instead.

select_contains_pt(*viewer*, *pt*)

set_data(***kwdargs*)

set_data_points(*points*)

>   Input `points` must be in data coordinates, will be converted to the coordinate space of the object and stored.

set_point_by_index(*i*, *pt*)

setup_edit(*detail*)

>   subclass should override as necessary.

swapxy(*x1*, *y1*, *x2*, *y2*)

sync_state()

>   This method called when changes are made to the parameters. subclasses should override if they need any special state handling.

use_coordmap(*mapobj*)

within_line(*viewer*, *points*, *p_start*, *p_stop*, *canvas_radius*)

>   Points `points` and line endpoints `p_start`, `p_stop` are in data coordinates. Return True for points within the line defined by a line from p_start to p_end and within `canvas_radius`. The distance between points is scaled by the viewer's canvas scale.

within_radius(*viewer*, *points*, *pt*, *canvas_radius*)

>   Points `points` and point `pt` are in data coordinates. Return True for points within the circle defined by a center at point `pt` and within canvas_radius. The distance between points is scaled by the canvas scale.

8.3 ginga.canvas.CompoundMixin Module

8.3.1 Classes

CompoundMixin()	A CompoundMixin is a mixin class that makes an object that is an aggregation of other objects.

CompoundMixin

class ginga.canvas.CompoundMixin.**CompoundMixin**

> Bases: object

> A CompoundMixin is a mixin class that makes an object that is an aggregation of other objects.

> It is used to make generic compound drawing types as well as (for example) layers of canvases on top of an image.

Methods Summary

add_object(obj[, belowThis])	
contains_pts(pts)	
copy([share])	
delete_all_objects()	
delete_object(obj)	
delete_objects(objects)	
draw(viewer)	
get_center_pt()	
get_edit_points(viewer)	
get_items_at(pt)	
get_llur()	Get lower-left and upper-right coordinates of the bounding box of this compound object.
get_objects()	
get_objects_by_kind(kind)	
get_objects_by_kinds(kinds)	
get_points()	

continues on next page

Table 2 – continued from previous page

get_reference_pt()	
has_object(obj)	
inherit_from(obj)	
initialize(canvas, viewer, logger)	
is_compound()	
lower_object(obj[, belowThis])	
move_delta_pt(off_pt)	
raise_object(obj[, aboveThis])	
roll_objects(n)	
rotate(theta[, xoff, yoff])	
rotate_deg(thetas, offset)	
scale_by_factors(factors)	
select_contains_pt(viewer, pt)	
select_items_at(viewer, pt[, test])	
set_attr_all(**kwdargs)	
set_edit_point(i, pt, detail)	
setup_edit(detail)	
swap_objects()	
use_coordmap(mapobj)	

Methods Documentation

add_object(*obj*, *belowThis=None*)

contains_pts(*pts*)

copy(*share=[]*)

delete_all_objects()

delete_object(*obj*)

delete_objects(*objects*)

draw(*viewer*)

get_center_pt()

get_edit_points(*viewer*)

get_items_at(*pt*)

get_llur()

Get lower-left and upper-right coordinates of the bounding box of this compound object.

> **Returns**
>
> > **x1, y1, x2, y2: a 4-tuple of the lower-left and upper-right coords**

get_objects()

get_objects_by_kind(*kind*)

get_objects_by_kinds(*kinds*)

get_points()

get_reference_pt()

has_object(*obj*)

inherit_from(*obj*)

initialize(*canvas*, *viewer*, *logger*)

is_compound()

lower_object(*obj*, *belowThis=None*)

move_delta_pt(*off_pt*)

raise_object(*obj*, *aboveThis=None*)

roll_objects(*n*)

rotate(*theta*, *xoff=0*, *yoff=0*)

rotate_deg(*thetas*, *offset*)

scale_by_factors(*factors*)

select_contains_pt(*viewer*, *pt*)

select_items_at(*viewer*, *pt*, *test=None*)

set_attr_all(***kwdargs*)

set_edit_point(*i*, *pt*, *detail*)

setup_edit(*detail*)

swap_objects()

use_coordmap(*mapobj*)

8.4 ginga.canvas.coordmap Module

8.4.1 Classes

NativeMapper(viewer)	A coordinate mapper that maps to the viewer's canvas in the viewer's canvas coordinates.
WindowMapper(viewer)	A coordinate mapper that maps to the viewer in 'window' coordinates.
PercentageMapper(viewer)	A coordinate mapper that maps to the viewer in 'percentage' coordinates.
CartesianMapper(viewer)	A coordinate mapper that maps to the viewer in Cartesian coordinates that do not scale (unlike DataMapper).
DataMapper(viewer)	A coordinate mapper that maps to the viewer in data coordinates.
OffsetMapper(viewer, refobj)	A coordinate mapper that maps to the viewer in data coordinates that are offsets relative to some other reference object.
WCSMapper(viewer)	A coordinate mapper that maps to the viewer in WCS coordinates.

NativeMapper

class ginga.canvas.coordmap.**NativeMapper**(*viewer*)

Bases: `BaseMapper`

A coordinate mapper that maps to the viewer's canvas in the viewer's canvas coordinates.

Methods Summary

data_to(data_pts[, viewer])	
offset_pt(pts, offset)	Offset a point specified by `pt`, by the offsets `offset`.
rotate_pt(pts, theta, offset)	Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`.
to_data(cvs_pts[, viewer])	

Methods Documentation

data_to(*data_pts*, *viewer=None*)

offset_pt(*pts*, *offset*)

Offset a point specified by `pt`, by the offsets `offset`. Coordinates are assumed to be in the space defined by this mapper.

rotate_pt(*pts*, *theta*, *offset*)

Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`. Coordinates are assumed to be in the space defined by this mapper.

> **to_data**(*cvs_pts*, *viewer=None*)

WindowMapper

class ginga.canvas.coordmap.**WindowMapper**(*viewer*)

  Bases: BaseMapper

  A coordinate mapper that maps to the viewer in 'window' coordinates.

Methods Summary

data_to(data_pts[, viewer])	
offset_pt(pts, offset)	Offset a point specified by pt, by the offsets offset.
rotate_pt(pts, theta, offset)	Rotate a point specified by pt by the angle theta (in degrees) around the point indicated by offset.
to_data(cvs_pts[, viewer])	

Methods Documentation

data_to(*data_pts*, *viewer=None*)

offset_pt(*pts*, *offset*)

  Offset a point specified by pt, by the offsets offset. Coordinates are assumed to be in the space defined by this mapper.

rotate_pt(*pts*, *theta*, *offset*)

  Rotate a point specified by pt by the angle theta (in degrees) around the point indicated by offset. Coordinates are assumed to be in the space defined by this mapper.

to_data(*cvs_pts*, *viewer=None*)

PercentageMapper

class ginga.canvas.coordmap.**PercentageMapper**(*viewer*)

  Bases: BaseMapper

  A coordinate mapper that maps to the viewer in 'percentage' coordinates.

Methods Summary

data_to(data_pts[, viewer])	
offset_pt(pts, offset)	Offset a point specified by pt, by the offsets offset.
rotate_pt(pts, theta, offset)	Rotate a point specified by pt by the angle theta (in degrees) around the point indicated by offset.
to_data(pct_pts[, viewer])	

Methods Documentation

data_to(*data_pts*, *viewer=None*)

offset_pt(*pts*, *offset*)

> Offset a point specified by `pt`, by the offsets `offset`. Coordinates are assumed to be in the space defined by this mapper.

rotate_pt(*pts*, *theta*, *offset*)

> Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`. Coordinates are assumed to be in the space defined by this mapper.

to_data(*pct_pts*, *viewer=None*)

CartesianMapper

class ginga.canvas.coordmap.**CartesianMapper**(*viewer*)

> Bases: `BaseMapper`
>
> A coordinate mapper that maps to the viewer in Cartesian coordinates that do not scale (unlike DataMapper).

Methods Summary

data_to(data_pts[, viewer])	
offset_pt(pts, offset)	Offset a point specified by `pt`, by the offsets `offset`.
rotate_pt(pts, theta, offset)	Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`.
to_data(crt_pts[, viewer])	

Methods Documentation

data_to(*data_pts*, *viewer=None*)

offset_pt(*pts*, *offset*)

> Offset a point specified by `pt`, by the offsets `offset`. Coordinates are assumed to be in the space defined by this mapper.

rotate_pt(*pts*, *theta*, *offset*)

> Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`. Coordinates are assumed to be in the space defined by this mapper.

to_data(*crt_pts*, *viewer=None*)

DataMapper

class ginga.canvas.coordmap.**DataMapper**(*viewer*)

    Bases: `BaseMapper`

    A coordinate mapper that maps to the viewer in data coordinates.

    #### Methods Summary

data_to(data_pts[, viewer])	
offset_pt(pts, offset)	Offset a point specified by `pt`, by the offsets `offset`.
rotate_pt(pts, theta, offset)	Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`.
to_data(data_pts[, viewer])	

    #### Methods Documentation

    data_to(*data_pts*, *viewer=None*)

    offset_pt(*pts*, *offset*)

        Offset a point specified by `pt`, by the offsets `offset`. Coordinates are assumed to be in the space defined by this mapper.

    rotate_pt(*pts*, *theta*, *offset*)

        Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`. Coordinates are assumed to be in the space defined by this mapper.

    to_data(*data_pts*, *viewer=None*)

OffsetMapper

class ginga.canvas.coordmap.**OffsetMapper**(*viewer*, *refobj*)

    Bases: `BaseMapper`

    A coordinate mapper that maps to the viewer in data coordinates that are offsets relative to some other reference object.

    #### Methods Summary

calc_offsets(pts)	
data_to(data_pts[, viewer])	
offset_pt(pts, offset)	Offset a point specified by `pt`, by the offsets `offset`.
rotate_pt(pts, theta, offset)	Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`.
to_data(delta_pt[, viewer])	

Methods Documentation

calc_offsets(*pts*)

data_to(*data_pts*, *viewer=None*)

offset_pt(*pts*, *offset*)

> Offset a point specified by `pt`, by the offsets `offset`. Coordinates are assumed to be in the space defined by this mapper.

rotate_pt(*pts*, *theta*, *offset*)

> Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`. Coordinates are assumed to be in the space defined by this mapper.

to_data(*delta_pt*, *viewer=None*)

WCSMapper

class ginga.canvas.coordmap.**WCSMapper**(*viewer*)

> Bases: `BaseMapper`
>
> A coordinate mapper that maps to the viewer in WCS coordinates.

Methods Summary

[*data_to*](data_pts[, viewer])	
[*offset_pt*](pts, offset)	Offset a point specified by `pt`, by the offsets `offset`.
[*rotate_pt*](pts, theta, offset)	Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`.
[*to_data*](wcs_pts[, viewer])	

Methods Documentation

data_to(*data_pts*, *viewer=None*)

offset_pt(*pts*, *offset*)

> Offset a point specified by `pt`, by the offsets `offset`. Coordinates are assumed to be in the space defined by this mapper.

rotate_pt(*pts*, *theta*, *offset*)

> Rotate a point specified by `pt` by the angle `theta` (in degrees) around the point indicated by `offset`. Coordinates are assumed to be in the space defined by this mapper.

to_data(*wcs_pts*, *viewer=None*)

8.5 ginga.canvas.DrawingMixin Module

8.5.1 Classes

DrawingMixin()	The DrawingMixin is a mixin class that adds drawing capability for some of the basic CanvasObject-derived types.

DrawingMixin

class ginga.canvas.DrawingMixin.**DrawingMixin**

 Bases: object

 The DrawingMixin is a mixin class that adds drawing capability for some of the basic CanvasObject-derived types. The set_surface method is used to associate a CanvasView object for layering on.

 Methods Summary

add_draw_mode(name, **kwargs)	
clear_selected()	
draw(viewer)	
draw_motion(canvas, event, data_x, data_y, ...)	
draw_poly_add(canvas, event, data_x, data_y, ...)	
draw_poly_delete(canvas, event, data_x, ...)	
draw_start(canvas, event, data_x, data_y, viewer)	
draw_stop(canvas, event, data_x, data_y, viewer)	
edit_delete()	
edit_delete_cb(canvas, event, data_x, ...)	
edit_motion(canvas, event, data_x, data_y, ...)	
edit_poly_add(canvas, event, data_x, data_y, ...)	
edit_poly_delete(canvas, event, data_x, ...)	
edit_rotate(delta_deg, viewer)	
edit_scale(delta_x, delta_y, viewer)	

edit_select(newobj)	
edit_start(canvas, event, data_x, data_y, viewer)	
edit_stop(canvas, event, data_x, data_y, viewer)	
enable_draw(tf)	
enable_edit(tf)	
get_draw_class(drawtype)	
get_draw_classes()	
get_draw_mode()	
get_drawparams()	
get_drawtype()	
get_drawtypes()	
get_edit_object()	
get_selected()	
is_drawing()	
is_editing()	
is_selected(obj)	
num_selected()	
pick_hover(canvas, event, data_x, data_y, viewer)	
pick_key(canvas, event, data_x, data_y, viewer)	
pick_motion(canvas, event, data_x, data_y, ...)	
pick_start(canvas, event, data_x, data_y, viewer)	
pick_stop(canvas, event, data_x, data_y, viewer)	
process_drawing()	
register_canvas_type(name, klass)	
register_for_cursor_drawing(viewer)	
select_add(obj)	

Table 3 – continued from previous page

select_remove(obj)	
set_draw_mode(mode)	
set_drawcolor(colorname)	
set_drawtype(drawtype, **drawparams)	
set_surface(viewer)	

Methods Documentation

add_draw_mode(*name*, ***kwargs*)

clear_selected()

draw(*viewer*)

draw_motion(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

draw_poly_add(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

draw_poly_delete(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

draw_start(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

draw_stop(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

edit_delete()

edit_delete_cb(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

edit_motion(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

edit_poly_add(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

edit_poly_delete(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

edit_rotate(*delta_deg*, *viewer*)

edit_scale(*delta_x*, *delta_y*, *viewer*)

edit_select(*newobj*)

edit_start(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

edit_stop(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

enable_draw(*tf*)

enable_edit(*tf*)

get_draw_class(*drawtype*)

get_draw_classes()

get_draw_mode()

get_drawparams()

get_drawtype()

get_drawtypes()

get_edit_object()

get_selected()

is_drawing()

is_editing()

is_selected(*obj*)

num_selected()

pick_hover(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

pick_key(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

pick_motion(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

pick_start(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

pick_stop(*canvas*, *event*, *data_x*, *data_y*, *viewer*)

process_drawing()

register_canvas_type(*name*, *klass*)

register_for_cursor_drawing(*viewer*)

select_add(*obj*)

select_remove(*obj*)

set_draw_mode(*mode*)

set_drawcolor(*colorname*)

set_drawtype(*drawtype*, ***drawparams*)

set_surface(*viewer*)

8.6 ginga.canvas.types.layer Module

8.6.1 Classes

CompoundObject(*objects, **kwdargs)	Compound object on a Ginga canvas.
Canvas(*objects, **kwdargs)	Class to handle canvas in Ginga.
DrawingCanvas(**kwdargs)	Drawing canvas.

CompoundObject

class ginga.canvas.types.layer.**CompoundObject**(**objects*, ***kwdargs*)

    Bases: *CompoundMixin*, *CanvasObjectBase*

    Compound object on a Ginga canvas. Parameters are the child objects making up the compound object. Objects are drawn in the order listed. Example:

    This makes a point inside a circle.

    Methods Summary

get_params_metadata()	

    Methods Documentation

    classmethod get_params_metadata()

Canvas

class ginga.canvas.types.layer.**Canvas**(**objects*, ***kwdargs*)

    Bases: *CanvasMixin*, *CompoundObject*

    Class to handle canvas in Ginga.

    Methods Summary

get_params_metadata()	

    Methods Documentation

    classmethod get_params_metadata()

DrawingCanvas

class ginga.canvas.types.layer.**DrawingCanvas**(***kwdargs*)

    Bases: UIMixin, *DrawingMixin*, *Canvas*

    Drawing canvas.

8.6.2 Class Inheritance Diagram
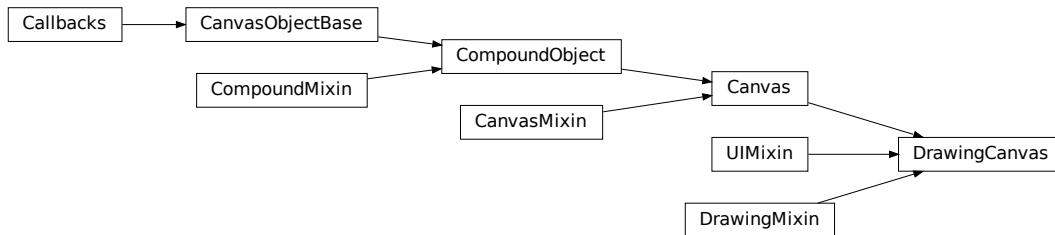
8.7 ginga.ImageView Module

This module handles image viewers.

8.7.1 Classes

ImageViewBase([logger, rgbmap, settings])	An abstract base class for displaying images represented by Numpy data arrays.

ImageViewBase

class ginga.ImageView.**ImageViewBase**(*logger=None*, *rgbmap=None*, *settings=None*)

Bases: `Callbacks`

An abstract base class for displaying images represented by Numpy data arrays.

This class attempts to do as much of the image handling as possible using Numpy array manipulations (even color and intensity mapping) so that only a minimal mapping to a pixel buffer is necessary in concrete subclasses that connect to an actual rendering surface (e.g., Qt, GTK, Tk, HTML5).

Parameters

logger
[`Logger` or `None`] Logger for tracing and debugging. If not given, one will be created.

rgbmap
[`RGBMapper` or `None`] RGB mapper object. If not given, one will be created.

settings
[`SettingGroup` or `None`] Viewer preferences. If not given, one will be created.

