`stginga`: Ginga Plugins for Data Analysis and Quality Assurance of HST and JWST Science Data

Pey Lian Lim (`lim@stsci.edu`), Space Telescope Science Institute
Eric Jeschke (`eric@naoj.org`), National Astronomical Observatory of Japan

P. L. Lim

Abstract

`stginga`[6] is an image visualization package to assist in data analysis and quality assurance of science data from Hubble Space Telescope (HST) and James Webb Space Telescope (JWST). It is based on the Ginga[3] toolkit for building scientific viewers. In this poster, we will describe the main plugins developed for data analysis and quality assurance tasks with `stginga`. We also discuss the basic outline of writing a Ginga plugin, with pointers to documentation and examples.

Introduction

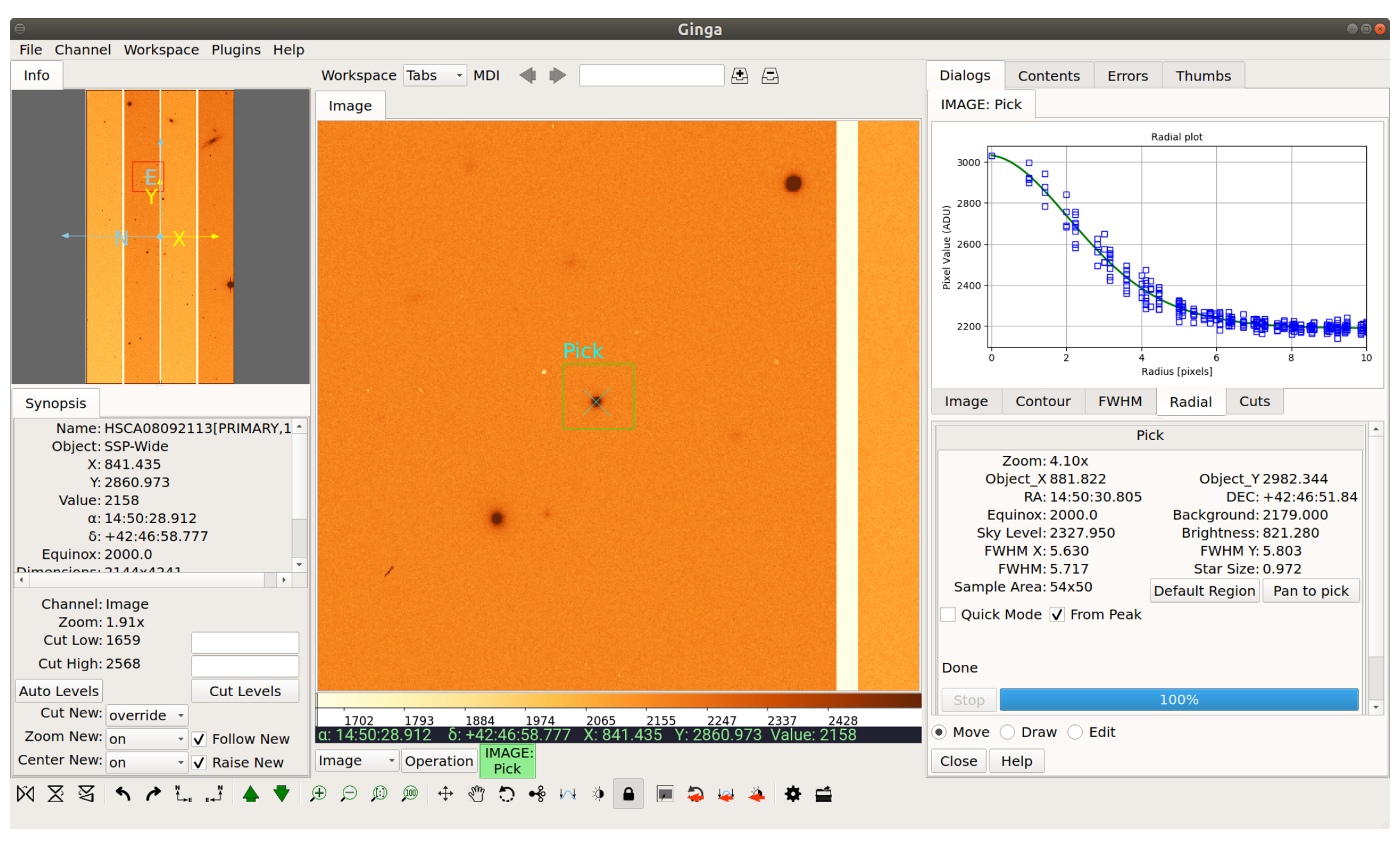


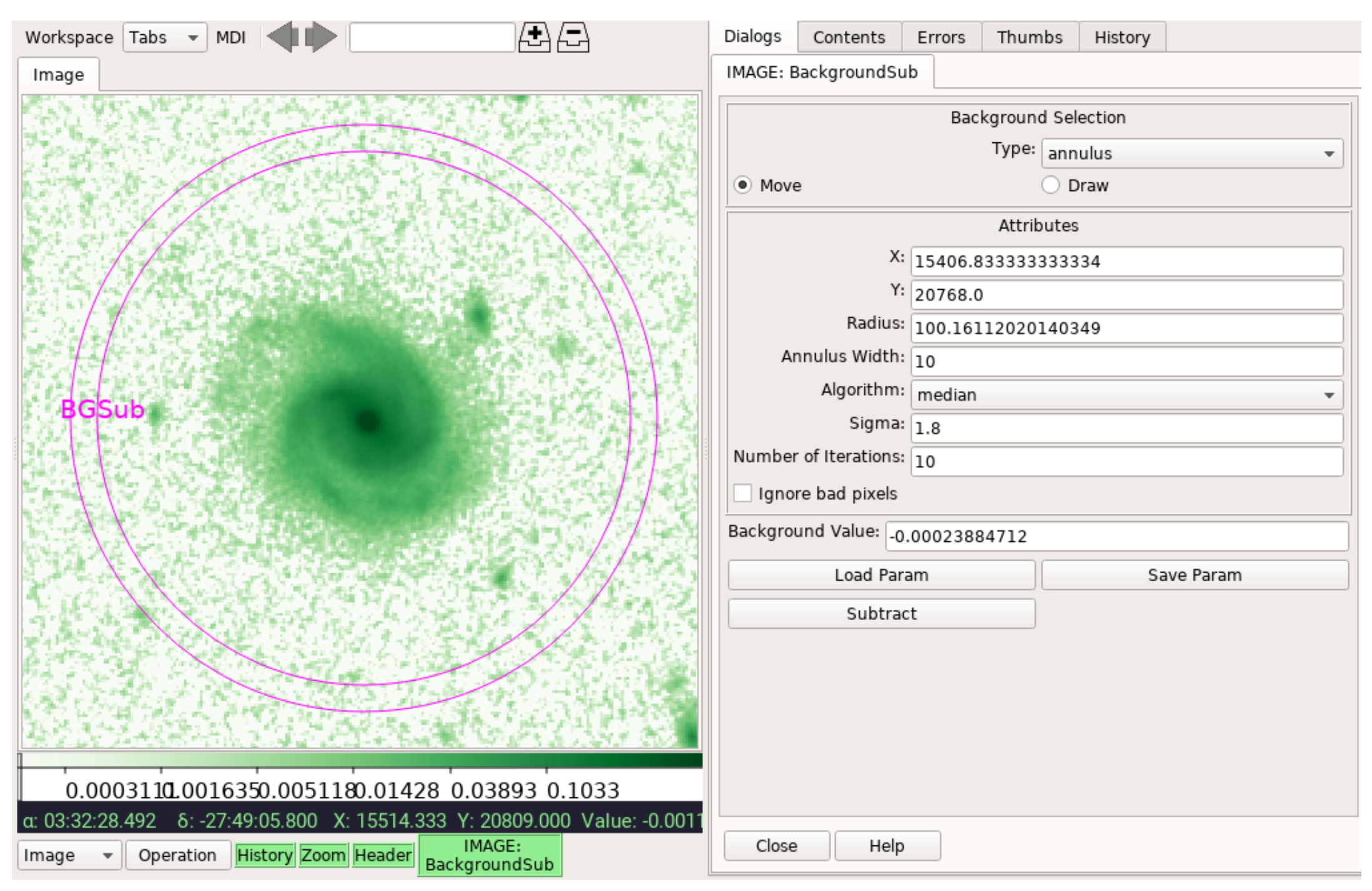
Figure 1: The Ginga reference viewer.

