Ginga: an open-source astronomical image viewer and toolkit

Eric Jeschke‡*

http://www.youtube.com/watch?v=n2KvYUUXCs

Abstract—Ginga is a new astronomical image viewer written in Python. It uses and inter-operates with several key scientific Python packages: NumPy, Astropy, and SciPy. A key differentiator for this image viewer, compared to older-generation FITS viewers, is that all the key components are written as Python classes, allowing for the first time a powerful FITS image display widget to be directly embedded in, and tightly coupled with, Python code.

We call Ginga a toolkit for programming FITS viewers because it includes a choice of base classes for programming custom viewers for two different modern widget sets: Gtk and Qt, available on the three common desktop platforms. In addition, a reference viewer is included with the source code based on a plugin architecture in which the viewer can be extended with plugins scripted in Python. The code is released under a BSD license similar to other major Python packages and is available on GitHub.

Ginga has been introduced only recently as a tool to the astronomical community, but since SciPy has a developer focus this talk concentrates on programming with the Ginga toolkit. We cover two cases: using the bare image widget to build custom viewers and writing plugins for the existing full-featured Ginga viewer. The talk may be of interest to anyone developing code in Python. We call Ginga a toolkit for programming FITS viewers because it includes a choice of base classes for programming custom viewers for two different modern widget sets: Gtk and Qt, available on the three common desktop platforms.

Users wishing to develop an imaging program employing Ginga can follow one of two logical development paths: starting from the reference viewer and building up around it, or starting from the reference viewer and customizing it via a plugin.

Getting and installing Ginga
Ginga is released under a BSD license similar to other major scientific Python packages and is available on GitHub: http://github.com/ejeschke/ginga. It is a distutils-compatible Python package, and is also available in PyPI. Installing it is as simple as:

```
pip install ginga
```
or:

```
python setup.py install
```

Use the latter if you have downloaded the latest source as a tarball from http://ejeschke.github.com/ginga or cloned the git repository from https://github.com/ejeschke/ginga.git. The package will be installed as "ginga" and the reference viewer will also be installed as `ginga` (but located wherever scripts are stored).

Prerequisites and dependences: Ginga will run under Python versions from 2.7 to 3.3. Note that as a minimum you will need to have at least installed `numpy` and one of the Python Gtk or Qt bindings (e.g. `pygtk`, `pyqt`). For full functionality you will also need `scipy` and `astropy` [Tol13]. Certain features in the reference viewer also be activated if `matplotlib` is installed.

Part 1: Developing with the Ginga Widget

When developing with the Ginga toolkit for visualizing FITS files there are two main starting points one might take:

- using only the Ginga widget itself, or
- starting with the full-featured reference viewer that comes with Ginga and customize it for some special purpose.

The first way is probably best for when the developer has a custom application in mind, needs a bare-bones viewer or wants

1. Flexible Image Transport System—the current standard for archiving and exchanging astronomical data as files.
to develop an entirely new full-featured viewer. The second way 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 easy to do. In this paper we address both of these approaches.

First, let’s take a look at how to use the "bare" Ginga FITS viewing widget by itself. The FitsImageZoom widget handles image display, scaling (zooming), panning, manual cut levels, auto cut levels with a choice of algorithms, color mapping, transformations, and rotation. Besides the image window itself there are no additional GUI (Graphical User Interface) components and these controls are handled programatically or directly by keyboard and mouse bindings on the window. Developers can enable as many of the features as they want, or reimplement them. The user interface bindings are configurable via a pluggable Bindings class, and there are a plethora of callbacks that can be registered, allowing the user to create their own custom user interface for manipulating the view.

Listing 1 shows a code listing for a simple graphical FITS viewer using this widget (screenshot in Figure 1) written in around 100 or so lines of Python. It creates a window containing an image view and two buttons. This example, included with the Ginga package, will open FITS files dragged and dropped on the image window or via a dialog popped up when clicking the "Open File" button.

Looking at the constructor for this particular viewer, you can see where we create a FitsImageZoom object. On this object we enable automatic cut levels (using the 'zscale' algorithm), auto zoom 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 panning, zooming, cut levels and simple transformations (flip x/y and swap axes). 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.

