12 *AIPS* FOR THE MORE SOPHISTICATED USER

The program AIPS uses the *POPS* language to communicate with you. *POPS* has many capabilities that have been hidden or taken for granted in the previous sections. Once you start to become familiar with *AIPS*, you will need to know more about *POPS* to take full advantage of the powerful features which are available. The first section below describes some of the shortcuts and conventions of *AIPS* while the second section describes program flow control options.. The third section describes multiple features of the *POPS* language including the constructions of procedures. The fourth section addresses the needs of "remote" users of *AIPS*, while the last section discusses how to begin writing your own *AIPS* tasks.

12.1 *AIPS* conventions

12.1.1 AIPS shortcuts

Some niceties of using *POPS* syntax in *AIPS* are:

1. More than one expression can be put on a line. These expressions must be separated by a semicolon (;). Exceptions are RESTORE, GET, RUN and a few other "pseudoverbs" which, with their arguments, must stand alone. For example, GET APMAP ; INDISK = 1 $^{C}_{R}$ will ignore the INDISK = 1. When in doubt, see the HELP files for the pseudoverb to find the restrictions on its use.

2. As in many other systems, recognized keywords in AIPS do not need to be typed in full. You must type only enough of the leading characters usually suffice. This "minimum-matching" has been exploited throughout this *CookBook*.

3. The parameter variables in AIPS are called "adverbs." They are assigned values by the equals verb (=), *e.g.*, INTAPE = 2 $^{C}_{R}$. The equals sign may usually be replaced by a space. The exception arises when the variable on the left is a subscripted array element and the expression on the right involves a unary minus or other function reference (*e.g.*, APARM(3) = -1 ; APARM(4) = SIN(X) $^{C}_{R}$).

4. Array adverbs are set to a constant value by putting a single value on the right hand side of the equals sign, *e.g.*, CELLSIZE = 1.5 $^{C}_{R}$. A list of values may be put in the array by putting the list on the right hand side of the = sign separated by commas (,), *e.g.*, LEVS = -1, 1, 2, 3, 6, 9 $^{C}_{R}$. The commas may be replaced by spaces in most cases. An exception occurs if an element is negative or some other arithmetic expression. Thus, SHIFT = -19 -2 $^{C}_{R}$ will produce SHIFT = -21, -21.

5. Adverbs can be used in arithmetical expressions or set equal to other adverbs, *e.g.*, OUTNAME = INNAME ; OUTSEQ = 2.5 * INSEQ + 3.

6. Both upper and lower case letters may be entered by the user, but adverb character string values are converted to upper case before being stored and used by AIPS.

As an example, these shortcuts allow the following AIPS command sequence:

> INNAME '3C138' $^{C}_{R}$

> INCLASS 'IMAP' $^{C}_{R}$

> INSEQ 0 $^{C}_{R}$

> BLC 200,200 $^{C}_{R}$
> TRC 300,300 $^{C}_{R}$
> OUTNAME '3C138' $^{C}_{R}$
> OUTSEQ 5 $^{C}_{R}$
> GO PRTIM $^{C}_{R}$

to be shortened to:

> Inn '3c138' ; INC 'imap' ; blc 200 ; Trc 300 $^{C}_{R}$ (Note use of upper and lower case.)
> OUTN INN ; OUTS INS + 5 $^{C}_{R}$
> go prti $^{C}_{R}$ (Task name can be in either case, too.)

12.1.2 Data-file names and formats

The physical name of the data file is generated internally, depends on the type of computer, and will not often concern you as a user. You will refer to an image by specifying its disk number, the type of image (`'MA'` for images, `'UV'` for *uv* data), your user identification number, and the following three parts of the image designation:

1. Name — A string of up to 12 legal characters.
2. Class — A string of up to 6 legal characters.
3. Seq — A number between 0 and 9999.

Each of these parts corresponds to separate input adverbs called `INNAME`, `INCLASS`, and `INSEQ` (and their variations). You can choose the image name arbitrarily and sensible choices will reduce other book-keeping. Many programs will choose a reasonable image name if you do not specify one.

A common set of conventions for the name adverbs is used throughout *AIPS*. INNAME ' ' $^{C}_{R}$ means "accept any image name with the specified class and sequence". INSEQ 0 $^{C}_{R}$ means "accept any image with the specified name and class" or, if only one image is to be used, "accept the image with the specified name and class having the highest sequence number". OUTNAME ' ' $^{C}_{R}$ means "use the actual `INNAME`". OUTCLASS ' ' $^{C}_{R}$ means "use the task name", except for tasks that write more than one output image, in which case task-based defaults will be used. OUTSEQ 0 $^{C}_{R}$ means "use a sequence number that is one higher than that of any files currently on disk with the same name and class as the requested output file". The name and class strings also support "wild-card" characters for input and output. This feature is especially powerful in tasks, such as `FITTP`, that can be told to operate on *all* images that match the specified name parameters. Type HELP INNAME $^{C}_{R}$ and HELP OUTNAME $^{C}_{R}$ for details.

Only the array data, in the form of 4-byte floating-point numbers, are stored in the image or *uv* data file. The header information is stored separately for each image or *uv* data set. Directory information is stored in a special file, called the Catalog File. Each disk has such a file and it contains directory information for all images and *uv* data sets for all users on the disk pack. On most systems, each user is assigned their own catalog file on each disk.

Extension files may be associated with any image data file. Each image can have (in principle) up to twenty types of extension files and up to 46655 "versions" of each type. These subordinate files contain additional information associated with the image and are designated by a two-letter type code. `'HI'` is a history file, `'CC'` is a Clean components file, `'PL'` is a plot file, `'AN'` is an antennas file, `'SL'` is a slice file. In *AIPS*, an extension file associated with an image is uniquely specified by the usual file-naming adverbs plus the extension file type (adverb `INEXT`) and the version number (adverb `INVERS`). The default convention for `INVERS` is reasonable — on input, zero means the highest (*i.e.*, most recent) version and, on output, the highest plus one.

Array elements in an image are designated by their pixel coordinates (counts). If $M(i,j,k,l,m,n,o)$ is a seven-dimensional array, the (1,1,1,1,1,1,1) pixel will be associated with the lower left hand corner of the image. The $i^{\rm th}$ (first) coordinate increments fastest and is associated with a column in each plane of the image. The $j^{\rm th}$ (second) coordinate is associated with a row in each plane. The other coordinates allow the image to be generalized to cover up to seven dimensions, *i.e.*, "cubes" and the like. The two adverbs `BLC` for bottom left corner and `TRC` for top right corner let you specify the desired sub-array in up to seven dimensions. When a sub-image is taken from an image, the pixel designation of any image element will usually change.

12.2 Process control features of *AIPS*

There are a number of tools available to assist you in scheduling and controlling the execution of *AIPS* tasks. They include taking the input from a text file rather than an interactive terminal, special adverbs to the verb `GO`, and even one or more detached, batch processing queues.

12.2.1 `RUN` files

If you have some lengthy sequence of commands to give to `AIPS`, especially if you will have to give essentially the same sequence more than once, you may find it helpful to prepare the sequence in a text file using your favorite and flexible text editor. In particular, it is cumbersome to write and edit procedures more than about five lines long in *AIPS* because of its primitive internal editor. Long procedures are best written as text files at your computer's monitor level, where the editing facilities will usually be much better. These text files can be transferred to `AIPS` easily using the `RUN` file facility.

Any commands that can be typed on the terminal to `AIPS` can be stored in your text file. The text files may be stored in a disk area of your choosing or in a public area identified by the logical name `$RUNFIL`. The name of the file must be all upper case letters, followed by a period, followed by your user number as a three-digit "extended-hexadecimal" number with leading zeros. (To translate between decimal and extended hexadecimal, use the *AIPS* procedures or the `AIPS` verbs called `EHEX` and `REHEX`.) To use the `RUN` file from your own disk area, define a logical name before starting `AIPS` to point to the area (see § 3.10.1). Then start up `AIPS` under your user number and enter

> VERSION = 'MYAREA' $^{\rm C}_{\rm R}$	where MYAREA is your disk area, or
> VERSION = ' ' $^{\rm C}_{\rm R}$	if `$RUNFIL` is to be used
> RUN *FILE* $^{\rm C}_{\rm R}$	to execute the file named *FILE.uuu*

where *uuu* is your user number if extended hexadecimal with leading zeros to make three digits. The file *FILE.uuu* or, if it does not exist, *FILE.001* will be executed by the above command. Note that minimum match also applies to `RUN` file names.

