The AIPS++ User Interface

Douglas O. S. Wood
NRAO/AOC

December 18, 1991

Introduction

The AIPS++ user interface, like its software tools, will be constantly changing and evolving. It might take any form. You may have in mind the look and feel of your favorite user interface, someone else might have dreams of the perfect user interface yet to come. Because the user interface is so highly personal, and because there are different UIs that may be more appropriate for different tasks, the AIPS++ user should be able to choose the user interface he or she would like to use. Having this flexibility designed into AIPS++ not only helps insure that users benefit from an interface that best suits their needs and tastes, it allows AIPS++ to take advantage of new technologies (input/display devices and system software) as they become available in the future.

In this document I try to avoid specifying anything that is characteristic of a particular UI (e.g. text based, menus, dialog boxes, flow charts, etc.) as much as possible. A discussion that is based on a particular UI is both biased and incomplete. I would like the UI specification for AIPS++ to be as independent of the details of a particular UI as much as possible.

Looking at the project from a practical point of view, however, it seems clear that the first AIPS++ interface to be written is likely to be text (ASCII) based. Clearly such a UI has the advantage that just about any device, from the most brain dead terminal to the most high-powered visualization workstation, will be compatible with it. Personally, I would prefer adopting a bit-mapped, windowing system as the base level UI so the design team could retire their VT100s as boat anchors. Although some may feel that there are advantages to a text based system (in particular when writing/programing procedures or 'scripts') it is my sincere hope that the designers of AIPS++ will not simply put other user interfaces "on top of" the text based system. Such a strategy is usually only marginally successful. An example of this is the mailtool provided by SUN with their SPARCstations. Although the mailtool is "graphical" it has an approach that is clearly mired in the fact that it is only a "front end" which composes text based `mail` commands. For example, the mailtool still uses the .mailrc file to create aliases and adjust all of its important mail related parameters.

Some of mailtool's parameters, however, are controlled via a graphical user interface using the mouse. This is inconsistent and reveals a 50% effort at a complete graphical user interface. Here are some examples of where mailtool is lacking as a graphical tool:

- icons or pictures should be used to guide the user in making choices or issuing commands (such as send mail)
- clicking somewhere (on an icon/picture of an address book?) should allow one to quickly lookup or add a name to an address book
- I would like to check off their names in my address book or send a message to everyone in a particular address book
- mail messages should include graphics, and sound (which are both possible on the SUNs)

Finally, I will try to avoid comparisons with the user interface of AIPS1, but there are always lessons to be learned from a previous attempt. Personally, I think one of the most frustrating things about using AIPS1 is the time spent setting and reviewing inputs to tasks. This would have been much worse had we not had the save/get and tsave/tget mechanisms. I think most users of AIPS1 found difficulty in getting inputs right and in being confused about what to set them to achieve the desired result. In a data reduction and analysis problem such as AIPS++, most of the work the astronomer does *is* in choosing the right inputs. Thus, *the design team must focus on the parameter editing and control system to make it as user ergonomic as possible*. This is where a graphical user interface has so much power. It can give visual clues and feedback as to what parameters need to be changed, it can present icons, menus, and other graphics to help users recall what they want to do, and it can present this information in an organized fashion with unnecessary detail hidden if desired by the user.

Principles

In this section I outline some of the guiding principles that any proposed AIPS++ user interface should follow:

Consistency

The most important aspect of the user interface is consistency. This is especially important because AIPS++ will undoubtedly involve hundreds of separate tasks and hundreds of input/output variables. A consistent user interface

- makes it easier for users to learn new tasks,

- makes it easier to remember how to use a task that is used infrequently, and
- reduces the amount of time users spends trying to remember unimportant, unscientific details in order to get their work done.

The system should be designed to help insure that as modules are added and changed in AIPS++ the UI remains consistent to the user. AIPS++ should be designed to make it easy for programmers or scientist/programers to add tasks that conform to the UI guidelines.

