The Astronomical Image Processing System

Eric W. Greisen

September 12, 1988

National Radio Astronomy Observatory
Edgemont Road, Charlottesville, VA 22903–2475
804–296–0211, x348; TWX 910–997–0174; Fax 804–296–0278
`egreisen@nrao.bitnet` or `egreisen@nrao.edu`
`6654::egreisen` or `...!uunet!nrao1!egreisen`

Introduction

The Astronomical Image Processing System, *AIPS* for short, is the National Radio Astronomy Observatory's contribution to the class of major image processing systems. It was one of the earliest of these systems and has grown to be, for a number of years, the largest and most widely used system in astronomy*. That success is due in large part to the scientific success of the telescope it was created to support, namely the Very Large Array (VLA) [1], built and operated by the NRAO. However, I like to think that some of the success is due to the design and capabilities of *AIPS* itself.

In the present paper, I will describe *AIPS* and the design considerations which have caused it to take its present form. I will also attempt to give some idea of the directions in which *AIPS* is presently evolving. It would be easy, with the advantage of hindsight, to pretend that we designed *AIPS* in its present shape from the beginning. That would not only be falsely self-serving, but would actually cause the reader to miss perhaps the most important aspect of the entire project — *AIPS* changes. I, as its chief designer, do not pretend to have all the best ideas, nor to be able to comprehend all of the system as a whole in detail even for today, yet say for years to come. I can guess, but I cannot know, what users will like and what they will find unnatural. And I cannot conceive some of the algorithms which will be invented in the future and come to be regarded as critical. All I really understand is that the system should be capable of growing, of absorbing good ideas from many people, and of correcting any mistakes or shortcomings. The ability to evolve has been the chief asset of the *AIPS* project.

---

* Other systems, primarily *IRAF* and *MIDAS*, are growing rapidly and this statement may already be dated.

Fundamental Design Considerations

The first thing that any software designer must do is determine what it is that he is expected to design. He must determine the actual needs of his employer and what computer software and hardware are already available to meet those needs. Employers often take a narrow view of this, basing their assessment on the most immediate problems and on a limited view of what current software can do. Designers should take a broader view, however, by trying to generalize the concept of the immediate needs and by trying to forsee future developments in those needs. It may be more expensive in the short run to take the time to think broad thoughts and to design and execute broad ideas, but it is cheaper in the long run to solve the general problem once than to solve specific problems separately over and over again. In particular, it is very much more expensive to write a new software system, when an existing one is already adequate to the requirements, or can be made adequate with a reasonable effort.

In our case, the employer is the National Radio Astronomy Observatory. The NRAO builds and operates radio telescopes under contract with the United States National Science Foundation to provide radio astronomical observing facilities to scientists from around the world. In 1978, the construction of NRAO's premier instrument, the Very Large Array, was approaching completion. The VLA is an aperture synthesis instrument with 27 individual elements and can produce very large quantities of data. (For example, a spectral-line observation can produce about 90000 measurements, each a complex number, every ten seconds.) NRAO's computer problem is to edit and calibrate these data, to convert the data into images of the sky, to remove instrumental effects from those images, and then to convert the images into scientifically useful numbers and displays. There was an active group at the VLA working on the calibration and imaging areas and there was also a separate package to do these operations in Charlottesville. Therefore, the obvious need was to "reduce" the images into scientifically useful results and this was the first charge to the "post-processing group." However, it was easy to predict that there would be a need to do more imaging away from the VLA site and to feed information gleaned from images back into improved editing and calibration. In particular, a new, "self-calibration" (or adaptive optics) algorithm was developed by Schwab in Charlottesville [2], and it was quickly clear that the future needs, even on a fairly short term, would include editing and calibration of the fringe visibility data and all parts of the imaging operation. On a longer scale, the NRAO was hoping to build (and is now building) the Very Long Baseline Array. This VLBA would need software to calibrate its data, and would not be able to use the software developed at the VLA. Thus, the future needs of the NRAO made it clear that the "post-processing" group was likely to become concerned with all phases of the processing. The data coming from the VLA in 1978 were single-channel continuum, and solutions for that problem were urgently needed. However, it was already clear that we would need to process spectral-line data and that we would want to compare images from other telescopes with those made at the VLA. Therefore, we could see easily that we needed to handle more general data forms than those which were our immediate problem.

Until 1978, it was assumed that scientists would come to the NRAO telescopes for their scheduled observing session and then remain long enough to reduce their data. With the high data rates of the VLA, this was no longer possible. It might be possible to provide the routine calibration and initial imaging facilities for visiting observers. However, from this point on, data reduction requires considerable careful thought and involvement by the scientist. Most visiting observers have obligations to their home institutions and cannot remain at the NRAO long enough to conduct a thoughtful final reduction of their data. Furthermore, I have found in writing my own astronomical papers that I always have a need to go back to the data to determine a few more numbers and to prepare yet another display, even long after the data have been "fully" reduced. Our visitors, when confronted with similar needs, have previously had to choose between waiting until their next visit to NRAO, writing their own software, or skipping some of the details in their analysis. If the users could run the NRAO's software at their home institutions, the scientific output of the VLA would be increased without costly duplication of programming effort.

Furthermore, the VLA is capable of producing data at rates that would overwhelm the supercomputers of 1988, yet say those of 1978. Therefore, the NRAO cannot hope to provide enough computer power to allow all visitors to complete the data analysis in a convenient and timely manner. If NRAO's software could run on computers at the visitors' home institutions, then the total computer power available for the VLA could be substantially more than NRAO could afford to provide. We cannot assume that astronomers will go out and buy exactly the same computers that NRAO has. This will never be economically feasible for more than a subset of our users and would not necessarily be desirable in a competitive computer market. Computer systems used to evolve on time scales of five years and now evolve in something closer to two years. A large-scale software system such as *AIPS* requires three years to get started and more than five to be scientifically valuable [3]. Thus, in the lifetime of the software, it will be expected to run on more than one generation of computer. Therefore, the prime design specification for *AIPS* was that it be machine independent, so that it can run on a number of different architectures that might be expected at the users' home institutions and on the next generation, or generations, of computers.