Attributes Summary

vname	
vtypes	

Methods Summary

apply_profile(profile[, keylist])	Apply a profile to the viewer.
apply_profile_or_settings(image)	Apply a profile to the viewer.
auto_levels([autocuts])	Apply auto-cut levels on the image view.
auto_orient()	Set the orientation for the image to a reasonable default.
autocut_params_cb(setting, value)	Handle callback related to changes in auto-cut levels.
calc_pan_pct([pad, min_pct, max_pct])	Calculate values for vertical/horizontal panning by percentages from the current pan position.
canvas_changed_cb(canvas, whence)	Handle callback for when canvas has changed.
canvascoords(data_x, data_y)	Same as *get_canvas_xy()*.
capture_default_viewer_profile()	
capture_profile(profile)	
center_cursor()	Center the cursor in the viewer's widget, in both X and Y.
center_image([no_reset])	Pan to the center of the image.
check_cursor_location()	Check whether the data location of the last known position of the cursor has changed.
checkpoint_profile()	
clear()	Clear the displayed image.
configure(width, height)	See *set_window_size()*.
configure_surface(width, height)	See *configure()*.
copy_attributes(dst_fi, attrlist[, share, ...])	Copy interesting attributes of our configuration to another image view.
copy_to_dst(target)	Extract our image and call *set_image()* on the target with it.
cut_levels(loval, hival[, no_reset])	Apply cut levels on the image view.
cut_levels_cb(setting, value)	Handle callback related to changes in cut levels.
data_to_offset(data_x, data_y)	Reverse of *offset_to_data()*.
define_cursor(cname, cursor)	Define a viewer cursor under a name.
delayed_redraw()	Handle delayed redrawing of the canvas.
enable_auto_orient(tf)	Set `auto_orient` behavior.
enable_autocenter(option)	Set `autocenter` behavior.
enable_autocuts(option)	Set `autocuts` behavior.
enable_autozoom(option)	Set `autozoom` behavior.
flip_x()	Transform view of the image by flipping the X axis.
flip_y()	Transform view of the image by flipping the Y axis.
get_autocenter_options()	Get all valid `autocenter` options.

continues on next page

get_autocut_methods()	Same as *ginga.AutoCuts.AutoCutsBase.get_algorithms()*.
get_autocuts_options()	Get all valid `autocuts` options.
get_autozoom_options()	Get all valid `autozoom` options.
get_bg()	Get the background color.
get_canvas()	Get the canvas object used by this instance.
get_canvas_image()	Get canvas image object.
get_canvas_pt(data_pt)	Similar to *get_canvas_xy()*, except that it takes a single array of points.
get_canvas_xy(data_x, data_y)	Reverse of *get_data_xy()*.
get_center()	Get image center.
get_color_algorithms()	Get available color distribution algorithm names.
get_coordmap(key)	Get coordinate mapper.
get_cursor(cname)	Get the cursor stored under the name.
get_cut_levels()	Get cut levels.
get_data(data_x, data_y)	Get the data value at the given position.
get_data_pct(xpct, ypct)	Calculate new data size for the given axis ratios.
get_data_pt(win_pt)	Similar to *get_data_xy()*, except that it takes a single array of points.
get_data_size()	Get the dimensions of the image currently being displayed.
get_data_xy(win_x, win_y)	Get the closest coordinates in the data array to those reported on the window.
get_dataobj()	Get the image currently being displayed.
get_datarect()	Get the approximate LL and UR corners of the displayed image.
get_desired_size()	Get desired size.
get_dims(data)	Get the first two dimensions of Numpy array data.
get_draw_rect()	Get the coordinates in the actual data corresponding to the area needed for drawing images for the current zoom level and pan.
get_fg()	Get the foreground color.
get_image()	Get the image currently being displayed.
get_image_as_array([order])	Get the current image shown in the viewer, with any overlaid graphics, in a numpy array with channels as needed and ordered.
get_image_as_buffer([output, order])	Get the current image shown in the viewer, with any overlaid graphics, in a IO buffer with channels as needed and ordered by the back end widget.
get_last_data_xy()	Get the last position of the cursor in data coordinates.
get_last_win_xy()	Get the last position of the cursor in window coordinates.
get_limits([coord])	Get the bounding box of the viewer extents.
get_logger()	Get the logger used by this instance.
get_pan([coord])	Get pan positions.
get_pan_rect()	Get the coordinates in the actual data corresponding to the area shown in the display for the current zoom level and pan.
get_pixel_distance(x1, y1, x2, y2)	Calculate distance between the given pixel positions.
get_plain_image_as_widget()	Get the current image shown in the viewer, without any overlaid graphics, in the format of an image widget in the back end toolkit.

Table 4 – continued from previous page

get_private_canvas()	Get the private canvas object used by this instance.
get_refresh_stats()	Return the measured statistics for timed refresh intervals.
get_rgb_image_as_buffer([output, format, ...])	Get the current image shown in the viewer, with any overlaid graphics, in a file IO-like object encoded as a bitmap graphics file.
get_rgb_image_as_bytes([format, quality])	Get the current image shown in the viewer, with any overlaid graphics, in the form of a buffer in the form of bytes.
get_rgb_image_as_widget([output, format, ...])	Get the current image shown in the viewer, with any overlaid graphics, in the form of a image widget in the toolkit of the back end.
get_rgb_order()	Get RGB order.
get_rgbmap()	Get the RGB map object used by this instance.
get_rotation()	Get image rotation angle.
get_rotation_info()	Get rotation information.
get_scale()	Same as *get_scale_max()*.
get_scale_base_xy()	Get stretch factors.
get_scale_limits()	Get scale limits.
get_scale_max()	Get maximum scale factor.
get_scale_min()	Get minimum scale factor.
get_scale_text()	Report current scaling in human-readable format.
get_scale_xy()	Get scale factors.
get_settings()	Get the settings used by this instance.
get_transforms()	Get transformations behavior.
get_vip()	Get the ViewerImageProxy object used by this instance.
get_window_size()	Get the window size in the underlying implementation.
get_zoom()	Get zoom level.
get_zoom_algorithm()	Get zoom algorithm.
get_zoomrate()	Get zoom rate.
getwin_array([order, alpha, dtype])	
getwin_buffer([order, alpha, dtype])	Same as *getwin_array()*, but with the output array converted to C-order Python bytes.
icc_profile_cb(setting, value)	Handle callback related to changes in output ICC profiles.
initialize_private_canvas(private_canvas)	Initialize the private canvas used by this instance.
interpolation_change_cb(setting, value)	Handle callback related to changes in interpolation.
invert_cmap()	Invert the color map.
is_compound()	Indicate if canvas object is a compound object.
is_redraw_pending()	Indicates whether a deferred redraw has been scheduled.
make_cursor(iconpath, x, y)	Make a cursor in the viewer's native widget toolkit.
make_timer()	Return a timer object implemented using the back end.
offset_to_data(off_x, off_y)	Get the closest coordinates in the data array to those in cartesian fixed (non-scaled) canvas coordinates.
offset_to_window(off_x, off_y)	Convert data offset to window coordinates.
onscreen_message(text[, delay, redraw])	Place a message onscreen in the viewer window.
onscreen_message_off()	Erase any message onscreen in the viewer window.

continues on next page

Table 4 – continued from previous page

pan_by_pct(pct_x, pct_y[, pad])	Pan by a percentage of the data space.
pan_cb(setting, value)	Handle callback related to changes in pan.
pan_center_px()	Pan to the center of the current pixel under the cursor.
pan_delta_px(x_delta_px, y_delta_px)	Pan by a delta in X and Y specified in pixels.
pan_lr(pct_vw, sign)	Pan left/right by an amount specified as a percentage of the visible width.
pan_omni(direction_deg, amount[, lock_x, lock_y])	Pan in a direction defined in degrees by an amount specified as a percentage.
pan_ud(pct_vh, sign[, msg])	Pan up/down by an amount specified as a percentage of the visible height.
panset_pct(pct_x, pct_y)	Similar to *set_pan()*, except that pan positions are determined by multiplying data dimensions with the given scale factors, where 1 is 100%.
panset_xy(data_x, data_y[, no_reset])	Similar to *set_pan()*, except that input pan positions are always in data space.
position_at_canvas_xy(data_pt, canvas_pt[, ...])	Position a data point at a certain canvas position.
position_cursor(data_x, data_y)	Position the current cursor to a location defined it data coords.
prepare_image(cvs_img, cache, whence)	This can be overridden by subclasses.
recalc_transforms([trcat])	Takes a catalog of transforms (`trcat`) and builds the chain of default transforms necessary to do rendering with most backends.
redraw([whence])	Redraw the canvas.
redraw_data([whence])	Render image from RGB map and redraw private canvas.
redraw_now([whence])	Redraw the displayed image.
refresh_timer_cb(timer, flags)	Refresh timer callback.
reload_image()	
reschedule_redraw(time_sec)	Reschedule redraw event.
reset_limits()	Reset the bounding box of the viewer extents.
restore_cmap()	Restores the color map from any rotation, stretch and/or shrinkage.
restore_contrast()	Restores the color map from any stretch and/or shrinkage.
rgbmap_cb(rgbmap)	Handle callback for when RGB map has changed.
rotate(deg)	Rotate the view of an image in a channel.
rotate_delta(delta_deg)	Rotate the view of an image in a channel by a delta.
rotation_change_cb(setting, value)	Handle callback related to changes in rotation angle.
save_plain_image_as_file(filepath[, format, ...])	Save the current image shown in the viewer, without any overlaid graphics, in a file with the specified format and quality.
save_profile(**params)	Save the given parameters into profile settings.
save_rgb_image_as_file(filepath[, format, ...])	Save the current image shown in the viewer, with any overlaid graphics, in a file with the specified format and quality.
scale_and_shift_cmap(scale_pct, shift_pct)	Stretch and/or shrink the color map.
scale_cb(setting, value)	Handle callback related to image scaling.
scale_to(scale_x, scale_y[, no_reset])	Scale the image in a channel.
set_autocenter(option)	Set `autocenter` behavior.
set_autocut_params(method, **params)	Set auto-cut parameters.
set_autocuts(autocuts)	Set the auto-cut algorithm.

Table 4 – continued from previous page

set_background(bg)	Set the background color.
set_bg(r, g, b)	Set the background color.
set_calg(dist)	Set color distribution algorithm.
set_canvas(canvas[, private_canvas])	Set the canvas object.
set_cmap(cm)	Set color map.
set_color_algorithm(calg_name, **kwdargs)	Set the color distribution algorithm.
set_color_map(cmap_name)	Set the color map.
set_coordmap(key, mapper)	Set coordinate mapper.
set_cursor(cursor)	Set the cursor in the viewer widget.
set_data(data[, metadata])	Set an image to be displayed by providing raw data.
set_dataobj(image[, add_to_canvas])	Set an image to be displayed.
set_desired_size(width, height)	See *set_window_size()*.
set_enter_focus(tf)	Determine whether the viewer widget should take focus when the cursor enters the window.
set_fg(r, g, b)	Set the foreground color.
set_foreground(fg)	Set the foreground color.
set_image(image[, add_to_canvas])	Set an image to be displayed.
set_imap(im)	Set intensity map.
set_intensity_map(imap_name)	Set the intensity map.
set_limits(limits[, coord])	Set the bounding box of the viewer extents.
set_name(name)	Set viewer name.
set_onscreen_message(text[, redraw])	Called by a subclass to update the onscreen message.
set_pan(pan_x, pan_y[, coord, no_reset])	Set pan position.
set_redraw_lag(lag_sec)	Set lag time for redrawing the canvas.
set_refresh_rate(fps)	Set the refresh rate for redrawing the canvas at a timed interval.
set_renderer(renderer)	Set and initialize the renderer used by this instance.
set_rgbmap(rgbmap)	Set RGB map object used by this instance.
set_scale(scale[, no_reset])	Scale the image in a channel.
set_scale_base_xy(scale_x_base, scale_y_base)	Set stretch factors.
set_scale_limits(scale_min, scale_max)	Set scale limits.
set_window_size(width, height)	Report the size of the window to display the image.
set_zoom_algorithm(name)	Set zoom algorithm.
set_zoomrate(zoomrate)	Set zoom rate.
shift_cmap(pct)	Shift color map.
show_color_bar(tf[, side])	
show_focus_indicator(tf[, color])	
show_mode_indicator(tf[, corner])	
show_pan_mark(tf[, color])	
start_refresh()	Start redrawing the canvas at the previously set timed interval.
stop_refresh()	Stop redrawing the canvas at the previously set timed interval.
swap_xy()	Transform view of the image by swapping the X and Y axes.
switch_cursor(cname)	Switch the viewer's cursor to the one defined under a name.

Table 4 – continued from previous page

take_focus()	Have the widget associated with this viewer take the keyboard focus.
transform(flip_x, flip_y, swap_xy)	Transform view of the image.
transform_cb(setting, value)	Handle callback related to changes in transformations.
update_widget()	Update the area corresponding to the backend widget.
viewable(dataobj)	Test whether `dataobj` is viewable by this viewer.
window_has_origin_upper()	Indicate if window of backend toolkit is implemented with an origin up or down.
window_to_offset(win_x, win_y)	Reverse of *offset_to_window()*.
zoom_fit([axis, no_reset])	Zoom to fit display window.
zoom_in([incr])	Zoom in a level.
zoom_out([decr])	Zoom out a level.
zoom_to(zoomlevel[, no_reset])	Set zoom level in a channel.
zoomsetting_change_cb(setting, value)	Handle callback related to changes in zoom.

Attributes Documentation

vname = 'Ginga Image'

vtypes = [<class 'ginga.BaseImage.BaseImage'>]

Methods Documentation

apply_profile(*profile*, *keylist=None*)

Apply a profile to the viewer.

>   **Parameters**

>   **profile**
>   [SettingGroup]

>   **This function is used to initialize the viewer to a known state. The keylist, if given, will limit the items to be transferred from the profile to viewer settings, otherwise all items are copied.**

apply_profile_or_settings(*image*)

Apply a profile to the viewer.

>   **Parameters**

>   **image**
>   [AstroImage or RGBImage] Image object.

>   **This function is used to initialize the viewer when a new image is loaded. Either the embedded profile settings or the default settings are applied as specified in the channel preferences.**

auto_levels(*autocuts=None*)

Apply auto-cut levels on the image view.

>   **Parameters**

autocuts

[subclass of *AutoCutsBase* or None] An object that implements the desired auto-cut algorithm. If not given, use algorithm from preferences.

auto_orient()

Set the orientation for the image to a reasonable default.

autocut_params_cb(*setting*, *value*)

Handle callback related to changes in auto-cut levels.

calc_pan_pct(*pad=0*, *min_pct=0.0*, *max_pct=0.9*)

Calculate values for vertical/horizontal panning by percentages from the current pan position. Mostly used by scrollbar callbacks.

> **Parameters**
>
> > **pad**
> >
> > [int (optional, defaults to 0)] a padding amount in pixels to add to the limits when calculating
> >
> > **min_pct**
> >
> > [float (optional, range 0.0:1.0, defaults to 0.0)]
> >
> > **max_pct**
> >
> > [float (optional, range 0.0:1.0, defaults to 0.9)]
>
> **Returns**
>
> > **res**
> >
> > [Bunch] calculation results, which include the following attributes: - rng_x : the range of X of the limits (including padding) - rng_y : the range of Y of the limits (including padding) - vis_x : the visually shown range of X in the viewer - vis_y : the visually shown range of Y in the viewer - thm_pct_x : the length of a X scrollbar arm as a ratio - thm_pct_y : the length of a Y scrollbar arm as a ratio - pan_pct_x : the pan position of X as a ratio - pan_pct_y : the pan position of Y as a ratio

canvas_changed_cb(*canvas*, *whence*)

Handle callback for when canvas has changed.

canvascoords(*data_x*, *data_y*)

Same as *get_canvas_xy()*.

capture_default_viewer_profile()

capture_profile(*profile*)

center_cursor()

Center the cursor in the viewer's widget, in both X and Y.

This should be implemented by subclasses.

center_image(*no_reset=True*)

Pan to the center of the image.

> **Parameters**
>
> > **no_reset**
> >
> > [bool] See *set_pan()*.

check_cursor_location()

Check whether the data location of the last known position of the cursor has changed. If so, issue a callback.

---

checkpoint_profile()

clear()

>   Clear the displayed image.

configure(*width*, *height*)

>   See *set_window_size()*.

configure_surface(*width*, *height*)

>   See *configure()*.

copy_attributes(*dst_fi*, *attrlist*, *share=False*, *whence=0*)

>   Copy interesting attributes of our configuration to another image view.

>   > **Parameters**

>   > > **dst_fi**
>   > >   [subclass of *ImageViewBase*] Another instance of image view.

>   > > **attrlist**
>   > >   [list] A list of attribute names to copy. They can be `'transforms'`, `'rotation'`, `'cutlevels'`, `'rgbmap'`, `'zoom'`, `'pan'`, `'autocuts'`, `'limits'`, `'icc'` or `'interpolation'`.

>   > > **share**
>   > >   [bool] If True, the designated settings will be shared, otherwise the values are simply copied.

copy_to_dst(*target*)

>   Extract our image and call *set_image()* on the target with it.

>   > **Parameters**

>   > > **target**
>   > >   Subclass of *ImageViewBase*.

cut_levels(*loval*, *hival*, *no_reset=False*)

>   Apply cut levels on the image view.

>   > **Parameters**

>   > > **loval, hival**
>   > >   [float] Low and high values of the cut levels, respectively.

>   > > **no_reset**
>   > >   [bool] Do not reset `autocuts` setting.

cut_levels_cb(*setting*, *value*)

>   Handle callback related to changes in cut levels.

data_to_offset(*data_x*, *data_y*)

>   Reverse of *offset_to_data()*.

define_cursor(*cname*, *cursor*)

>   Define a viewer cursor under a name. Does not change the current cursor.

>   > **Parameters**

>   > > **cname**
>   > >   [str] name of the cursor to define.

cursor
[object] a cursor object in the back end's toolkit

`cursor` is usually constructed from `make_cursor`.

delayed_redraw()
Handle delayed redrawing of the canvas.

enable_auto_orient(*tf*)
Set `auto_orient` behavior.

Parameters

tf
[bool] Turns automatic image orientation on or off.

enable_autocenter(*option*)
Set `autocenter` behavior.

Parameters

option
[{'on', 'override', 'once', 'off'}] Option for auto-center behavior. A list of acceptable options can also be obtained by `get_autocenter_options()`.

