September 12, 1991

To: Bob Hjellming

From: Rick Fisher

Subject: AIPS++ Requirements

Here's a first cut at putting my thoughts on paper on what I'd like to see in AIPS++. I don't have much experience with U-V data, so most of my examples, and certainly my point of view, are from the map/spectrum/time domains, although I suspect that most of what I say will have broader relevance. Since concepts are extremely difficult to communicate in the abstract, let me illustrate each 'wish' with a specific example with the understanding that there may be better ways to implement the general idea. If I illustrate a point with a counterexample, I don't mean to slight any particular data reduction package.

On a new observing project my data reduction style starts out very interactive and evolves to a set of complex operations to be performed more or less automatically. If I had a choice between a set of data reduction tools and a powerful but fairly rigid data reduction pipeline, I would take the set of tools. My ideal is a tool set and a well maintained library of data reduction procedures (programmed groups of tool applications) that I can copy and modify to suit my particular needs. Another ideal would be that differences between interactive and automatic data reduction be minimal. It should be easy to assemble a set of successful interactive operations into an automatic procedure.

There is no better way for a new observer to learn a data reduction system than to experiment with its components. Even just being able to look inside a procedure is much more instructive than user manual descriptions of black-box procedures. Necessary to the learning procedure is good feedback. What does the data look like before and after an operation is performed? What were the relevant parameters that affected the operation? Where is the new data? Did the data size change? Windows and graphical techniques can be of great help in the feedback area.

User command input methods are probably much more controversial than data display techniques, mainly because there is no universal best way. We tend to like what we are familiar with which impedes progress on user interfaces and makes agreement nearly impossible. The best I can do is to give you some impression from recently learning a couple of new (to me) software packages. A revolutionary new user interface is likely to be a big mistake. Let's take the best features of the astronomical and other packages now in existence and leave lots of room for evolution. Several general things are quite important to a good user interface: coherence and consistency, modest hierarchy of functions, lots of on-line help and table-guided setup, and a large percentage of tools whose uses are nearly self-evident to new users. In many cases, more than one way to invoke the same operation is entirely appropriate. In the end, as a user I'll go to considerable length to learn a new user interface if there are plenty of useful data reduction

tools in the package worth getting to.

Consistency or coherence in a user interface doesn't mean slavish adherence to a particular format. Some things lend themselves naturally to buttons and a mouse like 'print', 'scroll', 'select', etc., and others are far easier as a typed command like "spect1 = (spect2 - spect3) / spect3." Moving icons around to produce the effect of a typed command can be a real drag at times. Also, putting mouse commands into a macro seems to require a command line version of the mouse commands. At least I have not seen a satisfactory way to program mouse commands. The big advantage of GUI input methods is that the choices can be presented to the user at the same place where the selections are to be made. When done well, the options can be quite obvious.

Thinking about how observers visualize their data is probably a useful exercise when designing the tools or the user interface. My mental model is data chunks and operations involving one or more of these chunks at a time. Since data chunks are static and operations are transient, I probably think a lot more about data than about the operations. After an operation is completed, I generally forget about the operation itself, but I'm still interested in the data that went into and came out of that operation. In a complex data reduction layout, I'd rather worry only about the most recent form of the data and ask the package software to keep record of what I've done in case I want to check my logic or back up to an earlier stage and take another tack. This record could be the commands in a procedure, a log of interactive commands, and/or a graphical data-flow diagram.

This emphasis on data rather than operations seems to run counter of the emphasis on tasks, verbs, procedures, etc. in current reduction packages, so maybe I'm proposing a fairly big change in the way the system is presented to the user. When we write a manual for a data reduction system we don't describe the data, because that's not part of the system. But to the user the system is only a means to an end, not the end itself. None of this is meant to imply that operations aren't a key element in a data reduction package. To use an analogy, even a carpenter wouldn't describe her house in terms of the tools used to build it, but she is likely to describe to materials that went into it.

All that said, here are some preferences based on experience with data that wasn't anticipated by any existing data reduction package.

I'd like to see a fairly extensive base of tools which are as universal as possible. For example, if I add two two-dimensional arrays, it shouldn't matter whether the arrays are sky maps, gridded U-V data, or pulsar spectra. This makes the use of tools for unforeseen applications a lot easier. More specific tools can apply the universal tools and add what is known about the specific data being operated on. For example, the rotation of an image of an object around a specified RA/Dec could use the universal array "rotate" operator and add what is known about the real coordinates of the map to assign coordinates to the resulting array. If I want to use

"rotate" on, say, pulsar data, I shouldn't have to undo the parts of the map rotate operator that don't make sense to pulsar data to use the basic the array rotator.

Most of the applications that I can think of allow data to be put into discrete arrays of one or many dimensions. Hence, array manipulation operators are probably a large part of the tool set. That's probably why a package like IDL is so readily applied to astronomical data. One place where AIPS++ can excel over a more general package is in the convenience it can offer the astronomer in keeping track of data properties that are unique to our applications. If you do an image rotate, it should be very easy to determine what RA/Dec a pixel in the new array corresponds to. To strike a balance between universality of an operator and the desire for automatic housekeeping in specific applications, I'd like to see a way devised (if it doesn't already exist) for me to attach properties to a general data array that make the specific application that I have just devised more convenient. In the case of real-world coordinates, I'd like to be able to specify a transformation equation between pixel number and some variable like frequency or pulse phase, and have the various operators keep track of my real-world coordinate when they are applied. Each dimension of an array should accommodate many transformations from pixel number to real-world quantities, e.g., sky frequency, intermediate frequency, velocity (heliocentric, relativistic, optical definition, ...) at the same time. Also, real-world coordinates that are not parallel to the array dimensions must be accommodated in the user-specified transformation.