In addition to users of the VLA, there is a second significant class of customer for our software — those with data from astronomical instruments other than the VLA. This class includes, of course, VLA users and NRAO staff members who will already be using *AIPS* for their VLA data. It also includes scientists at other institutions who will be asked, by their VLA-oriented colleagues, to devote a non-trivial portion of their joint computer resources to *AIPS*. Many of the software requirements of scientists in this class are virtually identical to those of VLA users. Thus, the second primary specification for *AIPS* is flexibility and generality in its data base and software design. This generality has benefits for VLA users as well. It means that, when new capabilities are created in the VLA, the software may already be able to handle them with, at most, a few corrections. The new capabilities have not been "designed out" by too restrictive data formats and programmer thinking. Users will expend considerable effort to become proficient in the use of *AIPS* to reduce their VLA data. If that learning can also be applied to other problems, we have performed a considerable service.

A full list of the criteria for the design of *AIPS* — basically a "motherhood-and-apple-pie" list — was given in 1982 [4] as:

1. transportable
2. general, flexible
3. interactive
4. efficient
5. friendly
6. multi-user
7. allow batch
8. easy to add new software
9. documented
10. uniformity of software
11. run in small memory
12. minimize disk sizes.

An image processing system must be interactive, at least at some appropriate level. The system must allow the user to evaluate the progress of the reductions, and to discover, and to point out to the software, interesting regions within the images. On the other hand, it must be able to make full use of the computer's resources when performing major computations on large images. It should be "friendly" — relatively easy and simple for beginning users and yet powerful for advanced users. There should be easy access to the system and data both for multiple interactive users and for users with long, batch-like computations to perform. In order to develop, maintain, and distribute a large system over a considerable period of time, it should be easy to add software to the system, there must be extensive documentation at all levels, and there must be uniformity in software coding standards, conventions, service routines, and the like. I have added the last two criteria to the list because they have been important to the design, although there are no longer such obvious criteria. Modern computer systems now usually have substantial memories, virtual-memory operating systems, and significant disk storage. Even these machines do, however, get overloaded and begin to devote almost all their resources to page swapping, which is sufficient reason to keep the individual programs as small as possible. Cavalier memory use can produce truly gross page faulting that could be avoided with only minor cleverness. And no computer ever has enough disk. I have watched NRAO's mini-supercomputers run at less than half capacity because there was no disk space on which to write the results. Note that this was on a computer with four Gigabytes of disk and only a small number of users.

I have used the word "system" above, thereby incorporating an important, but implicit, additional criterion for the whole project. A system is more than a collection of independent data reduction programs. It is a highly correlated set of programs which access the same form or forms of data in very similar manners. A system lets numerous programmers contribute their efforts to the project and allows that project to be maintained even when personnel and computers change. System-wide routines permit improved performance of basic functions such as I/O and system standards enhance the quality of the code and facilitate its distribution and maintenance. From the users' point of view, a system presents a coherent picture with standard data, a common command language, and a predictable program philosophy — all of which reduce the difficulty of learning how to use the programs. Proper standards will allow the user training, the programmer knowledge and skills, and the source code and the data to be portable over both space and time.

A User's View of *AIPS*

In Figure 1, I have drawn an overview of the *AIPS* system as seen by a user. The main interactive program in the system is itself called `AIPS` and is represented by the heavy box at the upper left of the figure. The input to this program may be switched temporarily from the user's terminal to text files located in a disk area called `RUN`. This option allows users to prepare long procedures and operation sequences with the local text editor in an area protected from most changes in `AIPS`. The program `AIPS` consists, logically, of two parts — a command language processor and a set of subroutines which perform various application operations. The choice of a command language is one of the more visible decisions made by the system designer and discussions of this choice generate much heat. It is my belief, however, that the choice is not so important in the long run so long as the command language meets certain minimum requirements. First, it must be a *language*; it must provide the ability to define a variety of types of parameter values, to define new symbols, and to define sequences of operations including looping and logical branching. Second, the language must be divorced from the application enough that the language could be replaced without significantly changing the applications code. Third, the language must meet the requirements for portability so that the user sees the same command language on different computers at different times.

The command language we chose is called *POPS* or *People-Oriented Parsing Service* [5]. *POPS* works from an area of computer memory called the symbol table — a linked list of symbols known to it. These symbols are of three general types:

(1) *Verbs* are symbols which, when invoked, cause a portion of the application source code inside the program `AIPS` to be executed.

(2) *Adverbs* are parameters used to hold data values. These data are typically used to control the functions of verbs, procedures, and tasks, or as output from verbs and procedures. Adverbs may be reals or character strings and scalars or arrays.

(3) *Procedures* are symbols representing pre-compiled sequences of *POPS* symbols.

I should emphasize that the language processor and application subroutines are all written in Fortran. The only applications written in the *POPS* language are a few simple programmer-provided procedures and any procedures written by the current user. In its normal mode, *POPS* reads a line from the input device, parses and interprets the symbols and data in the line, compiles the statements, and then causes them to be executed. The user may, at run time, create procedures. In this mode, *POPS* does its usual parsing and compilation, one input line at a time, but it stores the compiled code away to be executed at a later time. The *POPS* language provides a fairly complete collection of mathematical and logical operators. It offers `FOR` and `WHILE` looping and the `IF`—`THEN`—`ELSE` construct. All the usual mathematical and string-handling functions are available. New adverbs may be created at run time. Procedures may invoke other procedures, may receive and pass arguments, and may act like function subroutines. In other words, *POPS* is a computer language with normal capabilities. It is a fairly simple language with a rather cumbersome (internal) implementation, but it appears to be more than adequate. To make things friendlier to the user, we employ minimum match whenever we search a symbol table or disk directory. Where possible, the compiler is also forgiving about missing equals signs and commas.

