AIPS++ Note 165:
High Level Requirements for the AIPS++ User Interface

Mark Holdaway

National Radio Astronomy Observatory

January 13, 1994

Introduction

This note and Chapter 1 of Note 163 cover very similar material. This document actually preceeds Note 163. This is a more formal presentation of what is required, and Note 163 is more of a vision of what should be available to the astronomical user as he sits down to the terminal.

The user interface consists of the sum of how information flows from the user into AIPS++ and how information flows from AIPS++ to the user. This is quite broad and includes setting up the environment, creating new objects, selecting existing objects, making associations between objects, getting help, setting input parameters, initiating and controlling tasks, reporting on task progress, getting objects and atomic data results produced by tasks, feeding these objects and atomic data results into other tasks or objects, invoking object's methods from the UI, viewing object contents and atomic data results both as they are being generated and after their generation is complete, and then being able to figure out what it is you just did. Many of these activities lend themselves to graphical representation, but the user must also be able to perform these actions from the command line, ie from a "dumb" terminal, from the command line in a windowing environment, or from a batch script. I present here some of my high level ideas about each of these areas.

Setting up the AIPS++ Environment

When starting up AIPS++, there are a lot of configurable things which must be defined. If the user is in a windowing environment, they have to specify which windows they want: message window, help window, graphics window, table browser. The user must specify the level of messages they wish

to send to the message window and to the history Table. They must specify which UI they want to use, which printers, cpu's, and disks they want to use, whether they want the programs to interactively request information or not, which projects to work on, etc. I'm sure this list has many other items.

AIPS++ will have a configuration file, .aips++, which will provide the default setup, which can be edited prior to running AIPS++, and which will record changes made to the configuration while running AIPS++. One should be able to change the configuration of AIPS++ without "restarting".

I am assuming here that AIPS++ will be run from its own shell and have its own command line interpreter. However, unix commands and utilities must be accessable from the AIPS++ shell.

Selecting Existing Objects

Just as AIPS Classic maintains a catalog of UV data and images, AIPS++ will maintain a catalog, but it will be able to contain many different types of objects: MeasurementSet, VisSet, TelescopeModel, MeasurementModel, Image, MultiFacetedImage, user defined ResultTables, and text files. The user must have a powerful means of selecting which object they are interested in. The entire catalog could be displayed and a single object could be selected. Selection could be made on object type, like UCAT and MCAT in AIPS Classic. We need the capability of selecting multiple objects which might be used in the input to a task all at once (the task will take a list of objects) or iteratively (a task or series of tasks will take a single object, but the tasks are repeated once per selected object). Sun's mailtool is a model for graphical selection of multiple objects from a list. The command line interpreter must support looping through objects and sorting of objects.

In order to access the actual data files via the operating system, we must also have a utility which indicates the files which map to a particular AIPS++ object as well as the resources they take up.

Associations Between Objects

Associations between two or more objects can be modeled either via the equivalent of pointers to the other objects within one or more of the objects, or as a container object which has references to its constituent, heterogeneous objects. Both models of associations will be useful in AIPS++. Currently,

the "pointer" model is being used for the MS-TM and MS-MM associations. The "project", a hierarchical organizational tool which can contain heterogeneous objects related only in that they deal with the same astronomical object or astronomical question, is an example of the container model (the "project" class does not exist yet, for now think of it as a directory in a UNIX file system). The user will need to make and break associations of both types. Furthermore, the user will need to be able to find out which objects are in an association. Both of these activities lend themselves to a GUI, but each operation is also required from the command line.

Setting Input Parameters for Tasks

Global parameters are no longer *en vogue*. The next step is to associate parameters with each task. However, I advocate a step further: to also associate parameters with each object. For example, the *ReceptorGains* telescope component would have a parameter called "IntegrationTime". Any calibration program would then find out what integration time was specified in the telescope component rather than request that information from the user upon task invocation. Of course, the user needs to specify the integration time in the telescope component during its construction, or change it at a later time.

We might want to look into the possibility of having a configurable "project-wide" parameter default block which can be fed into the input parameters to a task by an explicit consious action taken by the user.

A very few parameters will still be global, and most of these belong above in the *Setting up the AIPS++ Environment* section.

Many parameters will be associated with particular objects, and these will not need to be input unless their values needed to be changed.

Many parameters will still need to be task specific. These parameters will be associated with each task and will be recallable.

Astronomical objects will not be input parameters; rather, handles to get the astronomical objects out of the catalog system will be passed as input parameters.

When querried, each task and object will list the required input parameters, their types, their default values, and a brief description of what each parameter does. Such a list must be easily edited by the user for easy submission of input parameters from a script in batch mode.

At least two modes of entering parameters needs to be provided: a simple "KEYWORD = value" approach for the command line or scripts, and a "form filling" approach for a GUI. The software which controls parameter loading should have access to a file which contains default values, acceptable ranges of the values, or acceptable enumerated values. For "KEYWORD = value", the input value will be checked against the acceptable values after input, while the GUI form will be built to indicate the possible choices.