The first line of a `RUN` file is ignored by `AIPS`. You should type comments into your `RUN` files to remind you what they are doing. The first line, any line which begins with an `*` in column one, and all text following a `$` sign (in any line) are treated as comments which will not be compiled or executed by `AIPS`. All facilities in `AIPS` such as `GET`, `SAVE`, and `TGET` can be used in `RUN` files, with the sole exceptions of `RUN` itself and the pseudoverb `COMPRESS` which makes use of the `RUN` process.

AIPS programmers also provide a few `RUN` files for general use in each release *e.g.*, `VLACALIB`, `VLACLCAL` and `VLARESET`. They are normally used to create procedures helpful to some, but not all, *AIPS* users. These are stored under user number 1, but are available automatically to everyone. To use a procedure from one of them, type, for example:

> RUN VLAPROCS $^{C}_{R}$ to read and execute file VLAPROCS in AIPS. The text will be listed as it is read.

This file contains a number of procedure definitions including those named above. To execute the procedure, prepare the input adverbs as needed, and then type

> VLACALIB $^{C}_{R}$ to execute the calibration sequence of the simplified procedure.

AIPS— programmers will provide a file with the inputs and help information for all canned procedures. Thus you can do

> HELP VLACAL $^{C}_{R}$ to see help information on the procedure.

> INP VLACAL $^{C}_{R}$ to see the current inputs to the procedure.

The help and inputs functions do not require the RUN, but you must do the RUN to compile the procedure before you can use it.

12.2.2 More about GO

The verb GO is shown in examples throughout this *CookBook*. Don't overlook the fact that GO, like all other *AIPS* verbs, has its own inputs. You will be familiar with the ability to specify which task you want to execute either by an immediate argument, *e.g.*, GO UVSRT $^{C}_{R}$, or by the parameter TASK, *e.g.*, TASK 'UVSRT' ; GO $^{C}_{R}$. GO has two further parameters, DOWAIT and VERSION. The value of DOWAIT is passed to the task and instructs it to resume AIPS as soon as possible (DOWAIT FALSE $^{C}_{R}$) or to resume AIPS only after completing its operations (DOWAIT TRUE $^{C}_{R}$). The latter option lets the task return a meaningful error code to which AIPS may respond by aborting the current input line, procedure, FOR loop, etc. Note that the verb WAIT '*taskname*' also forces AIPS to wait for a task to complete, but it cannot respond to some failure in that task. For example, the line:

> GO UVSRT ; WAIT UVSRT ; GO UVMAP $^{C}_{R}$

may cause unwanted images to be generated by UVMAP if UVSRT fails for lack of disk space or some other reason. However, the line:

> DOWAIT TRUE ; GO UVSRT ; GO UVMAP $^{C}_{R}$

will not attempt to execute UVMAP if UVSRT fails. Note that AIPS will not get hung up when a task aborts even if DOWAIT is true. *This consideration is particularly important when you do multiple runs of* FITTP *to write your output tape. If one fails, the tape may even be rewound. If the next scheduled* FITTP *actually runs, it may write at the beginning of tape, destroying all previous data on the tape!*

VERSION, the last input to GO, is used to specify which version of the program you wish to execute. You might use this to select the TST or OLD versions of a task from the NEW version of AIPS, for example. VERSION also allows you to execute private versions of programs and even to check a list of areas for versions of your program. Type HELP VERSION $^{C}_{R}$ for details.

GO has another useful capability. Normally, in order to invoke a verb, you simply type its name, *e.g.*,:

> PRTMSG $^{C}_{R}$ to print the message file contents.

However, if you type, instead:

> GO PRTMSG $^{C}_{R}$

having forgotten that PRTMSG is a verb, then AIPS will actually execute:

> TPUT PRTMSG ; PRTMSG $^{C}_{R}$ to save the current PRTMSG inputs and then print the contents of the message file.

You can recover those inputs at a later time with TGET PRTMSG $^{C}_{R}$.

12.2.3 Batch jobs

The AIPS batch processor can be used to run jobs outside interactive AIPS. The job consists of a set of AIPS instructions which do not need user interaction. This excludes the TV, the Tektronix, and the tape-drive oriented tasks and verbs. RUN files may be used in batch jobs — as the batch editor facility is also primitive, they are particularly attractive to batch users. Older restrictions on which tasks may be run in queue 1 have been removed in the 15JAN96 release.

The instructions to be executed in the batch processor are prepared in a "workfile." The workfile can be made while in AIPS and detailed instructions are given by typing:

> HELP BATCHJOB $^{C}_{R}$

A simple example is given here:

> BATQUE = 2 ; BATCLEAR $^{C}_{R}$	to select queue 2 and clear its workfile.
> BATCH $^{C}_{R}$	to enter batch preparation mode.
< TASK = 'UVSRT' $^{C}_{R}$	(Notice < prompt. Begin typing as in AIPS.)
< INN = '3C16' ; INCL = 'UVDATA' $^{C}_{R}$	
< INSEQ 1 ; OUTN INN ; OUTCL INCL $^{C}_{R}$	
< OUTSEQ = 0 ; SORT = 'XY' $^{C}_{R}$	
< GO $^{C}_{R}$	(Batch AIPS always waits for a task to finish before continuing.)
< RUN XXXXX $^{C}_{R}$	(RUN files are good to use in a batch job.)
< GO $^{C}_{R}$	
< ENDBATCH $^{C}_{R}$	to leave batch preparation mode type in ENDBATCH spelled out in full.
>	(Resume normal interactive processing.)

To list a batch file, type:

> BATFLINE = 0 ; BATLIST $^{C}_{R}$

To edit line *n* in a batch file, type:

> BATEDIT *n* $^{C}_{R}$	
< *put text here* $^{C}_{R}$	to replace old line *n*.
< *some more text* $^{C}_{R}$	to insert more commands between old lines *n* and *n*+1.
< ENDBATCH $^{C}_{R}$	(spelled out in full.)
>	(Resume normal interactive processing.)

As with procedures (§ 12.3.2), if *n* is an integer, the existing line *n* is overwritten with the line or lines typed before ENDBATCH. If *n* is not an integer, the new lines are simply inserted between lines *n* and *n*+1. BAMODIFY provides, for workfiles, the same functions as MODIFY does for procedures.

Finally, the workfile can be submitted to the batch processor by typing:

> SUBMIT $^{C}_{R}$

The instructions are sent to a checking program which checks that the input is free of obvious errors. All RUN files are expanded and checked. If Checker (the task AIPSC*m* where *m* is some extended hexadecimal number < Z) approves, the job goes into the AIPS job queue, which is managed by QMNGR*n*. If you change your mind, the job can be removed from the queue and returned to the workfile with the verb UNQUE.

Batch has several limitations. First, devices which require interactive use (TV device, Tektronix device, and the tape drives) cannot be used in batch. Also, batch uses a different set of TPUT and TGET files. Thus, a TGET in batch does not get the adverbs from your last interactive use of the specified task. However, the AIPS facilities GET and SAVE are particularly useful for batch. You can use interactive AIPS to set up and test set(s) of procedures and adverb values and SAVE them in named files. These files may then be recovered by batch

for the routine processing of large sets of data. This is considerably more convenient than using the batch editor. Note that `SAVE` / `GET` files may become obsolete with new *AIPS* releases, but that improvements to the *POPS* language have made this quite unlikely.

At present, batch jobs are run after a short delay, on a first-come, first-served basis. After your job has been submitted successfully, type:

`> QUEUES` $^{C}_{R}$ to list jobs in the queue.