The other logical part of `AIPS` is the set of verb subroutines. Some verbs must be regarded as parts of the *POPS* language — from such simple things as the mathematical operators to the rather complicated verbs that control, for example, batch processing. One of these complicated verbs, called `GO`, implements one of the most important aspects of the *AIPS* system. All heavily computational operations — things which deal with full images or data bases — are implemented as separate programs, called *tasks*. Tasks are known to `AIPS`, not as part of the symbol table, but only via text files in an area called `HELP`. The verb `GO` uses the appropriate `HELP` file to determine which adverbs are required by the task and in what order. `GO` puts these values in a block on a standard *Task Data* file, wakes up the task, and suspends itself. The task reads the file to get its parameters and other information such as the user identification and which of the several possible `AIPS` running on the computer actually started it. Then, if so instructed *and if appropriate*, the task resumes `AIPS`. With this mechanism, the user can continue to interact with the language processor and the verbs inside `AIPS` while running one or more simultaneous, major computations. From a programming viewpoint, this structure has many obvious advantages as well. Each major algorithm is a separate program which can be developed individually and added to the system with *no* changes to the main `AIPS` program. Tasks spawned by the interactive `AIPS` may use the tape drives and the TV and graphics devices. Other than a single-word return code, however, tasks do not pass parameters back to `AIPS`. Instead, their actions are detected by the changes in the common data base and in the primary and interactive-device catalogs. Messages from tasks appear on a monitor screen and, like all input to and messages from `AIPS` itself, are also recorded in a message file. This structure allows a task to leave `AIPS` suspended and to carry on whatever communication with the user it desires using the main terminal, the TV, or other devices. Several tasks take advantage of this structure to do interactive data editing, masking, model fitting, and the like.

Most interactive operations are implemented as verbs. These include facilities for run-time documentation, facilities to manipulate and to discover the contents of tapes, and facilities to manage the user's data base. Verbs load and annotate images on the TV, carry out image enhancement functions like zooming and pseudo-coloring, perform roam, blink, and movie operations, and allow the user to determine, interactively, image pixel positions and values. Other verbs draw images on the graphics device and return positions and values using the crosshairs.

Another important concept in *AIPS* is its batch capabilities. The user prepares input text for his batch job with the assistance of several verbs inside `AIPS` or with a special, stand-alone batch preparation program. To cause his completed text to be executed, the user invokes the verb `SUBMIT`. This verb wakes a special version of `AIPS` called `AIPSC` or Checker. This program contains a full *POPS* language processor including all math, procedure building, and similar verbs. However, almost all other verbs are fully stubbed. It is Checker's job to read the user's text file, processing the *POPS* language statements and, if it finds no mistakes, to copy it to a new file and to enter appropriate information about the new file in a special *Batch queuing* file. If needed, Checker then wakes a queue manager program called `QMNGR`. This program uses the data entered in the batch queuing file to run one or more versions of `AIPSB` — the batch version of `AIPS`.

`AIPSB` is nearly identical to `AIPS` with two major exceptions: (1) it takes its input from a batch text file and (2) all fundamentally interactive verbs — those using the TV, graphics screen, and tapes — are stubbed. In particular, `AIPSB` can spawn tasks just as `AIPS` does. The main differences here are (1) that

AIPSB is not resumed until the task finishes — securing a sequential operation and ensuring a meaningful return code and (2) the messages do not go to the monitors. They are simply logged and left available for printing during and after the job.

To make things friendlier, there are the pseudo-verbs SAVE and GET. SAVE stores, in user-owned and named files, the full *POPS* symbol table including all current procedures and adverb values. GET recovers this "environment" either at some later time in an interactive session or in a batch job. Thus, the user will often prepare adverb values and procedures interactively, using them on a sample portion of his data. He may then, with a short batch input file, apply this environment to the rest of his data without continuing to occupy the interactive devices. RUN files may also be used in batch jobs to simplify the task of preparing the input commands.

A Programmer's View of *AIPS*

Language

A designer must begin his system design with a decision on the computer language to be used. In general, he should be prepared to invent his own language at some level since no widespread language will be entirely appropriate to the problem. The designer then writes a preprocessor program to translate the designer's language into a standard computer language. It is my belief, however, that more astronomy is done sooner with a language that is very close to a standard, well-known language than with one that has to be invented in all its parts. In the case of *AIPS*, there was only one choice of language that could be made in 1978-1979, namely Fortran 66. There were no other languages available which were well known to astronomers, for which there was a strict international standard, and for which there were compilers available on all machines. Fortran 77 was obviously coming. However, the requirements for small memory and small disk files forced us to use two-byte integers wherever possible, which was a direct violation of one of the most useful portions of the Fortran 77 standard. In fact, the specification that *AIPS* should be portable to PDP-11 machines, an architecture that was unable at the time to allocate meaningfully the same storage to an integer as to a floating point, required that we not use standard Fortran 77. Therefore, we adopted Fortran 66 and added to it three constructs: INCLUDE to insert the contents of a file in place of the INCLUDE statement and ENCODE and DECODE to do in-core reads and writes. The former was done for code management reasons, to provide system-wide definitions of the standard commons. The latter was made necessary by the need to place text in strings for writing to display devices and disk files. Preprocessors were written for some machines, but VMS did not require one.

One cannot write a reasonable image processing system without "structures", data units whose contents include smaller data units of various types and lengths. In particular, one describes an image with an image *header* which typically contains, for example, integers for image dimensions, single-precision floating-point numbers for brightness extrema, double-precision floating-point numbers for coordinates, and character strings for image names. This header needs to be treated as a unit to be stored on disk, passed between subroutines, and the like. The relative lengths of integers, floating numbers, and strings are machine-dependent, but the problem is still tractable in machine-independent Fortran 66. For *AIPS*, we have a machine-dependent subroutine, called by all programs, to set parameters describing the local machine including the various word lengths. Then, machine-independent code can `EQUIVALENCE INTEGER*2`, `INTEGER*4`, `REAL*4`, and `REAL*8` arrays to the header and compute the offsets within these arrays from the beginning of the header block to any desired portion of it. Character data are also stored in the `REAL*4` array, but they must be handled very carefully since the storage of such Hollerith data is quite machine-dependent. A full complement of routines to move, compare, parse, etc. various forms of these data has been developed. However, I will not belabor the details, since, as will be described later, *AIPS* is now undergoing a transformation of its language standard.