Raises

ginga.ImageView.ImageViewError
Invalid option.

enable_autocuts(*option*)
Set `autocuts` behavior.

Parameters

option
[{'on', 'override', 'once', 'off'}] Option for auto-cut behavior. A list of acceptable options can also be obtained by `get_autocuts_options()`.

Raises

ginga.ImageView.ImageViewError
Invalid option.

enable_autozoom(*option*)
Set `autozoom` behavior.

Parameters

option
[{'on', 'override', 'once', 'off'}] Option for zoom behavior. A list of acceptable options can also be obtained by `get_autozoom_options()`.

Raises

ginga.ImageView.ImageViewError
Invalid option.

flip_x()
Transform view of the image by flipping the X axis.

flip_y()
Transform view of the image by flipping the Y axis.

get_autocenter_options()

Get all valid `autocenter` options.

> **Returns**
>
> > **autocenter_options**
> > [tuple] A list of valid options.

get_autocut_methods()

Same as *ginga.AutoCuts.AutoCutsBase.get_algorithms()*.

get_autocuts_options()

Get all valid `autocuts` options.

> **Returns**
>
> > **autocuts_options**
> > [tuple] A list of valid options.

get_autozoom_options()

Get all valid `autozoom` options.

> **Returns**
>
> > **autozoom_options**
> > [tuple] A list of valid options.

get_bg()

Get the background color.

> **Returns**
>
> > **img_bg**
> > [tuple] RGB values.

get_canvas()

Get the canvas object used by this instance.

> **Returns**
>
> > **canvas**
> > [*DrawingCanvas*] Canvas.

get_canvas_image()

Get canvas image object.

> **Returns**
>
> > **imgobj**
> > [`NormImage`] Normalized image sitting on the canvas.

get_canvas_pt(*data_pt*)

Similar to *get_canvas_xy()*, except that it takes a single array of points.

get_canvas_xy(*data_x*, *data_y*)

Reverse of *get_data_xy()*.

get_center()

Get image center.

> **Returns**

ctr
    [tuple] X and Y positions, in that order.

get_color_algorithms()
    Get available color distribution algorithm names. See `ginga.ColorDist.get_dist_names()`.

get_coordmap(*key*)
    Get coordinate mapper.

    Parameters

    key
        [str] Name of the desired coordinate mapper.

    Returns

    mapper
        Coordinate mapper object (see `ginga.canvas.coordmap`).

get_cursor(*cname*)
    Get the cursor stored under the name. This can be overridden by subclasses, if necessary.

    Parameters

    cname
        [str] name of the cursor to return.

get_cut_levels()
    Get cut levels.

    Returns

    cuts
        [tuple] Low and high values, in that order.

get_data(*data_x*, *data_y*)
    Get the data value at the given position. Indices are zero-based, as in Numpy.

    Parameters

    data_x, data_y
        [int] Data indices for X and Y, respectively.

    Returns

    value
        Data value.

get_data_pct(*xpct*, *ypct*)
    Calculate new data size for the given axis ratios. See `get_limits()`.

    Parameters

    xpct, ypct
        [float] Ratio for X and Y, respectively, where 1 is 100%.

    Returns

    x, y
        [int] Scaled dimensions.

get_data_pt(*win_pt*)
    Similar to `get_data_xy()`, except that it takes a single array of points.

---

get_data_size()

> Get the dimensions of the image currently being displayed.
>
> > **Returns**
> >
> > > **size**
> > > [tuple] Image dimensions in the form of (`width, height`).

get_data_xy(*win_x*, *win_y*)

> Get the closest coordinates in the data array to those reported on the window.
>
> > **Parameters**
> >
> > > **win_x, win_y**
> > > [float or ndarray] Window coordinates.
> >
> > **Returns**
> >
> > > **coord**
> > > [tuple] Data coordinates in the form of (`x, y`).

get_dataobj()

> Get the image currently being displayed.
>
> > **Returns**
> >
> > > **image**
> > > [`AstroImage` or `RGBImage`] Image object.

get_datarect()

> Get the approximate LL and UR corners of the displayed image.
>
> > **Returns**
> >
> > > **rect**
> > > [tuple] Bounding box in data coordinates in the form of (`x1, y1, x2, y2`).

get_desired_size()

> Get desired size.
>
> > **Returns**
> >
> > > **size**
> > > [tuple] Desired size in the form of (`width, height`).

get_dims(*data*)

> Get the first two dimensions of Numpy array data. Data may have more dimensions, but they are not reported.
>
> > **Returns**
> >
> > > **dims**
> > > [tuple] Data dimensions in the form of (`width, height`).

get_draw_rect()

> Get the coordinates in the actual data corresponding to the area needed for drawing images for the current zoom level and pan. Unlike get_pan_rect(), this includes areas outside of the current viewport, but that might be viewed with a transformation or rotation subsequently applied.
>
> > **Returns**
> >
> > > **points**
> > > [list] Coordinates in the form of [(`x0, y0`), (`x1, y1`), (`x2, y2`), (`x3, y3`)] corresponding to the four corners.

get_fg()

    Get the foreground color.

        Returns

            img_fg
                [tuple] RGB values.

get_image()

    Get the image currently being displayed.

        Returns

            image
                [`AstroImage` or `RGBImage`] Image object.

get_image_as_array(*order=None*)

    Get the current image shown in the viewer, with any overlaid graphics, in a numpy array with channels as needed and ordered.

    This can be overridden by subclasses.

get_image_as_buffer(*output=None*, *order=None*)

    Get the current image shown in the viewer, with any overlaid graphics, in a IO buffer with channels as needed and ordered by the back end widget.

    This can be overridden by subclasses.

        Parameters

            output
                [a file IO-like object or None] open python IO descriptor or None to have one created

        Returns

            buffer
                [file IO-like object] This will be the one passed in, unless `output` is None in which case a BytesIO obejct is returned

get_last_data_xy()

    Get the last position of the cursor in data coordinates. This can be overridden by subclasses, if necessary.

get_last_win_xy()

    Get the last position of the cursor in window coordinates. This can be overridden by subclasses, if necessary.

get_limits(*coord='data'*)

    Get the bounding box of the viewer extents.

        Returns

            limits
                [tuple]

            Bounding box in coordinates of type `coord` in the form of
                (ll_pt, ur_pt).

get_logger()

    Get the logger used by this instance.

        Returns

            logger
                [`Logger`] Logger.

get_pan(*coord='data'*)

> Get pan positions.
>
> > **Parameters**
> >
> > > **coord**
> > > [{'data', 'wcs'}] Indicates whether the pan positions are returned in data or WCS space.
> >
> > **Returns**
> >
> > > **positions**
> > > [tuple] X and Y positions, in that order.

get_pan_rect()

> Get the coordinates in the actual data corresponding to the area shown in the display for the current zoom level and pan.
>
> > **Returns**
> >
> > > **points**
> > > [list] Coordinates in the form of `[(x0, y0), (x1, y1), (x2, y2), (x3, y3)]` from lower-left to lower-right.

get_pixel_distance(*x1*, *y1*, *x2*, *y2*)

> Calculate distance between the given pixel positions.
>
> > **Parameters**
> >
> > > **x1, y1, x2, y2**
> > > [number] Pixel coordinates.
> >
> > **Returns**
> >
> > > **dist**
> > > [float] Rounded distance.

get_plain_image_as_widget()

> Get the current image shown in the viewer, without any overlaid graphics, in the format of an image widget in the back end toolkit. Typically used for generating thumbnails. This should be implemented by subclasses.
>
> > **Returns**
> >
> > > **widget**
> > > [object] An image widget object in the viewer's back end toolkit

get_private_canvas()

> Get the private canvas object used by this instance.
>
> > **Returns**
> >
> > > **canvas**
> > > [*DrawingCanvas*] Canvas.

get_refresh_stats()

> Return the measured statistics for timed refresh intervals.
>
> > **Returns**
> >
> > > **stats**
> > > [float] The measured rate of actual back end updates in frames per second.

get_rgb_image_as_buffer(*output=None*, *format='png'*, *quality=90*)

> Get the current image shown in the viewer, with any overlaid graphics, in a file IO-like object encoded as a bitmap graphics file.
>
> This can be overridden by subclasses.
>
> > **Parameters**
> >
> > > **output**
> > > > [a file IO-like object or None] open python IO descriptor or None to have one created
> > >
> > > **format**
> > > > [str] A string defining the format to save the image. Typically at least 'jpeg' and 'png' are supported. (default: 'png')
> > >
> > > **quality: int**
> > > > The quality metric for saving lossy compressed formats.
> >
> > **Returns**
> >
> > > **buffer**
> > > > [file IO-like object] This will be the one passed in, unless `output` is None in which case a BytesIO obejct is returned

get_rgb_image_as_bytes(*format='png'*, *quality=90*)

> Get the current image shown in the viewer, with any overlaid graphics, in the form of a buffer in the form of bytes.
>
> > **Parameters**
> >
> > > **format**
> > > > [str] See *get_rgb_image_as_buffer()*.
> > >
> > > **quality: int**
> > > > See *get_rgb_image_as_buffer()*.
> >
> > **Returns**
> >
> > > **buffer**
> > > > [bytes] The window contents as a buffer in the form of bytes.

get_rgb_image_as_widget(*output=None*, *format='png'*, *quality=90*)

> Get the current image shown in the viewer, with any overlaid graphics, in the form of a image widget in the toolkit of the back end.
>
> > **Parameters**
> >
> > > See :meth:`get_rgb_image_as_buffer`.
> >
> > **Returns**
> >
> > > **widget**
> > > > [object] An image widget object in the viewer's back end toolkit

get_rgb_order()

> Get RGB order.
>
> > **Returns**
> >
> > > **rgb**
> > > > [str] Returns the order of RGBA planes required by the subclass to render the canvas properly.

`get_rgbmap()`

> Get the RGB map object used by this instance.

> > **Returns**

> > > **rgbmap**
> > > [RGBMapper] RGB map.

`get_rotation()`

> Get image rotation angle.

> > **Returns**

> > > **rot_deg**
> > > [float] Rotation angle in degrees.

`get_rotation_info()`

> Get rotation information.

> > **Returns**

> > > **info**
> > > [tuple] X and Y positions, and rotation angle in degrees, in that order.

`get_scale()`

> Same as *get_scale_max()*.

`get_scale_base_xy()`

> Get stretch factors.

> > **Returns**

> > > **stretchfactors**
> > > [tuple] Stretch factors for X and Y, in that order.

`get_scale_limits()`

> Get scale limits.

> > **Returns**

> > > **scale_limits**
> > > [tuple] Minimum and maximum scale limits, respectively.

`get_scale_max()`

> Get maximum scale factor.

> > **Returns**

> > > **scalefactor**
> > > [float] Scale factor for X or Y, whichever is larger.

`get_scale_min()`

> Get minimum scale factor.

> > **Returns**

> > > **scalefactor**
> > > [float] Scale factor for X or Y, whichever is smaller.

`get_scale_text()`

> Report current scaling in human-readable format.

> > **Returns**

> **text**
>> [str] '`<num> x`' if enlarged, or '`1/<num> x`' if shrunken.

get_scale_xy()

> Get scale factors.
>
>> **Returns**
>>
>>> **scalefactors**
>>>> [tuple] Scale factors for X and Y, in that order.

get_settings()

> Get the settings used by this instance.
>
>> **Returns**
>>
>>> **settings**
>>>> [`SettingGroup`] Settings.

get_transforms()

> Get transformations behavior.
>
>> **Returns**
>>
>>> **transforms**
>>>> [tuple] Selected options for `flip_x`, `flip_y`, and `swap_xy`.

get_vip()

> Get the ViewerImageProxy object used by this instance.
>
>> **Returns**
>>
>>> **vip**
>>>> [`ViewerImageProxy`] A ViewerImageProxy object.

get_window_size()

> Get the window size in the underlying implementation.
>
>> **Returns**
>>
>>> **size**
>>>> [tuple] Window size in the form of (`width, height`).

get_zoom()

> Get zoom level.
>
>> **Returns**
>>
>>> **zoomlevel**
>>>> [float] Zoom level.

get_zoom_algorithm()

> Get zoom algorithm.
>
>> **Returns**
>>
>>> **name**
>>>> [str] Name of the zoom algorithm in use.

get_zoomrate()

> Get zoom rate.
>
>> **Returns**

> **zoomrate**
>> [float] Zoom rate.

getwin_array(*order='RGB'*, *alpha=1.0*, *dtype=None*)

getwin_buffer(*order='RGB'*, *alpha=1.0*, *dtype=None*)

> Same as *getwin_array()*, but with the output array converted to C-order Python bytes.

icc_profile_cb(*setting*, *value*)

> Handle callback related to changes in output ICC profiles.

initialize_private_canvas(*private_canvas*)

> Initialize the private canvas used by this instance.

interpolation_change_cb(*setting*, *value*)

> Handle callback related to changes in interpolation.

invert_cmap()

> Invert the color map. See `ginga.RGBMap.RGBMapper.invert_cmap()`.

is_compound()

> Indicate if canvas object is a compound object. This can be re-implemented by subclasses that can overplot objects.

> > **Returns**

> > > **status**
> > >> [bool] Currently, this *always* returns `False`.

is_redraw_pending()

> Indicates whether a deferred redraw has been scheduled.

> > **Returns**

> > > **pending**
> > >> [bool] True if a deferred redraw is pending, False otherwise.

make_cursor(*iconpath*, *x*, *y*)

> Make a cursor in the viewer's native widget toolkit. This should be implemented by subclasses.

> > **Parameters**

> > > **iconpath**
> > >> [str] the path to a PNG image file defining the cursor

> > > **x**
> > >> [int] the X position of the center of the cursor hot spot

> > > **y**
> > >> [int] the Y position of the center of the cursor hot spot

make_timer()

> Return a timer object implemented using the back end. This should be implemented by subclasses.

> > **Returns**

> > > **timer**
> > >> [a Timer object]

offset_to_data(*off_x*, *off_y*)

> Get the closest coordinates in the data array to those in cartesian fixed (non-scaled) canvas coordinates.
>
> > **Parameters**
> >
> > > **off_x, off_y**
> > > [float or ndarray] Cartesian canvas coordinates.
> >
> > **Returns**
> >
> > > **coord**
> > > [tuple] Data coordinates in the form of `(x, y)`.

offset_to_window(*off_x*, *off_y*)

> Convert data offset to window coordinates.
>
> > **Parameters**
> >
> > > **off_x, off_y**
> > > [float or ndarray] Data offsets.
> >
> > **Returns**
> >
> > > **coord**
> > > [tuple] Offset in window coordinates in the form of `(x, y)`.

onscreen_message(*text*, *delay=None*, *redraw=True*)

> Place a message onscreen in the viewer window. This must be implemented by subclasses.
>
> > **Parameters**
> >
> > > **text**
> > > [str] the text to draw in the window
> > >
> > > **delay**
> > > [float or None] if None, the message will remain until another message is set. If a float, specifies the time in seconds before the message will be erased. (default: None)
> > >
> > > **redraw**
> > > [bool] True if the widget should be redrawn right away (so that the message appears). (default: True)

onscreen_message_off()

> Erase any message onscreen in the viewer window.

pan_by_pct(*pct_x*, *pct_y*, *pad=0*)

> Pan by a percentage of the data space. This method is designed to be called by scrollbar callbacks.
>
> > **Parameters**
> >
> > > **pct_x**
> > > [float (range 0.0][1.0)] Percentage in the X range to pan
> > >
> > > **pct_y**
> > > [float (range 0.0][1.0)] Percentage in the Y range to pan
> > >
> > > **pad**
> > > [int (optional, defaults to 0)] a padding amount in pixels to add to the limits when calculating
> > >
> > > **min_pct**
> > > [float (optional, range 0.0:1.0, defaults to 0.0)]

> **max_pct**
> [float (optional, range 0.0:1.0, defaults to 0.9)]

pan_cb(*setting*, *value*)

> Handle callback related to changes in pan.

pan_center_px()

> Pan to the center of the current pixel under the cursor.

pan_delta_px(*x_delta_px*, *y_delta_px*)

> Pan by a delta in X and Y specified in pixels.
>
> > **Parameters**
> >
> > > **x_delta_px**
> > > [float] Delta pixels in X
> > >
> > > **y_delta_px**
> > > [float] Delta pixels in Y

pan_lr(*pct_vw*, *sign*)

> Pan left/right by an amount specified as a percentage of the visible width.
>
> > **Parameters**
> >
> > > **pct_vw**
> > > [float (range 0.0][1.0)] Percent of visible width to pan
> > >
> > > **sign**
> > > [int (1 or -1)] -1 for left, 1 for right

pan_omni(*direction_deg*, *amount*, *lock_x=False*, *lock_y=False*)

> Pan in a direction defined in degrees by an amount specified as a percentage.
>
> > **Parameters**
> >
> > > **direction_deg**
> > > [float (range 0.0][360.0)] Direction in which to pan, where 0.0 is defined as
> > >
> > > **amount**
> > > [float (range 0.0][1.0)] Amount to distribute to X and Y according to direction
> > >
> > > **lock_x**
> > > [bool (optional, defaults to False)] If True, do not allow any panning in the X direction
> > >
> > > **lock_y**
> > > [bool (optional, defaults to False)] If True, do not allow any panning in the Y direction

pan_ud(*pct_vh*, *sign*, *msg=False*)

> Pan up/down by an amount specified as a percentage of the visible height.
>
> > **Parameters**
> >
> > > **pct_vh**
> > > [float (range 0.0][1.0)] Percent of visible height to pan
> > >
> > > **sign**
> > > [int (1 or -1)] -1 for up, 1 for down

panset_pct(*pct_x*, *pct_y*)

> Similar to *set_pan()*, except that pan positions are determined by multiplying data dimensions with the given scale factors, where 1 is 100%.