Note the `SUBMIT TIME` for your job. It will not start before that time. The messages generated by your batch job will be printed automatically into a text file. They are kept in your message file, however, and can be reprinted or examined later via `PRTMSG` with `PRNUMB` set to the *AIPS* number of the batch queue. Printer output for batch jobs is concatenated into a file either specified by the user with `OUTPRINT` or a file named `PRTFIL:BATCH`*jjj*`.`*nnn*, where *jjj* is the job number and *nnn* the user number both in extended hexadecimal. Note, this means batch job printouts are concatenated in (normally) one file and are not automatically printed. An interactive `AIPS` can interact with a batch job via `TELL`; see § 5.3.1.

12.3 `AIPS` language

`AIPS` contains a basic set of symbols and keywords which are needed to construct a computer language, as well as the symbols needed by the application code. A list of the basic symbols is given in the help file called `POPSYM`, reproduced below:

```
Type:  Symbols used in the POPS interpretive language

    VERB      USE        COMMENTS

                   ----Arithmetic expressions

     +       A + B       Add the expression A to B
     -       A - B       Subtract the expression B from A
     *       A * B       Multiply the expression A with B
     /       A / B       Divide the expression A by B
     **      A ** B      Calculate A to the power B
     ( )     (A+B)*C     Grouping expressions as desired
     =       A = B       Store the value of B into A
     ,       A = 3,5,4   Separator of elements in an array
     :       TO          Equivalent to the verb TO
     ;                   Separator between AIPS statements

                   ----Logical expressions

     >       A > B       A greater than B
     <       A < B       A less than B
     =       A = B       A equal B (numeric or string)
     >=      A >= B      A equal to or greater than B
     <=      A <= B      A equal to or less than B
     <>      A <> B      A not equal to B (numeric or string)
     !       A ! B       A or B
     &       A & B       A and B
     ^       ^ A         not A
```

```
                ----String expressions

!!       A !! B        string = string A followed by string B
SUBSTR   SUBSTR(A,i,j) string = chars i through j of string A
LENGTH   LENGTH(A)     position last non-blank in A
CHAR     CHAR(A)       convert number A to string
VALUE    VALUE(A)      convert string A to number


                ----Looping constructions

(FOR-TO-BY-END)        FOR I=1 TO 7 BY 2
                       <any valid set of AIPS syntax>
                       END

(WHILE-END)            WHILE <any logical expression>
                       <any valid set of AIPS syntax>
                       END

(IF-THEN-ELSE-END)     IF <any logical expression>
                       THEN <any valid set of AIPS syntax>
                       ELSE <any valid set of AIPS syntax>
                       END

                ----Built-in functions

ATAN          Arctangent (one argument)
ATAN2         Arctangent (two arguments)
COS           Cosine (degrees)
SIN           Sine (degrees)
TAN           Tangent (degrees)
EXP           Exponential
LN            Log base e
LOG           Log base 10
SQRT          Square-root
MAX           Maximum i.e. X = MAX (A, B)
MIN           Minimum i.e. X = MIN (A, B)
MODULUS       Root-square sum of two arguments
MOD(A,B)      A - (A/B) * B  i.e. remainder of A/B
CEIL(A)       Lowest integer >= A
FLOOR(A)      Highest integer <= A

                ----Procedure building verbs

PROC      PV  Begin building a procedure
PROCEDUR  PV  Begin building a procedure
LIST      pV  List a procedure
EDIT      PV  Edit a procedure
ENDEDIT   PV  End editing a procedure
ERASE     PV  Delete line(s) of a procedure
MODIFY    PV  Modify a line in a procedure
RETURN    V   Last statement in a procedure
FINISH    PV  End procedure building
```

```
                ----Variable declarations

SCALAR    pV  Declare scalars
ARRAY     pV  Declare arrays
STRING    pV  Declare strings

                ----Input/Output functions

PRINT     V   Print the following keyword value(s)
TYPE      V   Print the following keyword value(s)
READ      V   Read value(s) from terminal after # prompt

                ----Other information

CORE      pV  Amount of core left in POPS
COMPRESS  PV  Compress the core area, recovering lost space
CLRTEMP   V   Clear the temp data array
DEBUG     pV  Debug: turns on compiler debug information
DUMP      V   Dump K array on terminal screen
SCRATCH   PV  Remove procedures in POPS
$         PV  Makes rest of input line a comment
```

12.3.1 Using *POPS* outside of procedures

POPS variables are either numeric or character valued and may be multi-dimensional arrays. Once created, all variables are available everywhere, *i.e.*, they are global. You may manipulate these variables on the command line using most of the symbols listed above. In fact, you have been doing this while setting the adverbs for all the tasks and verbs described in preceding chapters. The more advanced user may wish to use some of the language features in order to simplify repetitive data processing. Here are some simple examples of uses of the `AIPS` language:

> TYPE (2 + 5 * 6) $^{C}_{R}$	32 is written on the terminal.
> TYPE 'X =', ATAN (1.0) $^{C}_{R}$	`X = 45` is written on the terminal.
> TYPE 'MAPNAME ', INNAME, INCLASS, INSEQ $^{C}_{R}$	`MAPNAME 3C138 IMAP 1` is written on the terminal.

The simplest loop capability in `AIPS` uses the pseudoverbs `FOR`, `TO`, and `BY` for repetitive operations. Such loops are primarily intended for use in "procedures" (see § 12.3.2). If a `FOR` loop can be typed fully on one input line, it will also work outside the procedure mode. The following example shows how to delete a series of images with the same name and class and with consecutive sequence numbers 1 through 10:

> INNA 'TEST' ; INCL 'IMAP' $^{C}_{R}$	to set (fixed) name parts.
> INDI 1 $^{C}_{R}$	to set (fixed) disk number.
> FOR INSEQ = 1 TO 10 ; ZAP ; END $^{C}_{R}$	to delete the files.

`FOR` loops must be terminated with an `END`. The following example shows how to delete every other file in a catalog with 20 entries:

> FOR INSEQ = 1 TO 20 BY 2 ; GETN(I) ; ZAP ; END $^{C}_{R}$

More extensive examples are shown in the sections below on procedures.

In some cases, you may wish to manipulate character strings to give your files meaningful names — particularly if your `RUN` file or procedure operates repetitively on many similar files. The verbs for character manipulation are listed above. As an example,

> OUTNAME = 'CLEAN' !! CHAR(BLC(3)) $^{C}_{R}$	to name each output file after the input image plane.

Note that trailing blanks are ignored. If you wanted a space after `CLEAN` before the plane number, use

> OUTNAME = 'CLEAN' $^{C}_{R}$	to set the basic form.
> SUBSTR (OUTNAME , 7 , 12) = CHAR (BLC(3)) $^{C}_{R}$	to alter only the last six characters of `OUTNAME`.

12.3.2 Procedures

Procedure building is a way to combine keywords in `AIPS` in any convenient way to obtain useful constructs. For complicated sequences, it is easier to prepare and debug procedures in `RUN` files (§ 12.2.1) than to prepare them in interactive *AIPS*. A procedure is given a name, with or without arguments, and then can be treated as an *AIPS* verb. As an example, consider a procedure to load an image on the TV, set the cursor, and fit for the maximum intensity. You could type the following on your terminal:

> PROC MFIT (I) $^{C}_{R}$	to define procedure `MFIT` with one argument `I`. (`I` and `J` are two dummy adverbs which are already defined in `AIPS`.)
: GETNAME(I) $^{C}_{R}$	(Notice the prompt symbol `:` . This means that we are in the procedure-building mode.)
: TVLOD ; IMXY ; MAXFIT $^{C}_{R}$	to load the image, produce and read the cursor, and fit the maximum near the cursor position when a TV button is pressed
: RETURN $^{C}_{R}$	to designate a return point in the procedure — normally not required at the end of a procedure unless a value is to be left on the stack, *i.e.*, a function.
: FINISH $^{C}_{R}$	to designate the end of the procedure-building mode and to get back into the normal (prompt `>`) mode.
>	Notice the prompt symbol, you are back to interactive input mode.