Data Formats

The most important design decision is whether or not to build a new image processing system and, if so, what are the immediate needs of its potential users. The next most important design decision is the determination of the form of the data. This decision will not only determine numerous practical details of data management and other system software, but will control how the designer, programmers, and users come to think about their problem. A simple form, suitable for the immediate problem, will help produce straightforward code more quickly. However, a package which has one-dimensional spectra as its basic form will seldom do two-dimensional imaging on celestial coordinates easily. Another package designed with two-dimensional images in separate files will be awkward at best when later asked to analyze spectra formed from a set of those files. A system designed with minimal header information will never handle coordinates well, while one with an excessively detailed header will fail when presented with some new form of data and will promote code that fails to support the header design fully and correctly.

No design will escape these pitfalls entirely. *AIPS* was the first major system designed after the FITS format was created and we benefited greatly from it. In fact, the *AIPS* header format can be said to be a restricted, binary representation of a FITS header. The basic FITS format [6], in which the image is regularly gridded, is supported by *AIPS* up to seven dimensions, with the dimensionality, type, coordinate value and increment, and reference pixel given in FITS-like fashion in the header for each of the axes. The random groups form of FITS [7] is also supported, but our tasks currently handle only interferometer fringe visibility data and some forms of single-dish radio data. Up to seven random parameters are supported now, but more will be handled correctly after the code transformation to be described later. *AIPS* is particularly strong in its representation and uniform handling of physical units and coordinate systems, especially in the cases of nonlinear projective geometries and rotated coordinates.

AIPS data live in disk files. Each disk has, for each user, a "catalog" or directory file listing the name and activity of each image on the disk belonging to the user. The header for each image is kept for safety in another file as a 256-integer binary record. The image data are in floating-point format in another file of type "MA" for basic FITS and "UV" for random-group FITS. Subsidiary data attached to the image are kept in more files called "extension" files. Up to 255 files of each of 10 types are allowed by the current header format and more types will be allowed soon. Extension files are used to hold the processing history of the image, to hold the components of models of the image (*e.g.*, CLEAN components), to hold device-independent instructions for displays of the image, to hold tables describing the sources and frequencies observed, to hold calibration values and editing commands for the data, and so forth. Many of the extension files now have a format which is a binary representation of the new FITS extensions and tables agreements ([8] and [9]).

All of this generality has come at a significant cost. *AIPS* is not an easy system in which to code, in part because the data forms and the corresponding header are so general and "standard" programs are expected to handle that generality. There are many subroutines to help with this task, but the sheer number represents a learning barrier. However, I believe the generality has been worth the cost in producing higher quality, flexible code which can do reasonable things with a diversity of data. *AIPS* has been used to reduce a wide variety of optical, X-ray, infra-red, and radio observations as well as being the standard reduction system for the VLA. Since *AIPS*' formats are so FITS-like, the reading and writing of FITS tapes is fully supported within *AIPS*. In fact, *AIPS* has been used as a test arena for all proposed extensions to the FITS formats.

Virtual Device Interfaces

An image processing system must have operating-system like functions. For example, it must be able to create, open, write, close, truncate, reopen, read, and delete disk files. To be friendly, it must be able to record and report the time and date and to talk to display devices of various kinds. To be flexible and powerful, it must allow programs to communicate with each other. And to send and receive data from the outside world, it must be able to translate between the internal binary formats and the external standards. One could do these functions simply by calling, for example, VMS system utilities from every program that needed them. However, this would not be machine independent, nor would it be a good "system" solution even if we were willing to sell our souls to a single manufacturer! (The UNIX operating system has increasingly been suggested as the "portable" cure for the use of VMS system calls in code. This is equally pernicious nonsense, since every new release of UNIX from every manufacturer, differs in non-trivial ways from previous releases of that and all other manufacturers.) Instead, a system designer needs to define a *virtual operating system* (VOS), a set of subroutine call sequences which perform the desired system-like operations as the designer chooses to define them. Then all machine dependencies can be hidden away in the subroutines which implement the VOS and the applications may be written in a machine-independent fashion. The VOS also serves the system-design purpose of providing unique system-wide methods of doing critical functions. Considerable effort can be put into the coding of the VOS routines to hide serious complexities from the applications programmers and to enhance performance. In *AIPS*, we call the VOS the "Z routines" since their names all begin with the letter Z. We have developed sets of Z routines for DEC's VMS, for several flavors of UNIX, and for COS (the Cray Operating System).

The VOS is an example of a *virtual device interface* (VDI), with the device being the host computer. *AIPS* implements two other virtual device interfaces openly. One of these is the "TV", a multi-level interactive display device. We have a specific idea of what characteristics a TV should have and have defined a set of subroutine call sequences which would drive such a TV. We also have defined a set of parameters to be used by the device-independent routines that define aspects of the actual TV and which are returned when the TV is opened. These parameters include the number of pixels in each row and column, the peak intensity in the display, the number of image memories, the number of one-bit graphics overlay channels, etc., as well as parameters defining the current state of the TV such as which channels are now visible, what zoom magnification and scroll are in effect, and so forth. The TV VDI is sufficiently powerful that we have implemented a truly "virtual" TV, in which the physical device is on a cooperating computer connected to the host by computer network. The device-independent code operating on the host has no knowledge of this and runs as if the TV device were directly connected on a DMA port with the user sitting in the room, rather than possibly across the country. The TV VDI is known in *AIPS* as the Y routines because the routine names start with the letter Y. Implementations for International Imaging Systems Models 70, 75, and IVAS, for Gould-DeAnza 8500, for Comtal Vision 1/20, for SUN under SunView, for ARGS, and for Lexidata are currently distributed. Other models are in use for *AIPS* at some of our user sites. Note that the Y routines should be operating-system-independent and call the VOS Z routines (or manufacturer-supplied routines) to perform such things as open, read, write, and close.