---

panset_xy(*data_x*, *data_y*, *no_reset=False*)

> Similar to [*set_pan()*](#), except that input pan positions are always in data space.

position_at_canvas_xy(*data_pt*, *canvas_pt*, *no_reset=False*)

> Position a data point at a certain canvas position. Calculates and sets the pan position necessary to position a data point precisely at a point on the canvas.
>
> > **Parameters**
> >
> > > **data_pt**
> > > > [tuple] data point to position, must include x and y (x, y).
> > >
> > > **canvas_pt**
> > > > [tuple] canvas coordinate (cx, cy) where data point should end up.
> > >
> > > **no_reset**
> > > > [bool] See [*set_pan()*](#).

position_cursor(*data_x*, *data_y*)

> Position the current cursor to a location defined it data coords. This should be implemented by subclasses.
>
> > **Parameters**
> >
> > > **data_x**
> > > > [float] the X position to position the cursor in data coords
> > >
> > > **data_y**
> > > > [float] the X position to position the cursor in data coords

prepare_image(*cvs_img*, *cache*, *whence*)

> This can be overridden by subclasses.

recalc_transforms(*trcat=None*)

> Takes a catalog of transforms (`trcat`) and builds the chain of default transforms necessary to do rendering with most backends.

redraw(*whence=0*)

> Redraw the canvas.
>
> > **Parameters**
> >
> > > **whence**
> > > > [int or float] Optimization flag that reduces the time to refresh the viewer by only recalculating what is necessary:
> > > >
> > > > > 0: New image, pan/scale has changed 1: Cut levels or similar has changed 2: Color mapping has changed 2.3: ICC profile has changed 2.5: Transforms have changed 2.6: Rotation has changed 3: Graphical overlays have changed

redraw_data(*whence=0*)

> Render image from RGB map and redraw private canvas.
>
> ---
>
> **Note:** Do not call this method unless you are implementing a subclass.
>
> ---
>
> > **Parameters**
> >
> > > **whence**
> > > > See [*redraw()*](#).

---

redraw_now(*whence=0*)

> Redraw the displayed image.

> > **Parameters**

> > > **whence**
> > > See `redraw()`.

refresh_timer_cb(*timer*, *flags*)

> Refresh timer callback. This callback will normally only be called internally.

> > **Parameters**

> > > **timer**
> > > [a Ginga GUI timer] A GUI-based Ginga timer

> > > **flags**
> > > [dict-like] A set of flags controlling the timer

reload_image()

reschedule_redraw(*time_sec*)

> Reschedule redraw event.

> This should be implemented by subclasses.

> > **Parameters**

> > > **time_sec**
> > > [float] Time, in seconds, to wait.

reset_limits()

> Reset the bounding box of the viewer extents.

> > **Parameters**

> > > **None**

restore_cmap()

> Restores the color map from any rotation, stretch and/or shrinkage. See `ginga.RGBMap.RGBMapper.restore_cmap()`.

restore_contrast()

> Restores the color map from any stretch and/or shrinkage. See `ginga.RGBMap.RGBMapper.reset_sarr()`.

rgbmap_cb(*rgbmap*)

> Handle callback for when RGB map has changed.

rotate(*deg*)

> Rotate the view of an image in a channel.

> ---

> **Note:** Transforming the image is generally faster than rotating, if rotating in 90 degree increments. Also see `transform()`.

> ---

> > **Parameters**

> > > **deg**
> > > [float] Rotation angle in degrees.

rotate_delta(*delta_deg*)

> Rotate the view of an image in a channel by a delta.

> **Note:** Transforming the image is generally faster than rotating, if rotating in 90 degree increments. Also see *transform()*.

>> **Parameters**

>>> **delta_deg**
>>> [float] Incremental rotation angle in degrees.

rotation_change_cb(*setting*, *value*)

> Handle callback related to changes in rotation angle.

save_plain_image_as_file(*filepath*, *format='png'*, *quality=90*)

> Save the current image shown in the viewer, without any overlaid graphics, in a file with the specified format and quality. Typically used for generating thumbnails. This should be implemented by subclasses.

>> **Parameters**

>>> **filepath**
>>> [str] path of the file to write

>>> **format**
>>> [str] See *get_rgb_image_as_buffer()*.

>>> **quality: int**
>>> See *get_rgb_image_as_buffer()*.

save_profile(***params*)

> Save the given parameters into profile settings.

>> **Parameters**

>>> **params**
>>> [dict] Keywords and values to be saved.

save_rgb_image_as_file(*filepath*, *format='png'*, *quality=90*)

> Save the current image shown in the viewer, with any overlaid graphics, in a file with the specified format and quality. This can be overridden by subclasses.

>> **Parameters**

>>> **filepath**
>>> [str] path of the file to write

>>> **format**
>>> [str] See *get_rgb_image_as_buffer()*.

>>> **quality: int**
>>> See *get_rgb_image_as_buffer()*.

scale_and_shift_cmap(*scale_pct*, *shift_pct*)

> Stretch and/or shrink the color map. See `ginga.RGBMap.RGBMapper.scale_and_shift()`.

scale_cb(*setting*, *value*)

> Handle callback related to image scaling.

---

scale_to(*scale_x*, *scale_y*, *no_reset=False*)

> Scale the image in a channel. This only changes the viewer settings; the image is not modified in any way. Also see *zoom_to()*.
>
> > **Parameters**
> >
> > > **scale_x, scale_y**
> > > [float] Scaling factors for the image in the X and Y axes, respectively.
> > >
> > > **no_reset**
> > > [bool] Do not reset `autozoom` setting.

set_autocenter(*option*)

> Set `autocenter` behavior.
>
> > **Parameters**
> >
> > > **option**
> > > [{'on', 'override', 'once', 'off'}]] Option for auto-center behavior. A list of acceptable options can also be obtained by *get_autocenter_options()*.
> >
> > **Raises**
> >
> > > **ginga.ImageView.ImageViewError**
> > > Invalid option.

set_autocut_params(*method*, ***params*)

> Set auto-cut parameters.
>
> > **Parameters**
> >
> > > **method**
> > > [str] Auto-cut algorithm. A list of acceptable options can be obtained by *get_autocut_methods()*.
> > >
> > > **params**
> > > [dict] Algorithm-specific keywords and values.

set_autocuts(*autocuts*)

> Set the auto-cut algorithm.
>
> > **Parameters**
> >
> > > **autocuts**
> > > [subclass of *AutoCutsBase*] An object that implements the desired auto-cut algorithm.

set_background(*bg*)

> Set the background color.
>
> > **Parameters**
> >
> > > **bg**
> > > [str or tuple of float] color name or tuple of floats, between 0 and 1, inclusive.

set_bg(*r*, *g*, *b*)

> Set the background color.
>
> > **Parameters**
> >
> > > **r, g, b**
> > > [float] RGB values, which should be between 0 and 1, inclusive.

set_calg(*dist*)

>   Set color distribution algorithm. See `ginga.RGBMap.RGBMapper.set_dist()`.

set_canvas(*canvas*, *private_canvas=None*)

>   Set the canvas object.

>   **Parameters**

>>   **canvas**
>>   [*DrawingCanvas*] Canvas object.

>>   **private_canvas**
>>   [*DrawingCanvas* or `None`] Private canvas object. If not given, this is the same as `canvas`.

set_cmap(*cm*)

>   Set color map. See `ginga.RGBMap.RGBMapper.set_cmap()`.

set_color_algorithm(*calg_name*, ***kwdargs*)

>   Set the color distribution algorithm.

>   Available color distribution algorithm names can be discovered using `ginga.ColorDist.get_dist_names()`.

>   **Parameters**

>>   **calg_name**
>>   [str] The name of a color distribution algorithm.

>>   **kwdargs**
>>   [dict] Keyword arguments for color distribution object (see `ColorDist`).

set_color_map(*cmap_name*)

>   Set the color map.

>   Available color map names can be discovered using `get_names()`.

>   **Parameters**

>>   **cmap_name**
>>   [str] The name of a color map.

set_coordmap(*key*, *mapper*)

>   Set coordinate mapper.

>   **Parameters**

>>   **key**
>>   [str] Name of the coordinate mapper.

>>   **mapper**
>>   Coordinate mapper object (see *ginga.canvas.coordmap*).

set_cursor(*cursor*)

>   Set the cursor in the viewer widget. This should be implemented by subclasses.

>   **Parameters**

>>   **cursor**
>>   [object] a cursor object in the back end's toolkit

set_data(*data*, *metadata=None*)

Set an image to be displayed by providing raw data.

This is a convenience method for first constructing an image with `AstroImage` and then calling `set_image()`.

> **Parameters**

> > **data**
> > [ndarray] This should be at least a 2D Numpy array.

> > **metadata**
> > [dict or `None`] Image metadata mapping keywords to their respective values.

set_dataobj(*image*, *add_to_canvas=True*)

Set an image to be displayed.

If there is no error, the `'image-unset'` and `'image-set'` callbacks will be invoked.

> **Parameters**

> > **image**
> > [`AstroImage` or `RGBImage`] 2D Image object.

> > **add_to_canvas**
> > [bool] Add image to canvas.

set_desired_size(*width*, *height*)

See `set_window_size()`.

set_enter_focus(*tf*)

Determine whether the viewer widget should take focus when the cursor enters the window.

> **Parameters**

> > **tf**
> > [bool] If True the widget will grab focus when the cursor moves into the window.

set_fg(*r*, *g*, *b*)

Set the foreground color.

> **Parameters**

> > **r, g, b**
> > [float] RGB values, which should be between 0 and 1, inclusive.

set_foreground(*fg*)

Set the foreground color.

> **Parameters**

> > **fg**
> > [str or tuple of float] color name or tuple of floats, between 0 and 1, inclusive.

set_image(*image*, *add_to_canvas=True*)

Set an image to be displayed.

If there is no error, the `'image-unset'` and `'image-set'` callbacks will be invoked.

> **Parameters**

> > **image**
> > [`AstroImage` or `RGBImage`] 2D Image object.

> **add_to_canvas**
>> [bool] Add image to canvas.

set_imap(*im*)

> Set intensity map. See `ginga.RGBMap.RGBMapper.set_imap()`.

set_intensity_map(*imap_name*)

> Set the intensity map.
>
> Available intensity map names can be discovered using `ginga.imap.get_names()`.
>
>> **Parameters**
>>
>>> **imap_name**
>>>> [str] The name of an intensity map.

set_limits(*limits*, *coord='data'*)

> Set the bounding box of the viewer extents.
>
>> **Parameters**
>>
>>> **limits**
>>>> [tuple or None] A tuple setting the extents of the viewer in the form of (`ll_pt, ur_pt`).

set_name(*name*)

> Set viewer name.

set_onscreen_message(*text*, *redraw=True*)

> Called by a subclass to update the onscreen message.
>
>> **Parameters**
>>
>>> **text**
>>>> [str] The text to show in the display.

set_pan(*pan_x*, *pan_y*, *coord='data'*, *no_reset=False*)

> Set pan position.
>
>> **Parameters**
>>
>>> **pan_x, pan_y**
>>>> [float] Pan positions in X and Y.
>>>
>>> **coord**
>>>> [{'data', 'wcs'}] Indicates whether the given pan positions are in data or WCS space.
>>>
>>> **no_reset**
>>>> [bool] Do not reset `autocenter` setting.

set_redraw_lag(*lag_sec*)

> Set lag time for redrawing the canvas.
>
>> **Parameters**
>>
>>> **lag_sec**
>>>> [float] Number of seconds to wait.

set_refresh_rate(*fps*)

> Set the refresh rate for redrawing the canvas at a timed interval.
>
>> **Parameters**
>>
>>> **fps**
>>>> [float] Desired rate in frames per second.

---

set_renderer(*renderer*)

> Set and initialize the renderer used by this instance.

set_rgbmap(*rgbmap*)

> Set RGB map object used by this instance. It controls how the values in the image are mapped to color.
>
> > **Parameters**
> >
> > > **rgbmap**
> > > [RGBMapper] RGB map.

set_scale(*scale*, *no_reset=False*)

> Scale the image in a channel. Also see *zoom_to()*.
>
> > **Parameters**
> >
> > > **scale**
> > > [tuple of float] Scaling factors for the image in the X and Y axes.
> > >
> > > **no_reset**
> > > [bool] Do not reset `autozoom` setting.

set_scale_base_xy(*scale_x_base*, *scale_y_base*)

> Set stretch factors.
>
> > **Parameters**
> >
> > > **scale_x_base, scale_y_base**
> > > [float] Stretch factors for X and Y, respectively.

set_scale_limits(*scale_min*, *scale_max*)

> Set scale limits.
>
> > **Parameters**
> >
> > > **scale_min, scale_max**
> > > [float] Minimum and maximum scale limits, respectively.

set_window_size(*width*, *height*)

> Report the size of the window to display the image.
>
> **Callbacks**
>
> Will call any callbacks registered for the `'configure'` event. Callbacks should have a method signature of:

```
(viewer, width, height, ...)
```

> **Note:** This is called by the subclass with `width` and `height` as soon as the actual dimensions of the allocated window are known.

> > **Parameters**
> >
> > > **width**
> > > [int] The width of the window in pixels.
> > >
> > > **height**
> > > [int] The height of the window in pixels.

set_zoom_algorithm(*name*)

> Set zoom algorithm.

> > **Parameters**

> > > **name**
> > > > [str] Name of a zoom algorithm to use.

set_zoomrate(*zoomrate*)

> Set zoom rate.

> > **Parameters**

> > > **zoomrate**
> > > > [float] Zoom rate.

shift_cmap(*pct*)

> Shift color map. See `ginga.RGBMap.RGBMapper.shift()`.

show_color_bar(*tf*, *side='bottom'*)

show_focus_indicator(*tf*, *color='white'*)

show_mode_indicator(*tf*, *corner='ur'*)

show_pan_mark(*tf*, *color='red'*)

start_refresh()

> Start redrawing the canvas at the previously set timed interval.

stop_refresh()

> Stop redrawing the canvas at the previously set timed interval.

swap_xy()

> Transform view of the image by swapping the X and Y axes.

switch_cursor(*cname*)

> Switch the viewer's cursor to the one defined under a name.

> > **Parameters**

> > > **cname**
> > > > [str] name of the cursor to switch to.

take_focus()

> Have the widget associated with this viewer take the keyboard focus. This should be implemented by subclasses, if they have a widget that can take focus.

transform(*flip_x*, *flip_y*, *swap_xy*)

> Transform view of the image.

> > ---
> > **Note:** Transforming the image is generally faster than rotating, if rotating in 90 degree increments. Also see *rotate()*.
> > ---

> > **Parameters**

> > > **flipx, flipy**
> > > > [bool] If True, flip the image in the X and Y axes, respectively

> **swapxy**
>> [bool] If True, swap the X and Y axes.

transform_cb(*setting*, *value*)

> Handle callback related to changes in transformations.

update_widget()

> Update the area corresponding to the backend widget. This must be implemented by subclasses.

classmethod viewable(*dataobj*)

> Test whether `dataobj` is viewable by this viewer.

window_has_origin_upper()

> Indicate if window of backend toolkit is implemented with an origin up or down.

>> **Returns**

>>> **res**
>>>> [bool] Returns True if the origin is up, False otherwise.

window_to_offset(*win_x*, *win_y*)

> Reverse of *offset_to_window()*.

zoom_fit(*axis='lock'*, *no_reset=False*)

> Zoom to fit display window. Pan the image and scale the view to fit the size of the set limits (usually set to the image size). Parameter `axis` can be used to set which axes are allowed to be scaled; if set to 'lock' then all axes are scaled in such a way as to keep the scale factor uniform between axes. Also see *zoom_to()*.

>> **Parameters**

>>> **axis**
>>>> [str] One of: 'x', 'y', 'xy', or 'lock' (default).

>>> **no_reset**
>>>> [bool] Do not reset `autozoom` setting.

zoom_in(*incr=1.0*)

> Zoom in a level. Also see *zoom_to()*.

>> **Parameters**

>>> **incr**
>>>> [float (optional, defaults to 1)] The value to increase the zoom level

zoom_out(*decr=1.0*)

> Zoom out a level. Also see *zoom_to()*.

>> **Parameters**

>>> **decr**
>>>> [float (optional, defaults to 1)] The value to decrease the zoom level

zoom_to(*zoomlevel*, *no_reset=False*)

> Set zoom level in a channel. This only changes the relevant settings; The image is not modified. Also see *scale_to()*.

> **Note:** In addition to the given zoom level, other zoom settings are defined for the channel in preferences.

>> **Parameters**

>   **zoomlevel**
>
>   > [int] The zoom level to zoom the image. Negative value to zoom out; positive to zoom in.
>
>   **no_reset**
>
>   > [bool] Do not reset `autozoom` setting.

zoomsetting_change_cb(*setting*, *value*)

> Handle callback related to changes in zoom.

8.8 ginga.Bindings Module

8.8.1 Classes

BindingMapError	
BindingMapper(logger[, btnmap, mode_map, ...])	Map physical events to logical events.
ImageViewBindings(logger[, settings])	Configurable event handling (bindings) for UI events.
KeyEvent([key, state, mode, modifiers, ...])	A key press or release event in a Ginga viewer.
PanEvent([button, state, mode, modifiers, ...])	A pinch event in a Ginga viewer.
PinchEvent([button, state, mode, modifiers, ...])	A pinch event in a Ginga viewer.
PointEvent([button, state, mode, modifiers, ...])	A mouse/pointer/cursor event in a Ginga viewer.
ScrollEvent([button, state, mode, ...])	A mouse or trackpad scroll event in a Ginga viewer.
UIEvent([viewer])	Base class for user interface events.

BindingMapError

exception ginga.Bindings.**BindingMapError**

BindingMapper

class ginga.Bindings.**BindingMapper**(*logger*, *btnmap=None*, *mode_map=None*, *modifier_map=None*)

> Bases: `Callbacks`
>
> Map physical events to logical events.
>
> The BindingMapper class maps physical events (key presses, button clicks, mouse movement, etc) into logical events. By registering for logical events, plugins and other event handling code doesn't need to care about the physical controls bindings. The bindings can be changed and everything continues to work.