Ginga[2] is a Python package that implements a toolkit for building scientific viewers. It provides a *reference viewer* (see Fig. 1), which features a plugin architecture in which nearly every graphical feature of the program is implemented by a Python plugin. By implementing some new plugins for the HST and JWST data analysis and quality assurance tasks, and combining these with a curated selection of the distributed "stock" plugins, we were able to fairly quickly develop a tool for use in the HST and JWST community.

The reference viewer separates image data into virtual holding pens called *channels*, named and organized by the user. Plugins are categorized as *global* or *local*. A global plugin applies to all images across all channels: only one instance can be opened in the whole Ginga session, whereas a local plugin is associated with the channel it is started from: one instance can be opened per channel and different instances can be configured separately in the same Ginga session.

Currently, all the plugins in `stginga` are local plugins. As with many Ginga plugins, they can be customized via a file in the user's home directory: `~/.ginga/plugin_`*`PluginName`*`.cfg`. Some plugins currently in Ginga originated from `stginga` (e.g., *ChangeHistory*, *SaveImage*, *TVMark*, and *TVMask*) when they were identified to be useful in general beyond HST or JWST. The viewer can use different toolkit *backends*; all the screenshots shown in this poster used Qt5 backend, although they should also work with the Qt4, PySide, GTK 2, and GTK 3 backends.

BackgroundSub



Dialogs | Contents | Errors | Thumbs | History

IMAGE: BackgroundSub

Background Selection
Type: annulus
(●) Move ( ) Draw

Attributes
X: 15406.833333333334
Y: 20768.0
Radius: 100.16112020140349
Annulus Width: 10
Algorithm: median
Sigma: 1.8
Number of Iterations: 10
☐ Ignore bad pixels

