Astronomical Data Analysis Software and Systems: XXIV
ASP Conference Series, Vol. 495
A. R. Taylor and E. Rosolowsky, eds.
©2015 Astronomical Society of the Pacific

Hackable User Interfaces In Astronomy with Glue

Christopher Beaumont,[1] Alyssa Goodman,[1] and Perry Greenfield[2]

[1]*Harvard-Smithsonian Center for Astrophysics, 60 Garden St, MS 42, Cambridge MA, 02138*

[2]*Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218*

Abstract. Astronomers typically choose between Graphical User Interfaces and custom-written computer code when exploring and analyzing data. Few tools are designed to encourage both of these workflows, despite their complementary strengths. We believe that such hybrid **hackable user interfaces** could enable more agile data exploration, combining the fluidity that comes from a GUI with the precision and reproducibility that comes from writing code. In this article we articulate the different strengths and weaknesses of both workflows and discuss how to enable both in a single tool. We focus on Glue (`http://glue-viz.org`) as a case study and examine how the goal of creating a hackable user interface has influenced the design of Glue.

1. Introduction

Most of the tools that astronomers use for data exploration and analysis fall into one of two categories. The first set of tools are primarily programmatic interfaces that users operate by writing code. The second set of tools are Graphical User Interfaces (GUIs). Each workflow has complementary strengths and weaknesses.

Reproducibility – Programmatic tools tend to be better suited for performing reproducible computation. This is due to the fact that scientists are required to translate their high-level ideas into fully-unambiguous descriptions that a computer can execute. The resulting script or program can then be refined to express the exact computation the scientist wants and stored in a fully self-contained, reproducible workflow.

GUIs are mediated by inherently less-precise means – mouse movements, key presses, etc. While these interactions can in principle be logged to create a "reproducible" workflow, the resulting log is not usually a meaningful summary for a human to read.[1]

Expressiveness – Modern research involves analyzing data in increasingly novel and customized ways. Programming languages are better-suited to address this long tail of analysis applications, since they are more expressive. GUIs, on the other hand, are better suited to perform specific common tasks very easily, since they provide a more restricted, specialized interface.

[1]A notable exception to this is the Chandra Imaging and Plotting System (ChIPS; Germain et al. 2006), which can export the state of a GUI session to a Python script of meaningful analysis steps.

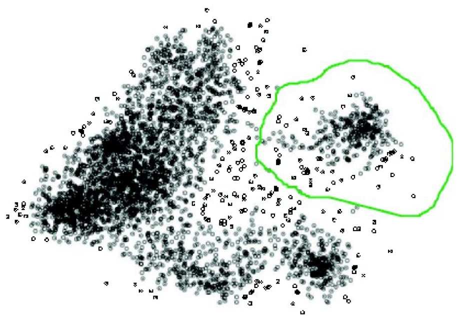


Figure 1. Geometrically complex selections like the one drawn above are trivial to specify interactively, but tedious to specify programmatically.

Cognitive Load, Creativity, and Exploration – While computer code is more reproducible, expressive, and scalable than working with a GUI, it is also significantly harder to use. The act of translating a high-level scientific idea into a low-level computational one takes time and effort, and requires that a scientist switch mental contexts from science to programming. All of this effort is orthogonal to actually doing science.