The other VDI in *AIPS* is to a vector processor. The initial *AIPS* implementations at NRAO were on a ModComp Classic and a Vax 11/780, both equipped with Floating Point Systems (FPS) AP-120B array processors. The most compute-intensive tasks were written to take advantage of these peripherals to improve performance by factors of 8–14. We developed a virtual device interface, called the Q routines, to act as an interface to the FPS libraries and to some NRAO-written microcode and vector-function-chainer routines. Then it was straightforward to also write a separate version of the Q routines which used the host CPU to perform the same functions on the data. The array processor memory is represented by a Fortran `COMMON` and Fortran and assembly routines operate on the data in the common. These "pseudo-AP" routines allow *AIPS* to run on computers that do not have array processors, with no changes to the higher-level, processor-independent code. The selection of real or pseudo array processor is made solely by choosing which Q library is included in the link edit of the task. We have taken advantage of this concept to develop fast implementations of *AIPS* for Crays and mini-supercomputers such as those made by Convex and Alliant. Such computers are vector machines which effectively do array processor computations within the host cpu, fully supported by the host Fortran compiler. Many of the pseudo-AP routines are fully vectorized by these compilers, but for improved performance, we have written a few specialized versions of the Q routines as well. The AP-like model of a vector processor has been criticized because it requires data movement between the body of the program and the AP or pseudo-AP memory. However, such a model allows *AIPS* to make use of powerful array processors, while we have found the data movement costs to be negligible on vector computers.

Documentation

No discussion of an image processing system is complete without some description of its documentation. A designer should, very early in the project, devise standards for user- and programmer-level documentation and methods for enforcing those standards. The designer should also develop means for automatically generating documentation from the source code files themselves. If he (or she) does not do these things carefully, then the system will develop the standard failing of incomplete and outdated documentation. Good will and the solemn promises of the programmers to document their work are not enough. *AIPS* has done well in this area, yet still suffers some of this failing.

For the users, there must be both run-time and printed documentation. The run-time information serves three functions: (1) reminding users of the parameters of a verb or task and their current values, (2) describing the parameters and functions in somewhat greater detail, and (3) really educating the user about the more critical and/or difficult processes. In *AIPS* there are currently 725 separate on-line "help" files for tasks, verbs, and adverbs, containing about 2 megabytes of text. Tasks cannot be started by `AIPS` without a proper help file. This design feature keeps `AIPS` independent of the tasks, but has the desired side effect of forcing the programmers to write most of the required on-line documentation. The files for verbs and tasks have three sections, corresponding to the three functions listed above, which we call "inputs", "help", and "explain". Programmers write the first two sections, but when possible, we prefer that scientific users write the explain section.

Printed documentation for users comes in three forms, a tutorial document, a reference book, and periodic information about changes. The first is provided in *AIPS* with the *AIPS COOKBOOK* [10]. It was initially written by Bridle in 1981 as a step-by-step guide to help beginners find their way into the system. Current editions retain this emphasis, but have added chapters to introduce users to more advanced topics such as spectral-line and VLBI reductions and greater use of *POPS*' facilities. The reference book should be produced partly by hand and partly by selectively reproducing and organizing the help files. The code to do this used to exist in *AIPS*, but has fallen into disrepair in part due to lack of user interest. The third form is provided by the *AIPSLETTER* which we publish about four times per year. Its function is to report the changes to the system in both summary and detailed forms, as well as to carry other articles of interest to the *AIPS* user and programmer community.

There are many possible levels for programmer documentation and a system which provides a wide variety will be more successful. There should be a tutorial document to take users from simple beginnings to understanding the virtual device interfaces and other subroutine packages of the system. There must be reference documents to spell out the details of file formats and other conventions. There needs also to be documents which provide a programmer with a "shopping list" of the available subroutines in the system using a variety of selection criteria. Such lists must be generated from the source code files themselves in order that they remain current and able to help programmers find their way through large systems. In *AIPS*, the primary documentation for programmers is the two-volume book called *Going AIPS* [11]. It begins with a couple of chapters aimed at scientists who wish to write an *AIPS* task without having to learn "too much". The text then goes on to describe in greater detail how to write tasks, how to handle the header and the image catalog file, and how to invoke the various input/output routines. Volume 1 ends with a discussion of programmer tools and the source code directory structure. Volume 2 treats devices (tape, graphics), TV displays, plotting, array processors, tables, FITS, Z routines, and calibration and editing routines and files. Additional text files which describe formats and other details may be found with the *AIPS* source code, but are not routinely distributed in printed form. The shopping list function will be provided in the future, but our old methods, which did not use the source-code files, failed to maintain usable lists.

The Future of *AIPS*

Code Overhaul

When we initially designed *AIPS*, Fortran 66 was the only widely known, widely supported, standard language. The initial requirements for small memory and disk utilization were in conflict with other requirements, especially ease of coding, efficiency, and transportability. Thus, we devised a system that was very portable, but rather cumbersome and confusing in some of its coding requirements and vulnerable to the quality and reliability of manufacturers' Fortran compilers. The computers now in use in astronomy have changed a great deal and comparatively large memories and disk capacities are now available. Thus, criteria 11 and 12 of page four no longer make sense and we have decided to invoke the "*AIPS* changes" rule again.