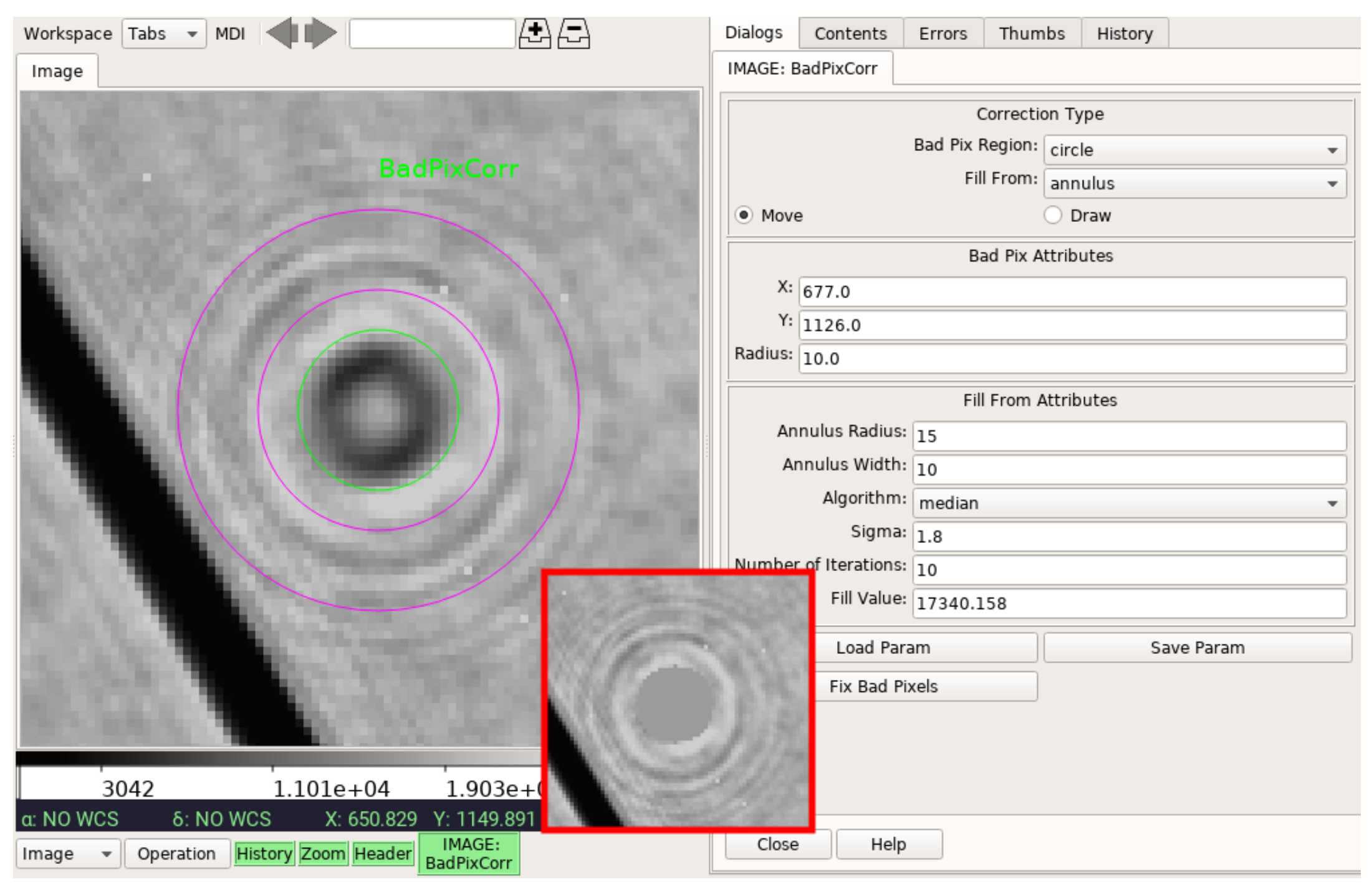
Background Value: -0.00023884712

Figure 2: BackgroundSub plugin for background subtraction.

BackgroundSub (see Fig. 2) is used to calculate and subtract background value. User draws a shape (e.g., annulus) to define the region from which background is calculated. In the "Attributes" box, parameters controlling the calculation can be adjusted. As user modifies the region or changes the parameters, background value would be recalculated accordingly. Optionally, if a data quality (DQ) extension is available, pixels marked as "not good" can be excluded from calculations as well. Subtraction parameters can be saved to a JSON file, which then can be reloaded.

Finally, if desired, the calculated background can be subtracted off the displayed image in Ginga. However, the subtracted image only exists in an in-memory cache in Ginga; if the cache fills up Ginga will eventually eject the image if it is not being viewed. To save the subtracted image out to a different file, use the *SaveImage* plugin in Ginga. As of this writing, *BackgroundSub* only handles constant background, therefore unsuitable for when background has a gradient or a pattern.