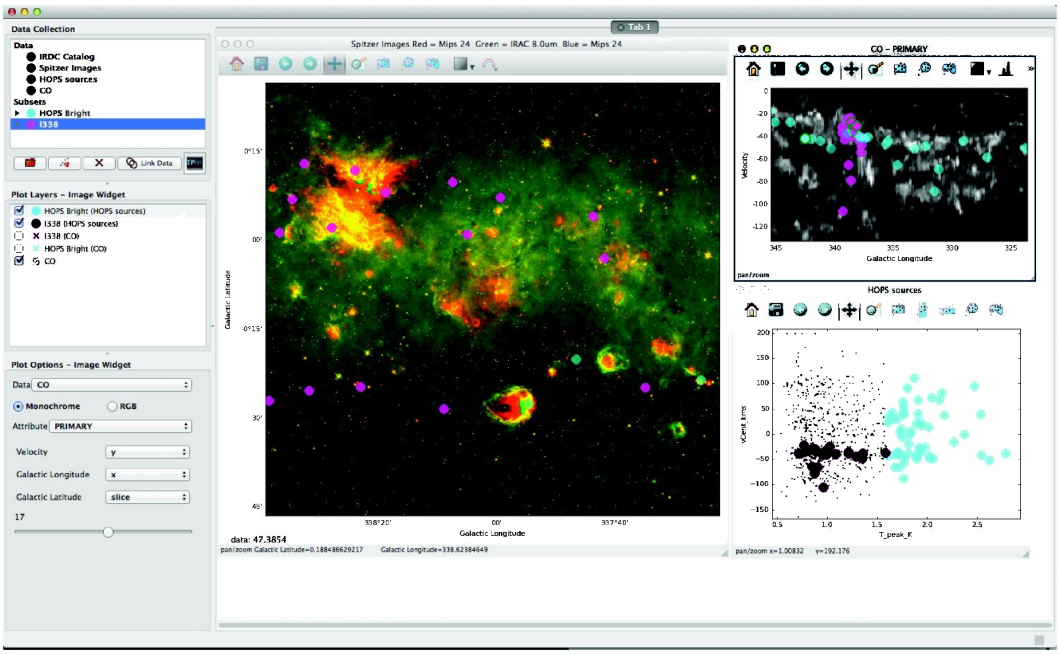
While GUIs are more restrictive in the tasks they perform, they are also a more natural interface when designed effectively. It is generally easier to remain focused on scientific questions when using a GUI; this promotes more creative exploration of data.

Visual Analysis – Finally, many analytical tasks are inherently more visual than verbal, and thus more naturally accommodated by graphical user interfaces. This is particularly true in astronomy. One common example is image adjustment – it is much more natural to adjust parameters like field of view, contrast, intensity transfer functions, etc. by seeing the immediate result of interacting with widgets than it is to type out these settings textually. Another example is selecting regions of parameter space, particularly if those regions are geometrically complex or defined by subtle patterns in the data. For example, the selection in Figure 1 is trivial if a user can draw regions on a plot but much more tedious to specify programmatically.

Neither the graphical nor the programmatic workflow is ideal for the entire data analysis lifecycle. This suggests that hybrid tools – so-called **hackable user interfaces** that combine graphical and programmatic interaction – might better enable analysts to explore their data than tools focused on a single interaction mode. This idea has been a key motivation in the development of Glue, a Python library for building linked, interactive visualizations of related datasets.

2. Introduction to Glue

Glue was originally developed to more easily inter-compare related datasets across several files. The key interaction that Glue supports is the ability to create linked-view visualizations of image and catalog data. All visualizations in Glue are brushable; users can draw arbitrary shapes on a plot, and these shapes filter the data lying inside these re-



Data CO
Attribute PRIMARY
Velocity y
Galactic Longitude x
Galactic Latitude slice
17
data: 47.3854
pan/zoom Galactic Latitude=0.188486629217 Galactic Longitude=338.62384649
pan/zoom x=1.00832 y=192.176

Figure 2. Screenshot of Glue visualizing Spitzer imagery, a CO spectral data cube, and catalog of pre-stellar cores.

gions. Upon selecting data in one visualization, the selection is highlighted in all visualizations. Furthermore, users can specify logical relationships that exist between related datasets, which allows selections to propagate across files. Figure 2 shows a screenshot of Glue. Linked-view visualizations are an effective, well-established technique to understand high-dimensional datasets, and they have a long history inside and outside astronomy (Goodman 2012). Glue extends this technique across file boundaries, making inter-dataset comparisons easier while minimizing the amount of preparatory data cleaning.

