NOTE 195
Getting Started with Glish for AIPS++

Rick Fisher, NRAO

Text last updated : 1999 April 30 by Jim Braatz

A postscript version of this note is available.

1 Introduction

This document is an AIPS++ applications-specific supplement to "The Glish 2.7 User Manual.". Glish is both a programming language and an environment for data acquisition and analysis. It contains many of the mathematical features of other languages such as Fortran and C, and it is particularly strong on array arithmetic. To the user, glish becomes AIPS++ or a telescope control system with the addition of functions and "clients" for those applications. In fact, both applications may be run from the same invocation of glish, if so desired.

Here we shall cover the main language features with simple examples drawn from test data acquired by the GBT hardware systems now under development. The emphasis will be on the tools for writing your own data and control functions rather than on a review of AIPS++ library functions. Also, only a few of the built-in glish functions will be mentioned. See section 10.12 of the Glish User Manual for a complete function summary, and the AIPS++ User Reference Manual for information on the AIPS++ library. A few elements of the AIPS++ library are used below; when discussed the relevant section of the AIPS++ User Reference Manual is available via the link.

I shall assume that you have access to a working AIPS++ installation. Details about setting up your AIPS++ environment are given in Getting Started in AIPS++.

2 Starting AIPS++ and Glish.

In a new directory you can establish an AIPS++ and glish environment with the following UNIX command:

```
For csh-like shells:
$ source /aips++/stable/aipsinit.csh

For bash-like shells:
% . /aips++/stable/aipsinit.sh
```

Note that the directory name, `/aips++/stable`, may vary depending upon your installation; see Getting Started in AIPS++ for more details.

Then start AIPS++, open the example files, and load the example glish scripts with

```
$ aips++
- include 'glishtutorial.g'
```

The script 'glishtutorial.g' converts the example FITS data files used in this tutorial into tables which can be read directly in aips++. When the conversion process has completed the glish prompt will return.

AIPS++ without the example files and scripts can be started with simply

```
$ aips++
```

The glish prompt is either '-' or '+'. To test glish and plot window operation run the commands:

```
- mp := pgplotter()
- mp.plotxy([1:100]/5.0 ,sin([1:100] / 5.0), title="sine wave")
```

You should see about three cycles of a sine wave. This is plotted using the AIPS++ utility pgplotter.

3 Variable Assignment and Automatic Typing

Glish variables are automatically assigned a variable type when they are created. In other words, there is no variable declaration statement. After the following assignments:

```
- x := 2
- z := 23.786
- check := F
- status := T
- amp := 1.66 - 0.994i
```

'x' is an integer, 'z' is a double, 'check' and 'status' are booleans (true or false), and 'amp' is double complex. The available data types are boolean, byte, short, integer, float, double, complex, and double complex. Floating point constants are assumed to be double, and integer constants are assumed to be integer. In most cases the data type is self-evident, but, if your code needs to know a variable's type, there are built-in test functions such as is_boolean(), is_float(), is_dcomplex(), etc., which return a boolean value T or F. Also, the function type_name() will return a character string describing the type. For example,

```
- is_float(z)
F

- type_name(z)
double
```

Automatic variable typing is quite convenient, but it provides no warning when you inadvertently reassign a variable. The assignment

```
- status := 1.65e7
```

is perfectly legal resulting in 'status' now being a double. You can protect a variable from reassignment with the 'const' keyword as in

```
- const Pi := 3.14159
```

Values of variables and constants in an arithmetic expression are all converted to the type of highest numeric precision in the expression before performing the mathematical operations. In an expression using the variables above

```
a := x * z * check
```

'a' is a double. Variable typing may be forced for purposes such as integer truncation or storage economy with the built-in functions as_integer(), as_float(), etc. Keep in mind, however, that a variable is retyped every time it is assigned, so, on a few occasions, your expressions may need to be very explicit about type.

The arithmetic operators are

```
+  -  *  /  %  ^
```

where % is the modulus operator that converts its operand values to integer and returns an integer result, and ^ is the exponentiation operator with floating point arguments and result.

```
- 26.8 % 4.2
2
- 3.01^1.99
8.960811
```

The value of a variable may be printed with the print statement or, in interactive mode, by simply typing the variable name without the ':=' assignment operator. The following two statements are equivalent:

```
- print x
- x
```

Be careful with array names! You can get a real screen full.

4 Arrays

Array operations are one of glish's strong points for both code economy and computing efficiency. The notation takes a bit of getting used to.

An array variable may be created either by assigning to it an array constant or the result of an array expression,

```
- indices := 1:512
- ar3 := ar2 * ar1
```

or by using the array initialization function,

```
- image1 := array(0.0, 256, 256)
```

The array variable, 'image1', becomes the type of the first argument of 'array()', which is the array initialization value. The first argument itself may be a scalar or an array. The rest of the arguments are array dimensions of any length and number that will fit in memory.