Methods Summary

add_mode(keyname, mode_name[, mode_type, msg])	
add_modifier(keyname, modname)	
clear_button_map()	

Table 5 – continued from previous page

[clear_event_map]()	
[clear_mode_map]()	
[clear_modifier_map]()	
[current_mode]()	
[get_button](btncode)	
[get_buttons]()	
[get_default_mode_type]()	
[get_modes]()	
[get_modifiers]()	
[has_mode](mode_name)	
[map_button](btncode, alias)	For remapping the buttons to different names.
[map_event](mode, modifiers, trigger, eventname)	
[mode_key_down](viewer, keyname)	This method is called when a key is pressed and was not handled by some other handler with precedence, such as a subcanvas.
[mode_key_up](viewer, keyname)	This method is called when a key is pressed in a mode and was not handled by some other handler with precedence, such as a subcanvas.
[register_for_events](viewer)	
[remove_mode](mode_name)	
[reset_mode](viewer)	
[set_default_mode_type](mode_type)	
[set_mode](name[, mode_type])	
[set_mode_map](mode_map)	
[window_button_press](viewer, btncode, data_x, ...)	
[window_button_release](viewer, btncode, ...)	
[window_enter](viewer)	
[window_focus](viewer, has_focus)	
[window_key_press](viewer, keyname)	

Table 5 – continued from previous page

window_key_release(viewer, keyname)
window_leave(viewer)
window_map(viewer)
window_motion(viewer, btncode, data_x, data_y)
window_pan(viewer, state, delta_x, delta_y)
window_pinch(viewer, state, rot_deg, scale)
window_scroll(viewer, direction, amount, ...)

Methods Documentation

add_mode(*keyname*, *mode_name*, *mode_type='held'*, *msg=None*)

add_modifier(*keyname*, *modname*)

clear_button_map()

clear_event_map()

clear_mode_map()

clear_modifier_map()

current_mode()

get_button(*btncode*)

get_buttons()

get_default_mode_type()

get_modes()

get_modifiers()

has_mode(*mode_name*)

map_button(*btncode*, *alias*)

> For remapping the buttons to different names. 'btncode' is a fixed button code and 'alias' is a logical name.

map_event(*mode*, *modifiers*, *trigger*, *eventname*)

mode_key_down(*viewer*, *keyname*)

> This method is called when a key is pressed and was not handled by some other handler with precedence, such as a subcanvas.

mode_key_up(*viewer*, *keyname*)

> This method is called when a key is pressed in a mode and was not handled by some other handler with precedence, such as a subcanvas.

register_for_events(*viewer*)

remove_mode(*mode_name*)

reset_mode(*viewer*)

set_default_mode_type(*mode_type*)

set_mode(*name*, *mode_type=None*)

set_mode_map(*mode_map*)

window_button_press(*viewer*, *btncode*, *data_x*, *data_y*)

window_button_release(*viewer*, *btncode*, *data_x*, *data_y*)

window_enter(*viewer*)

window_focus(*viewer*, *has_focus*)

window_key_press(*viewer*, *keyname*)

window_key_release(*viewer*, *keyname*)

window_leave(*viewer*)

window_map(*viewer*)

window_motion(*viewer*, *btncode*, *data_x*, *data_y*)

window_pan(*viewer*, *state*, *delta_x*, *delta_y*)

window_pinch(*viewer*, *state*, *rot_deg*, *scale*)

window_scroll(*viewer*, *direction*, *amount*, *data_x*, *data_y*)

ImageViewBindings

class ginga.Bindings.**ImageViewBindings**(*logger*, *settings=None*)

Bases: object

Configurable event handling (bindings) for UI events.

This module handles the mapping of user interface events (keyboard, mouse, trackpad, gestures) to operations on a Ginga viewer.

Attributes Summary

action_prefixes

Methods Summary

[add_cursor](viewer, curname, curpath)	
[add_mode_obj](mode_obj)	
[enable](**kwargs)	General enable function encompassing all user interface features.
[enable_all](tf)	
[enable_cmap](tf)	Enable the color map to be warped interactively (True/False).
[enable_cuts](tf)	Enable the cuts levels to be set interactively (True/False).
[enable_flip](tf)	Enable the image to be flipped interactively (True/False).
[enable_pan](tf)	Enable the image to be panned interactively (True/False).
[enable_rotate](tf)	Enable the image to be rotated interactively (True/False).
[enable_zoom](tf)	Enable the image to be zoomed interactively (True/False).
[get_direction](direction[, rev])	Translate a direction in compass degrees into 'up' or 'down'.
[get_feature_allow](feat_name)	
[get_mode_obj](mode_name)	
[get_settings]()	
[initialize_settings](settings)	
[merge_actions](viewer, bindmap, obj, tups)	
[mode_set_cb](bm, mode, mode_type, viewer)	
[parse_combo](combo, modes_set, modifiers_set, pfx)	Parse a string into a mode, a set of modifiers and a trigger.
[reset](viewer)	
[set_bindings](viewer)	
[set_mode](viewer, name[, mode_type])	
[setup_settings_events](viewer, bindmap)	
[window_map](viewer)	

Attributes Documentation

action_prefixes = ['kp_', 'ms_', 'sc_', 'pi_', 'pa_']

Methods Documentation

add_cursor(*viewer*, *curname*, *curpath*)

add_mode_obj(*mode_obj*)

enable(***kwargs*)

    General enable function encompassing all user interface features. Usage (e.g.): viewer.enable(rotate=False, flip=True)

enable_all(*tf*)

enable_cmap(*tf*)

    Enable the color map to be warped interactively (True/False).

enable_cuts(*tf*)

    Enable the cuts levels to be set interactively (True/False).

enable_flip(*tf*)

    Enable the image to be flipped interactively (True/False).

enable_pan(*tf*)

    Enable the image to be panned interactively (True/False).

enable_rotate(*tf*)

    Enable the image to be rotated interactively (True/False).

enable_zoom(*tf*)

    Enable the image to be zoomed interactively (True/False).

get_direction(*direction*, *rev=False*)

    Translate a direction in compass degrees into 'up' or 'down'.

get_feature_allow(*feat_name*)

get_mode_obj(*mode_name*)

get_settings()

initialize_settings(*settings*)

merge_actions(*viewer*, *bindmap*, *obj*, *tups*)

mode_set_cb(*bm*, *mode*, *mode_type*, *viewer*)

parse_combo(*combo*, *modes_set*, *modifiers_set*, *pfx*)

    Parse a string into a mode, a set of modifiers and a trigger.

reset(*viewer*)

set_bindings(*viewer*)

set_mode(*viewer*, *name*, *mode_type='oneshot'*)

setup_settings_events(*viewer*, *bindmap*)

window_map(*viewer*)

KeyEvent

class ginga.Bindings.**KeyEvent**(*key=None*, *state=None*, *mode=None*, *modifiers=None*, *data_x=None*, *data_y=None*, *viewer=None*)

> Bases: *UIEvent*
>
> A key press or release event in a Ginga viewer.
>
> > **Attributes**
> >
> > > **key**
> > > > [str] The key as it is known to Ginga
> > >
> > > **state: str**
> > > > 'down' if a key press, 'up' if a key release
> > >
> > > **mode**
> > > > [str] The mode name of the mode that was active when the event happened
> > >
> > > **modifiers**
> > > > [set of str] A set of names of modifier keys that were pressed at the time
> > >
> > > **data_x**
> > > > [float] X part of the data coordinates of the viewer under the cursor
> > >
> > > **data_y**
> > > > [float] Y part of the data coordinates of the viewer under the cursor
> > >
> > > **viewer**
> > > > [subclass of *ImageViewBase*] The viewer in which the event happened

PanEvent

class ginga.Bindings.**PanEvent**(*button=None*, *state=None*, *mode=None*, *modifiers=None*, *delta_x=None*, *delta_y=None*, *data_x=None*, *data_y=None*, *viewer=None*)

> Bases: *UIEvent*
>
> A pinch event in a Ginga viewer.
>
> > **Attributes**
> >
> > > **button**
> > > > [str] The name of the button as set up in the configuration
> > >
> > > **state: str**
> > > > 'start' (gesture starting), 'move' (in action) or 'stop' (done)
> > >
> > > **mode**
> > > > [str] The mode name of the mode that was active when the event happened
> > >
> > > **modifiers**
> > > > [set of str] A set of names of modifier keys that were pressed at the time
> > >
> > > **delta_x**
> > > > [float] Amount of scroll movement in the X direction

> **delta_y**
>> [float] Amount of scroll movement in the Y direction
>
> **data_x**
>> [float] X part of the data coordinates of the viewer under the cursor
>
> **data_y**
>> [float] Y part of the data coordinates of the viewer under the cursor
>
> **viewer**
>> [subclass of *ImageViewBase*] The viewer in which the event happened

PinchEvent

class `ginga.Bindings.`**PinchEvent**(*button=None*, *state=None*, *mode=None*, *modifiers=None*, *rot_deg=None*, *scale=None*, *data_x=None*, *data_y=None*, *viewer=None*)

> Bases: *UIEvent*
>
> A pinch event in a Ginga viewer.
>
> > **Attributes**
> >
> > **button**
> >> [str] The name of the button as set up in the configuration
> >
> > **state: str**
> >> 'start' (gesture starting), 'move' (in action) or 'stop' (done)
> >
> > **mode**
> >> [str] The mode name of the mode that was active when the event happened
> >
> > **modifiers**
> >> [set of str] A set of names of modifier keys that were pressed at the time
> >
> > **rot_deg**
> >> [float] Amount of rotation in degrees
> >
> > **scale**
> >> [float] Scale of the pinch shrink or enlargement
> >
> > **data_x**
> >> [float] X part of the data coordinates of the viewer under the cursor
> >
> > **data_y**
> >> [float] Y part of the data coordinates of the viewer under the cursor
> >
> > **viewer**
> >> [subclass of *ImageViewBase*] The viewer in which the event happened

PointEvent

class `ginga.Bindings.`**PointEvent**(*button=None*, *state=None*, *mode=None*, *modifiers=None*, *data_x=None*, *data_y=None*, *viewer=None*)

> Bases: *UIEvent*
>
> A mouse/pointer/cursor event in a Ginga viewer.
>
> > **Attributes**

button

[str] The name of the button as set up in the configuration

state: str

'down' if a press, 'move' if being dragged, 'up' if a release

mode

[str] The mode name of the mode that was active when the event happened

modifiers

[set of str] A set of names of modifier keys that were pressed at the time

data_x

[float] X part of the data coordinates of the viewer under the cursor

data_y

[float] Y part of the data coordinates of the viewer under the cursor

viewer

[subclass of *ImageViewBase*] The viewer in which the event happened

ScrollEvent

class ginga.Bindings.**ScrollEvent**(*button=None*, *state=None*, *mode=None*, *modifiers=None*, *direction=None*, *amount=None*, *data_x=None*, *data_y=None*, *viewer=None*)

Bases: *UIEvent*

A mouse or trackpad scroll event in a Ginga viewer.

Attributes

button

[str] The name of the button as set up in the configuration

state: str

Always 'scroll'

mode

[str] The mode name of the mode that was active when the event happened

modifiers

[set of str] A set of names of modifier keys that were pressed at the time

direction

[float] A direction in compass degrees of the scroll

amount

[float] The amount of the scroll

data_x

[float] X part of the data coordinates of the viewer under the cursor

data_y

[float] Y part of the data coordinates of the viewer under the cursor

viewer

[subclass of *ImageViewBase*] The viewer in which the event happened

UIEvent

class ginga.Bindings.**UIEvent**(*viewer=None*)

> Bases: object

> Base class for user interface events.

Methods Summary

accept()	
was_handled()	

Methods Documentation

accept()

was_handled()

8.9 ginga.rv.main Module

This module handles the main reference viewer.

8.9.1 Classes

ReferenceViewer([layout, plugins])	This class exists solely to be able to customize the reference viewer startup.

ReferenceViewer

class ginga.rv.main.**ReferenceViewer**(*layout=['seq', {}, ['vbox', {'name': 'top', 'width': 1400, 'height': 700},
{'row': ['hbox', {'name': 'menu'}], 'stretch': 0}, {'row': ['hpanel',
{'name': 'hpnl'}, ['ws', {'name': 'left', 'wstype': 'tabs', 'width': 300,
'height': -1, 'group': 2}, [('Info', ['vpanel', {}, ['ws', {'name': 'uleft',
'wstype': 'stack', 'height': 250, 'group': 3}], ['ws', {'name': 'lleft',
'wstype': 'tabs', 'height': 330, 'group': 3}]])]]], ['vbox', {'name':
'main', 'width': 600}, {'row': ['ws', {'name': 'channels', 'wstype':
'tabs', 'group': 1, 'use_toolbar': True, 'default': True}], 'stretch': 1},
{'row': ['ws', {'name': 'cbar', 'wstype': 'stack', 'group': 99}], 'stretch':
0}, {'row': ['ws', {'name': 'readout', 'wstype': 'stack', 'group': 99}],
'stretch': 0}, {'row': ['ws', {'name': 'operations', 'wstype': 'stack',
'group': 99}], 'stretch': 0}], ['ws', {'name': 'right', 'wstype': 'tabs',
'width': 400, 'height': -1, 'group': 2}, [('Dialogs', ['ws', {'name':
'dialogs', 'wstype': 'tabs', 'group': 2}])]]], 'stretch': 1}, {'row': ['ws',
{'name': 'toolbar', 'wstype': 'stack', 'height': 40, 'group': 2}], 'stretch':
0}, {'row': ['hbox', {'name': 'status'}], 'stretch': 0}]],
plugins=[{'module': 'Operations', 'workspace': 'operations', 'start':
True, 'hidden': True, 'category': 'System', 'menu': 'Operations [G]',
'ptype': 'global'}, {'module': 'Toolbar', 'workspace': 'toolbar', 'start':
True, 'hidden': True, 'category': 'System', 'menu': 'Toolbar [G]',
'ptype': 'global'}, {'module': 'Pan', 'workspace': 'uleft', 'start': True,
'hidden': True, 'category': 'System', 'menu': 'Pan [G]', 'ptype':
'global'}, {'module': 'Info', 'tab': 'Synopsis', 'workspace': 'lleft', 'start':
True, 'hidden': True, 'category': 'System', 'menu': 'Info [G]', 'ptype':
'global'}, {'module': 'Thumbs', 'tab': 'Thumbs', 'workspace': 'right',
'start': True, 'hidden': True, 'category': 'System', 'menu': 'Thumbs
[G]', 'ptype': 'global'}, {'module': 'Contents', 'tab': 'Contents',
'workspace': 'right', 'start': True, 'hidden': True, 'category': 'System',
'menu': 'Contents [G]', 'ptype': 'global'}, {'module': 'Colorbar',
'workspace': 'cbar', 'start': True, 'hidden': True, 'category': 'System',
'menu': 'Colorbar [G]', 'ptype': 'global'}, {'module': 'Cursor',
'workspace': 'readout', 'start': True, 'hidden': True, 'category':
'System', 'menu': 'Cursor [G]', 'ptype': 'global'}, {'module': 'Errors',
'tab': 'Errors', 'workspace': 'right', 'start': True, 'hidden': True,
'category': 'System', 'menu': 'Errors [G]', 'ptype': 'global'}, {'module':
'Downloads', 'tab': 'Downloads', 'workspace': 'right', 'start': False,
'menu': 'Downloads [G]', 'category': 'Utils', 'ptype': 'global'},
{'module': 'Blink', 'tab': 'Blink Channels', 'workspace': 'right', 'start':
False, 'menu': 'Blink Channels [G]', 'category': 'Analysis', 'ptype':
'global'}, {'module': 'Blink', 'workspace': 'dialogs', 'menu': 'Blink
Images', 'category': 'Analysis', 'ptype': 'local'}, {'module': 'Crosshair',
'workspace': 'left', 'category': 'Analysis', 'ptype': 'local'}, {'module':
'Cuts', 'workspace': 'dialogs', 'category': 'Analysis', 'ptype': 'local'},
{'module': 'LineProfile', 'workspace': 'dialogs', 'category':
'Analysis.Datacube', 'ptype': 'local'}, {'module': 'Histogram',
'workspace': 'dialogs', 'category': 'Analysis', 'ptype': 'local'},
{'module': 'Overlays', 'workspace': 'dialogs', 'category': 'Analysis',
'ptype': 'local'}, {'module': 'Pick', 'workspace': 'dialogs', 'category':
'Analysis', 'ptype': 'local'}, {'module': 'PixTable', 'workspace':
'dialogs', 'category': 'Analysis', 'ptype': 'local'}, {'module': 'TVMark',
'workspace': 'dialogs', 'category': 'Analysis', 'ptype': 'local'},
{'module': 'TVMask', 'workspace': 'dialogs', 'category': 'Analysis',
'ptype': 'local'}, {'module': 'WCSMatch', 'tab': 'WCSMatch',
'workspace': 'right', 'start': False, 'menu': 'WCS Match [G]',
'category': 'Analysis', 'ptype': 'global'}, {'module': 'Command', 'tab':
'Command', 'workspace': 'lleft', 'start': False, 'menu': 'Command Line
[G]', 'category': 'Debug', 'ptype': 'global'}, {'module': 'Log', 'tab':
'Log', 'workspace': 'right', 'start': False, 'menu': 'Logger Info [G]',
'category': 'Debug', 'ptype': 'global'}, {'module': 'MultiDim',
'workspace': 'lleft', 'category': 'Navigation', 'ptype': 'local'},*

Bases: `object`

This class exists solely to be able to customize the reference viewer startup.

Methods Summary

add_default_options(argprs)	Adds the default reference viewer startup options to an ArgumentParser instance `argprs`.
add_default_plugins([except_global, ...])	Add the ginga-distributed default set of plugins to the reference viewer.
add_plugin_spec(spec)	
add_separately_distributed_plugins()	
clear_default_plugins()	
main(options, args)	Main routine for running the reference viewer.

Methods Documentation

add_default_options(*argprs*)

>   Adds the default reference viewer startup options to an ArgumentParser instance `argprs`.

add_default_plugins(*except_global=[]*, *except_local=[]*)

>   Add the ginga-distributed default set of plugins to the reference viewer.

add_plugin_spec(*spec*)

add_separately_distributed_plugins()

clear_default_plugins()

main(*options*, *args*)

>   Main routine for running the reference viewer.
>
>   `options` is a ArgumentParser object that has been populated with values from parsing the command line. It should at least include the options from add_default_options()
>
>   `args` is a list of arguments to the viewer after parsing out options. It should contain a list of files or URLs to load.