3. Hackable User Interface Patterns

3.1. Automation via Programmatic Interface Control

The first general technique for making GUIs more hackable is to allow users to control a program programmatically. With this functionality, users can automate custom workflows. A common tedium of using GUIs is the "cold start problem" – loading data into a GUI often involves a lot of repetitive actions (navigating through an open file dialog, adjusting window sizes and positions, setting good initial parameters like image contrast or point size, etc.). When a tool is scriptable, however, repetitive tasks can be consolidated into macro-like utilities.

When building Glue, we have sought out actions that users most likely want programmatic control over. We then aim to make the interface for performing these actions as simple as possible. One such action, traditionally a pain point with GUIs, is file loading. Glue allows users to write custom functions to parse arbitrary files. Since Python has a rich ecosystem across many different scientific domains, these parsing functions are often simple wrappers around third party tools, making it easy to ingest arbitrary data into Glue. Listing 0.1 shows an example function to parse medical MRI data into Glue.

```
import numpy as np
from dicom import read_file
from glue.config import data_factory

@data_factory('DICOM Medical Image Loader')
def load_mri_data(path):
    dicom_file = read_file(path)
    data = np.fromstring(dicom_file.PixelData, dtype=np.short)
    return data.reshape(dicom_file.Rows, dicom_file.Columns)
```

Listing 0.1 A custom Glue function to parse medical image data.

Glue users can also write startup scripts to pre-load datasets and define the links between these datasets. Since researchers often examine the same data dozens of times over the course of a project, these scripts enjoy multiple reuse, and eliminate the tedium of guiding the initial state of a GUI to where a user prefers to start analysis.

3.2. Extensibility via Plugins

The programmatic control discussed in the previous section addresses the fact that writing code is a better way to automate repetitive tasks than using a GUI. Another, more serious drawback of traditional GUIs is that they are designed to facilitate a narrow set of analytical tasks, relative to a programming language. GUIs may make it far easier to carry out the computation they were designed for (say, contrast adjustment in ds9; Smithsonian Astrophysical Observatory 2000) but performing other tasks is often impossible (say, edge detection in ds9). There is a dual penalty to this: first, a ds9 user interested in running edge detection on her data must switch tools and mental context; second, she loses the ability to easily explore the results of the analysis in ds9 (she will likely convert the result into a new image, save it to disk, and re-load it in ds9. This tends to clutter the filesystem with temporary or intermediate data products).

To address this limitation, GUIs typically provide a plugin architecture to let users extend the interface with custom functionality. Most of the major astronomy data exploration tools have such a functionality. ds9 has a mechanism for users to call custom analysis scripts written in any language, passing along information about the current state of the viewer (currently image, mouse location, defined regions, etc.).[2] Users can create Aladin plugins (Bonnarel et al. 1999) by building a Java subclass of an AladinPlugin class.[3] TOPCAT (Taylor 2005) lets users write custom code for creating new derived table columns from other columns (again using Java).[4]

It seems that these features are not well-used – a web search reveals only a handful of ds9 or Aladin plugins developed by the community. Why might this be? Successful community-driven projects like yt (Turk et al. 2011), Astropy (Astropy Collaboration

