Interactive Data Analysis Environments Birds-of-a-Feather Session

Joseph Harrington and Paul E. Barrett

Codes 693 and 660, NASA Goddard Space Flight Center, Greenbelt, MD 20771-0001

Abstract. We conducted a discussion of interactive environments in which scientists handle data. Traditional astronomical packages (AIPS, IDL, IRAF, MIDAS, etc.) lack many modern language and interactive features, whereas modern interactive/scripting/rapid prototyping environments (PERL, Tcl, etc.) lack convenient and efficient numerical capability and a way to access existing astronomical code. Our focus was thus how best to bring the capabilities of modern environments to astronomical data processing. To follow developments in this evolving field, we have set up a WWW page[1] that compares the available environments.

1. Introduction

Even with the data-volume explosion of modern astronomy, desktop computers can easily keep pace with the calculations we require of them. However, our workstations do not themselves know what calculations must be performed. Developing an analysis approach generally takes an individual astronomer much longer than the observations and calculations themselves. Reducing this bottleneck should therefore be a high priority. In recent years the community of programmers on the Internet has created interactive environments that accelerate the development of data handling software. Although these systems are often extremely powerful and flexible, they tend to have better support for textual data manipulation than for the numerical calculation and image display that astronomers need. Most astronomical environments, rich in task-specific packages of routines and capable of the necessary numerics, were written long before the advances in interactive environments and feel cumbersome when compared to the newer systems. The need for individual astronomers to handle larger and more complex data sets continues to grow, so it is time to bring the state of the art in interactive environments to astronomy.

We conducted a Birds-of-a-Feather (BoF) discussion about interactive environments for handling astronomical data. Representatives from environments both in astronomy and from outside the field formed a panel that took audience questions. Panel members were Brian Glendenning for Glish/AIPS++, Wayne Landsman for IDL, Michael Fitzpatrick for IRAF, Arnold Rots for Khoros, Klaus Banse for MIDAS, Lee Rottler for Python, and Nicholas Elias for Tcl. We

[1] http://lheawww.gsfc.nasa.gov/users/barrett/IDAE/table.1.html

wanted to find out what the community wanted in an interactive environment and how it felt about certain specific issues. We also wanted to inform the community of the possibilities now available with state-of-the-art systems outside of astronomy. The BoF ran for 90 minutes and maintained an attendance of about 50 participants, with some flux into and out of the room. The range of expertise in the audience was large, so the discussion was punctuated by a number of well-made points rather than an approach to consensus or disagreement. In this article we will focus on the points made during the discussion. Length restrictions preclude making a detailed case for a change in astronomical data environments or for discussing specific systems.

2. Context: Our Investment in Software

The task of bringing modern language and interactive capability to astronomical data analysis is difficult for two reasons. First, the developments in computing range from standardized command-line interfaces to network-transparent interprocess communication. We need to find out what capabilities are available, which are most important to researchers, and what are the relative difficulties of implementation. Second, we cannot afford to abandon or manually convert the huge quantity of code we currently use. We will not likely agree on a single environment, yet we will want to be able to use each others' code more than we can now. Any solution we choose must not leave us in a position to suffer from the same problem again in the future.

Our discussion focuses on languages and interactive environments, but we must always keep the larger second problem in mind: We desire a solution that accommodates the simultaneous use of multiple languages and allows for easy transition from one to the next as languages evolve. One possibility is the "Gaming Table" concept of Jan Noordam and Will Deich's Future of Astronomical Data Systems BoF. Similar strategies are currently under consideration by several of the major astronomical packages, most notably AIPS++. We mention this concept here because any system that is not capable of using code written in other languages (particularly compiled languages) or of being run by code written in other systems is a bad, short-term investment.

3. Ideas from the Discussion

Key features of the interactive environment include command-line editors (standards like GNU Readline are not yet supported by the astronomical packages), real-time numerical efficiency, debuggers, and graphics display. Scripting language issues include friendly syntax, fast learning curve, good array notation, and access to compiled code and code written in other scripting languages.

Although they are different concepts, the issues of interactive environments and scripting languages are tied together. A script may have been prepared in advance and developed with a plan, or it may evolve as the user thinks of things and types them. Whereas a script file is read monotonically from beginning to end, an interactive user may need to abandon a train of thought, backtrack, correct errors, etc. Interactive users are more efficient when they can re-use portions of earlier commands, when they can call up information such as documentation, when they type to standard interfaces, and when their overall typing burden is reduced. The author of a non-interactive script (one contained in a file) can select the editing environment to suit personal taste and is usually much less concerned about extra typing than an interactive user. For example, many non-interactive languages (such as C or PERL) use semicolons to signal the end of a line. Signaling the end of an interactive line with anything more than a return is a nuisance. Likewise, non-interactive function calls often have the form `function(`*arg1*`, `*arg2*`, ...)`. The most common interactive environments (such as the shells) dispense with the parentheses and commas and just use spaces as delimiters. There are many reasons to desire the interactive and scripting languages to be the same, but the best interactive syntax may be very different from the best script syntax. Flexible systems like IRAF cl allow both styles.