When you type such a procedure into `AIPS`, the code is compiled as you type. Most syntax errors are spotted immediately and will unceremoniously dump you out of procedure mode. However, all lines written before the detected error are kept and the procedure editor can be used to continue.

The *AIPS* procedure editing capabilities are quite primitive. If you want to build procedures longer than about five lines, we therefore recommend using permanent storage files in the "`RUN`" area, as discussed in § 12.2.1 below.

To list the procedure `MFIT`, type:

> LIST MFIT $^{C}_{R}$

This will produce the following:

```
1PROC MFIT(I)
2GETNAME(I)
3TVLOD ; IMXY ; MAXFIT
4RETURN
5FINISH
```

The procedure is identical to what you typed, with line numbers added.

Procedures are edited line by line. To edit line 2 in the above procedure, type:

> EDIT MFIT 2 $^{C}_{R}$	to enter Procedure editing mode.

; GETNAME(I) ; TVLOD $^{C}_{R}$ (Notice prompt symbol ; for procedure-editing mode.) This change replaces the old line 2 adding a `TVLOD`.

; IMXY ; MAXFIT $^{C}_{R}$ to add a line between the changed line 2 and old line 3.

; GETNAME(I+1) $^{C}_{R}$ to add yet another line after 2.

; ENDEDIT $^{C}_{R}$ to terminate procedure editing.

> LIST MFIT $^{C}_{R}$

Listing the modified procedure will give:

```
1PROC MFIT(I)
2GETNAME(I) ; TVLOD
3IMXY ; MAXFIT
4GETNAME(I+1)
5TVLOD ; IMXY; MAXFIT
6RETURN
7FINISH
```

To delete lines *n* through *m* from a procedure, type:

> ERASE *xxxxxxx n* : *m* $^{C}_{R}$ where *xxxxxxx* is the name of the procedure.

To insert one or more lines between lines 3 and 4 of a procedure, type:

> EDIT *xxxxxxx* 3.5 $^{C}_{R}$

; *(Type additional lines as needed.)*

; ENDEDIT $^{C}_{R}$

Notice that the lines are renumbered after any `EDIT` or `ERASE`. Use `LIST` to determine the new line numbers.

The pseudoverb `MODIFY` lets you modify characters within a line of a procedure to correct the line or change its meaning. The grammar is:

> MODIFY *proc-name line-number* where *proc-name* is the name of the procedure and *line-number* is the line number in the procedure as shown by `LIST`.

`MODIFY` begins by showing the existing line with a `?` as a prefix. Then it prompts for input with a `?` To keep the character of the original line immediately above the cursor, type a blank (space-bar). To delete that character, type a `$` (dollar-sign). To replace that character, type the new character (to get a new blank character, type an `@` sign). Insertions complicate things. To insert text prior to the character immediately above the cursor, type a \ followed by the desired text followed by another \. You may continue to `MODIFY` the remainder of the line, but you must remember that the current character position in the old line is to the left of the current cursor position by the number of inserted characters (including the 2 \'s). `MODIFY` will display the resulting line of code after you hit a carriage return ($^{C}_{R}$) and does not change the line number. Example:

```
> MODIFY  ED  2 |CR
?TYPE 'THIS IS EDS PROC'
?              MY@\NEW\     @FOR@EXAMPLE' |CR
TYPE 'THIS IS MY NEW PROC FOR EXAMPLE'
> MODIFY  ED  2 |CR
?TYPE 'THIS IS MY NEW PROC FOR EXAMPLE'
?                 $$$$    \EDURE,\ |CR
TYPE 'THIS IS MY PROCEDURE, FOR EXAMPLE'
```

More information about procedure building and editing can be found by typing:

> HELP PROCEDUR $^{C}_{R}$

Procedure creation and editing uses up the limited memory of the *POPS* processor. When the memory is gone, the message `BLEW CORE!` will appear and you can do no more procedure writing without starting over

(*i.e.*, RESTORE 0 $^{C}_{R}$). CORE $^{C}_{R}$ will tell you how much memory is left. If the memory remaining appears small, try COMPRESS to recover the lost memory (in 15JAN96 and later releases). COMPRESS might even work after a BLEW CORE! if you are lucky.

The procedure MFIT can be executed by:

> MFIT(*n*) $^{C}_{R}$ where *n* is the slot number of the appropriate image.

(It is assumed that the correct disk unit number has already been set.) This procedure can also be part of another procedure or put in a loop. For example:

> FOR I= 1 TO 10 BY 2; MFIT(I) ; END $^{C}_{R}$

will load the TV and fit the maximum for the first ten images on the appropriate disk.

All the syntax available in AIPS is available for use inside procedures *except for certain pseudoverbs*. The "prohibited" pseudoverbs include SAVE, GET, STORE, RESTORE, PROCEDURE, EDIT, ENDEDIT, MODIFY, LIST, CORE, SCRATCH and COMPRESS. Others do not make much sense in procedures, including MSGKILL, DEBUG, and ABORTASK. Other pseudoverbs are, however, particularly useful in procedures. These include TGET, TPUT, and GO.

Several verbs are extremely useful in procedures. To set the image name adverbs to those visible on the TV, use TVNAME. When GETN accesses an empty slot, an error condition is raised and the procedure dies. To handle this error condition in your procedure, use EGETN *n* instead and test the adverb ERROR which will be "true" if the slot is empty. CHKNAME may be used similarly to check on the existence of files with computed names. Some tasks require image-data dependent inputs. To help handle this in general procedures, the verb GETHEAD allows all header parameters to be fetched into adverbs. Type EXPLAIN GETHEAD $^{C}_{R}$ for details. There are numerous arithmetic functions, useful looping constructions, and powerful methods of building arithmetic, logical, and string expressions in *POPS*. See § 12.3 above for a list of these. CLRTEMP may be used in procedures which do a lot of looping. It clears the temporary space used to hold substrings and other temporary constants. A procedure that does much string manipulation is likely to overflow this area after a number of iterations. The message BLEW TEMP C! usually accompanies the overflow.

Once a procedure is written and edited, it can be stored in a SAVE file for later use. Procedures are lost when another GET file is obtained. Procedures can be stored more permanently in RUN files which are described in § 12.2.1 above. To list the names of all procedures currently in your AIPS environment, type:

> HELP PROCS $^{C}_{R}$

This will list internal *AIPS* procedures as well as your own.

Several procedures have been built into *AIPS*. In particular, some procedures are defined in the system RUN file VLAPROCS to aid *routine* calibration of VLA data. Currently, these are VLACALIB, VLACLCAL and VLARESET. They may be useful templates for your own procedures. If you are unfamiliar with the use of *AIPS* procedures, looking at these system-supplied ones will help you to understand, and see the power of, this feature of *AIPS*.

12.3.3 Writing your own programs with *POPS*

You may want to add your own programs to *AIPS*. It is not a trivial matter to generate an *AIPS*-standard FORTRAN program (see HELP NEWTASK $^{C}_{R}$, the *Going AIPS* manuals, and § 12.5 below). Simple but powerful programs may however be built as procedures that use existing verbs and tasks.

Consider a the following example. (This example is presented as if it were typed into an interactive AIPS. In practice, you will probably prefer to prepare such a complicated procedure as a RUN file.) We wish to determine the average value and rms scatter at any pixel location in a set of *n* images. We shall demand

that the n images all have the same `INNAME` and `INCLASS` with sequence numbers between 1 and n. The `RENAME` verb can be used to name the images appropriately. We could call this procedure:

```
AVGRMS (PIXXY, N, AVG, RMS)
```

where

PIXXY	is the pixel location in the images,
N	is the number of images,
AVG	is the average value at the pixel location, and
RMS	is the rms value at the pixel location.