Most arrays are created by assignment. The primary use for the 'array()' function is to create an empty array to be filled piecemeal later. Again, remember that an assignment can change the size of an array or even turn it into a scalar.

Users of IDL, PV-Wave, and, to some extent, IRAF will find glish's array index notation familiar. Fortran and C programmers will be tempted to attack arrays with integer indices and iterative loops. Resist the temptation when possible because the intrinsic array operations are much faster.

To avoid an abstraction of arrays let's load a continuum receiver data column from one of the sample data tables. A small portion of this table looks like

```
RECEIVER_ID  PHASE  CAL  SUBSCN    DATA
              0          0      0       1     600885
              1          0      0       1     572540
              0          1      1       1     649341
              1          1      1       1     617254
              0          0      0       2     601413
              1          0      0       2     573158
              0          1      1       2     649827
              1          1      1       2     617841
              0          0      0       3     602472
              1          0      0       3     574106
```

There are many more rows and columns to this table, but these include the interesting columns for the moment.

Now load the 'DATA' column from table t1 into a glish array. This table has already been opened using the AIPS++ table system if you included 'glishtutorial.g'.

```
- data := t1.getcol('DATA')
```

If you want a complete picture of this table, browse the table with

```
- t1.browse()
```

The vector 'data' (1-D array) contains interleaved data from two receiver channels with the noise calibration signal turned on and off. To look at a single element just use an integer index.

```
- data[3]
649341
```

Notice that indices start at 1. To access a contiguous range of values in the vector use an index range.

```
- data[1:5]
[600885 572540 649341 617254 601413]
```

Any subset of vector contents may be acquired with an index array containing the indices of the values you want.

```
- indx := [2,5,9]
- data[indx]
[572540 601413 602472]
```

or just simply

```
- data[[2,5,9]]
[572540 601413 602472]
```

Notice the second set of square brackets which define the index list as members of an integer vector rather than indices of different dimensions of a three-dimensional array. To digress a moment, a multidimensional array is addressed as follows:

```
- mda := array(0.0, 16, 16, 16)
- mda[5, 3, 14] # column 5, row 3, slice 14
- mda[ , 6, 5] # all values in row 6, slice 5
- mda[11, , ] # values in the column 11 plane
- mda[ , ,3:5] # 3 planes of slices 3, 4, and 5
```

The # character is the comment delimiter. Everything from it to the end of the line is ignored by glish.

If you like, you can load an example of a two-dimensional array of data from a different table that contains a column of spectra.

```
- spectra := t7.getcol('DATA')
```

Glish variables have attributes, a few of which are predefined. The dimensions of an array are contained in its 'shape' attribute. The syntax for an attribute variable is

```
- spectra::shape
[1024 33]
```

In this case, there are 33 spectra of 1024 channels each.

Back to our 'data' vector, rather than generate an index vector it is more efficient to create a boolean vector with the same number of elements as the 'data' vector and use it as an element mask. For example, to get all of the 'data' values for receiver 1 with the calibration noise "on" get the RECEIVER_ID and CAL columns.

```
- rcvr := t1.getcol('RECEIVER_ID');
- cal := t1.getcol('CAL')
```

Then make a boolean vector of the same length and use it as an index mask to the 'data' vector.

```
- mask := (rcvr == 1) & (cal == 1)
- mask[1:10]
[F F F T F F F T F F]

- data_cal_1 := data[mask]
- data_cal_1[1:5]
[617254 617841 618768 619804 620325]

- print length(data), length(data_cal_1)
960 240
```

From the values and length of 'data_cal_1' we can see that it contains the values from the 'data' vector where the boolean vector, 'mask' , is true. The function length() returns the number of elements in the array, not the number of bytes. The equivalent subarray access could be written without the intermediate 'mask' variable.

```
- data_cal_1 := data[rcvr == 1 & cal == 1]
```

We have also used the operator precedence rules to eliminate the parentheses, but, when in doubt, use parentheses. The comparison operators are

```
<   >   <=   >=   ==   !=
```

which are less than, greater than, less than or equal to, greater than or equal to, equal to, and not equal to, respectively. The boolean operators are "and", "or", and "not".

```
&   |   !
```

All of the arithmetic, comparison, and boolean operators operate on arrays element by element. Therefore, all arrays in an expression must be of the same size in each dimension. For example,

```
- [1:3] * [1:3]
[1 4 9]
```

It is not a vector dot product.