Language syntax was a contentious area. Individuals preferred different styles and were unwilling to agree on a single language. Some were not willing to learn a new language unless there was overwhelming reason to do so. Some liked object-oriented languages, others thought that a waste of time, at least for scripting. Some were picky about the exact style of a language's syntax, others said they could adapt to anything without much cost.

A strong appeal of analysis languages like IDL and Python is the ability to manipulate arrays simply. If `a` and `b` are arrays, then `c=a*b` multiplies corresponding elements of `a` and `b` and `b=a[20:30,19:29]` extracts a sub-array from `a`. This "syntactic sugar" makes the code much easier to read and write than function calls or explicit loops. Furthermore, the operation runs at machine speed, making it practical to reduce large datasets using an interpreted language. However, complicated expressions like $a = b \times (c + d/2)^e$ are done as a series of simpler whole-array operations. Modern compilers can take advantage of machine instructions that optimize such expressions, passing the data through the CPU only once. We hope to see interpreters that do this soon.

Those who value rapid prototyping (quickly developing scripts that do a job well, then converting them to a compiled language that runs fast) prefer a language that is as close as possible to the ultimate compiled language, but that is interactive, doesn't have declarations, has conveniences like array operations in the syntax, etc. Some thought that researchers should want to write something quickly in a high-level system and have it run reasonably well without further work. They felt rapid prototyping was for production work, not research.

As soon as each new system arrives, people reinvent a lot of code, such as a FITS file processor. This is a foolish waste of time. We should be learning how to work between languages, and spend our valuable time writing new code.

The point was made that if a large project chooses a proprietary system for its basic calibration environment, it makes access to their project much more difficult and/or costly to data customers who do not have licenses for that particular product. Likewise, the project places its investment in coding at risk should the provider of that product cease to support it or cease to exist. ISO was cited as having committed this error in choosing IDL. Basing work on packages distributed under the GNU General Public License assures access for anyone.

No one (not even panelists) claimed to prefer the interactive environments of existing astronomical systems over the modern languages. However, the financial constraints on the programming groups (particularly IRAF) that support these packages are very tight and programming teams are small, usually much fewer than 5 people. Their home institutions generally hold the development of new application code for that institution as highest priority, regardless of the desires of the programming group and the community at large (even if grant money supports the project). Thus, a replacement of the IRAF cl is unlikely anytime soon, at least from NOAO. The command line interpreters for astronomical environments were designed for a special purpose and not as general programming languages. One panelist suggested that had they the opportunity to redesign the command line shell, they would choose a more modern design or even an existing general purpose language. This is the route chosen by the AIPS++ group, who use Glish.

Some groups are making their systems' core routines available in C or FORTRAN libraries. This would allow compiled code and interactive languages with dynamic linking capability to use those routines. However, it may be some time before all the applications of a given system are available in this way. Some limitations of the system designs, like storing arrays in disk files rather than memory, may make linking against these packages' routines from external programs less attractive to those who demand efficiency.

The major analysis environments strongly influence the work of many people, but rarely do their prospective users have the opportunity to influence the product during the critical design and development stages. These projects are community investments, and there is much expertise to be gained from the community. We encourage developers to hold open design reviews and to take seriously the comments they receive.

4. Conclusions

The list of desirable features is long and we are told that modernizing old systems is unlikely. However, several of the new languages (including Glish, GUILE, PERL, and Python) have or are gaining the numerical features needed for data analysis. What remains is the means to connect languages to each other and to existing code, and we can expect to see some experimental systems here soon. In the mean time, we should develop our new code with an eye toward the future. Codes requiring top efficiency or guaranteed longevity are best written in compiled languages. Modularity will pay off because tools such as SWIG[2] will read application source code and automatically generate wrapper scripts that make the routines available to a number of interactive languages. This means the programmer needn't write a main routine or graphical user interface, since the interactive languages provide them.

Because this is a fast-developing situation, we have created a WWW site[3] that compares many systems. Each has an entry in a feature comparison table, a textual description, and links to relevant sites. Several have programming examples that were prepared by the BoF panelists. We welcome additions and corrections.

[2] http://www.cs.utah.edu/%7Ebeazley/SWIG/swig.html

[3] http://lheawww.gsfc.nasa.gov/users/barrett/IDAE/table.1.html