The array adverb `PIXXY` is a standard `AIPS` adverb, but the variables `N`, `AVG`, and `RMS` are unknown to `AIPS`. These must be defined before we can write the procedure `AVGRMS`. This is done by a short dummy procedure which we will call `DAVGRMS`:

```
> PROC DAVGRMS CR                          to define dummy procedure.
:  SCALAR N, AVG, RMS CR                   to define scalar adverbs.
:  FINISH CR                               to exit from dummy procedure.
```

Now begin the procedure `AVGRMS`:

```
> PROC AVGRMS (PIXXY, N, AVG, RMS) CR      to enter procedure building mode.
:  SCALAR SUM, SUM2 CR                     to define more variables.
:  ARRAY VAL(20) CR                        to define an array.
:  RMS = 0 ; SUM = 0 ; SUM = 0 CR          to zero some variables.
:  FOR INSEQ =1 TO N CR                    to begin summing loop.
:      QIMVAL CR                           to get pixel value at PIXXY in image
                                           INNAME INCLASS INSEQ.
:      VAL(INSEQ) = PIXVAL CR              to save pixel value (placed in PIXVAL by
                                           IMVAL) in our array.
:      SUM = SUM + PIXVAL CR               to sum for averaging.
:      SUM2 = SUM2 + PIXVAL * PIXVAL CR    to sum for rms.
:      END CR                              to mark end of FOR loop.
:  AVG = SUM / N CR                        to get average value.
:  IF N > 1.5 THEN CR                      to check if N > 1.
:      RMS = SQRT((SUM2 - N*AVG*AVG) / (N * (N-1))) CR   to calculate rms if N > 1.
:  ELSE ; TYPE 'N TOO SMALL', N CR         to warn the user.
:      END CR                              to mark end of IF clause.
:  TYPE 'AVG=',AVG,'RMS=',RMS,'AT PIXEL',PIXXY CR
:  TYPE ' # ',' VAL ',' ERROR ' CR         to print a heading.
:  FOR INSEQ = 1 TO N CR                   to begin another loop.
:      SUM = AVG - VAL(INSEQ) CR           to get residual.
:      TYPE INSEQ, VAL(INSEQ), SUM CR      to print data and residual.
:      END CR                              to mark end of FOR loop.
:  FINISH CR                               to return to regular AIPS mode.
>
```

The above procedure could be run as follows. First fill in the adverbs `INNAME`, `INCLASS` and `PIXXY` with the desired values. Then type:

```
> AVGRMS (PIXXY, n, AVG, RMS) CR
```

where n is the number of images to average. The average and rms will be calculated and written on the terminal and in the message file. This procedure could be used by another procedure. Suppose we wanted

to determine the average and rms of the pixels within a rectangular area. If we set `BLC` and `TRC` in the usual way to define the rectangular boundary, then the procedure:

```
> PROC AVGARRAY (BLC, TRC) CR                    to define new proc.
:  FOR I = BLC(1) TO TRC(1) CR                   to loop over x-coordinate
:      FOR J = BLC(2) TO TRC(2) CR               to loop over y-coordinate.
:          PIXXY = I , J CR                      to set pixel coordinates for AVGRMS.
:          AVGRMS (PIXXY, N, AVG, RMS) CR
:          END ; END CR                          to end y loop, then x loop.
:  FINISH CR                                     to end the proc., RETURN not needed.
>
```

will calculate the average value and rms at this array of pixel locations. Please note that this is just an example. The verb `IMSTAT` performs this function much more efficiently.

12.4 Remote use of *AIPS*

AIPS users do not always find themselves seated in front of the main display screen of the computer which they intend to use for their data analysis. *AIPS* provides facilities for such users which depend to some extent on the nature and location of the workstation or terminal at which the user is seated. Nearly seamless function is provided to a user seated at a workstation on the local Ethernet well known to the *AIPS* installation. Substantial capabilities are still available to the user at a more distant workstation capable of X-Windows display, especially if that workstation can also run *AIPS* programs such as the TV server and remote tape server. Even the user at a simple terminal or workstation, capable of emulating a Tektronix 4010, can still get some interactive displays. It is only the users at very simple terminals who will be rather limited in their interactive use of *AIPS*.

12.4.1 Connections via X-Windows

In the following discussion, we will assume that you need to do your computing on a computer called *Server* and that you are sitting in front of a workstation called *MyHost*.

If *Server* and *MyHost* are both on the same local area network and both have the same byte ordering, then they should have *AIPS* installed with both of them shown as being at the same *AIPS* "site." In this case, you simply `rlogin` or `telnet` into *Server* from a window on *MyHost* and issue the `aips` command as described in § 2.2.3. The `aips` command will recognize that you are coming in from *MyHost* and will start the message, graphics, and TV servers on *MyHost* if needed. If you want to share data areas between the two computers, you may add a `da=`*MyHost* on the command line and `AIPS` will run with all data areas from both machines. (The disk systems must be auto-mountable between the two computers.) The `aips` command also lets you select the most convenient printer within your local area network for use in your *AIPS* session. Other forms of data transfer, including magnetic tapes, will be discussed later.

If *Server* and *MyHost* have different byte orders or are not both on the same local area network, then they cannot be at the same *AIPS* site. If both machines have *AIPS* installed and the version of *AIPS* on *Server* is at least `15JAN96`, then you may run on *Server* with *MyHost* treated as a "guest" (if the *AIPS* Manager of *Server* has allowed for one or more guests). Such guests do not interfere with normal users of *Server* — or other guests — in any way so long as limits on the number of interactive *AIPS* or number of guests are not exceeded. Again, `rlogin` or `telnet` into *Server* from a window on *MyHost*. Make sure that the environment variable `DISPLAY` on *Server* is set to *MyHost*`:0` and that *Server* is mentioned in your `.rhosts` file on *MyHost*. Then issue the usual `aips` command. You do not have to give the `tv=` option, but

you may give `tv=`*MyHost* if you wish.. This will start the message, graphics, and TV servers on *MyHost* if needed while giving some messages to indicate your "guest" status. If the servers fail to start and messages such as "`Cannot start remote TV servers...`" appear, then you must start the servers using the `aips` command on *MyHost*. The displays will work without restarting `AIPS` on *Server*. Thereafter, give the `aips` option `tvok` to *Server* (rather than `tv=`) to suppress the annoying messages. The main consequence of being a guest is that the files which describe the state of the TV and graphics displays reside on *Server* and are not the same as the files on *MyHost* which also purport to describe the state of the displays (on *MyHost*). Thus, if you alter the TV display from more than one computer, there will be some confusion as to its state. The display status will be maintained properly, so long as you use the displays only from one computer at a time. (These status files are shared among the computers within an AIPS site, allowing more than one to know the display status and contents in detail at the same time.) Since the display refreshing is handled locally by programs running in *MyHost*, this level of connection supports nearly full interactivity including such demanding displays as `TVBLINK` and `TVMOVIE`.

If *MyHost* does not have AIPS installed, or if the Manager of *Server* has not made provision for guests, then you may run the message, graphics, and TV servers on *Server*, with the X-Windows display set to *MyHost*. You must ask for this explicitly, setting `tv=`*MyHost*`:` (note the trailing colon). This mode of operation is not encouraged since there can be only one of each of the servers actually running on *Server*. Thus, if you are running them displaying on *MyHost*, then the user at the console of *Server* cannot get his/her usual displays. Furthermore, the display refreshing has to be transmitted over the network, making some of the interactive displays too slow to be useful. Nonetheless, there are circumstances in which this mode of operation is the only one available. If the compute server lacks an X-Windows display console or if it is run for one user at a time, then this mode is not unreasonable. Be sure to close down the display servers as soon as Use the `Esc` key in the `AIPSTV` (`XAS`) window, `CTRL-C` in the `MSGSRV` window, and `CTRL`-left-mouse-button you finish using them in order to make them available to other users. in the `TEKSRV` window.

12.4.2 Connections to a terminal

If you do not wish to hog the display servers on *Server*, you may still do some interactive AIPSing if your workstation window is able to emulate a Tektronix 4010 terminal, or you are at a terminal capable of this emulation. (Note that most `xterm` displays may be switched between a "Tek mode" and the normal "VT mode" by pressing the `Control` key and the middle mouse button.) To operate in this mode, log in to *Server* from your terminal or workstation window and start `AIPS` with the command-line option `REMOTE`. This option will disable all TV functions, will cause all task messages to come to your terminal or window, and will cause any graphics ("TK") displays to be sent to your terminal or window. You may display an "image" by creating a contour drawing with `CNTR` and then displaying the plot file with `TKPL` Interactive cursor verbs and procedures such as `TKXY`, `TKPOS`, and `TKWIN` may then be used. The old 4010 Tektronix display mixed plotting and text in a less than elegant fashion which is slavishly honored by most emulations. This is unfortunate, but usually does not prevent the display from being used for simple position selection and the like.