We are now converting *AIPS* from Fortran 66 to Fortran 77. Our new language, after conversion by a pre-processor program, will be in strict adherence to the ANSI standard. We require that every system use a pre-processor program to translate any special *AIPS* statements. At present, we use a *limited* subset of Fortran 77, and we add a `HOLLERITH` statement and an `INCLUDE` statement, including a `LOCAL INCLUDE`. The `HOLLERITH` data type is required in order to place character-valued data into data structures for input/output and transmission to devices. Any use of the character-valued data must be in the form of standard `CHARACTER` variables, so `HOLLERITH` variables will mostly appear only in calls to the routines which translate between `HOLLERITH` and `CHARACTER`, in equates to other `HOLLERITH` variables, and in `EQUIVALENCE` statements used to construct data structures. Following Fortran 77 rules, all `INTEGER`, `REAL`, `LOGICAL`, and `HOLLERITH` variables have the same length and the new *AIPS* explicitly assumes that that is at least 32 bits. We differ a little from Fortran 77 in not requiring `DOUBLE PRECISION` to be twice as long as `REAL`, thereby allowing us to use 64 bits for everything on 64-bit architectures.

Although it is esthetically pleasing to use a more recent, standard language, that is not the main reason for doing all this work. Instead, we have found that computations with 16-bit integers are slower than those with 32-bit integers on both our Vax and our Convex computers. This result was discovered accidentally using a program which should spend most of its time doing computations in floating point, so we are a bit mystified, but we hope to get significant performance improvements. We have encountered numerous errors in the Fortran compilers we have used, which have complicated our work at times and have prevented us from compiling most of the code with optimization. Since most of these errors were related to the use of short integers, we hope that the new *AIPS* may be compiled without compiler errors and with optimization. Many problems which we are now handling require us to count higher than 32767. The use of short integers throughout most of the code has made this difficult and has become a serious source of error. We should be able to use longer buffers in the future when that will speed the computations. The conversion to Fortran 77 has allowed us also to simplify the code by removing a variety of complexities required to insure portability, primarily of short integers.

We are also adding to each file some lines of text which may be used to print indexed lists of the routines available in *AIPS* with a one-line description of what they do. This will be enforced by our code-management system, so we should be able to maintain up-to-date "shopping" lists for programmers. We are also adding a copyright statement to each file in *AIPS*. The revised *AIPS* will be regarded as a new product and will be distributed only under license. This license will be free of charge to non-profit institutions engaged in basic research in astronomy. Our intention is not to limit the use of *AIPS* or its code, but only to retain legal rights to NRAO's "intellectual property" and to prevent unauthorized distribution of *AIPS* to third parties.

Development Plans

In order to manage the project, I have maintained a "Wishlist" of items to which we would like to devote our attention. This document is released periodically through the *AIPS* Memo Series [12] and presently contains some 10–20 man-years of bugs, improvements, and new projects. Figure 2 gives a summary list of some of the items we actually hope to work on in the next year. It is beyond the scope of this manuscript to explain in detail what is intended by the items listed.

The Distribution of *AIPS*

AIPS Site Survey

In order to determine the use of *AIPS*, we conduct an annual survey of the sites that we know to have received a copy of *AIPS*. The results of the 1987 survey appear in detail in [13] and [14] and will only be summarized here. The survey reports 132 computers on which *AIPS* is in active use, up from 54 in 1985 and 90 in 1986. Of the 132 computers, 99 run the VMS operating system, 27 run UNIX, and 6 run other systems. The machines are in 15 countries — Argentina (1), Australia (3), Canada (10), China (1), Denmark (1), France (3), Holland (5), India (2), Italy (4), Japan (2), Spain (1), Sweden (2), the United Kingdom (6), and the United States (87), of which 7 are at the NRAO.

We estimate computer power in units of a VAX 780 with FPS array processor executing full-time on *AIPS*. In these units, we take a Convex C–1 to have a weight of 3.5 and a Cray X–MP a weight of 25. The surveys have found that the total compute power actually running *AIPS* has increased from 9.1 in 1985 to 25.2 in 1986 to 37.5 in 1987, using these units. Of this 37.5, 6.3 are in the NRAO, 11.7 in the U.S. outside the NRAO, and 19.5 in the rest of the world. Exporting *AIPS* has provided a six-fold increase in the power available to our users, and a four-fold increase in the machine power in use specifically for VLA data reduction under *AIPS*. Despite the predominance of VMS machines in the machine and user counts, the computer power in use for *AIPS* under VMS is now down to about 25% of the total; it was 64% in the 1985 survey. This change is due to the increase in the number of vector computers running *AIPS*, none of which use the VMS operating system.

Management and Exportation of *AIPS*

A system designer must also plan code management and export policies for his/her system. The code must be protected from accidental change and from changes by two programmers at the same time. An audit trail of any changes must also be maintained. We do this in *AIPS* with special source-code directories and code-management software to handle "check-out", "put-back", "history", and the like. The designer must also allow sufficient time and use to debug the software and provide tools to report and repair the problems that will be encountered. A large, general system has too many options, combinations, and capabilities to be debugged by the programming staff alone. The code of *AIPS* is managed in the form of quarterly releases and there are always three versions. The "tst" version is under active software development and is used by NRAO staff members to help with the debugging. The "new" version is in heavy use by visitors to the NRAO to check and correct any remaining bugs. And the "old" version is not allowed to be modified and is the one shipped to non-NRAO sites. At each quarterly "freeze date", the versions migrate from tst to new and from new to old. We publish the *AIPSLETTER* in conjunction with this freeze.

`AIPS` has a verb called `GRIPE` by which a user may enter a complaint or suggestion. These gripes are collected regularly from NRAO computers and may be sent to NRAO by the non-NRAO sites. They are reviewed immediately and then reviewed regularly until answered by the programmers. This process greatly increases our communication with the users, our pool of new suggestions, and our information on bugs. If we had enough programmers to process the gripes more promptly, they would be even more significant in adapting *AIPS* to the needs of the users. Manpower also limits the amount of telephone consultation that we can offer *AIPS* users and site managers. System designers should include the needs for interaction with their user community in the planning of the system and, especially, the manpower requirements.