Recognition rather than recall

Users should feel that they are simply recognizing rather than having to recall a command, a parameter name or whatever. This is the principal advantage of a menu based system: select a menu and one simply has to recall which item in the menu they want, they don't have to remember a text based command that activates a particular command. The user interface should *help* the user with a minimum of memorization required.

Hide complexity

Unnecessary detail or complications should be hidden from the user when ever possible. For example, in AIPS1's LISTR, once one has set OPTYPE to 'SCAN' nearly all of the other inputs to LISTR are irrelevant, yet they have to scroll through them and might wonder if any do apply. If one is reducing VLA data, one should not have to view and reject andy inputs that are purely VLBI or single dish parameters. This might be accomplished with some sort of configuration set up for a particular UI or an entirely different UI for different telescopes or data reduction tasks.

User centered approach

If the design is about to call for some restrictions on the user interface (such as requiring single quotes around strings or limiting task names to 6 characters) the following question should be asked:

"Is this being done for the convenience of the programmer rather than the user?"

If the answer is "yes" the restriction should *not* be imposed.

Modeless operation

Modes (e.g. POPS for loops in AIPS1) are definitely to be avoided. The user should be in complete control at all times. They should have the power to stop any procedure (i.e. break out of a processing loop or abort a task) at any time.

Intelligence

If the user specifies that she wants to image a source that is not in the currently selected database, the system should inform her there is no source by that name. It should offer a suggestion as to how to fix the problem and offer on-line, context sensitive help to fix the problem. The user interface should also be intelligent enough not to let a user make undesirable parameter choices or choices that are clearly in conflict.

Feedback

User's should receive visual or other feedback that their actions have had the desired effect. The messaging system in AIPS1 is an example of this, but it must be extended further. It should be possible for the user to specify the amount of feedback that is desired (i.e. the verbosity). Messages should be clear and concise. For especially long tasks the user should be informed of the progress (e.g. percent of calculations completed, etc.).

Aesthetic integrity

The interface should be aesthetically pleasing and well integrated. Although this is particularly true for a graphical user interface, it is also true for a text base one. Users should be free to personalize the interface as much as possible to suit their needs and desires.

Separate tools for separate jobs

Tasks that do different things should be kept separate. This is were a tool based system is much more powerful than monolithic tasks. If it is possible to write the new function as a tool it should be done that way, rather than loading it on to an existing application.

Helpful forgiveness

If different users want to do things differently the user interface should accommodate them. For example, the interface should allow users to compose commands differently as long as the meaning can't be confused (e.g. 'inname = test' *vs.* 'inname test').

Error messages

Because AIPS++ involves the rather complicated manipulation of numerical data, the error reporting system of AIPS++ must receive a significant amount of attention. First, the user should be told if they are about to do something that the program is either going to object to later (e.g. conflicting parameters) or might produce potentially disastrous results (e.g. bad choices for parameters). In most cases the user should be allowed to enter risky

territory if they want, but not without a warning that they are potentially making a mistake.

The second form of error message is that produced during execution. These messages should be clear (not just "I-O Error 6 in ZSDKQU"). They should we written using consistent terminology astronomical terminology and plain English. They should also suggest a possible solution to the problem (e.g. "decrease the cell size" or "have you reached the end of the tape?") or provide a context sensitive link to an on-line help facility that will explain possible reasons for the error in more detail.

Parameter specification

Most of the time users are setting and resetting parameters. Of course some of this can't be avoided and that is where the 'art' comes in. But AIPS++ should make every effort to prevent users from having to specify and respecify the same things over and over again. Some solution might be for AIPS++ to provide a set of tools or system that can be used to define and edit parameter definitions and save them for later use. These 'parameter definitions' or 'parameter expressions' could then be used throughout an AIPS++ session. A simple example of a parameter definition might be a list of the names of all phase calibrators in a database saved under the name 'pcals'. Many of these definitions will look like database queries. Some other examples are:

- The interior of some volume defined by a threshold level in a 'cube'
- A polygon (or pixel mask) on a continuum image
- All observations made during a particular phase of a time series (e.g. the "on" phase of a pulsar
- A slice through a data cube or image
- A list of antennae
- A stokes mask (e.g 'RR' in AIPS1)
- All data flagged by the modcomps

One might even be able to write expressions that combine parameter definitions using simple operators (e.g. AND, OR, NOT, etc.) to construct expressions like "all calibrators observed at X band that are not 3C286." Users should be able to attach a brief note to describe each expression so they can remember what to use it for later.