8.10 ginga.colors Module

Module to handle colors supported by Ginga.

8.10.1 Functions

recalc_color_list()	Recalculate `ginga.colors.color_list`.
lookup_color(name[, format])	Find RGB or hex values for a supported color.
resolve_color(color)	Return RGB tuple of a given color.
add_color(name, tup)	Add (`color`, `RGB` to `ginga.colors.color_list`.
remove_color(name)	Remove a given color from `ginga.colors.color_list`.
get_colors()	Return `ginga.colors.color_list`.
scan_rgbtxt(filepath)	Parse colors from a given filename.
scan_rgbtxt_buf(buf)	Parse colors from a given string buffer.

recalc_color_list

ginga.colors.**recalc_color_list**()

> Recalculate `ginga.colors.color_list`.

lookup_color

ginga.colors.**lookup_color**(*name*, *format='tuple'*)

> Find RGB or hex values for a supported color.
>
> **name**
>> [str] Color name (e.g., `'red'`) or hash (e.g., `'#ff0000'`). Color name is case-sensitive.
>
> **format**
>> [{'tuple', 'hash'}] Desired output to be an RGB tuple or hash.
>
>> **Returns**
>>
>>> **color**
>>>> [tuple or str] Color value as specified by `format`.
>>
>> **Raises**
>>
>>> **KeyError**
>>>> Color name is not supported.
>>
>>> **ValueError**
>>>> Invalid format.

resolve_color

ginga.colors.**resolve_color**(*color*)

> Return RGB tuple of a given color.

add_color

ginga.colors.**add_color**(*name*, *tup*)

    Add (`color`, `RGB` to `ginga.colors.color_list`.

remove_color

ginga.colors.**remove_color**(*name*)

    Remove a given color from `ginga.colors.color_list`.

get_colors

ginga.colors.**get_colors**()

    Return `ginga.colors.color_list`.

scan_rgbtxt

ginga.colors.**scan_rgbtxt**(*filepath*)

    Parse colors from a given filename.

scan_rgbtxt_buf

ginga.colors.**scan_rgbtxt_buf**(*buf*)

    Parse colors from a given string buffer.

8.11 ginga.AutoCuts Module

8.11.1 Functions

get_autocuts(name)	Return the class object used to implement an autocuts algorithm.
get_autocuts_names()	Return the list of algorithm names for the available autocuts methods.

get_autocuts

ginga.AutoCuts.**get_autocuts**(*name*)

    Return the class object used to implement an autocuts algorithm.

        Parameters

            name

                [str] The name of the algorithm

        Returns

            klass

                [subclass of *AutoCutsBase*] A class implementing the auto cut levels algorithm

get_autocuts_names

ginga.AutoCuts.`get_autocuts_names`()

> Return the list of algorithm names for the available autocuts methods.

>> **Returns**

>>> **names**
>>>> [list of str] A list of the known auto cut levels algorithm names

8.11.2 Classes

AutoCutsBase(logger)	Base class for auto cuts algorithms.
Minmax(logger)	Calculate the cut levels as the minimum and maximum of the data.
Histogram(logger[, usecrop, sample, ...])	Calculate the cut levels based on a histogram analysis of the data.
StdDev(logger[, usecrop, sample, ...])	Calculate the cut levels based on a standard deviation analysis of the data.
MedianFilter(logger[, num_points, length])	Calculate the cut levels based on a median filtering analysis of the data.
ZScale(logger[, contrast, num_points])	Calculate the cut levels based on a median filtering analysis of the data.

AutoCutsBase

class ginga.AutoCuts.**AutoCutsBase**(*logger*)

> Bases: object

> Base class for auto cuts algorithms.

Methods Summary

calc_cut_levels(image)	Calculate the cut levels of an image according to the parameters.
calc_cut_levels_data(data_np)	Calculate the cut levels of a ndarray according to the parameters.
cut_levels(data_np, loval, hival[, vmin, vmax])	Apply the cut levels to data.
get_algorithms()	Return the list of autocuts algorithms.
get_autocut_levels(image)	Calculate the cut levels of an image according to the parameters.
get_crop(image[, crop_radius])	Get a cropped region of data from an image.
get_crop_data(data[, crop_radius])	Get a cropped region of data from a ndarray.
get_full(image[, px_limit])	Get the full (or cropped) array of data from an image.
get_params_metadata()	
get_sample(image[, num_points])	Return a sample from the full data array of the passed image.
get_sample_data(data[, num_points])	Return a sample from a data array.
update_params(**param_dict)	

Methods Documentation

calc_cut_levels(*image*)

Calculate the cut levels of an image according to the parameters.

> **Parameters**
>
> > **image**
> > [subclass of `BaseImage`] Image object from which the cut levels should be calculated
>
> **Returns**
>
> > **cut_levels**
> > [tuple of float (loval, hival)] The cut levels that were calculated

calc_cut_levels_data(*data_np*)

Calculate the cut levels of a ndarray according to the parameters.

> **Parameters**
>
> > **data_np**
> > [ndarray] A numpy array of data from which the cut levels should be calculated
>
> **Returns**
>
> > **cut_levels**
> > [tuple of float (loval, hival)] The cut levels that were calculated

cut_levels(*data_np*, *loval*, *hival*, *vmin=0.0*, *vmax=255.0*)

Apply the cut levels to data.

> **Parameters**
>
> > **data_np**
> > [ndarray] A numpy array of data to which the cut levels should be applied

> **loval**
> [float] The low cut level

> **hival**
> [float] The high cut level

> **vmin**
> [float (optional, default 0.0)] The floor of the range which is the output of the cut levels

> **vmax**
> [float (optional, default 255.0)] The ceiling of the range which is the output of the cut levels

> **Returns**

> **result**
> [ndarray] The result of applying the cut levels to the input array

get_algorithms()

Return the list of autocuts algorithms.

> **Parameters**

> **None**

> **Returns**

> **names: tuple of str**
> The names of the autocut algorithms

get_autocut_levels(*image*)

Calculate the cut levels of an image according to the parameters.

> **Parameters**

> **image**
> [subclass of `BaseImage`] Image object from which the cut levels should be calculated

> **Returns**

> **cut_levels**
> [tuple of float (loval, hival)] The cut levels that were calculated

get_crop(*image*, *crop_radius=None*)

Get a cropped region of data from an image.

> **Parameters**

> **image**
> [subclass of `BaseImage`] Image object from which the cut levels should be calculated

> **crop_radius**
> [int (optional, default `None`)] The radius of a crop region to extract from the image. If `None`, then the radius will default to the `crop_radius` attribute.

> **Returns**

> **result**
> [ndarray] The cropped data

get_crop_data(*data*, *crop_radius=None*)

Get a cropped region of data from a ndarray.

> **Parameters**

> **data**
>> [ndarray] Image data from which the data should be cropped
>
> **crop_radius**
>> [int (optional, default None)] The radius of a crop region to extract from the image data. If None, then the radius will default to the `crop_radius` attribute.
>
> **Returns**
>
> **result**
>> [ndarray] The cropped data

get_full(*image*, *px_limit=None*)

> Get the full (or cropped) array of data from an image.
>
> **Parameters**
>
> **image**
>> [subclass of BaseImage] Image object from which the cut levels should be calculated
>
> **px_limit**
>> [int (optional, default None)] The limit for extracting the full image data. If the number of pixels in the data is larger than this value, a crop from the data will be used instead.
>
> **Returns**
>
> **result**
>> [ndarray] The (possibly cropped) data

classmethod get_params_metadata()

get_sample(*image*, *num_points=None*)

> Return a sample from the full data array of the passed image.
>
> **Parameters**
>
> **image**
>> [subclass of BaseImage] Image object from which the cut levels should be calculated
>
> **num_points**
>> [int (optional, default None)] Specifies the number of points to sample. If None, the number pixels will be calculated to a "reasonable representative sample".
>
> **Returns**
>
> **result**
>> [ndarray] The sampled data

get_sample_data(*data*, *num_points=None*)

> Return a sample from a data array.
>
> **Parameters**
>
> **data**
>> [ndarray] Image data from which the data should be sampled
>
> **num_points**
>> [int (optional, default None)] Specifies the number of points to sample. If None, the number pixels will be calculated to a "reasonable representative sample".
>
> **Returns**
>
> **result**
>> [ndarray] The sampled data

update_params(***param_dict*)

Minmax

class ginga.AutoCuts.**Minmax**(*logger*)

Bases: *AutoCutsBase*

Calculate the cut levels as the minimum and maximum of the data.

The calculation is:
loval = min(sample_data) hival = max(sample_data)

> **Parameters**
>
> > **logger**
> > [Logger] Logger for tracing and debugging.

Methods Summary

calc_cut_levels(image)	See subclass documentation.
calc_cut_levels_data(data_np)	See subclass documentation.

Methods Documentation

calc_cut_levels(*image*)

> See subclass documentation.

calc_cut_levels_data(*data_np*)

> See subclass documentation.

Histogram

class ginga.AutoCuts.**Histogram**(*logger*, *usecrop=None*, *sample='crop'*, *full_px_limit=None*, *num_points=None*, *pct=0.999*, *numbins=2048*)

Bases: *AutoCutsBase*

Calculate the cut levels based on a histogram analysis of the data.

> **Parameters**
>
> > **logger**
> > [Logger] Logger for tracing and debugging.
> >
> > **sample**
> > [str (optional, 'crop', 'grid', or 'full', default: 'crop')] Specifies how to access the image for calculation: - crop: crop an area from the middle of the image - grid: sample data in a grid pattern across the image - full: use the full image data
> >
> > **full_px_limit**
> > [int (optional, defaults to 1M)] Specifies the limit for using the full data if sample == 'full'. If the number of pixels in the image is larger than this, then the image will fall back to using a crop.

num_points
>   [int (optional, defaults to None)] Specifies the number of points in the grid if sample == 'grid', or the diameter of the crop, if sample == 'crop' (or 'full' and number of pixels exceeds `full_px_limit`). If None, the number pixels will be calculated to a "reasonable representative sample".

pct
>   [float (optional, range: 0.0 - 1.0, defaults to 0.999)] Specifies the percentage of the histogram bins to retain

numbins
>   [int (optional, defaults to 2048)] Specifies the number of bins used in calculating the histogram

Methods Summary

calc_cut_levels(image)	See subclass documentation.
calc_cut_levels_data(data_np)	See subclass documentation.
calc_histogram(data[, pct, numbins])	Internal function used by this class.
get_params_metadata()	

Methods Documentation

calc_cut_levels(*image*)
>   See subclass documentation.

calc_cut_levels_data(*data_np*)
>   See subclass documentation.

calc_histogram(*data*, *pct=1.0*, *numbins=2048*)
>   Internal function used by this class.

classmethod get_params_metadata()

StdDev

class ginga.AutoCuts.**StdDev**(*logger*, *usecrop=None*, *sample='grid'*, *full_px_limit=None*, *num_points=None*, *hensa_lo=-1.5*, *hensa_hi=4.0*)

Bases: *AutoCutsBase*

Calculate the cut levels based on a standard deviation analysis of the data.

The calculation is:
>   loval = hensa_lo * sdev(sample_data) + mean(sample_data) hival = hensa_hi * sdev(sample_data) + mean(sample_data)

Parameters

logger
>   [Logger] Logger for tracing and debugging.

sample

> [str (optional, 'crop', 'grid', or 'full', default: 'crop')] Specifies how to access the image for calculation: - crop: crop an area from the middle of the image - grid: sample data in a grid pattern across the image - full: use the full image data

full_px_limit

> [int (optional, defaults to 1M)] Specifies the limit for using the full data if sample == 'full'. If the number of pixels in the image is larger than this, then the image will fall back to using a crop.

num_points

> [int (optional, defaults to None)] Specifies the number of points in the grid if sample == 'grid', or the diameter of the crop, if sample == 'crop' (or 'full' and number of pixels exceeds `full_px_limit`). If None, the number pixels will be calculated to a "reasonable representative sample".

hensa_lo

> [float (optional, defaults to -1.5)] Specifies the low cut multiplication factor to apply to the standard deviation before adding the median (usually < 0)

hensa_hi

> [float (optional, defaults to 4.0)] Specifies the low cut multiplication factor to apply to the standard deviation before adding the median (usually > 0)

Methods Summary

[`calc_cut_levels`](image)	See subclass documentation.
[`calc_cut_levels_data`](data_np)	See subclass documentation.
[`calc_stddev`](data[, hensa_lo, hensa_hi])	Internal function used by this class.
[`get_params_metadata`]()	

Methods Documentation

calc_cut_levels(*image*)

> See subclass documentation.

calc_cut_levels_data(*data_np*)

> See subclass documentation.

calc_stddev(*data*, *hensa_lo=-1.5*, *hensa_hi=4.0*)

> Internal function used by this class.

classmethod get_params_metadata()

MedianFilter

class ginga.AutoCuts.**MedianFilter**(*logger*, *num_points=2000*, *length=5*)

   Bases: *AutoCutsBase*

   Calculate the cut levels based on a median filtering analysis of the data.

   The calculation is:
      out = median_filter(sample_data, size=length) loval, hival = min(out), max(out)

   Parameters

      logger
         [Logger] Logger for tracing and debugging.

      num_points
         [int (optional, defaults to None)] Specifies the number of points to sample making up the grid. If None, the number of points will be calculated to a "reasonable representative sample".

      length
         [int (optional, defaults to 5)] Specifies the size of the median filter to apply to the data

   Methods Summary

calc_cut_levels(image)	See subclass documentation.
calc_cut_levels_data(data_np)	See subclass documentation.
calc_medianfilter(data[, length])	Internal function used by this class.
get_params_metadata()	

   Methods Documentation

   calc_cut_levels(*image*)
      See subclass documentation.

   calc_cut_levels_data(*data_np*)
      See subclass documentation.

   calc_medianfilter(*data*, *length=5*)
      Internal function used by this class.

   classmethod get_params_metadata()

ZScale

class ginga.AutoCuts.**ZScale**(*logger*, *contrast=0.25*, *num_points=None*)

   Bases: *AutoCutsBase*

   Calculate the cut levels based on a median filtering analysis of the data.

   Based on STScI's numdisplay implementation of IRAF's ZScale.

   The calculation is:
      loval, hival = zscale(sample_data, contrast)

Parameters

logger
[Logger] Logger for tracing and debugging.

contrast
[float (optional, range: 0.0 - 1.0, defaults to 0.25)] Specifies the contrast parameter to use in the zscale calculation

num_points
[int (optional, defaults to None)] Specifies the number of points to sample making up the grid. If None, the number of points will be calculated to a "reasonable representative sample".

Methods Summary

calc_cut_levels(image)	See subclass documentation.
calc_cut_levels_data(data_np)	See subclass documentation.
calc_zscale(data[, contrast, num_points])	Internal function used by this class.
get_params_metadata()	

Methods Documentation

calc_cut_levels(*image*)
See subclass documentation.

calc_cut_levels_data(*data_np*)
See subclass documentation.

calc_zscale(*data*, *contrast=0.25*, *num_points=1000*)
Internal function used by this class.

classmethod get_params_metadata()

8.12 ginga.util.io.io_asdf Module

Module wrapper for loading ASDF files.

Note: The API for this module is currently unstable and may change in a future release.

8.12.1 Functions

load_file(filepath[, idx, logger])	Load an object from an ASDF file.
load_asdf(asdf_obj[, idx, logger])	Load data from an open ASDF object.
load_from_asdf(asdf_obj[, data_key, ...])	Load from an ASDF object.

load_file

ginga.util.io.io_asdf.**load_file**(*filepath*, *idx=None*, *logger=None*, ***kwargs*)

>   Load an object from an ASDF file. See `ginga.util.loader()` for more info.

load_asdf

ginga.util.io.io_asdf.**load_asdf**(*asdf_obj*, *idx=None*, *logger=None*)

>   Load data from an open ASDF object.

>   **Parameters**

>   >   **asdf_obj**
>   >   >   [obj] ASDF or ASDF-in-FITS object.

>   >   **idx**
>   >   >   [None] Currently unused. Reserved for future use to identify specific data set of an ASDF object containing the data of interest.

>   >   **logger**
>   >   >   [python logging object] Currently unused. Reserved for future use in logging

>   **Returns**

>   >   **data**
>   >   >   [ndarray or None] Image data, if found.

>   >   **wcs**
>   >   >   [obj or None] GWCS object or models, if found.

>   >   **ahdr**
>   >   >   [dict] Header containing metadata.

load_from_asdf

ginga.util.io.io_asdf.**load_from_asdf**(*asdf_obj*, *data_key='data'*, *wcs_key='wcs'*, *header_key='meta'*)

>   Load from an ASDF object. This method is primarily used internally to extract the correct parts of an ASDF file to reconstruct an image with WCS and metadata.

>   **Parameters**

>   >   **asdf_obj**
>   >   >   [obj] ASDF or ASDF-in-FITS object.

>   >   **data_key, wcs_key, header_key**
>   >   >   [str] Key values to specify where to find data, WCS, and header in ASDF.

>   **Returns**

data
[ndarray or None] Image data, if found.

wcs
[obj or None] GWCS object or models, if found.

ahdr
[dict] Header containing metadata.

8.12.2 Classes

ASDFFileHandler(logger)	For loading ASDF data files.

ASDFFileHandler

class ginga.util.io.io_asdf.**ASDFFileHandler**(*logger*)

Bases: BaseIOHandler

For loading ASDF data files.