The AIPS task `TXPL` is a powerful tool for remote users without any, or correct, Tektronix 4010 emulation. It reads an extension file of type `PL` and translates the graphics commands in that file to an alphanumeric display for a "dumb" terminal. `TXPL` may be exactly what you need for AIPS applications that depend on scanning the *shape* of a plot rather than its fine detail. Common examples are viewing the shape of visibility functions produced by `UVPLT` (to guide self-calibration or to diagnose interference) or examining calibration solution plots from `SNPLT`. `TXPL` can also usefully interpret simple contour plots or even grey scales(!) for a remote user. It is often much faster to use `TXPL` to diagnose the state of your AIPS data processing over a low-bandwidth link than to use `TKPL` to execute a stream of Tektronix graphics instructions (even if you have full Tektronix 4010 emulation).

12.4.3 Remote data connections

Like the message, graphics, and TV servers, there is also a remote data server in AIPS in the form of a remote "tape" dæmon called TPMON. All computers that run AIPS run a copy of TPMON to serve FITS-disk files plus a copy of TPMON for each real AIPS tape device on the computer. Some computers start these TPMONs when the are booted while others wait until someone runs AIPS on them or orders the tape servers to run by using the tp= option for the computer from some other computer (§ 2.2.3). If you are computing on the remote *Server* and wish to use an AIPS tape drive on your computer, type on *Server*:

% aips tp=*MyHost* $^{C}_{R}$ to start AIPS on *Server* and to start the TPMONs on *MyHost*.

Then, within AIPS, enter:

> REMHOST '*MyHost*' $^{C}_{R}$ to specify your computer as the tape server.

> REMTAPE *n* $^{C}_{R}$ to specify the n^{th} AIPS tape device on *MyHost*.

> INTAPE *m* $^{C}_{R}$ to specify a *Server* tape number one or two higher than the number of real tape devices on *Server*. These match those shown for REMOTE as you started AIPS; see § 3.9 for an example.

> MOUNT $^{C}_{R}$ to mount the remote tape.

You may then use the remote tape as you would any other tape in AIPS. See § 3.9, § 4.1, § 5.1, and § 6.1 for examples of normal tape usage.

If you have a FITS-disk file (§ 3.10.3) on *MyHost* which you wish to read into *Server*, then you may set INFILE to *MyHost*::*logical*:*filename*. Note that double colons connect the host name to the logical name for the disk area and a single colon connects the logical name to the name of the file within that disk area. Note also that the logical name must be one known to TPMON when it is started. Put your file in a standard area such as FITS, which is known to all of AIPS, or create the logical variable in a window on *MyHost* (*e.g.*, with setenv or export — see § 3.10.1)and then start the TPMONs from that window with the aips command. Similarly, you may use FITTP to write a FITS-disk file onto *MyHost* of an AIPS image or *uv* data set. Enter OUTFILE with the form shown above for INFILE.

Unlike the display servers, TPMON can read and write disk files and magnetic tape devices. Given the hostile environment now found on the Internet, this poses a security problem. Therefore, beginning with the 15JAN96 release (and patched versions of 15JUL95), TPMON checks every connection request to see if the remote computer has permission to use its services. To do the remote tape operations described above, you must have the AIPS Manager for *MyHost* alter the appropriate files to give *Server* permission to use the TPMONs on *MyHost*. This should have been done already if *Server* and *MyHost* are on the same AIPS "site" or are routinely used together.

12.4.4 File transfer connections

The techniques discussed above apply to many computer configurations and to most tools within AIPS. They do not, however, handle the outputs of printing and plotting tasks. Nor do they provide much support for small computer systems that have no AIPS capability of their own. For these situations, the results of your computations will need to be written onto disk on *Server* and then transferred over the network to *MyHost*.

AIPS does not support remote printers explicitly. However, all AIPS tasks and verbs which generate printer output support the OUTPRINT adverb. With this adverb, you may specify a disk text file to receive the printer text. If you specify the same file for successive printer verbs or tasks, the outputs will be concatenated. (AIPS batch jobs do this automatically to concatenate all printer displays for the job.) You may then copy the text file to *MyHost* for editing, printing, or whatever. If you wish to print the whole file on a PostScript printer on *MyHost*, you may wish to run F2PS on the text file on *Server* and copy the

result to *MyHost*. *AIPS* provides a "filter" program to convert plain (or Fortran) text files to PostScript for printing on PostScript printers. The command

`$ F2PS -`*nn* `<` *file* `>` *outfile*

will convert text file *file* to PostScript format file *outfile*. The parameter *nn* is the number of lines per page used inside *AIPS*; use 97 for a small font in "portrait" form or 61 for a larger font in "landscape" form.

The *AIPS* task **LWPLA** also has an option to write its output to a disk file in encapsulated PostScript form using the **OUTFILE** parameter. Similarly, **TVRGB** and **TVCPS** use the same adverb to write PostScript text files containing their three-color displays. Numerous other *AIPS* tasks offer the option to write details of the operation to a text file specified with **OUTFILE**. These include **SLICE** (slice), **IMEAN** (histogram), **GAL** (fit results), **POSSM** (spectrum), **FRPLT** (spectrum), **HITEXT** (history), **UVCRS** (*uv*-plane crossings), **CONPL** (convolving functions), etc.

files may be written with **FITTP** and read with **FITLD**, **UVLOD** and **IMLOD**. For efficiency reasons, these are binary files rather than the text files produced by everything else. *AIPS* table files will have their contents transferred by these tasks along with the main data files. To put an *AIPS* table in a disk file in text form, use **TBOUT** to write a simple text file or **TAPQL** to write a **POSTQUEL** script to be read by the (free) **POSTGRES** database program. Files in the form written by **TBOUT** may be read back into *AIPS* with **TBIN**.

To transfer the FITS-disk files and text files between *MyHost* and *Server*, some standard network file transfer must be used. For example, use **ftp** on *MyHost* with

`%` `cd` *MyArea* $^{C}_{R}$	to switch to the disk area on *MyHost* used for your files.
`%` `ftp` *Server* $^{C}_{R}$	to start **ftp** to the remote system.
`Name (`*Server*`:...):` *loginame* $^{C}_{R}$	to log in to account *loginame*.
`Password:` *password* $^{C}_{R}$	to give the account's password.
`ftp>` `cd` *directory* $^{C}_{R}$	to change to the *directory* name containing the file on *Server*.
`ftp>` `binary` $^{C}_{R}$	to allow reading of a binary file — required for FITS-disk files, okay for text files.
`ftp>` `hash` $^{C}_{R}$	to get progress symbols as the copy proceeds — a good idea for large files.
`ftp>` `put` *filename* $^{C}_{R}$	to send *filename* from *MyHost* to *Server*.
`ftp>` `get` *anothername* $^{C}_{R}$	to send *anothername* from *Server* to *MyHost*.
`ftp>` `quit` $^{C}_{R}$	to exit from **ftp**.

The files should then be in the desired directories. You may have to rename them, however, to a name in all upper-case letters since that is required if they are to be read in by *AIPS*.

12.5 Adding your own tasks to *AIPS*

This Section is a brief guide for the user who wants to modify an existing task in *AIPS* or to write a new task. While it is difficult to write an *AIPS* task from scratch, *AIPS* contains several template tasks that are designed to hide most of the work from the "occasional" programmer. Anyone familiar with FORTRAN and a little of the system services of their local computer should be able to add convenient tasks to their local version of *AIPS* with a little practice and patience.