Some highly interactive tasks will require a custom built GUI. Custom built GUIs will require a great amount of communication between astronomers and the GUI programmer. With limited manpower, custom GUIs should be avoided unless its utility can be clearly demonstrated for a particular task.

Fairly extensive help should be available from the command line interface. The help may cause a new window to pop up, depending upon how the user has configured the system. Help will take many forms: *apropos* will search the ¡summary¿ documentation extractor directive in the top level tasks and procedures; a lower form of *apropos* will direct the user to object methods which are available from the command line interface; *algorithm* will access the ¡algorithm¿ preprocessor directive and return information about the algorithms used; help will need to be cross-referenced; help should direct the user to the code location so the user can look at it or get it for modification; some sort of tutorial will be required from help.

Initiating and Controlling Tasks

I will consider the simple case of starting up a top level task on one machine. At some point in the future, we will need to deal with a primary task initiating subtasks asynchronously on other machines or processors.

Once the input parameters have been set, the primary task can be initiated. The primary task may initiate secondary processes. For example, a maximum entropy task may initiate a small graphics window which displays a continuously updated graph of several diagnostic parameters (total flux, off source noise level, entropy, fit) as a function of iteration number. These data, or *output parameters* must be in a standard form and must be created in a standard manner within tasks (see below).

There must be a standardized way for a task to request further user input after the task has been initiated. For example, a user may want to check on the sigmas of parameters in a fit which occurs on many time slices

of data. He would set a global environment variable to "INTERACTIVE", which would cause the task to display the fit parameters and sigmas either graphically or with text. Following the display, the task would prompt the user for an "OK", an "ABORT", or a directive to enter "BATCHMODE". Once in batch mode, the task would no longer display this information or request user intervention. The prompting must be configurable: prompting for some things will interactive, while prompting for other things will be "batched-out". Global BATCHMODE will result in no prompts.

Similarly, there must be a standardized way of feeding information into a task operating in batch mode. For example, if a CLEAN is running non-interactively and the user wants to change the number of iterations, the user must be able to send that information to the task, and the task must be programmed to accept the new information.

The user must be able to abort interactive tasks without killing the AIPS++ shell and environment.

Reporting on Task Progress

Tasks which take a long time to complete will need to send information to the user indicating how far along they are. There are two ways I can think of to do this: simply send some text to the message system stating how much more work needs to be done, or send a number to the outputs block (see below) which can be plotted on the screen, graphically showing the program's progress.

There are many issues to be sorted out about messages. The message system design allows messages to be send to arbitrarilly many "message sinks", which can be Tables or screens. The message Tables could be attached to other objects (the object's personal history), or the message Table could be its own object (a global history file). Each message sink can filter out the messages it is not interested in. One or more message sinks will be a screen or message window. When running AIPS++ on multiple machines but controlling them from a single screen, one probably wants to have different message windows for each machine. Some very long tasks should probably have their own message windows. It is unclear when these message windows should go away. Probably the best thing to do is to design a highly configurable system which allows the user to define the message window action they prefer, and to make setup easy to understand and to change.

Getting Output Parameters from Tasks

There are some verbs in AIPS which set adverbs which can then be used as inputs to another task. This behavior should be formalized and extended. Just as there is something like an *inputs block* which task A will get when it is initiated, as task A runs, it should update an *outputs block*. The outputs block will have some information which is continually updated (for example, the number of clean components cleaned so far and the flux the algorithm has reached, even showing tickmarks depicting the start of major cycles), and some information which is calculated once at the end of the task. The philosophy for using the outputs should be: let no useful information just scroll off the screen again, put that information somewhere it can be accessed automatically. For continuously updated information like the clean progress data, there should also be some kind of *notify* which will tell a graphics process to come and get the latest clean progress information.

By default, all of the non-debugging outputs information should be written into the outputs block. The user should also be able to edit a form for each outputs block which indicates which specific outputs are desired and which ones should not be saved.

The outputs block from task A cannot be directly accessed by any task B, which can only access $B.inputs$. $A.output$ can be accessed from the command line, or perhaps graphically, and the values of $B.inputs$ can be set equal to certain values from $A.outputs$ from the command line. Likewise, the values in $A.inputs$ can also be accessed form the command line to be given to $B.inputs$.

The structure of the inputs block and outputs block will be quite similar. While the contents of the inputs block is clearly specified by what is required to run the task, the contents of the outputs block is not so clearly specified. For example, the cpu time and real time it takes to run the task are not required from an astronomical point of view, but if they were included, it would certainly help out people like Eric Greisen who is in the process of optimizing MX. We should make clear guidelines for the contents and structure of the inputs and outputs blocks.

Invoking Objects' Methods

Most of the functionality in any software system is not accessable to the top level user who is only able to access linked *main* programs. On many

occasions in SDE, I have written a shell *main* program around a lower level routine such as *imagemultiply* so that the top level user had some of the functionality available within programs. That level of functionality is desired at the level of the command line and non-compiled scripts.

