NOTE 194
The SDCalc User Environment (Draft 1)

Paul Shannon, NRAO

June 25, 1996

Introduction

This note proposes a user environment for SDCalc. The emphasis here is *not* upon graphical interface design, nor upon the flow of data through a particular data reduction sequence. Rather, the emphasis is squarely on the user environment as a *programming* environment.

This is not to say that SDCalc will not provide, right from the start, a number of integrated GUI applications that run in the environment, and which appear to be stand-alone applications. But the programming environment will always be present, will always be easily accessible, and will be used by everyone. It is of such central importance that its design must be tackled first, before the specifics of the widget layout and data flow of a particular data reduction task. This approach is based upon the view that data reduction will always be at least partly unpredictable. There are many standard paths in data analysis, and we will start right away to turn them into GUI applications running within the environment. But there will always be a dynamic balance between anticipated analysis paths (which appear in a GUI) and the customized or novel analysis paths the astronomer inevitably needs in addition to the standard paths.

The design of the user environment is based upon two principles: programmability and transparency. The environment must provide powerful and convenient tools for constructing programs (programmability), and the environment must support the easy decomposition of existing functions, commands and programs into their constituent parts (transparency), so that they can be understood, and used as building blocks for new capabilities.

SDCalc will always be evolving, always taking on new ideas and new code, and new ways of navigating through data. We should view this as normal and expected behavior, and actively support it. The way to do this is to cast the user environment, not as a bunch of applications with some modest support for programmability, but instead as an execellent programming environment, in which a steadily growing (and steadily refined) set of analysis packages have been written, and in which they can be run.

Thus, we have two symmetric goals: make it easy to construct progressively more convenient, specialized and (often) graphical programs; and make it easy to take them apart. The environment must provide both programmability and transparency.

Overview: Glish, Tk widgets, Compiled Clients and Distributed Objects

The Glish interpreter is the heart of the user interface. It is always present, and futhermore the user will almost always be *aware* of its presence. While Glish is very useful on its own, as a full-featured, interpreted programming language with convenient whole-array operations, it is particularly valuable for SDCalc because of its extensibility. It is easy to add new capability to Glish by writing functions, and by creating (in compiled programming languages) new “clients' or “distributed objects”. These programs typically provide fast and efficient implementation of astronomical data reduction algorithms; they appear in Glish as simple function calls, indistinguishable from built-in functions. For example, the *sin ()* function is built-in to Glish, but the *fitGaussian ()* function was written by an aips++ programmer, and actually runs as a separate process.

Over the course of the last year, the popular Tk widget set has been added to Glish. Tk widgets are easily added to any Glish script; they can be used in a small and restricted way – for example, to open a table at the command line:

```
tbl := open_table (file_selection_dialog ())
```

or they may be woven together into a complicated user interface providing paths through some sort of data analysis.

The programmability and extensibility of Glish, combined with the integrated Tk widget set, create the core of a powerful and flexible general purpose programming environment. In the presence of a substantial collection of astronomical data reduction tools, compiled as Glish clients or distributed objects, Glish becomes a programming environment for single dish radio astronomy.

This one environment can take on many different personalities: in the hands of a seasoned and savvy analyst, who prefers typing text commands and composing Glish functions, the aips++ environment will be mostly straight text, perhaps with occasional plotting of vectors and images to plotting tools we provide, and which can be called from Glish.

At the other end of the spectrum, a novice user will be likely to start up aips++ in all its GUI glory: for instance, running a Glish script for single dish spectral line analysis, with buttons, menus, listboxes, etc., all cooperating to form a polished, well-focused analysis package, and all used without the observer paying any heed to the Glish command line.

Most of our users will fall between these two extremes, and will move back and forth between them according to their circumstances. An observer may begin with an integrated graphical application running in Glish, only to discover some data that needs further examination, beyond that which is provided in the gui. It is essential that the aips++ environment make it easy for the observer to choose exactly the level of abstraction, of data and of operation, that they want.

Just as users will move back and forth between using SDCalc GUI's and the command line, so will SDCalc developers do the same thing, as they create new integrated applications. In a sense, therefore, all SDCalc users are at least potentially programmers, and we must, therefore, make the environment a good programmer's environment from the start. There are 3 elements to this:

- Compose and evaluate code interactively.
- Gain ready access to already existing higher-level functions, to see how they get at their data, and to see what services they invoke.
- Easy access to every possible level of information about data structures and data operations.

These needs can also be expressed in terms of rapid prototyping: both the astronomical user and the application developer need the best possible tools to move from what exists, to what is needed.