Attributes Summary

mimetypes
name

Methods Summary

check_availability()	
close()	
load_asdf(asdf_f[, idx])	
load_asdf_hdu_in_fits(fits_f, hdu, **kwargs)	*** This is a special method that should only be called from WITHIN io_fits.py to open up ASDF-embedded-in-FITS ***
load_file(filepath[, idx])	Load a single data object from a file.
load_idx(idx, **kwargs)	**Parameters**
load_idx_cont(idx_spec, loader_cont_fn, **kwargs)	**Parameters**
open_file(filepath, **kwargs)	

Attributes Documentation

mimetypes = ['image/asdf']

name = 'asdf'

Methods Documentation

classmethod check_availability()

close()

load_asdf(*asdf_f*, *idx=None*, ***kwargs*)

load_asdf_hdu_in_fits(*fits_f*, *hdu*, ***kwargs*)

> *** This is a special method that should only be called from WITHIN io_fits.py to open up ASDF-embedded-in-FITS ***

load_file(*filepath*, *idx=None*, ***kwargs*)

> Load a single data object from a file.
>
> > **Parameters**
> >
> > > **idx**
> > >
> > > > [:py:Object] A Python value that matches describes the path or index to a single data data object in the file
> > >
> > > **kwargs**
> > >
> > > > [optional keyword arguments] Any optional keyword arguments are passed to the code that loads the data from the file
> >
> > **Returns**
> >
> > > **data_obj**
> > >
> > > > [subclass of `ViewerObjectBase`] A supported data wrapper object for a Ginga viewer

load_idx(*idx*, ***kwargs*)

> > **Parameters**
> >
> > > **idx**
> > >
> > > > [:py:class:object or None] A Python value that matches describes the path or index to a single data data object in the file. Can be None to indicate opening the default (or first usable) object. Acceptable formats are defined by the subclass.
> > >
> > > **kwargs**
> > >
> > > > [optional keyword arguments] Any optional keyword arguments are passed to the code that loads the data from the file
> >
> > **Returns**
> >
> > > **data_obj**
> > >
> > > > [subclass of `ViewerObjectBase`] A supported data wrapper object for a Ginga viewer

load_idx_cont(*idx_spec*, *loader_cont_fn*, ***kwargs*)

> > **Parameters**
> >
> > > **idx_spec**
> > >
> > > > [str] A string in the form of a pair of brackets enclosing some index expression matching data objects in the file

loader_cont_fn
[func (data_obj) -> None] A loader continuation function that takes a data object generated from loading an HDU and does something with it

kwargs
[optional keyword arguments] Any optional keyword arguments are passed to the code that loads the data from the file

open_file(*filepath*, ***kwargs*)

8.12.3 Class Inheritance Diagram
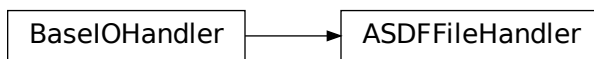
8.13 ginga.util.wcsmod Package

We are fortunate to have several possible choices for a python WCS package compatible with Ginga: astlib, kapteyn, starlink and astropy. kapteyn and astropy wrap Mark Calabretta's "WCSLIB", astLib wraps Jessica Mink's "wcstools", and I'm not sure what starlink uses (their own?).

Note that astlib requires pyfits (or astropy) in order to create a WCS object from a FITS header.

To force the use of one, do:

```
from ginga.util import wcsmod
wcsmod.use('kapteyn')
```

before you load any images. Otherwise Ginga will try to pick one for you.

Note that you can register custom WCS types using:

```
from ginga.util.wcsmod.common import register_wcs
register_wcs('mywcs', MyWCSClass, list_of_coord_types)
```

Look at the implemented WCS wrappers for details.

8.13.1 Functions

get_wcs_class(name)	Get a WCS class corresponding to the registered name.

get_wcs_class

ginga.util.wcsmod.**get_wcs_class**(*name*)

> Get a WCS class corresponding to the registered name. Will raise a KeyError if a class of the given name does not exist.

8.14 ginga.util.wcs Module

This module handles calculations based on world coordinate system.

8.14.1 Functions

hmsToDeg(h, m, s)	Convert RA hours, minutes, seconds into an angle in degrees.
dmsToDeg(sign, deg, min, sec)	Convert dec sign, degrees, minutes, seconds into a signed angle in degrees.
decTimeToDeg(sign_sym, deg, min, sec)	Convert dec sign, degrees, minutes, seconds into a signed angle in degrees.
degToHms(ra)	Converts the ra (in degrees) to HMS three tuple.
degToDms(dec[, isLatitude])	Convert the dec, in degrees, to an (sign,D,M,S) tuple.
arcsecToDeg(arcsec)	Convert numeric arcseconds (aka DMS seconds) to degrees of arc.
hmsStrToDeg(ra)	Convert a string representation of RA into a float in degrees.
dmsStrToDeg(dec)	Convert a string representation of DEC into a float in degrees.
trans_coeff(eq, x, y, z)	This function is provided by MOKA2 Development Team (1996.xx.xx) and used in SOSS system.
eqToEq2000(ra_deg, dec_deg, eq)	Convert Eq to Eq 2000.
get_xy_rotation_and_scale(header)	CREDIT: See IDL code at http://www.astro.washington.edu/docs/idl/cgi-bin/getpro/library32.html?GETROT
get_rotation_and_scale(header[, skew_threshold])	Calculate rotation and CDELT.
get_relative_orientation(image, ref_image)	Computes the relative orientation and scale of an image to a reference image.
simple_wcs(px_x, px_y, ra_deg, dec_deg, ...)	Calculate a set of WCS keywords for a 2D simple instrument FITS file with a 'standard' RA/DEC pixel projection.
deg2fmt(ra_deg, dec_deg, format)	Format coordinates.
dispos(dra0, decd0, dra, decd)	Compute distance and position angle solving a spherical triangle (no approximations).
deltaStarsRaDecDeg1(ra1_deg, dec1_deg, ...)	Spherical triangulation.
deltaStarsRaDecDeg2(ra1_deg, dec1_deg, ...)	
get_starsep_RaDecDeg(ra1_deg, dec1_deg, ...)	Calculate separation.
add_offset_radec(ra_deg, dec_deg, ...)	Algorithm to compute RA/Dec from RA/Dec base position plus tangent plane offsets.
get_RaDecOffsets(ra1_deg, dec1_deg, ra2_deg, ...)	Calculate offset.
lon_to_deg(lon)	Convert longitude to degrees.
lat_to_deg(lat)	Convert latitude to degrees.
ra_deg_to_str(ra_deg[, precision, format])	
dec_deg_to_str(dec_deg[, precision, format])	

hmsToDeg

ginga.util.wcs.**hmsToDeg**(*h*, *m*, *s*)

> Convert RA hours, minutes, seconds into an angle in degrees.

dmsToDeg

ginga.util.wcs.**dmsToDeg**(*sign*, *deg*, *min*, *sec*)

> Convert dec sign, degrees, minutes, seconds into a signed angle in degrees.

decTimeToDeg

ginga.util.wcs.**decTimeToDeg**(*sign_sym*, *deg*, *min*, *sec*)

> Convert dec sign, degrees, minutes, seconds into a signed angle in degrees.
>
> `sign_sym` may represent negative as either '-' or numeric -1.

degToHms

ginga.util.wcs.**degToHms**(*ra*)

> Converts the ra (in degrees) to HMS three tuple. H and M are in integer and the S part is in float.

degToDms

ginga.util.wcs.**degToDms**(*dec*, *isLatitude=True*)

> Convert the dec, in degrees, to an (sign,D,M,S) tuple. D and M are integer, and sign and S are float.

arcsecToDeg

ginga.util.wcs.**arcsecToDeg**(*arcsec*)

> Convert numeric arcseconds (aka DMS seconds) to degrees of arc.

hmsStrToDeg

ginga.util.wcs.**hmsStrToDeg**(*ra*)

> Convert a string representation of RA into a float in degrees.

dmsStrToDeg

ginga.util.wcs.**dmsStrToDeg**(*dec*)

> Convert a string representation of DEC into a float in degrees.

trans_coeff

ginga.util.wcs.**trans_coeff**(*eq*, *x*, *y*, *z*)

> This function is provided by MOKA2 Development Team (1996.xx.xx) and used in SOSS system.

eqToEq2000

ginga.util.wcs.**eqToEq2000**(*ra_deg*, *dec_deg*, *eq*)

> Convert Eq to Eq 2000.

get_xy_rotation_and_scale

ginga.util.wcs.**get_xy_rotation_and_scale**(*header*)

> CREDIT: See IDL code at http://www.astro.washington.edu/docs/idl/cgi-bin/getpro/library32.html?GETROT

get_rotation_and_scale

ginga.util.wcs.**get_rotation_and_scale**(*header*, *skew_threshold=0.001*)

> Calculate rotation and CDELT.

get_relative_orientation

ginga.util.wcs.**get_relative_orientation**(*image*, *ref_image*)

> Computes the relative orientation and scale of an image to a reference image.
>
> > **Parameters**
> >
> > > **image**
> > > > AstroImage based object
> > >
> > > **ref_image**
> > > > AstroImage based object
> >
> > **Returns**
> >
> > > **result**
> > > > Bunch object containing the relative scale in x and y and the relative rotation in degrees.

simple_wcs

ginga.util.wcs.**simple_wcs**(*px_x*, *px_y*, *ra_deg*, *dec_deg*, *px_scale_deg_px*, *rot_deg*, *cdbase=[1, 1]*)

> Calculate a set of WCS keywords for a 2D simple instrument FITS file with a 'standard' RA/DEC pixel projection.
>
> > **Parameters**
> >
> > > **px_x**
> > > > (ZERO-based) reference pixel of field in X (usually center of field)
> > >
> > > **px_y**
> > > > (ZERO-based) reference pixel of field in Y (usually center of field)
> > >
> > > **ra_deg**
> > > > RA (in deg) for the reference point

dec_deg
    DEC (in deg) for the reference point

px_scale_deg_px
    Pixel scale (deg/pixel); can be a tuple for different x,y scales

rot_deg
    Rotation angle of the field (in deg)

cdbase
    CD base

Returns

res
    [dict] Ordered dictionary object containing WCS headers.

deg2fmt

ginga.util.wcs.**deg2fmt**(*ra_deg*, *dec_deg*, *format*)

    Format coordinates.

dispos

ginga.util.wcs.**dispos**(*dra0*, *decd0*, *dra*, *decd*)

    Compute distance and position angle solving a spherical triangle (no approximations).

    Source/credit: Skycat Author: A.P. Martinez

Parameters

dra0
    [float] Center RA in decimal degrees.

decd0
    [float] Center DEC in decimal degrees.

dra
    [float] Point RA in decimal degrees.

decd
    [float] Point DEC in decimal degrees.

Returns

phi
    [float] Phi in degrees (East of North).

dist
    [float] Distance in arcmin.

deltaStarsRaDecDeg1

ginga.util.wcs.**deltaStarsRaDecDeg1**(*ra1_deg*, *dec1_deg*, *ra2_deg*, *dec2_deg*)

    Spherical triangulation.

deltaStarsRaDecDeg2

ginga.util.wcs.**deltaStarsRaDecDeg2**(*ra1_deg*, *dec1_deg*, *ra2_deg*, *dec2_deg*)

get_starsep_RaDecDeg

ginga.util.wcs.**get_starsep_RaDecDeg**(*ra1_deg*, *dec1_deg*, *ra2_deg*, *dec2_deg*)

    Calculate separation.

add_offset_radec

ginga.util.wcs.**add_offset_radec**(*ra_deg*, *dec_deg*, *delta_deg_ra*, *delta_deg_dec*)

    Algorithm to compute RA/Dec from RA/Dec base position plus tangent plane offsets.

get_RaDecOffsets

ginga.util.wcs.**get_RaDecOffsets**(*ra1_deg*, *dec1_deg*, *ra2_deg*, *dec2_deg*)

    Calculate offset.

lon_to_deg

ginga.util.wcs.**lon_to_deg**(*lon*)

    Convert longitude to degrees.

lat_to_deg

ginga.util.wcs.**lat_to_deg**(*lat*)

    Convert latitude to degrees.

ra_deg_to_str

ginga.util.wcs.**ra_deg_to_str**(*ra_deg*, *precision=3*, *format='%02d:%02d:%02d.%03d'*)

dec_deg_to_str

ginga.util.wcs.**dec_deg_to_str**(*dec_deg*, *precision=2*, *format='%s%02d:%02d:%02d.%02d'*)

8.15 ginga.util.ap_region Module

This module provides Ginga support for DS9 type region files and objects via the `astropy-regions` package.

8.15.1 Functions

astropy_region_to_ginga_canvas_object(r[, ...])	Convert an astropy-region object to a Ginga canvas object.
add_region(canvas, r[, tag, redraw])	Convenience function to plot an astropy-regions object on a Ginga canvas.
ginga_canvas_object_to_astropy_region(obj[, ...])	Convert a Ginga canvas object to an astropy-region object.

astropy_region_to_ginga_canvas_object

ginga.util.ap_region.**astropy_region_to_ginga_canvas_object**(*r*, *logger=None*)

>   Convert an astropy-region object to a Ginga canvas object.

>   >   **Parameters**

>   >   >   **r**

>   >   >   >   [subclass of `PixelRegion`] The region object to be converted

>   >   >   **logger**

>   >   >   >   [a Python logger (optional, default: None)] A logger to which errors will be written

>   >   **Returns**

>   >   >   **obj**

>   >   >   >   [subclass of *CanvasObject*] The corresponding Ginga canvas object

add_region

ginga.util.ap_region.**add_region**(*canvas*, *r*, *tag=None*, *redraw=True*)

>   Convenience function to plot an astropy-regions object on a Ginga canvas.

>   >   **Parameters**

>   >   >   **canvas**

>   >   >   >   [*DrawingCanvas*] The Ginga canvas on which the region should be plotted.

>   >   >   **r**

>   >   >   >   [subclass of `PixelRegion`] The region object to be plotted

>   >   >   **tag**

>   >   >   >   [str or None (optional, default: None)] Caller can optionally pass a specific tag for the canvas object

redraw
    [bool (optional, default: True)] True if the viewers of the canvas should be updated immediately

ginga_canvas_object_to_astropy_region

ginga.util.ap_region.**ginga_canvas_object_to_astropy_region**(*obj*, *frame='icrs'*, *logger=None*)

    Convert a Ginga canvas object to an astropy-region object.

    Parameters

        obj
            [subclass of `CanvasObjectBase`] The Ginga canvas object to be converted

        frame
            [str (optional, default: 'icrs')] The type of astropy frame that should be generated for Sky regions

        logger
            [a Python logger (optional, default: None)] A logger to which errors will be written

    Returns

        r
            [subclass of `PixelRegion` or `SkyRegion`] The corresponding astropy-region object

8.16 ginga.util.iqcalc Module

Module to handle image quality calculations.

8.16.1 Functions

get_mean(data_np)	Calculate mean for valid values.
get_median(data_np)	Like *get_mean()* but for median.

get_mean

ginga.util.iqcalc.**get_mean**(*data_np*)

    Calculate mean for valid values.

    Parameters

        data_np
            [ndarray] Input array. Can contain masked values.

    Returns

        result
            [float] Mean of array values that are finite. If array contains no finite values, returns NaN.

get_median

ginga.util.iqcalc.**get_median**(*data_np*)

>   Like *get_mean()* but for median.

8.16.2 Classes

| *IQCalcError* | Base exception for raising errors in this module. |
| *IQCalc*([logger]) | Class to handle model fitting and FWHM calculations. |

IQCalcError

exception ginga.util.iqcalc.**IQCalcError**

>   Base exception for raising errors in this module.

IQCalc

class ginga.util.iqcalc.**IQCalc**(*logger=None*)

>   Bases: object

>   Class to handle model fitting and FWHM calculations.

>   >   **Parameters**

>   >   >   **logger**
>   >   >   >   [obj or None] Python logger. If not given, one will be created.

>   >   **Attributes**

>   >   >   **lock**
>   >   >   >   [threading.RLock] For mutex around scipy.optimize, which seems to be non-threadsafe.

>   >   >   **skylevel_magnification, skylevel_offset**
>   >   >   >   [float] For adjustments to sky background level.

Methods Summary

brightness(x, y, radius, medv, data)	Return the brightness value found in a region defined by input location and radius.
calc_fwhm(arr1d[, medv, method_name])	Calculate FWHM for the given input array.
calc_fwhm_gaussian(arr1d[, medv, gauss_fn])	FWHM calculation on a 1D array by using least square fitting of a Gaussian function on the data.
calc_fwhm_moffat(arr1d[, medv, moffat_fn])	FWHM calculation on a 1D array by using least square fitting of a Moffat function on the data.
centroid(data, xc, yc, radius)	Calculate centroid from center of mass.
cut_cross(x, y, radius, data)	Cut data vertically and horizontally at the given position with the given radius.
cut_region(x, y, radius, data)	Return a cut region.
encircled_energy(data)	Return a function of encircled energy across pixel indices.
ensquared_energy(data)	Return a function of ensquared energy across pixel indices.
evaluate_peaks(peaks, data[, fwhm_radius, ...])	Evaluate photometry for given peaks in data array.
find_bright_peaks(data[, threshold, sigma, ...])	Find bright peak candidates in in the given data.
fwhm_data(x, y, data[, radius, method_name])	Equivalent to *get_fwhm()*.
gaussian(x, p)	Evaluate Gaussian function in 1D.
get_fwhm(x, y, radius, data[, medv, method_name])	Get the FWHM values of the object at the given coordinates and radius.
get_threshold(data[, sigma])	Calculate threshold for *find_bright_peaks()*.
moffat(x, p)	Evaluate Moffat function in 1D.
objlist_select(objlist, width, height[, ...])	Filter output from *evaluate_peaks()*.
pick_field(data[, peak_radius, fwhm_radius, ...])	Pick the first good object within the given field.
qualsize(image[, x1, y1, x2, y2, radius, ...])	Run *pick_field()* on the given image.
starsize(fwhm_x, deg_pix_x, fwhm_y, deg_pix_y)	Calculate average FWHM in arcseconds.

Methods Documentation

brightness(*x*, *y*, *radius*, *medv*, *data*)

Return the brightness value found in a region defined by input location and radius. Region is cut using *cut_region()*.

> **Parameters**
>
> > **x, y**
> > [int] Indices of central pixel.
> >
> > **radius**
> > [int] Half-width in both X and Y directions.
> >
> > **medv**
> > [float] Background to subtract off.
> >
> > **data**
> > [array-like] Data array.
>
> **Returns**
>
> > **res**
> > [float] Brightness.

`calc_fwhm`(*arr1d*, *medv=None*, *method_name='gaussian'*)

Calculate FWHM for the given input array.