As described in previous sections, the policy of exporting *AIPS* to all possible sites has been fundamental in the system since its inception. We do this through the many design decisions described above and implement it by making *AIPS* and the documentation available without charge to non-profit institutions engaged in basic research in astronomy. To order, please write to the address given at the start of this article. The documentation in the form of the *AIPS COOKBOOK*, *Going AIPS*, the *AIPS* Memo Series or the *AIPSLETTER* may also be obtained from that address. The new *AIPS*, beginning probably with the `15APR89` release, will require a license agreement with all sites. The details of the agreement have not been determined as yet, so please write for information in advance of your order. *AIPS* is also available to commercial institutions at a reasonable cost; write or call for details.

AIPS itself is available on 2400-foot magnetic tape in three different formats: "compressed FITS", VMS `BACKUP`, and UNIX `tar`. The first of these is intended for not-yet-supported operating systems and is an *AIPS* suggestion for a FITS standard to transmit text using a simple technique to compress consecutive blanks. A sample program to read the tape is provided, but, after that, the recipient is on his or her own. The `BACKUP` and `tar` tapes are system-specific and provide not only the text of all *AIPS* code and documentation files, but also procedures for automatically installing *AIPS* on the recipient's computer. The VMS tape also includes binary copies of the link libraries and load modules, so that, in many cases, the installation can go very quickly. Installation tapes are accompanied by substantial documentation. When ordering, please specify the operating system, computer model, and tape density. For VMS, I recommend the use of 6250-bpi tapes; we have had to omit some of the load modules from the 1600-bpi tapes in order to fit the system on two tapes at that density. Free, limited telephone consultations are available to help with installation and usage problems.

AIPS is a substantial system and you will need significant disk space in order to install it. At the `15JUL88` freeze, the compressed FITS tape contained text data for 90 directories. This included 3841 text files, comprising 681,471 lines or 22,239,430 bytes of compressed text. On our VAX, this release occupied 96 Megabytes of disk. Non-NRAO installations will not require all of the files all of the time, but you can count on needing at least 35 Megabytes at run time for the help files and load modules alone. UNIX systems usually require more disk space since they typically generate rather large load module files.

Acknowledgements

This project has used the knowledge, skills, and plain hard work of a great many people. A list, which I hope is reasonably complete, is given in Figure 3. The author would also like to thank Alan Bridle, Bob Burns, Bill Cotton, Don Wells, and Nancy Wiener for their comments on the present manuscript.

References

[1] Thompson, A. R., Clark, B. G., Wade, C. M., Napier, P. J., 1980, "The Very Large Array", *Ap. J. Suppl.*, **44**, 151.

[2] Schwab, F.R., 1980, "Adaptive Calibration of Radio Interferometer Data", *Proc. Soc. Photo-Opt. Instrum. Eng.*, **231**, 18.

[3] Wells, D. C., 1987, "Data Analysis Systems", *Selected Topics on Data Analysis in Astronomy*, Eds. Di Gesù, V., Scarsi, L., Crane, P., World Scientific Publishing Company, Singapore.

[4] Greisen, E. W., 1982, "An Overview of the NRAO-*AIPS* Image Processing System", presented to URSI, Boulder, Colorado.

[5] Sume, A., 1978, "POPS, An Interactive Terminal Language with Applications in Radio Astronomy", Internal Report No. 115, Research Laboratory of Electronics and Onsala Space Observatory, Chalmers University of Technology, Gothenburg, Sweden. *POPS* was invented by Jerry Hudson and extended to applications by Tom Cram, but neither have generated a definitive reference.

[6] Wells, D. C., Greisen, E. W., Harten, R. H., 1981, "FITS: a Flexible Image Transport System", *Astr. & Astrophys. Suppl. Ser.*, **44**, 363.

[7] Greisen, E. W., Harten, R. H., 1981, "An Extension of FITS for Groups of Small Arrays of Data", *Astr. & Astrophys. Suppl. Ser.*, **44**, 371.

[8] Grosbøl, P., Harten, R. H., Greisen, E. W., Wells, D. C., 1988, "Generalized Extensions and Blocking Factors for FITS", *Astr. & Astrophys. Suppl. Ser.*, **73**, 359.

[9] Harten, R. H., Grosbøl, P., Greisen, E. W., Wells, D. C., 1988, "The FITS Table Extension", *Astr. & Astrophys. Suppl. Ser.*, **73**, 365.

[10] *AIPS COOKBOOK*, Eds. Bridle, A. H., Greisen, E. W., National Radio Astronomy Observatory, Charlottesville, Virginia, 15 October, 1986.

[11] *Going AIPS*, Eds. Cotton, W. D., Greisen, E. W., National Radio Astronomy Observatory, Charlottesville, Virginia, 15 April, 1987.

[12] Greisen, E. W., 1987, "The AIPS Wishlist", *AIPS* Memo Series, Number 55.

[13] Bridle, A., Wells, D., Wiener, N., 1988, "The 1987 AIPS Site Survey", *AIPS* Memo Series, Number 56.

[14] Wells, D., Bridle, A., Wiener, N., 1988, "1987 AIPS Site Directory", *AIPS* Memo Series, Number 57.

The National Radio Astronomy Observatory is operated by Associated Universities, Inc. under contract with the United States National Science Foundation.

Figures

Figure 1. Block diagram of *AIPS* from a user point of view. Various communication paths are shown between the main interactive program `AIPS`, the batch program `AIPSB`, and the collection of separate tasks.