We probably don't need to be able to access all the public methods of all objects. Many methods of many objects primarilly perform operations within subsystems and as such do not belong in the public interface to that subsystem. There is no mechanism in C++ to specify subsystems and the public interface to those subsystems, and in AIPS++, we only have vague ideas about what those subsystems are and what their public interfaces should be. Once we have a better idea about how this will work out, we will be in a good position to identify which classes and which of their public methods must be accessable from the command line. I think it would be a mistake to allow all the public methods of all classes to be accessible from the command line, but it must be trivial to switch a method from *inaccessible* to *accessible* or back.

We obviously need a nice GUI to display the allowed methods of an object, to provide help with the required parameters for each method, to accept input, and to invoke the method. From the command line, we will need the same. The command line syntax for invoking an object's methods should be the C++ syntax.

If this requirement is unreasonable, it must be made known immediately, as much of the imagined system depends strongly on this.

Creating Objects in AIPS++

Most objects in AIPS++ will be created by tasks, by the methods of other objects, or by functions which can be accessed from the command line. However, the user must also have the ability to create an object "out of the vacuum". The prime example of such an object is a Table of user specified results. For example, the user might be interested in all sources brighter than 3σ in each of many fields. The user could create a table with column names "RA", "DEC", "TOTAL FLUX", "PEAK FLUX", "MAJOR", "MINOR", "PA", "IMAGE". The data in the columns would be obtained from the output block of some program like *search and destroy*. "IMAGE" specifies the handle of the image in which the source was found, and the other parameters are descriptive of the sources. This example also underlines the

necessity of being able to invoke objects' methods (the table's methods) from the command line as well as the need to get at the calculated results of a task or procedure from the command line. These capabilities will allow an astronomer to specify the quantities they want to glean from one or more observations, and will provide a personal database from which they will be able to draw astronomical or statistical conclusions. The main point is that the software which processes telescope data is the same software which collects, plots, and analyses astronomical information from the processed telescope data.

Viewing Data

There must be an easy way to display the structure of an object and its Tables. This includes associations with other objects and optionally the heirarchical structure of tables within each object. This utility will not show the actual data and is therefore different from the Table Browser.

Line drawing and 2-D graphic has been largely ignored in AIPS++ so far. While this is not the sexiest way to look at ones data, 2-D graphics are still the most common and most general way of viewing a wide variety of data.

I have already mentioned a few examples in which a small graphics window (a bit larger than a *performance meter*) associated with a particular task will pop up and display the progress of an algorithm. Such an application requires a standardized, simple, but general interface to the 2-D graphics system. The obvious place to start is to interface with Tables, but that will not be sufficient. Bob Hjellming's recent note on mathematical requirements (draft Nov 11 1993) is a good place to look at the operations which will be required. I would advise not worrying right now about any of the operations which border on 3-D displays (ie, spin, brush, movie, perspective displays).

It would be nice to have other displays which indicate the algorithm progress: for example, AIPS++ could optionally display an image which indicates where the clean components are being taken from. When clean components begin appearing in regions of the image where the user believes no emission should be found, the user can terminate the clean.

I would like to see the properties of a graphical display controlled by the user rather than by the programmer: the user should be able to click up a menu which gives the option of changing the axes to linear or to log, to change the font, positioning, and contents of the labeling, to change the color

of the data displayed, and actually be able to move data points on the graph while changing their values in the underlying database. (When the data have been changed in the database, we need a way of indicating what the original value was, and of undoing the change.) It is also important to be able to get a handle to exactly which data in a table or array corresponds to a particular point on the graph.

We also require something like the Table Browser. There will probably be many applications which are based on the Table Browser but have tailored interfaces. For example, since the messages will be logged in a Table, the message viewer would be a restricted type of Table Browser. One could search or sort on time, object name, task name, project, etc.

Scripting

Line editing and command recall is required. The style of the line editor will be set in the .aips++ defaults file.

The user would like to be able to write powerful scripts which can be tested quickly, ie, without compilation. This would serve as a prototyping environment for complex algorithms if the user has access to the data objects and thier methods from the command line. A history mechanism will log all of the user input from the command line. An edit window should allow one to edit the last N issued commands, removing commands that ended up being mistakes, generalizing inputs from specific numbers and strings to script variables or values obtained from output blocks, and adding commands, resulting in an AIPS++ script. Template scripts, complete with places for recommended documentation, would also be available.

Loop, if, case, and while constructs need to be supported. The scripting language needs to be powerful enough so that parameter names or task names could be contained in a script variable:

```
if ($PEAKFLUX > 1.0)
  CALTASK=strongcalib;
else
  CALTASK=calib;

......
other stuff
```

```
.....

$CALTASK < $CALTASK.inputs
```

Unix commands must be accessible from within AIPS++ scripts.