BadPixCorr



Dialogs | Contents | Errors | Thumbs | History

IMAGE: BadPixCorr

Correction Type
Bad Pix Region: circle
Fill From: annulus
(●) Move ( ) Draw

Bad Pix Attributes
X: 677.0
Y: 1126.0
Radius: 10.0

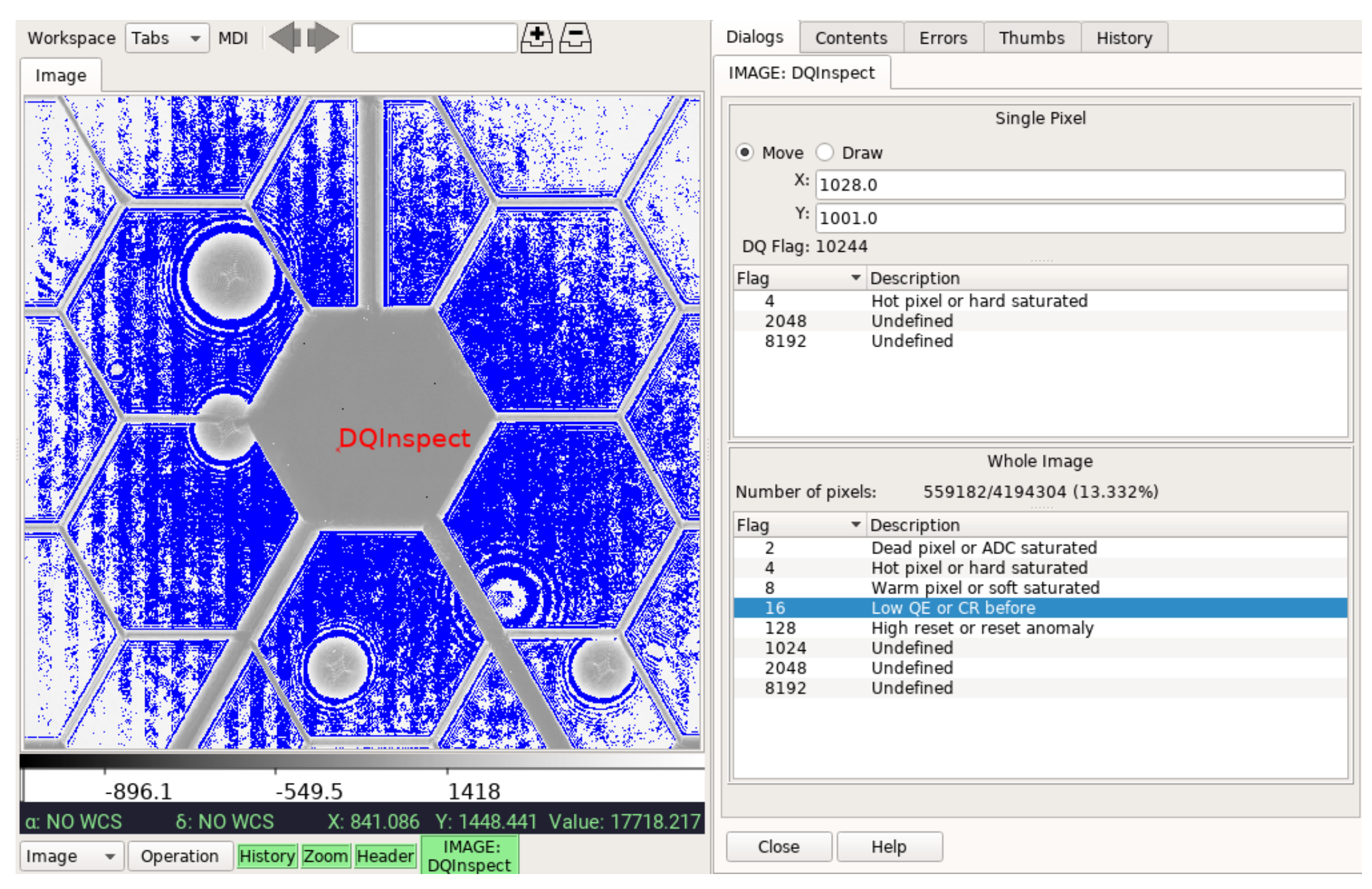
Fill From Attributes
Annulus Radius: 15
Annulus Width: 10
Algorithm: median
Sigma: 1.8
Number of Iterations: 10
Fill Value: 17340.158

Figure 3: BadPixCorr plugin for bad pixel correction.

BadPixCorr (see Fig. 3) is a plugin for performing interactive bad pixel correction on an image. Currently, it only handles fixing a single bad pixel or bad pixels within a circular region. The bad pixel(s) can be filled either by a user-defined constant, a constant calculated from an annulus (not unlike *BackgroundSub*), or Scipy `griddata` interpolation using the annulus. If DQ extension is present, the corresponding DQ flags will also be set to the given new flag value (default is 0 for "good").

Like *BackgroundSub*, it supports saving/loading parameters to/from JSON file and the corrected image only exists in the Ginga in-memory cache; if the cache fills up Ginga will eventually eject the image if it is not in use. To save the result image, use *SaveImage*.

DQInspect



Dialogs | Contents | Errors | Thumbs | History

IMAGE: DQInspect

Single Pixel
(●) Move ( ) Draw
X: 1028.0
Y: 1001.0
DQ Flag: 10244

Flag	Description
4	Hot pixel or hard saturated
2048	Undefined
8192	Undefined

Whole Image
Number of pixels: 559182/4194304 (13.332%)