Parameters

arr1d

[array-like] 1D array cut in either X or Y direction on the object.

medv

[float or None] Median of the data. If not given, it is calculated from `arr1d`.

method_name

[{'gaussian', 'moffat'}] Function to use for fitting.

Returns

res

[Bunch] Fitting results.

`calc_fwhm_gaussian`(*arr1d*, *medv=None*, *gauss_fn=None*)

FWHM calculation on a 1D array by using least square fitting of a Gaussian function on the data.

Parameters

arr1d

[array-like] 1D array cut in either X or Y direction on the object.

medv

[float or None] Median of the data. If not given, it is calculated from `arr1d`.

gauss_fn

[func or None] Gaussian function for fitting. If not given, *gaussian()* is used.

Returns

res

[Bunch] Fitting results.

Raises

IQCalcError

Fitting failed.

`calc_fwhm_moffat`(*arr1d*, *medv=None*, *moffat_fn=None*)

FWHM calculation on a 1D array by using least square fitting of a Moffat function on the data.

Parameters

arr1d

[array-like] 1D array cut in either X or Y direction on the object.

medv

[float or None] Median of the data. If not given, it is calculated from `arr1d`.

moffat_fn

[func or None] Moffat function for fitting. If not given, *moffat()* is used.

Returns

res

[Bunch] Fitting results.

Raises

> **IQCalcError**
>> Fitting failed.

centroid(*data*, *xc*, *yc*, *radius*)

> Calculate centroid from center of mass.

>> **Parameters**

>>> **data**
>>>> [array-like] Data array.

>>> **xc, yc**
>>>> [int] X and Y indices of the approximate center.

>>> **radius**
>>>> [float] Half-width of the region to consider around the given center.

>> **Returns**

>>> **x, y**
>>>> [float] Centroid indices.

>> **Raises**

>>> **IQCalcError**
>>>> Missing dependency.

cut_cross(*x*, *y*, *radius*, *data*)

> Cut data vertically and horizontally at the given position with the given radius.

>> **Parameters**

>>> **x, y**
>>>> [int] Indices where vertical and horizontal cuts meet.

>>> **radius**
>>>> [float] Radius of both cuts.

>>> **data**
>>>> [array-like] Data array to cut from.

>> **Returns**

>>> **x0**
>>>> [array-like] Starting pixel of horizontal cut (in X).

>>> **y0**
>>>> [array-like] Starting pixel of vertical cut (in Y).

>>> **xarr**
>>>> [array-like] Horizontal cut (in X).

>>> **yarr**
>>>> [array-like] Vertical cut (in Y).

cut_region(*x*, *y*, *radius*, *data*)

> Return a cut region.

>> **Parameters**

>>> **x, y**
>>>> [int] Indices of central pixel.

>>> **radius**
>>>> [int] Half-width in both X and Y directions.

data
    [array-like] Data array to cut from.

Returns

x0, y0
    [int] Origin of the region.

arr
    [array-like] Cut region (a view, not copy).

encircled_energy(*data*)

    Return a function of encircled energy across pixel indices.

    Ideally, data is already a masked array and is assumed to be centered.

ensquared_energy(*data*)

    Return a function of ensquared energy across pixel indices.

    Ideally, data is already a masked array and is assumed to be centered.

evaluate_peaks(*peaks*, *data*, *fwhm_radius=15*, *fwhm_method='gaussian'*, *ee_total_radius=10*,
            cb_fn=None, *ev_intr=None*)

    Evaluate photometry for given peaks in data array.

    Parameters

    peaks
        [list of tuple] List of `(x, y)` tuples containing indices of peaks.

    data
        [array-like] Data array that goes with the given peaks.

    fwhm_radius, fwhm_method
        See *get_fwhm()*.

    ee_total_radius
        [float] Radius, in pixels, where encircled and ensquared energy fractions are defined as 1.

    cb_fn
        [func or None] If applicable, provide a callback function that takes a `ginga.misc.Bunch.`
        `Bunch` containing the result for each peak. It should not return anything.

    ev_intr
        [`threading.Event` or None] For threading, if applicable.

    .. note:: unused parameter `bright_radius` was removed in
        release 4.0

    Returns

    objlist
        [list of `ginga.misc.Bunch.Bunch`] A list of successful results for the given peaks. Each
        result contains the following keys:

        • `objx`, `objy`: Fitted centroid from *get_fwhm()*.

        • `pos`: A measure of distance from the center of the image.

        • `oid_x`, `oid_y`: Center-of-mass centroid from *centroid()*.

        • `fwhm_x`, `fwhm_y`: Fitted FWHM from *get_fwhm()*.

        • `fwhm`: Overall measure of fwhm as a single value.

- `fwhm_radius`: Input FWHM radius.

- `brightness`: Average peak value based on *get_fwhm()* fits.

- `elipse`: A measure of ellipticity.

- `x`, `y`: Input indices of the peak.

- `skylevel`: Sky level estimated from median of data array and `skylevel_magnification` and `skylevel_offset` attributes.

- `background`: Median of the input array.

- `ensquared_energy_fn`: Function of ensquared energy for different pixel radii.

- `encircled_energy_fn`: Function of encircled energy for different pixel radii.

find_bright_peaks(*data*, *threshold=None*, *sigma=5*, *radius=5*)

> Find bright peak candidates in in the given data.

> > **Parameters**

> > > **data**
> > > [array-like] Input data to find peaks from.

> > > **threshold**
> > > [float or `None`] Detection threshold. Below this value, an object is not considered a candidate. If not given, a default is calculated using *get_threshold()* with the given `sigma`.

> > > **sigma**
> > > [float] Sigma for the threshold.

> > > **radius**
> > > [float] Pixel radius for determining local maxima. If the desired objects are larger in size, specify a larger radius.

> > **Returns**

> > > **peaks**
> > > [list of tuple] A list of candidate object coordinate tuples (`x`, `y`) in data.

fwhm_data(*x*, *y*, *data*, *radius=15*, *method_name='gaussian'*)

> Equivalent to *get_fwhm()*.

gaussian(*x*, *p*)

> Evaluate Gaussian function in 1D. See *calc_fwhm()*.

> > **Parameters**

> > > **x**
> > > [array-like] X values.

> > > **p**
> > > [tuple of float] Parameters for Gaussian, i.e., (`mean`, `stddev`, `amplitude`).

> > **Returns**

> > > **y**
> > > [array-like] Y values.

get_fwhm(*x*, *y*, *radius*, *data*, *medv=None*, *method_name='gaussian'*)

> Get the FWHM values of the object at the given coordinates and radius.

> > **Parameters**

x, y
  [int] Indices of the object location in data array.

radius
  [float] Radius of the region encompassing the object.

data
  [array-like] Data array.

medv, method_name
  See *calc_fwhm()*.

Returns

fwhm_x, fwhm_y
  [float] FWHM in X and Y, respectively.

ctr_x, ctr_y
  [float] Center in X and Y, respectively.

x_res, y_res
  [dict] Fit results from *calc_fwhm()* in X and Y, respectively.

get_threshold(*data*, *sigma=5.0*)
  Calculate threshold for *find_bright_peaks()*.

  Parameters

  data
    [array-like] Data array.

  sigma
    [float] Sigma for the threshold.

  Returns

  threshold
    [float] Threshold based on good data, its median, and the given sigma.

moffat(*x*, *p*)
  Evaluate Moffat function in 1D. See *calc_fwhm()*.

  Parameters

  x
    [array-like] X values.

  p
    [tuple of float] Parameters for Moffat, i.e., (`x_0`, `gamma`, `alpha`, `amplitude`), where `x_0` a.k.a. mean and `gamma` core width.

  Returns

  y
    [array-like] Y values.

objlist_select(*objlist*, *width*, *height*, *minfwhm=2.0*, *maxfwhm=150.0*, *minelipse=0.5*, *edgew=0.01*)
  Filter output from *evaluate_peaks()*.

  Parameters

  objlist
    [list of `ginga.misc.Bunch.Bunch`] Output from *evaluate_peaks()*.

> **width, height**
> > [int] Dimension of data array from which `objlist` was derived.
>
> **minfwhm, maxfwhm**
> > [float] Limits for desired FWHM, where (`minfwhm, maxfwhm`).
>
> **minelipse**
> > [float] Minimum value of desired ellipticity (not inclusive).
>
> **edgew**
> > [float] Factor between 0 and 1 that determines if a location is too close to the edge or not.

> **Returns**
>
> > **results**
> > > [list of `ginga.misc.Bunch.Bunch`] Elements of `objlist` that contain desired FWHM, ellipticity, and not too close to the edge.

`pick_field`(*data*, *peak_radius=5*, *fwhm_radius=15*, *threshold=None*, *minfwhm=2.0*, *maxfwhm=50.0*, *minelipse=0.5*, *edgew=0.01*, *ee_total_radius=10*)

> Pick the first good object within the given field.

> **Parameters**
>
> > **data**
> > > [array-like] Data array of the field.
> >
> > **peak_radius, threshold**
> > > See *find_bright_peaks()*.
> >
> > **fwhm_radius, ee_total_radius**
> > > See *evaluate_peaks()*.
> >
> > **minfwhm, maxfwhm, minelipse, edgew**
> > > See *objlist_select()*.
> >
> > **.. note:: unused parameter `bright_radius` was removed in**
> > > release 4.0

> **Returns**
>
> > **result**
> > > [`ginga.misc.Bunch.Bunch`] This is a single element of `objlist` as described in *evaluate_peaks()*.

> **Raises**
>
> > **IQCalcError**
> > > No object matches selection criteria.

`qualsize`(*image*, *x1=None*, *y1=None*, *x2=None*, *y2=None*, *radius=5*, *fwhm_radius=15*, *threshold=None*, *minfwhm=2.0*, *maxfwhm=50.0*, *minelipse=0.5*, *edgew=0.01*, *ee_total_radius=10*)

> Run *pick_field()* on the given image.

> **Parameters**
>
> > **image**
> > > [`ginga.AstroImage.AstroImage`] Image to process.
> >
> > **x1, y1, x2, y2**
> > > [int] See `ginga.BaseImage.BaseImage.cutout_data()`.
> >
> > **radius, threshold**
> > > See *find_bright_peaks()*.

> **fwhm_radius, ee_total_radius**
>> See *evaluate_peaks()*.
>
> **minfwhm, maxfwhm, minelipse, edgew**
>> See *objlist_select()*.
>
> **.. note:: unused parameter `bright_radius` was removed in**
>> release 4.0
>
> **Returns**
>
> **qs**
>> [ginga.misc.Bunch.Bunch] This is a single element of `objlist` as described in
>> *evaluate_peaks()*.

starsize(*fwhm_x*, *deg_pix_x*, *fwhm_y*, *deg_pix_y*)

> Calculate average FWHM in arcseconds.
>
> **Parameters**
>
> **fwhm_x**
>> [float] FWHM in X (pixels).
>
> **deg_pix_x**
>> [float] Plate scale from CDELT1 in degrees per pixel.
>
> **fwhm_y**
>> [float] FWHM in Y (pixels).
>
> **deg_pix_y**
>> [float] Plate scale from CDELT2 in degrees per pixel.
>
> **Returns**
>
> **fwhm**
>> [float] Average FWHM in arcseconds.

8.17 ginga.util.iqcalc_astropy Module

Module to handle image quality calculations using `astropy`.

8.17.1 Classes

IQCalc(*args, **kwargs)	This is *ginga.util.iqcalc.IQCalc* that uses `astropy`.

IQCalc

class ginga.util.iqcalc_astropy.**IQCalc**(**args*, ***kwargs*)

Bases: *IQCalc*

This is *ginga.util.iqcalc.IQCalc* that uses `astropy`.

This subclass has an extra `self.fitter` attribute for `astropy` fitting.

Methods Summary

calc_fwhm(arr1d[, medv, method_name])	Calculate FWHM for the given input array.
calc_fwhm_gaussian(arr1d[, medv])	FWHM calculation on a 1D array by using least square fitting of a Gaussian function on the data.
calc_fwhm_lorentz(arr1d[, medv])	FWHM calculation on a 1D array by using least square fitting of a Lorentz function on the data.
calc_fwhm_moffat(arr1d[, medv])	FWHM calculation on a 1D array by using least square fitting of a Moffat function on the data.
centroid(data, xc, yc, radius)	Calculate centroid from center of mass.
find_bright_peaks(data[, threshold, sigma, ...])	Find bright peak candidates in in the given data.
gaussian(x, p)	Evaluate Gaussian function in 1D.
lorentz(x, p)	Evaluate Lorentz function in 1D.
moffat(x, p)	Evaluate Moffat function in 1D.

Methods Documentation

calc_fwhm(*arr1d*, *medv=None*, *method_name='gaussian'*)

Calculate FWHM for the given input array.

> **Parameters**
>
> > **arr1d**
> > [array-like] 1D array cut in either X or Y direction on the object.
> >
> > **medv**
> > [float or None] Median of the data. If not given, it is calculated from `arr1d`.
> >
> > **method_name**
> > [{'gaussian', 'moffat', 'lorentz'}] Function to use for fitting.
>
> **Returns**
>
> > **res**
> > [Bunch] Fitting results.
>
> **Raises**
>
> > **NotImplementedError**
> > Given function is not supported.

calc_fwhm_gaussian(*arr1d*, *medv=None*, ***kwargs*)

FWHM calculation on a 1D array by using least square fitting of a Gaussian function on the data.

> **Parameters**
>
> > **arr1d**
> > [array-like] 1D array cut in either X or Y direction on the object.

> **medv**
> [float or None] Median of the data. If not given, it is calculated from `arr1d`.

> **kwargs**
> [dict] Not used; for backward-compatible API call only.

Returns

> **res**
> [Bunch] Fitting results.

Raises

> **IQCalcError**
> Fitting failed.

`calc_fwhm_lorentz`(*arr1d*, *medv=None*, ***kwargs*)

> FWHM calculation on a 1D array by using least square fitting of a Lorentz function on the data.

> **Parameters**

> > **arr1d**
> > [array-like] 1D array cut in either X or Y direction on the object.

> > **medv**
> > [float or None] Median of the data. If not given, it is calculated from `arr1d`.

> > **kwargs**
> > [dict] Not used; for backward-compatible API call only.

> **Returns**

> > **res**
> > [Bunch] Fitting results.

> **Raises**

> > **IQCalcError**
> > Fitting failed.

`calc_fwhm_moffat`(*arr1d*, *medv=None*, ***kwargs*)

> FWHM calculation on a 1D array by using least square fitting of a Moffat function on the data.

> **Parameters**

> > **arr1d**
> > [array-like] 1D array cut in either X or Y direction on the object.

> > **medv**
> > [float or None] Median of the data. If not given, it is calculated from `arr1d`.

> > **kwargs**
> > [dict] Not used; for backward-compatible API call only.

> **Returns**

> > **res**
> > [Bunch] Fitting results.

> **Raises**

> > **IQCalcError**
> > Fitting failed.

centroid(*data*, *xc*, *yc*, *radius*)

　Calculate centroid from center of mass.

Parameters

data
　[array-like] Data array.

xc, yc
　[int] X and Y indices of the approximate center.

radius
　[float] Half-width of the region to consider around the given center.

Returns

x, y
　[float] Centroid indices.

Raises

IQCalcError
　Missing dependency.

find_bright_peaks(*data*, *threshold=None*, *sigma=5*, *radius=5*)

　Find bright peak candidates in in the given data.

Parameters

data
　[array-like] Input data to find peaks from.

threshold
　[float or None] Detection threshold. Below this value, an object is not considered a candidate. If not given, a default is calculated using `get_threshold()` with the given `sigma`.

sigma
　[float] Sigma for the threshold.

radius
　[float] Pixel radius for determining local maxima. If the desired objects are larger in size, specify a larger radius.

Returns

peaks
　[list of tuple] A list of candidate object coordinate tuples `(x, y)` in data.

gaussian(*x*, *p*)

　Evaluate Gaussian function in 1D.

Parameters

x
　[array-like] X values.

p
　[tuple of float] Parameters for Gaussian, i.e., (`mean`, `stddev`, `amplitude`).

Returns

y
　[array-like] Y values.

lorentz(x, p)

Evaluate Lorentz function in 1D.

Parameters

x
[array-like] X values.

p
[tuple of float] Parameters for Lorentz, i.e., (`x_0, fwhm, amplitude`).

Returns

y
[array-like] Y values.

moffat(x, p)

Evaluate Moffat function in 1D.

Parameters

x
[array-like] X values.

p
[tuple of float] Parameters for Moffat, i.e., (`x_0, gamma, alpha, amplitude`), where `x_0` a.k.a. mean and `gamma` core width.

Returns

y
[array-like] Y values.

Some training videos are available in the downloads page on Github.

Be sure to also check out the Ginga wiki.

Part VII

Bug Reports

Please file an issue with the issue tracker on Github.

Ginga has a logging facility, and it would be most helpful if you can invoke Ginga with the logging options to capture any logged errors:

```
ginga --loglevel=20 --log=ginga.log --stderr
```

If the difficulty is with non-display or non-working World Coordinate System (WCS) for a particular image file please be ready to supply the file for our aid in debugging.

Part VIII

Developer Info

In the source code `examples/*` directories, see `example{1,2}_gtk.py` (Gtk), `example{1,2}_qt.py` (Qt), `example{1,2}_tk.py` (Tk), or `example{1,2,3,4,5}_mpl.py` (Matplotlib). There is more information for developers in the *Developer's Manual*.

See also the *Reference/API* or Module Index for a list of the available modules.

Part IX

Etymology

"Ginga" is the romanized spelling of the Japanese word "NN" (Hiragana: NNN), meaning "galaxy" (in general) and, more familiarly, the Milky Way. This viewer was written by software engineers at Subaru Telescope, National Astronomical Observatory of Japan, thus the connection.

Part X

Pronunciation

Ginga the viewer may be pronounced "ging-ga" (proper Japanese) or "jing-ga" (perhaps easier for Westerners). The latter pronunciation has meaning in the Brazilian dance/martial art Capoeira: a fundamental rocking or back and forth swinging motion. Pronunciation as "jin-ja" is considered poor form.

PYTHON MODULE INDEX

N

O

P

S