Some examples of where these personalized parameter definitions could be used are:

- To select the calibrators to use for a calibration operation
- To identify what data to use when making an image
- To select all data from IF 2

- To choose the 'line free' channels of a spectral line data cube
- To specify the 'blank' region of a moment map
- To select a portion of an image to make a contour plot
- To specify contour levels
- To flag some data

The primary advantage here is that once the user had learned how to use the tools to generate, display, edit, and combine parameter definitions, she can use the definitions again and again. Regions would also assure a greater uniformity of selection techniques throughout AIPS++.

Automation/Batch Processing

It should be easy for the user to ask AIPS++ to repeat a particular set of calibration/imaging/analysis operations on a different set of data or for a different set of parameters. Ideally I would like to be able to 'say' to AIPS++:

"Now that you have seen how I calibrated the data for line1, do the same thing for all the other sources observed in this line but at different velocities/frequencies."

"Repeat the procedure you saw me use last week on this week's data."

"Make the data cube over again, but this time use the new continuum image I have just constructed and increase the taper by 30%."

"Calibrate and image the data using each of the different bandpass files I have just created."

I realize that I probably will be required to write these scripts either from scratch or by editing a copy of a history file. The batch processing system command language should include process control constructs like for, while and if-then-else. Users should have tools to debug and edit their scripts. They must be able to intervene at various times in the execution of a script for some tasks (such as data editing) which may require some interactive 'judgement' calls that are difficult to program.

The user should also be able to 'pause' the procedure to examine the results so far to see how things are going and modify the course of the processing (e.g. "stop cleaning now and go on") if necessary. There should also be a mechanism to send the user a notification (perhaps a mail message) that the background process has finished.

Testing

It cannot be emphasized enough that the principle ingredient for a successful user interface is testing with real users. The programers and designers should observe users *as they actually interact with system.* They should passively watch how new, experienced and expert users interact with AIPS++. If users like or dislike something about a new version of the UI, if enough are puzzled by something or expected something to happen when it didn't, AIPS++ should be rewritten to accommodate them.

The programmers staff should have in their hearts the conviction that the only thing that matters is the satisfaction of the users. They should be willing and eager to make the system perform the way the users want it to perform. **User satisfaction is the sole measure of their success.**

On-Line Help

Extensive, but not a core dump of the manual

The on-line help must be extensive, well organized, well indexed, and hierarchical. There should be several ways to navigate the help information, e.g. via a road map, a list of topics, an index, by searching for some text, etc., etc. On bit-mapped systems the on-line help should include graphics when appropriate. For these reasons, Unix `man` is not an acceptable help system

Context Sensitive

For example, the help system should know that if a particular error message is generated, the user may want to be directed to the help information that will explain that error. If the user is reducing VLA data they should be able to tell the help system hide any discussion of VLBI data reduction techniques. The on-line help should be completely integrated with the printed manuals with extensive cross-referencing.

Modeless

Users should be able to view the help information while they are using AIPS++. Help should not be a separate mode that is different than using the program itself. They should be able to read and browse the help system (in another window) while they do their work.

Users shouldn't need to know help to get help

The help itself system should be structurally simple and self-explanatory. It may require a few words of instruction, but it should never be more complicated than the application it describes. It should use a familiar metaphor, such as a book for its basic organization.

Help the user frame their question

Framing the question is often the first problem. Users may want to ask the help "How do I calibrate continuum data" or "What do I do if my image is blank?" or "What is the cell size?"

Search and browse

It should be possible to search the help information for a search string, topic or keyword. The browser should present a road map or a list of topics so the user does not have to type a particular keyword to navigate the help system.

Customizing

Users and programmers should find it easy to edit the help file to document their work and to customize it to their tastes or needs.