[2] `http://ds9.si.edu/ref/analysis.html`

[3] `http://aladin.u-strasbg.fr/java/nph-aladin.pl?frame=plugins`

[4] `http://www.star.bris.ac.uk/~mbt/topcat/sun253/sun253.html#jelExtend`

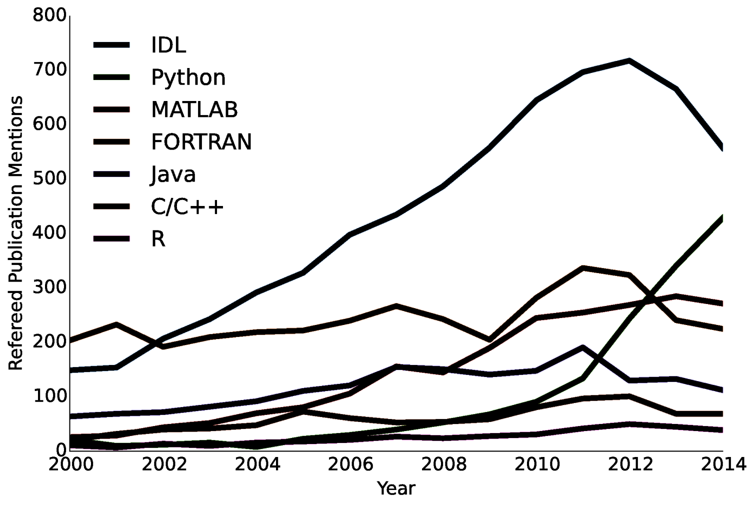


Figure 3. How often various programming languages are mentioned in articles on the SAO/NASA Astrophysics Data System. Scripting languages – particularly IDL and Python – dominate the discussion of programming languages in the astrophysical literature.

et al. 2013), and SunPy (SunPy Development Team 2014) suggest the problem isn't lack of astronomers interested in writing software – these projects each receive contributions from hundreds of unique scientists. The problem is more likely a steep learning curve and/or superficial payoff for the required time investment. For example, writing plugins in Aladin or TOPCAT require writing Java code, which few astronomers write (Figure 3 shows how often various programming languages are mentioned in refereed articles on the SAO/NASA Astrophysics Data System).[5] Furthermore, an architecture like Aladin's requires that developers work directly with the internal Aladin API, whose structure is geared more towards professional software developers than scientists. Given this high initial barrier, and a lack of good examples or documentation of how plugins might be used, such features are not adopted.

These ideas have influenced how we've designed plugins for Glue. Importantly, Glue is written in Python, using the same libraries that many scientists use for day-to-day data analysis. Because of this, it is fundamentally easier to expose programmatic functionality to scientists at a conceptual level that they are comfortable with.

Glue's custom data viewers are an example of this design philosophy. The vast majority of Python-using astronomers use Matplotlib (Hunter 2007) to visualize data, and they use that library to develop a wide array of customized visualizations. Because Glue also uses Matplotlib to build interactive, selectable, linked-view visualizations, there is a natural opportunity to allow users to use their own custom visualization code within Glue. The problem to address is that, of course, visualizations in Glue are more than static Matplotlib plots: they allow users to overlay multiple data layers interactively, automatically update when selections change, and provide widgets for adjusting

[5]`http://bit.ly/ads_popularity`

parameters of the plot. While many scientists *are* interested in designing custom visualizations, most are *not* interested in thinking about user interface logic.

Glue minimizes the amount of user interface logic needed to create a custom visualization, and strives to make the code look as much as possible like what scientists already write. Listing 0.2 shows a simple scatterplot viewer as an example – the code creates the viewer shown in Figure 4. This code reveals several ways in which Glue simplifies the act of creating custom viewers:

```
import numpy as np
from glue import custom_viewer

def scale(x):
    if x.size > 0:
        x = 80.0 * x / np.nanmax(x) + 10
    return x

simple_scatter = custom_viewer('Custom Scatterplot',
                               opacity=(0.0, 1.0),
                               x='att',
                               y='att',
                               point_size='att'
                               )

@simple_scatter.plot_data
def draw_data(axes, x, y, point_size, opacity, style):
    axes.scatter(x, y, s=scale(point_size),
                 alpha=opacity, color=style.color)

@simple_scatter.plot_subset
def draw_subset(axes, x, y, point_size, style):
    axes.scatter(x, y, s=scale(point_size),
                 color=style.color)

@simple_scatter.select
def select(roi, x, y):
    return roi.contains(x, y)
```

Listing 0.2 A custom Glue function to parse medical image data.