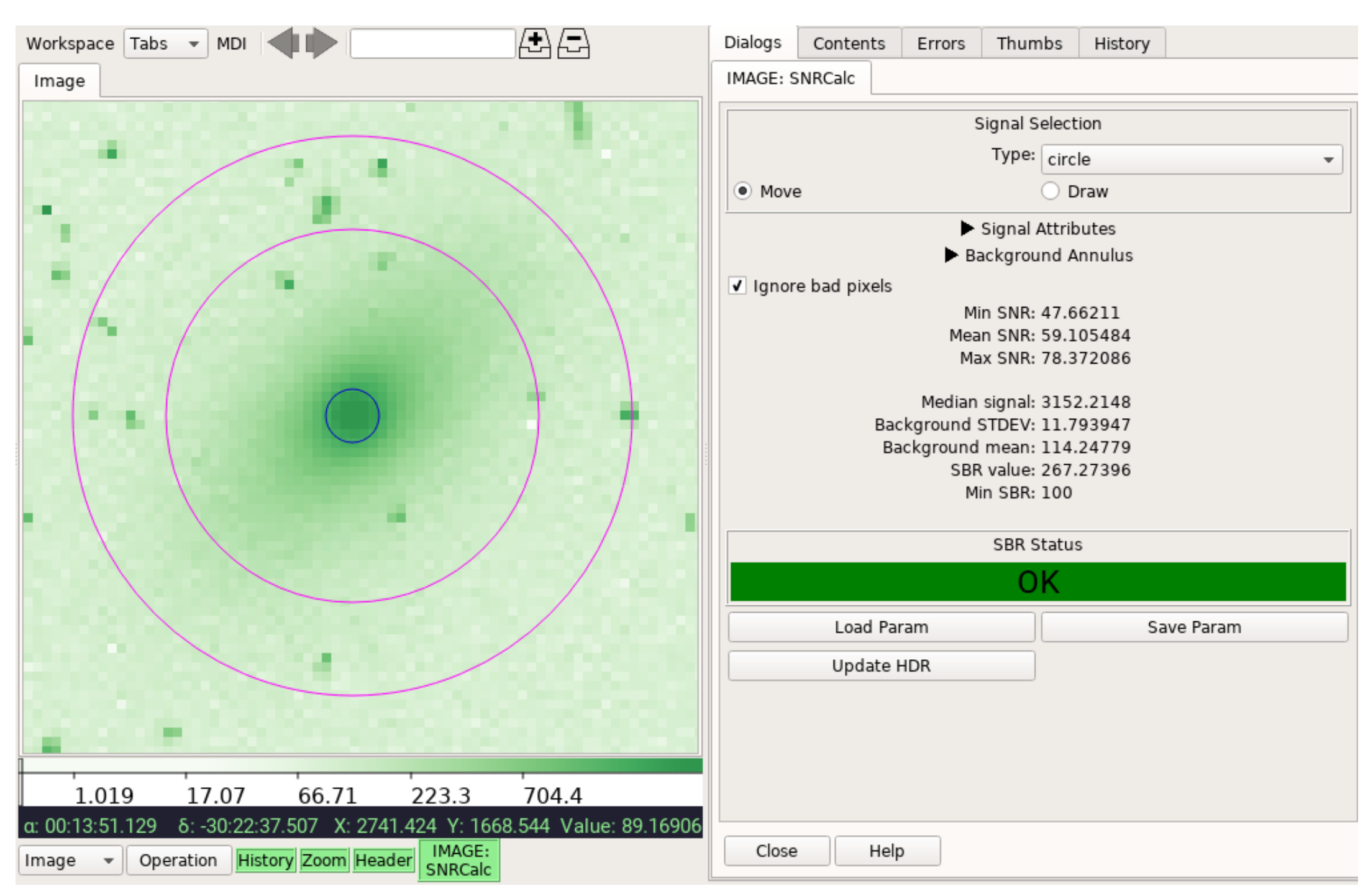
Flag	Description
2	Dead pixel or ADC saturated
4	Hot pixel or hard saturated
8	Warm pixel or soft saturated
16	Low QE or CR before
128	High reset or reset anomaly
1024	Undefined
2048	Undefined
8192	Undefined

Figure 4: DQInspect plugin for data quality inspection.

DQInspect (see Fig. 4) is used to visualize the associated DQ array stored as an HDU within an image. It shows the different DQ flags (top table) that went into a selected pixel (marked by a red "x") and also the overall mask of the selected DQ flag(s) (blue pixels; bottom table). For overall mask, when multiple flags are selected, each flag is assigned a different mask color at a reduced opacity for each. User has the option to customize flag definitions for different instruments.

SNRCalc

SNRCalc (see Fig. 5) is used to calculate Signal-to-Noise Ratio (SNR) and Surface Background Ratio (SBR) on an image. Given the selected science (S) and background (B) regions, SBR is defined by Ball Aerospace[1] as the median of S divided by the standard deviation of B. If the image has an accompanying error (E) extension, SNR can also be calculated by dividing S by E over the same region and then computing its min, max, and mean.