In some cases, one or more of the dimensions of a data array will not correspond to some continuous coordinate. For example, one dimension might be a phase number of a switching cycle (signal, reference, cal-on, etc.). Another dimension might be an IF number with arbitrary center frequencies, etc. Many array operators (add, multiply, transpose, etc.) make sense for this kind of data, so I'd like to see this sort of data chunk accommodated by AIPS++.

Subarrays of data should be as easy to access and manipulate as full arrays. The IRAF notation of 'arrayname[1,*,3:56]' to access element 1 of the first dimension, all of the second dimension, and elements 3 through 56 of the third dimension is pretty good. There may be others that are as good or better.

At least in the command language part of the user input interface, the very heavily used unary and binary array operators (add, multiply, etc.) should be assigned to operator symbols (+, *, etc.) where such assignments are pretty obvious. Maybe just (+, -, /, *) are enough. I seem to use these a lot. Complex expressions should be permitted with these and other operators, e.g., map3 = (map2 - map1) / map1. Subarrays and matching lists of arrays should be legal operands with the operator smart enough to figure out whether the operator/operand combination is legal, e.g., matching list lengths, (sub)array sizes, and numbers of dimensions.

For more complex operators, a shallow hierarchy seems preferable to a proliferation of operators on one level. My main rationale here is to make quick work of finding an operator

that looks like it will do what I want. Rather than wading through many many keywords like 'mean', 'median', 'addclip', 'average', etc., I'd like a relatively short list (fits in half of a modern display screen) of pretty obvious categories like 'combine', 'fit1d', 'fit2d', 'shift' that narrows the search in a big hurry. Whether there are a few major operators with lots of options or a lot of more restricted operators is not as important as being able to find them with some fairly intuitive search strategy.

Unfortunately (from the complexity standpoint), many operators may have a zillion options and parameters. Three ways of setting these options make sense in different contexts. They are: standard defaults, local specification of a few options/parameters when the operator is invoked, and user-named setup tables for each operator. I like local as opposed to global parameters. They are a bit more work in the short run, but I prefer the assurance that I'm not going to get surprises from unexpected sources.

If I had to make an enormously sweeping statement about a user interface it would be something like "Give me the feeling that I know where to find things, that I can review at a glance where my data has come from and what it looks like, and that I'm not going to get blind-sided by too many things that I've forgotten about or that someone else thought I didn't need to know."

In the data -> operator -> data model, keywords like fetch, get, store, and other implementation specific concepts are considerably reduced. I think that is a good thing, as a rule, but there may be cases where the system cannot read the observer's mind well enough, or the user might want to tune the system for better efficiency. One idea that comes to mind is that I want to tell the system that I expect to use a particular data chunk more than once so don't bother reading and writing it on disk every time it is used; keep it in memory, if possible. Something like assigning an image to a register would be one way.

Access to header values should be easy at high levels in the user interface. My specific thought is something like wanting to scale a pulsar data array by normalizing it and multiplying by the system temperature value in the header. I'd suggest a very small but general set of access functions, one each for real, integer, and string values, that allow access to all present and future header parameters by, for example, FITS name, e.g., tsys = head("TSYS").

Sending and retrieving data, plots, listings, and so forth to and from I/O devices shouldn't be complicated. There should be a simple way for each site to configure their system so that all devices can be presented to the user in a menu or routing diagram in which the user may establish temporary connections or data may be quickly sent to a device at the push of a button. The UNIX method of treating devices like files is a step in the right direction, but we don't want the extreme generality and setup complexity to be evident to the user or even to the AIPS++ site system administrator.

I don't foresee the day when one data reduction package serves all my needs, nor do I even

wish that it might happen. Hence, make transfer of data to and from other data reduction packages as easy as possible. It should be possible for the user to have AIPS++ open in one window and some other package open in another and be able to toss data back and forth without worrying about what format it needs to be in to make the transfer.

A good relational data base for retrieving any sort of data that can be handled by AIPS++ would be great. The business of retrieving stuff by scan number or trying to think up clever names for data cubes to tell what's in them can be pretty trying when there is lots of data or the processing is complex. Retrieving data by all sorts of parameters is desirable, e.g., source name, position, IF number, time, date, array configuration, center frequency, velocity range .... Combinations and wildcards are very useful. Ideally, retrieval would be possible for data of any age (last record, yesterday's data, or last year's data) without the user needing to know something different for each era. If data isn't on disk when requested, the archival tape/disk/whatever number and location should be returned to the user.

Additions to the local version of AIPS++ should be easy for the user who knows C, FORTRAN, or C++. This means well defined access to data arrays and headers, good firewalls to protect the system, but not necessarily the data, from coding mistakes, and simple but powerful fill-in-the-blanks 'make' files for compiling new code and linking help files and other user-interface features. Writing code for a friendly system that takes care of display, peripheral I/O, and data housekeeping should and can be fun.