12.5.1 Initial choices to make

The simplest way to write an *AIPS* task is to modify one of the four template tasks, **TAFFY**, **CANDY**, **FUDGE** or **UVFIL**. These tasks handle the *AIPS* I/O, contain extensive documentation, need to be modified in only

a few well-defined places, and can be easily interfaced to user subroutines. They are limited in their ability to handle many input and output images, however, and generally operate on one image row or one visibility point at a time. Softening of these limitations will be discussed in § 12.5.5.

`TAFFY` — This template task reads an existing image in *AIPS* line by line, modifies the line of data as desired and then creates and writes the modified image in *AIPS*. This template task might, for example, be used to blank pixels in an input image in some specified manner.

`CANDY` — This task is similar to `TAFFY` except the input data are contained in an auxiliary file which is FORTRAN readable and outside *AIPS*. The task can transfer an image in any reasonable format outside *AIPS* into the *AIPS* data structure.

`FUDGE` — This template task reads an existing *AIPS uv* database, modifies this database point by point, and then creates and writes this modified output *uv* database in the *AIPS* catalog. This template task might, for example, be used to modify a *uv* data base for which the time parameter has been incorrectly written.

`UVFIL` — This task is similar to `FUDGE` except that the input *uv* data are contained in one or two auxiliary files, which are FORTRAN readable and outside *AIPS*. The task is useful for translating visibility data in an arbitrary format into an *AIPS* cataloged *uv* data set, or for computing a *uv* data set from model sources.

If you wish to make a minor change in an existing *AIPS* task, it will be simpler to copy the *AIPS* task itself and modify it as needed. For example, if you wanted to add a option in `COMB` to combine two input images in a new way to obtain a resultant image, it would be better to start with `COMB` itself than with one of the templates. However, non-trivial changes to major tasks will require a careful look at the code, which is generally well documented and segmented.

Changes to tasks that make plots and/or use the TV and other graphics devices can be tricky, but are useful sometimes. Changes to imaging and deconvolution software should not be attempted unless the changes are almost trivial. Modifying existing verbs or creating new ones require you to find which `AU`*xx*`.FOR` routine is involved and then require the entire `AIPS` program to be relinked. *Going AIPS* describes some of the considerations. The new pseudoverbs `VERB` and `PSEUDOVB` allow you to update your `AIPS` vocabulary using your own procedure rather than by changing the supplied vocabulary files and executing `POPSGN` as described in *Going AIPS*. In all cases, do not put your modified code into the standard *AIPS* code areas unless your local *AIPS* Manager agrees and has made a backup copy of the original code and executables.

12.5.2 Getting started

Even if you have special privileges in *AIPS*, it is wise to generate and link new code in your own user directory rather than in your installation's designated local *AIPS* directory. After the code has been written and tested, check with your local *AIPS* Manager about installing it in a public area. Decide which template task or other *AIPS* task you want to modify, then follow these instructions after logging into your private area.

Under Unix

% source /AIPS/LOGIN.CSH $^{C}_{R}$	to get the basic *AIPS* system logicals under a c-shell, or
% . /AIPS/LOGIN.SH $^{C}_{R}$	to get the basic *AIPS* system logicals under a korn, bourne, or bash shell. (These assume that your *AIPS* system home directory is called `/AIPS`).
% $CDTST (or $CDNEW) $^{C}_{R}$	to set up the logical assignments for programming.
% PROG *TASK* $^{C}_{R}$	to locate the task called *TASK*.

% LIBS $*AREA* > *NTASK*.OPT $^{C}_{R}$	to create a file of linking information for a task found in area *AREA*, *e.g.*, **APLPGM** for **FUDGE** and **TAFFY**, **APGNOT** for **CANDY** and **UVFIL**, **YPGM** for **IMEAN**, etc.
% cp $*AREA*/*TASK*.FOR *NTASK*.FOR $^{C}_{R}$	to copy code for *TASK* into your area and to give it a new name (≤ 5 letters).
% cp $HLPFIL/*TASK*.HLP *NTASK*.HLP $^{C}_{R}$	to copy and rename the inputs/help file for the task.
% setenv MYAIPS 'pwd' $^{C}_{R}$	to define **MYAIPS** (must be uppercase) as your disk area for *AIPS* programs, or
% export MYAIPS='pwd' $^{C}_{R}$	to define the **MYAIPS** environment variable under korm. bash, and bourne shells.

12.5.3 Initial check of code and procedures

Before modifying the task in any way, it is wise to compile, link and execute the unchanged task to check that the original code is sound and that all of the *AIPS* logical assignments have been properly set. The duplicated *AIPS* task should run identically to the original; the template tasks are set up to duplicate the input data set with no changes and default parameters. In this way errors or other problems associated with the generation of new tasks can be found and corrected before getting into the quagmire of bugs that you are about to add. Note that **CANDY** and **UVFIL** require external data files and are therefore not so easily checked.

One change is needed in the FORTRAN program before checking. In the main program, about 60 lines down, there is a data statement which identifies the task.

`DATA PRGM /'`*TASK* `'/`

Change this to

`DATA PRGM /'`*NTASK*`'/`

The maximum number of characters in a task name is five and the **DATA** statement must be in the above form. There is no need to change the help file yet, as long as it is named **NTASK.HLP**.

To compile and link the task, type:

% COMLNK *NTASK* *NTASK*.OPT $^{C}_{R}$

To compile the task for debugging, add the **DEBUG NOPURGE** options to the **COMLNK** line. There should be no error messages and no significant warnings.

After successful compilation and linking, try executing the task. Under Unix, stay logged in to your area. To initiate the **AIPS** program, type:

% aips $^{C}_{R}$	or
% aips debug $^{C}_{R}$	to use the debugger (*e.g.*, **dbx**) on tasks compiled with the **DEBUG** option. You will have to tell the procedure which debugger you want and that you do not want to start **AIPS** itself under that debugger.

Your *AIPS* Manager may have to make some arrangements for you to activate *AIPS* from your logon. Once you have started *AIPS*, type

> VERSION 'MYAIPS' $^{C}_{R}$	to specify the location of inputs and help information and of the task executable.

Set up the input parameters, then

> INP $^{C}_{R}$	to review the input parameters. You should be told **AIPS 1: Found in Version=MYAIPS** at the start of the inputs display. If not, there is something wrong with the logical or task name or the location of the inputs/help file.

`> GO` $^{C}_{R}$ to run the task. The `Found` ... line should appear again.

12.5.4 Modifying an *AIPS* task

If you are modifying an existing task, the only ground rules are to read the task carefully and note the locations where changes must be made. Unlike the template tasks which are organized so that additional code need be inserted in only one or two places, an *AIPS* task may need revisions in a variety of places. Some guidelines are:

1. Change the data statement `DATA PRGM` ... near the beginning of the program and the `PROGRAM` statement at the very beginning, if you have not already done this.
2. Make changes in the introductory text which describes the task. This is particularly important if you are adding or removing adverbs.
3. The subroutine `GTPARM` obtains the adverb values from the input table in order. If you add or subtract adverbs, change `INPRMS`, the amount of input information. Be careful in the translation of the adverbs, which are usually listed in a `COMMON` named `/INPARM/`.
4. Change code as desired. Try to modify `HI`story file entries as well.
5. Liberally sprinkle `PRINT` statements in crucial places to help debug the program. If much of the new code can be put in a subroutine, write and debug the subroutine outside *AIPS*. Then, add it to the task.
6. Revise the file `NTASK.HLP`. At least, change all references to `TASK` to `NTASK`! If there are any changes in the adverb list, look at these changes carefully. Further changes in the `HELP` and `EXPLAIN` portions of the file may be needed to document your work for for others and for yourself at a later date.
7. Compile and link the modified task following the instructions in the previous section. When it compiles and links without errors, try it out in *AIPS*.

12.5.5 Modifying an *AIPS* template task.

The template tasks are:

1. `TAFFY` — modifies an existing image file and writes a new image.
2. `CANDY` — writes a new image. Input data from outside *AIPS*.
3. `FUDGE` — modifies an existing *uv* database and writes a new database.
4. `UVFIL` — writes a new *uv* database taking input data from outside *AIPS*.