- Users do not worry directly about event handling or state synchronization. Instead, they divide their logic into a few separate functions, and provide some annotation to describe what tasks each function performs. Glue calls the `@plot_data` function to visualize a dataset, `@plot_subset` to visualize a subset, and the `@select` to determine how to filter data points based on a user-drawn selection region on a plot.
- Users do not directly write code to create widgets. Instead, they specify various adjustable properties, and sensible widgets are auto generated from these. For example, the snippet `opacity=(0.0, 1.0)` creates a slider widget for setting

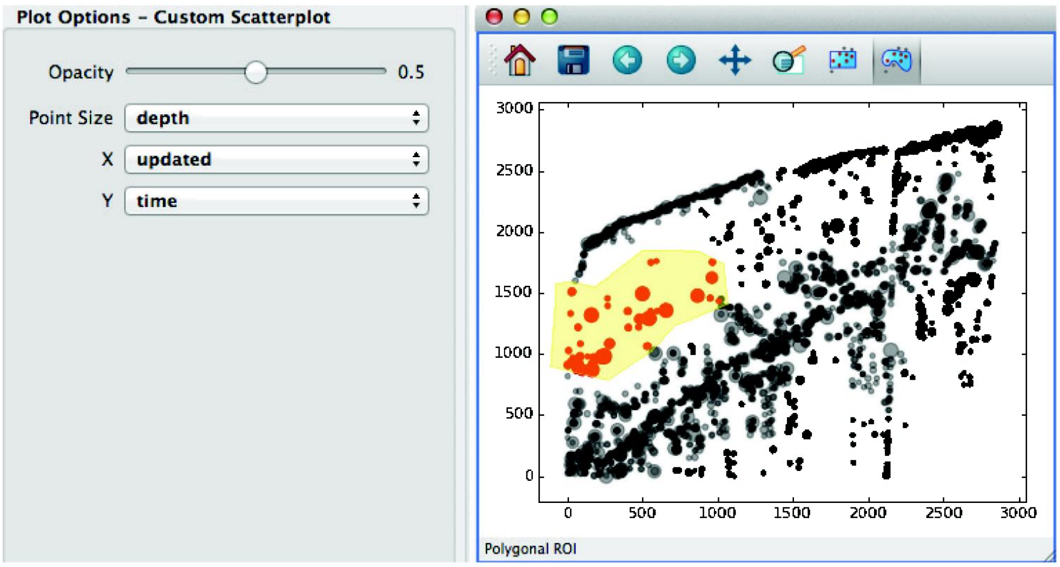

Plot Options – Custom Scatterplot
Opacity 0.5
Point Size depth
X updated
Y time

Figure 4. The custom scatter plot generated from the code in Listing 0.2.

the plot opacity, and the current value of this setting is passed into functions like `plot_data`. Likewise, the `x="att"` snippet creates a drop down box allowing users to select a particular data attribute, which the plotting functions can use however they see fit (in this case, to determine the x location of the points).

- As much as possible, these functions use data structures that scientists are familiar with. For example, the state of the opacity widget is passed into functions as a number, and data attributes are passed into functions as NumPy arrays. This minimizes the amount of Glue-specific API that a user has to learn to build a useful visualization.

Glue's custom viewer framework exposes nearly the full power of Matplotlib to the user, while minimizing the amount of extra work needed to turn static plots into interactive, selectable Glue viewers. This design was inspired by R's Shiny library (RStudio, Inc 2013) and IPython's interactive widget functionality (Pérez & Granger 2007). These projects also allow users to make interactive visualizations without worrying about GUI logic, and have seen broad community adoption in both academic and commercial settings (there are several thousand examples of code using these libraries on GitHub).

3.3. Extensibility via Data Sharing

Plugins address limitations in the tasks GUIs can perform by allowing others to "send code in" to the application. Another approach is to "pull data out" of an application to explore in another context, optionally sending the results of that analysis back to the GUI. This workflow is also enabled by the fact that Glue is written in Python. Glue uses the IPython kernel/client architecture, which allows multiple resources to share the same Python session. This allows Glue to run in parallel with, say, the IPython

Notebook.[6] Furthermore, both programs share access to the same data, which enables some interesting workflows. For example, the code snippet in Listing 0.3 extracts three columns of a catalog loaded into Glue, uses the third-party Scikit-Learn library (Pedregosa et al. 2011) to cluster these data into three groups, and sends the groups back into Glue as subsets. Glue makes it easy to extract data out of the interface, perform arbitrary analysis in Python, and send results back into Glue as new datasets or selections. Selections can also be easily extracted, making Glue a useful way to visually filter out particular data points for a given analysis.