Listing 1:

```python
#!/usr/bin/env python
#
# example1_qt.py -- Simple, configurable FITS viewer.
#
import sys, os
import logging
from ginga.AstroImage import pyfits
from ginga.qtw.QtHelp import QtGui, QtCore
from ginga.qtw.FitsImageQt import FitsImageZoom

class FitsViewer(QtGui.QMainWindow):
    def __init__(self, logger):
        super(FitsViewer, self).__init__()
        self.logger = logger

        fi = FitsImageZoom(self.logger)
        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_setActive(True)
        self.fitsimage = fi

        bd = fi.get_bindings()
        bd.enable_pan(True)
        bd.enable_zoom(True)
        bd.enable_cuts(True)
        bd.enable_flip(True)

        w = fi.get_widget()
        w.resize(512, 512)

        vbox = QtGui.QVBoxLayout()
        vbox.setContentsMargins(
            QtCore.QMargins(2, 2, 2, 2))
        vbox.setSpacing(1)
        vbox.addWidget(w, 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")
        self.connect(wquit,
            QtCore.SIGNAL("clicked()"),
            self, QtCore.SLOT("close()"))

        hbox.addWidget(wopen, stretch=0)
        hbox.addWidget(wquit, stretch=0)

        vbox.addStretch(1)
        hbox.addStretch(1)

        for w in (wopen, wquit):
            vbox.addWidget(w, stretch=0)

        hw = QtGui.QWidget()
        hw.setLayout(vbox)
        vbox.addWidget(hw, stretch=0)

        vw = QtGui.QWidget()
        self.setCentralWidget(vw)
        vw.setLayout(vbox)

    def load_file(self, filepath):
        fitsobj = pyfits.open(filepath, 'readonly')
        data = fitsobj[0].data

        if (data == None) and (len(fitsobj) > 1) \
            and isinstance(fitsobj[1], pyfits.core.CompImageHDU):
```

Fig. 1: A simple, "bare bones" FITS viewer written in Qt.
```python
data = fitsobj[1].data
fitsobj.close()
self.fitsimage.set_data(data)
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].encode('ascii')
    else:
        fileName = str(res)
    self.load_file(fileName)
def drop_file(self, fitsimage, paths):
    fileName = paths[0]
    self.load_file(fileName)
def main(options, args):
    app = QtGui.QApplication(sys.argv)
    app.connect(app,
        QtCore.SIGNAL('lastWindowClosed()'),
        app, QtCore.SLOT('quit()'))
    logger = logging.getLogger("example1")
    logger.setLevel(logging.INFO)
    stderrHdlr = logging.StreamHandler()
    logger.addHandler(stderrHdlr)
    w = FitsViewer(logger)
    w.resize(524, 540)
    w.show()
    if len(args) > 0:
        w.load_file(args[0])
    app.exec_()
```

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. As shown, `load_file` uses Astropy to open the file and extract the first usable HDU as a NumPy data array. It then passes this array to the viewer via the `set_data()` method. The Ginga widget can take in data either as 2D NumPy arrays, Astropy/pyfits HDUs or Ginga’s own AstroImage wrapped images.

A second class `FitsImageCanvas` (not used in this example, but shown in Figure 2), adds scalable object plotting on top of the image view plane. A variety of simple graphical shapes are available, including lines, circles, rectangles, points, polygons, text, rulers, compasses, etc. Plotted objects scale, transform and rotate seamlessly with the image. See the example2 scripts in the Ginga package download for details.

**Part 2: Developing Plugins for Ginga**

We now turn our attention to the other approach to developing with Ginga: modifying 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 viewing widget 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, shown in Figure 2 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: local and global. 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

![Fig. 2: An example of a FitsImageCanvas widget with graphical overlay.](image-url)
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 5 shows an example of the Pick local plugin occupying a notebook 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.

```
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'
```

The purpose of each method is as follows.

- **__init__(self, fv, fitsimage):** This method is called when the plugin is loaded for the first time. `fv` is a reference to the Ginga shell and `fitsimage` is a reference to the FitsImageCanvas 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.

- **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.

- **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.

- **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. This method may be omitted if there is no special cleanup required when stopping.

- **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 or defocused. The method may be omitted if there is no user event handling to disable.

```python
def __str__(self):
    return 'myplugin'
```
Fig. 4: Two global plugins: Pan (top) and Info (bottom), shown sharing a tab.