Dialogs | Contents | Errors | Thumbs | History

IMAGE: SNRCalc

Signal Selection
Type: circle
(●) Move ( ) Draw
▶ Signal Attributes
▶ Background Annulus
☑ Ignore bad pixels

Min SNR: 47.66211
Mean SNR: 59.105484
Max SNR: 78.372086

Median signal: 3152.2148
Background STDEV: 11.793947
Background mean: 114.24779
SBR value: 267.27396
Min SBR: 100

SBR Status
OK

Figure 5: SNRCalc plugin for SNR calculation.

While SNR is more popular, SBR is useful for an image without existing or reliable error values. User may define a minimum limit for SBR check, so that the GUI can provide a quick visual indication on whether the selected region achieves the desired SBR or not. As part of the statistics, mean background value is also provided, albeit not used in SBR nor SNR calculations. Optionally, if DQ extension is available, pixels marked as "not good" can be excluded from calculations as well.

Like *BackgroundSub*, it supports saving/loading parameters to/from JSON file. Calculated values can be saved in the image header using the "Update HDR" button. The image with updated header only exists in the Ginga cache until it is forced out by viewing other images; to save it, use *SaveImage*.

Writing a Ginga plugin

Instructions for writing a plugin is available at `https://bit.ly/writeplugins`. Existing plugins in Ginga and `stginga` code repositories can be used as examples. It is recommended that you play with the existing ones and choose one that is the closest to your desired functionality as a starting point.

Local plugins

A local plugin at its simplest is a Python class defined in a file. The class should inherit from `ginga.GingaPlugin.LocalPlugin` and provide `__init__()`, `build_gui()`, `start()`, and `stop()` methods. These methods are used to initialize the plugin, build the user interface, and to do any necessary tasks at the start and stop of the plugin, respectively. Typically, you would also want to implement the `redo()` method, which is called when there are new data loaded into the viewer to which the running plugin should respond.

Inside the file, any modules that are available in the user's Python environment may be imported and used, allowing huge flexibility in the kinds of things a plugin can do; i.e., open files, connect to sockets or other communication frameworks, or call a myriad of astronomical Python packages. It also has a reference to the viewer with which it is associated so it can access the viewer data (as Numpy array) and can manipulate canvas overlays with graphics on the viewer (as shown in the sections above) or manipulate the viewer settings (e.g., panning, scale, color map).

Global plugins

Writing a global plugin is similiar to the process for writing a local one. The difference is that the plugin ostensibly must be able to update its state when the user switches channels, since there is only one instance of the plugin allowed to be open; There are callbacks for which you can register to be alerted of these events. Otherwise, the API is quite similar to that of a local plugin.

Distributing plugins

When you want to distribute your plugin(s), the best way is to use the `ginga-plugin-template` [4]. This template allows one or more plugins to be installed as a separate package, and be discovered by the reference viewer when it starts up. If you want more control over the layout of the viewer and the set of included plugins, you can follow the path blazed by `stginga` and make your own startup script for the reference viewer with a curated mix of the stock plugins with your own.

Conclusion

`stginga` utilizes Ginga plugins to support HST and JWST data analysis, which includes background subtraction, bad pixel correction, DQ flags inspection, and signal-to-noise calculations.

Writing Ginga plugins can be an expedient way to develop graphical data analysis and quality assurance tasks, by leveraging the combination of Python, a lean Ginga plugin API, and the burgeoning number of open-source astronomical Python modules.

Both `stginga` and `ginga` are installable via `pip`. Alternatively, if you use `conda`, they are also available on AstroConda[5], in addition to `ginga` being in `conda-forge` too. Their development versions could be cloned from GitHub. Both are open-source and licensed under 3-clause BSD.

References

[1] S. Acton. Image Pre-Processor SBR. Private communications, 2015.

[2] E. Jeschke, T. Inagaki, and R. Kackley. Enhancements to Ginga: a Python package for building astronomical data viewers. In A. R. Taylor and E. Rosolowsky, editors, *Astronomical Data Analysis Software and Systems XXIV*. ASP, 2015. Vol. 495.

[3] National Astronomical Observatory of Japan. Ginga. https://github.com/ejeschke/ginga, 2018.

[4] National Astronomical Observatory of Japan. ginga-plugin-template. https://github.com/ejeschke/ginga-plugin-template, 2018.

[5] Space Telescope Science Institute. AstroConda. https://astroconda.readthedocs.io, 2018.

[6] Space Telescope Science Institute. stginga. https://github.com/spacetelescope/stginga, 2018.