These tasks are described in detail in Chapter 2 of *Going AIPS*. The template tasks and the code for each are extensively documented there, so only the major points are discussed here.

`TAFFY` reads a selected subset of an image, one row at a time, to a user interface subroutine `DIDDLE`. An output image is created, cataloged and filled with values calculated in `DIDDLE` (or attached subroutines). The dimensionality of the output image need not be the same as the input image.

1. If the output image has identical dimensions to the input image, and an output image row can be generated from each input image row, then `TAFFY` is relatively straight-forward to use and the accompanying documentation should suffice.

2. TAFFY has the option of deferring the writing of an output row until some number of input rows have been read. This would allow the output to depend on some function of several, or even all, input rows. Examples would include smoothing in x and y and other sorts of spatial filtering.

3. It is possible to handle several input images and several output images with TAFFY by using multi-dimensional images and the axis transposing task, TRANS. Suppose you want to calculate the spectral index from a set of four images. First, use MCUBE to put the four images into one data cube. (You will have to redefine the frequency axis of each image to 1,2,3,4 in order to get MCUBE to do what you want.) Then, transpose the cube so that the frequency axis is first with right ascension and declination as the second and third axes. Use this image as the input to TAFFY and use CPARM to specify the frequency values. Each input row will then be the intensity at a pixel for the four frequencies — from which the spectral index and other parameters can be calculated. The output dimension can be specified arbitrarily. For example, you might want to write out the spectral index, the error, the curvature and the flux intercept at some fiducial frequency. When the task has completed, transpose the output cube so that the celestial coordinates are again the first two axes and the spectral index and friends are the third axis.

4. Subroutine NEWHED may be modified to require certain axis types, operation codes, etc. It must be used to change the output image dimensions if they are not to be the same as the input. An option to omit from the output image the first input axis is available and must be selected in NEWHED.

5. If you can write and debug outside AIPS any subroutines that DIDDLE will call in calculating the output image, you may speed up the debugging of your algorithms.

CANDY lets you create an image one row at a time. Input information is obtained through a file which is external to AIPS.

1. Subroutine NEWHED shows an example of how the external file may be defined and how it can be used. The first few records of the external file should contain information for defining the header of the output image and updating the appropriate catalog blocks. The pointers to the AIPS catalog are given in Chapter 5 of *Going AIPS*.

2. The subroutine MAKMAP reads further records from the external file until a full row of the output image is obtained.

3. Many adverbs are built into CANDY. If you need more or wish to change them, read the information at the beginning of subroutine CANIN.

FUDGE reads an existing *uv* database point by point and creates a new *uv* database with the modified data. It can easily be used to make simple changes in a *uv* database; for example, if the u, v, or w terms are to be recalculated, if all phases are to be changed in sign, etc.

1. If the output image has different dimensions than the input image (*e.g.*, by combining several spectral-line channels), changes must be made in the subroutine FUDGIN.

2. To combine several input data points, use the task UVSRT to put the data points adjacent in the file.

3. If you wish to calculate quantities from an input *uv* data set without creating an output file, use this task. In the subroutine DIDDLE set IRET = -1 to avoid writing an the output *uv* data set.

UVFIL reads input from one or two FORTRAN-readable files which are outside AIPS to create, catalog, and fill a *uv* database into AIPS. This task is useful for transcribing *uv* data in an arbitrary format into an AIPS database.

1. The task code has copious notes to help the user. The first auxiliary file should contain header information about the observations and the telescopes. This file is read in the subroutine `NEWHED`.
2. The second auxiliary file contains the actual *uv* data in some format and it is converted into an AIPS *uv* database, point by point, in the subroutine `FIDDLE`. The comments are extensive and an example is given in the code.

12.5.6 Further remarks

1. Try to use the AIPS coding standards as described in *Going AIPS*. This will make your code more readable and more portable. It will also save other AIPS programmers lots of work if your new task comes into general use.
2. Declare all variables that you use.
3. While debugging, use the FORTRAN `PRINT *, ...` statement in your code to obtain temporary output on your terminal during execution. If you want more permanent output, use the AIPS message file facilities with the appropriate message level (4 is recommended for information, 6 for warnings, 8 for errors).
4. You may use a debugger (such as `dbx`) on AIPS tasks by specifying `DEBUG` to the `COMLNK` and `aips` procedures. Inside `AIPS`, set the "hidden" adverb `SETDEBUG` to 0 for normal operation and to 20 to run tasks with your specified debugger.

12.6 Additional recipes

12.6.1 Banana nut bread

1. Cream 1 cup `sugar` and 1/2 cup `margarine` together.
2. Add 2 `eggs`, 2 cups `flour`, 1/2 teaspoon `salt`, and 1 teaspoon `baking soda` and mix thoroughly.
3. Add 1 cup chopped `nuts` (walnuts or pecans), 3/4 cup mashed `bananas`, and, lastly, 4 teaspoons `sour milk` and mix well.
4. Put in greased loaf pan.
5. Bake in 350 deg oven for 1 hour.

12.6.2 Frozen Push-Ups

1. Peel 2 `bananas` and slice into blender or food processor.
2. Add 1 6-ounce can frozen `orange juice` (thawed), 1/2 cup instant non-fat `dry milk`, 1/2 cup `water`, and 1 cup plain low-fat `yogurt`.
3. Cover and blend until foamy. Pour into small paper cups and freeze.
4. To eat, squeeze bottom of cup.

Thanks to Ruthe Eshleman *The American Heart Association Cookbook.*

12.6.3 Banana poundcake

1. Mix in large bowl until blended:
   - $1\frac{1}{3}$ cups mashed `bananas` (4 medium)
   - 1 pkg. ($18\frac{1}{2}$ oz.) `yellow cake mix`
   - 1 pkg. ($3\frac{3}{4}$ oz.) instant `vanilla pudding mix`
   - $\frac{1}{3}$ cup `salad oil`
   - $\frac{1}{2}$ cup `water`
   - $\frac{1}{2}$ teaspoon `cinnamon`
   - $\frac{1}{2}$ teaspoon `nutmeg`
   - 4 `eggs` at room temperature
2. Beat at medium speed for 4 minutes.
3. Turn batter into greased and lightly floured 10-inch tube pan.
4. Bake in 350 deg oven for 1 hour or until cake tester inserted in cake comes out clean.
5. Cool in pan 10 minutes, then turn out onto rack and cool completely.
6. If desired, dust with confectioners sugar before serving.

Thanks to the United Fresh Fruit and Vegetable Association.

12.6.4 Chewy banana split dessert

1. Prepare and bake one package (19.8 0z) chewy fudge (or other favorite) `brownie mix`. Allow to cool thoroughly, four hours or more.
2. Peel 2 large ripe `bananas` and place very thin slices on top of brownie.
3. Cover bananas evenly with one 12-oz. container of `whipped topping` (thawed) and drizzle 1/2 cup `chocolate syrup` over that.
4. Refrigerate to chill completely. Cut into squares to serve.

12.6.5 Little banana cream tarts

1. Preheat oven to 325 deg.
2. Combine 6 tablespoons `margarine` or butter (softened), 1/4 cup packed `brown sugar`, 1/4 cup `powdered sugar`, and 1/2 teaspoon `vanilla extract`.
3. Stir in 2/3 cups crushed `cereal` (2 cups un-crushed Multi-Bran Chex suggested), 1/2 cup all-purpose `flour`, and 1/3 cup finely chopped `nuts` (optional).
4. Divide dough evenly into 12 balls. Place each ball in 2.5-inch muffin cup; press into sides. Bake 8 to 10 minutes.
5. Let stand in pan 15 minutes. Use knife to remove each tart carefully from pan. Tarts will be very soft. Let cool completely.
6. Melt 2 tablespoons `margarine` in skillet over low heat.
7. Stir in 2 tablespoons `heavy cream`, 4 tablespoons packed `brown sugar`, and 1/8 teaspoon `allspice`. Cook until sugar is dissolved, stirring occasionally.
8. Stir in 3 medium `bananas`, sliced. Divide filling evenly among the cooled tarts and garnish with whipped cream.

Thanks to Ralston Purina Company.