Fig. 5: The Pick local plugin, shown occupying a tab.

The plugin is active. This method may be omitted.

Putting it All Together: 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 on the canvas in Figure 2 and the plugin GUI shown in Figure 6.

```python
from ginga.qtw.QtHelp import QtGui, QtCore
from ginga.qtw import QtHelp
from ginga import GingaPlugin

class Ruler(GingaPlugin.LocalPlugin):
    def __init__(self, fv, fitsimage):
        # superclass saves and defines some variables
        # for us, like logger
        super(Ruler, self).__init__(fv, fitsimage)

        self.rulecolor = 'lightgreen'
```
Fig. 6: The Ruler local plugin GUI, shown occupying a tab.

```python
def build_gui(self, container):
    sw = QtGui.QScrollArea()
    twidget = QtHelp.VBox()
    ap = QtGui.QSizePolicy(QtGui.QSizePolicy.MinimumExpanding, QtGui.QSizePolicy.Fixed)
    twidget.setSizePolicy(ap)
    vbox1 = twidget.layout()
    vbox1.setContentsMargins(4, 4, 4, 4)
    vbox1.setSpacing(2)
    sw.setWidgetResizable(True)
    sw.setWidget(twidget)

    msgFont = QtGui.QFont("Sans", 14)
    tw = QtGui.QLabel()
    tw.setFont(msgFont)
    tw.setWordWrap(True)
    self.tw = tw

    fr = QtHelp.Frame("Instructions")
    fr.layout().addWidget(tw, stretch=1,
        alignment=QtCore.Qt.AlignTop)
    vbox1.addWidget(fr, stretch=0,
        alignment=QtCore.Qt.AlignLeft)

captions = ("Units", "combobox"),
    w, b = QtHelp.build_info(captions)
    self.w = b

    vbox1.addWidget(w, stretch=1,
        alignment=QtCore.Qt.AlignLeft)
    vbox1.addWidget(b, stretch=0,
        alignment=QtCore.Qt.AlignLeft)

    container.addWidget(sw, stretch=1)
```

The Ginga package includes a rich set of classes and there are also many methods that can be called in the shell or in the FitsImageCanvas object for plotting or manipulating the view. Length constraints do not permit us to cover even a portion of what is possible in this paper. The best way to get a feel for these APIs is to look at the source of one of the many plugins distributed with Ginga. Most of them are not very long or complex. In general, a plugin can include any Python packages or modules that it wants and programming one is essentially similar to writing any other Python program.

Writing a Global Plugin

This 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, because the plugin is expected to be active across the entire session, and is not associated with any particular channel. build_gui() and start() are called when the Ginga shell starts up, and stop() is never called until the program terminates. pause() and resume() can safely be omitted because they should never be called. Like local plugins, build_gui() can be omitted if there is no GUI associated with the plugin. Take a look at some of the global plugins distributed with the viewer for more information and further examples. The IRAF plugin, which handles IRAF/ginga interaction similarly to IRAF/ds9, is an example of a plugin without a GUI.

Conclusion

The Ginga FITS viewer and toolkit provides a set of building blocks for developers wishing to add FITS image visualization to their Python-based application, or end users interested in a Python-scriptable, extensible viewer. Two avenues of development are possible: a "blue sky" approach by using a flexible FitsImageCanvas display widget and building up around that, or by starting with the plugin-based reference viewer and customizing by modifying or writing new plugins. In either case, the software can be targeted to two different widget sets (Gtk and Qt) across the common desktop platforms that Python is available on today. The code is open-sourced under a BSD license and is available via the GitHub code repository or via PyPI.

Future plans for Ginga mostly center around the development of some additional plugins to enhance capabilities. Ideas suggested by users include:

- mosaicking of images
- simple, user-customizable pipelines for handling flat fielding, bias frames, dark frame subtraction, bad pixel masking, etc.
- improving the set of graphical plotting elements
- semi-transparent colored overlays, for showing masks, etc.
- improving PDF and postscript output options

REFERENCES

[2] Unless the user reloads the plugin. Most plugins in Ginga can be dynamically reloaded using the Debug plugin, which facilitates debugging tremendously, since Ginga itself does not have to be restarted, data does not have to be reloaded, etc.