Glue also supports live-monitoring of files it has loaded data from and, if these files change , Glue refreshes all the data and plots. This provides an easy and general way to integrate other analysis tools. For example, calibrated datasets are often produced by custom data reduction scripts. Users can interactively refine these scripts by loading the output datasets in Glue, tweaking and re-running the reduction, and examining how the data change – without ever having to restart Glue or rebuild plots.

```
from glue import glue
from sklearn.cluster import KMeans
import numpy as np

# start glue
session = qglue()

# ... user loads data from the UI

# extract the loaded data
dc = session.data_collection
data = dc[0]

# extract 3 attributes from the data, cluster
X = np.column_stack((data['x'], data['y'], data['z']))
clusters = KMeans(n_clusters=3).fit_predict(X)

# add cluster_id as a new attribute
c = data.add_component(clusters, label='cluster_id')

# create 3 new subsets, that select each value in clusters
dc.new_subset_group(label='Cluster 1', subset_state = (c == 0))
dc.new_subset_group(label='Cluster 2', subset_state = (c == 1))
dc.new_subset_group(label='Cluster 3', subset_state = (c == 2))
```

Listing 0.3 Code to launch Glue, extract data from a running session, perform a clustering analysis in Python, and send the identified clusters back into Glue as new selections.

[6]Glue also provides a IPython terminal widget in the main UI

4. Conclusion

Astronomers embrace both programmatic and graphical user interfaces for data exploration, but rarely at the same time. The friction at this interface between interfaces tends to push researchers towards one workflow or the other, despite the fact that the two workflows have complementary strengths – GUIs are better-suited for visual exploration, code is better suited for expressing and automating precise computation. Tools could be designed to better facilitate switching between workflows. Such hackable user interfaces would better combine the benefits of each style into a more flexible data exploration tool. The design of Glue has been guided by this idea. Glue is built using the same software suite many astronomers user for their analysis. This makes it easier to build a GUI that researchers can take programmatic advantage of, without dealing with the tedious details of GUI programming.

Acknowledgments. Glue is supported by a contract with NASA. Figure 3 is adapted from an analysis from Tom Robitaille[7].

References

Astropy Collaboration, Robitaille, T. P., et al. 2013, A&A, 558, A33. `1307.6212`

Bonnarel, F., et al. 1999, in Astronomical Data Analysis Software and Systems VIII, edited by D. M. Mehringer, R. L. Plante, & D. A. Roberts, vol. 172 of Astronomical Society of the Pacific Conference Series, 229

Germain, G., et al. 2006, in Astronomical Data Analysis Software and Systems XV, edited by C. Gabriel, C. Arviset, D. Ponz, & S. Enrique, vol. 351 of Astronomical Society of the Pacific Conference Series, 57

Goodman, A. A. 2012, Astronomische Nachrichten, 333, 505. `1205.4747`

Hunter, J. D. 2007, Computing In Science & Engineering, 9, 90

Pedregosa, F., et al. 2011, Journal of Machine Learning Research, 12, 2825

Pérez, F., & Granger, B. E. 2007, Computing in Science and Engineering, 9, 21. URL `http://ipython.org`

RStudio, Inc 2013, Easy web applications in R. URL: `http://www.rstudio.com/shiny/`

Smithsonian Astrophysical Observatory 2000, SAOImage DS9: A utility for displaying astronomical images in the X11 window environment, Astrophysics Source Code Library. `0003.002`

SunPy Development Team 2014, SunPy: Python for Solar Physicists, Astrophysics Source Code Library. `1401.010`

Taylor, M. B. 2005, in Astronomical Data Analysis Software and Systems XIV, edited by P. Shopbell, M. Britton, & R. Ebert, vol. 347 of Astronomical Society of the Pacific Conference Series, 29

Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., & Norman, M. L. 2011, ApJS, 192, 9. `1011.3514`

[7] `http://astrofrog.github.io/blog/2013/10/02/acknowledging-tools-services-in-papers/`