This freedom is relatively easy to provide. The remainder of this paper explains how we can do this, and uses a small example from single dish spectral line data analysis to make it concrete, and concluding with a slightly more detailed description of the three tools mentioned above.

Example: Fitting a Baseline

A fundamental operation in single dish spectral line data analysis is to fit and subtract a baseline from a spectrum. We already have a GUI prototype of this in an assemblage of Glish script. These steps are involved:

1. An aips++ table is selected which contains the scans of interest.
2. One particular scan is selected.
3. Receiver and cal mode are selected.
4. The raw spectrum is plotted.
5. The region to be fitted (the tails of the spectrum) are marked.
6. A polynomial of specified order is fitted to the marked data.
7. The fitted curve is subtracted from the raw spectrum.

8. The baseline-subtracted spectrum is plotted.

If this program were to make some assumptions, the GUI could be simplified: it could present only a simple listbox in which the users chooses a scan number, and then a button labelled **BASELINE**. When that button is pressed, some automatic region selections are made, a plausible polynomial order is chosen, and all of the other steps are performed invisibly. The resulting spectrum is plotted for the user to see. Another button would save the information generated by this operation in some appropriate way.

Simple Customization: Fitting Repeated Baselines

Let's imagine that the observer is satisifed with the spectra that result from the simple operations of the previous section. Now he wants to apply the same operations to many spectra, and wants to do that automatically. He knows how to write a Fortran "do" loop, and a quick look at the online Glish tutorial, plus a little bit of experimenting in the Glish interpreter, convinces him he knows how to write a Glish for loop. The next need is to find out what the actual operations are that lie behind the **BASELINE** button. Here the transparency tool comes into play: when invoked for the **BASELINE** button (perhaps by setting a mode in the GUI), the button prints Glish statements into a Tk text buffer widget, for study, editing or copy-and-pasting into the observer's own Glish code.

Not all of the Glish commands revealed by the transparency tool will make complete sense to the observer. Perhaps the second argument to the *fitpoly* command is unclear. Here is where the aips++ Help Browser comes into play (see AIPS++ Note 187). This tool is designed to give the observer as much (or as little) information about aips++ software entities as they want.

After some experimentation, the observer's simple iteration code is debugged − though probably it is unlikely to be robust enough for general use. The observer uses it to fit all of the baselines in the dataset, and saves the results with an appropriate Glish command. He may then elect to wrap up his code in a Glish function, and save it to a file he keeps of his personal, or project-specific Glish code. If the observer wishes, this function can be linked to a new button, or menu item, dynamically created on the existing GUI and saved for future use along with the code for the function it calls.

More Complicated Customization: Parameterized Fitting Repeated Baselines

Many's the time when default fitting parameters, and default fitting regions, will not be adequate. The observer sometimes will want to average spectra, fit baselines to the result, and then subtract that baseline to all of the individual spectra, or some subset of the original. There will be many variations. While some of the variations are predictable and subject to easy parameterization (for example, a box car average of n spectra), and thus are forms that we will want to capture in a sensible GUI application early on, it must be easy for the observer to compose, perhaps attach to some widgets in the GUI, and save and run their own variations. The same steps outlined in the previous section – decompose stock code, get help on the underlying functions, cut and paste and edit stock code, test and debug the resulting functions – apply here, though the programming challenge is a little greater.

Please note that, at a certain level of sophistication and intended generality, this work of creating more complicated customization is indistinguisable from an SDCalc developer writing new analysis programs. The tools and techniques proposed here will serve all levels of sophistication.

Three Tools

The Glish language, and especially its ability to be easily extended with Glish Clients and distributed objects (usally written in C++ with the AIPS++ classes) – these are the backbone of the aips++ user environment. But they must be accompanied by some tools so that programmability and transparency are as good as they can be. There are three tools we need right away:

1. Transparency Tool. This only needs to work at the level of Glish functions – there is no need to interactively expand or decompose the objects and member functions of compiled code. This tool would take a single argument – the name of a Glish function – and would return a nicely formatted version of the functions text. Further decomposition could be achieved by calling the tool on a function which appeared in the first decomposition. (We may need to promote a style of Glish/Tk programming in which every gui action resolves to a single function call, so that decomposition of a gui action could easily achieved by the transparency tool as well.) It should be possible to redirect the function listing to the Glish tty, to a Tk edit buffer widget, or to a named file.
2. Evaluation Tool. We need a text editing buffer, probably a Tk text widget, from which text will be sent to the Glish interpreter. We should be able to send selected text, the whole buffer, or the function in which the cursor is currently located.
3. Online, context-sensitive help. Please see AIPS++ Note 187.