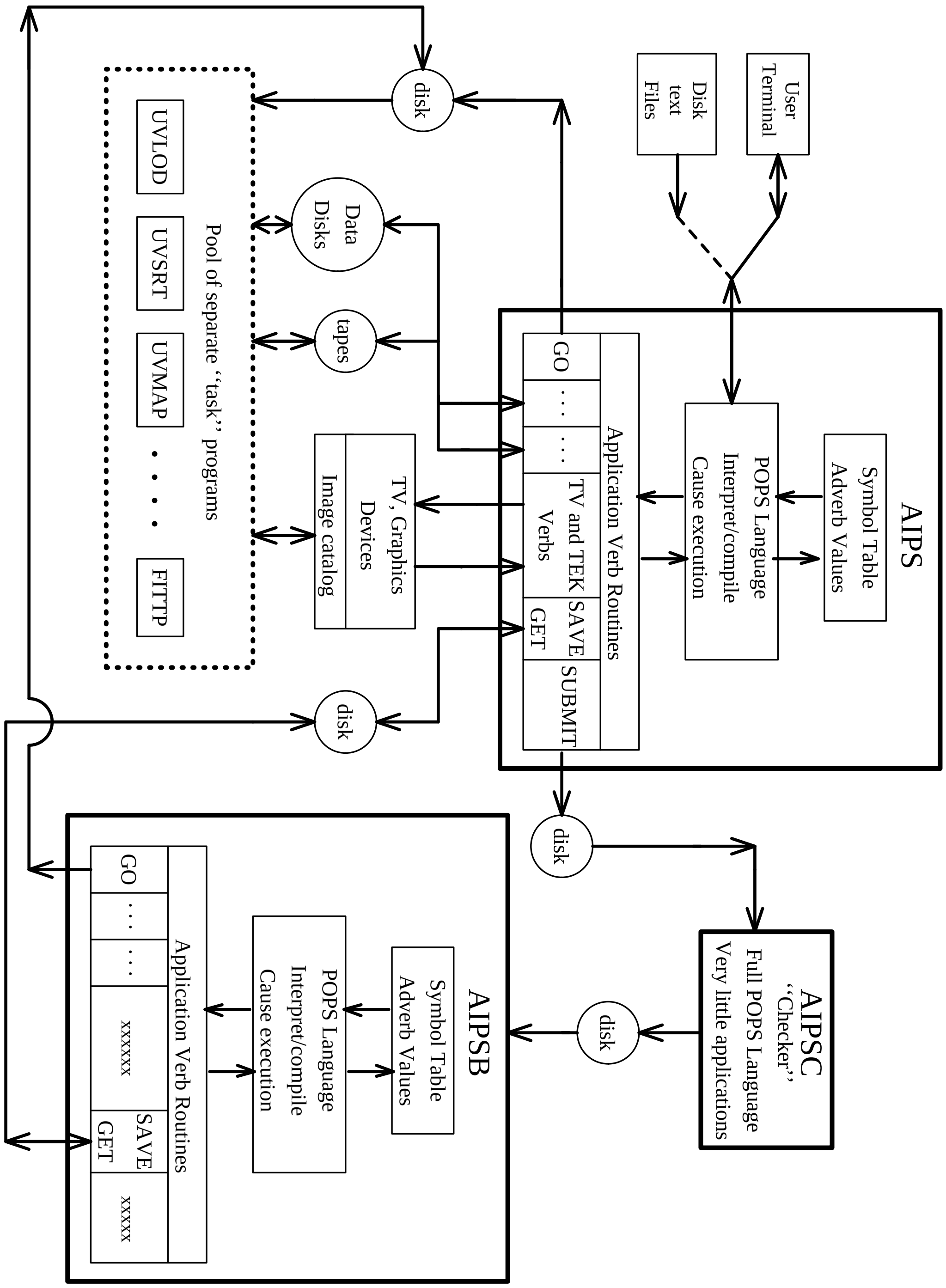

Figure 2. Development plans in *AIPS*.

Imaging

- data compression — *uv* data on disk in "16 bits"
- sortless imaging
- wide-field imaging (3-dimensions demonstrated by Cornwell)
- multi-resolution Clean

Distributed *AIPS* per Array Telescope Computing Plan

- virtual TV extension to other systems
- wider window support for workstations, X-Windows
- Network File System studies

Calibration, primarily VLBA

- Doppler tracking
- autocorrelation spectra to calibrate line data
- graphics-based interactive editing
- fringe-rate mapping
- studies continued on full calibration of VLB data

Spectral line

- improve structure of baseline and Gaussian fitting tasks
- allow wider range of displays in them

Documentation

- *COOKBOOK* up to date, add calibration chapter
- *Going AIPS* update for the code conversion
- *AIPS* Managers' Manual

Other

- tables into and out of *AIPS*
- faster message files
- graphics package interface to *AIPS*
- better user interface(s)
- international conventions on world coordinates

Figure 3. *AIPS* Contributors

current *AIPS* group

Ernie Allen	—	tape and documentation distribution
Bill Cotton	—	calibration and imaging software, VLB
Phil Diamond	—	spectral-line software, VLB
Eric Greisen	—	project design and management, general applications
Kerry Hilldrup	—	UNIX and Cray systems, Z routines
Nancy Wiener	—	Gripes, documentation, general assistance

former *AIPS* group

David Brown	—	VMS and ModComp systems
Tom Cram	—	initial design discussions
Gary Fickling	—	VMS systems, applications software
Ed Fomalont	—	scientific advisor, applications software
Walter Jaffe	—	applications and basic software
Thad Polk	—	geometric corrections software
Gustaf van Morsel	—	spectral-line analysis software
Don Wells	—	management and software design advisor

advisors

Alan Bridle	—	scientific friend and advisor
Bob Burns	—	management advisor, Head NRAO Computer Division
Ron Ekers	—	scientific and management advisor

software assistance

John Benson	—	VLB software
Stuart Button	—	early general applications
Tim Cornwell	—	mosaicing and maximum entropy tasks
David Garrett	—	preliminary UNIX implementation
Brian Glendenning	—	SUN image display routines
Jerry Hudson	—	*POPS* language
Neil Killeen	—	image analysis tasks
Pat Moore	—	VLA *AIPS* manager
Arnold Rots	—	TV display applications
Fred Schwab	—	self-calibration and other mathematical tasks