For a graphical illustration of what we have done with the extracted data column get one more column, 'Time', from the table and plot the two data vectors against it.

```
- times := t1.getcol('Time')
- times -:= as_integer(times) # remove day numbers
- mp := pgplotter()
- mp.env(.96294,.96304,5.6e5,6.6e5,0,0)
- mp.line(times[1:32], data[1:32])
- sometimes := (times[1:32])[mask[1:32]]
- mp.eras()
- mp.env(.96294,.96304,6.1e5,6.3e5,0,0)
- mp.line(sometimes, data_cal_1[1:length(sometimes)])
```

The second line above illustrates one of the combined arithmetic and assignment operators (+:= -:= *:= /:= %:= ˆ:= &:= —:=) that are also found in C. The assignment of 'sometimes' is a combined use of vector ranges and a mask.

If we assume that the receiver 1 noise calibration intensity is 1.4 Kelvins, we can plot a calibrated scan with the additional commands

```
- data_cal_off_1 := data[rcvr == 1 & cal == 0]
- dataK := 1.4 * ((data_cal_1 + data_cal_off_1) / 2.0) /
+ (data_cal_1 - data_cal_off_1)
- mp.eras()
- mp.plotxy([1:len(dataK)],dataK)
```

Note that a different (and simpler) plotting function, mp.plotxy(), replaces the mp.env() and mp.line() commands used previously. The latter functions were used at first because of the unusual requirements on the axes ranges.

You should see four scans through a radio source placed end to end. These are actually four separate scans in the same table that were taken with a cross pointing scan procedure on the 140-foot. Note that the + on the third line in the previous example is the glish prompt and not an indication that you should type the plus character.

All of the other column names in this table may be listed with

```
- t1.colnames()
```

Here are some additional indexing ideas.

Rotate a vector five places.

```
- rot_vector := data[[5:data::shape,1:4]]
```

Delete out-of-range values.

```
- purged := data[data < 700000]
```

Make a new time vector to match the 'purged' vector.

```
- ptime := times[data < 700000]
```

Reverse a vector.

```
- flipped := data[data::shape:1]
```

Splice two vectors together.

```
- spliced := [v1, v2]
```

5 Scripts and Functions

Glish code may be written in a file with an editor and executed with the 'include' command. These scripts contain glish code that is identical to the text that you type at the glish prompt. Script files are conventionally given the name extension '.g'. The glish command syntax is

```
- include 'myscript.g'
```

The most frequent use of script files is to edit and load function definitions. The syntax may be either

```
function <fn name> ( <arguments> ) {
    statements
}
```

or

```
<fn name> := function ( <arguments> ) {
    statements
}
```

The 'function' keyword may be abbreviated 'func'. The curly brackets are to group all of the statements together. Without the brackets, only the one statement following the function arguments would be part of the function. Glish is pretty liberal about whether brackets and statements appear on same or different lines and about how you use indentation, as long as the meaning is unambiguous. A function may return any variable type including whole arrays.

Here is a simple function for plotting data from one receiver of one scan in a table.

```
plot_scan := function (tab, scan_no, receiver=1)
{
    mp := pgplotter()
    data := tab.getcol('DATA');
    scan := tab.getcol('SCAN');
    rcvr := tab.getcol('RECEIVER_ID');

    mask := scan == scan_no & receiver == (rcvr + 1);
    scan_data := data[mask];
    if (length(scan_data) == 0) {
        fail paste('No data found for scan', scan_no,
                   'receiver', receiver);
    }
    mp.plotxy([1:len(data[mask])], data[mask],
              title=paste('Scan', scan_no, 'Receiver', receiver));
           return
}
```

Type this function into a script file, say, 'test.g'. Load it into glish with

```
- include 'test.g'
```

and execute the function with

```
- plot_scan(t1, 48)
```

There are a few new items in the function definition above. The lines are terminated with semicolons. This is optional both in a script file and at the glish prompt. If you put two lines of code on the same physical line, they must be separated by semicolons. The function contains an if() statement whose argument is normally a boolean expression, but it can be a numeric variable. Zero is false, and everything else is true. 'pgplotter()' and 'mp.plotxy()' are AIPS++ functions. 'paste()' is a glish built-in string concatenation function. 'fail' is a glish built-in that causes an error return; use it like this when you want to signal an error.

Function arguments may be defaulted by assigning them a value in the function definition, as we did with the 'receiver' argument. When a function is called, if no value is given for a defaulted argument, the argument assumes the value given in the function definition. Undefaulted arguments must be given values in a function call. Normally, arguments are given values according to their position in the argument list, but they can be given in any order, if they are named. For example, the plot_scan function call above could also have been done with

```
- plot_scan(scan_no=48, receiver=1, tab=t1)
```

The first arguments may be specified by position, but after an argument is named all the rest must be named.

Variables defined by assignment within functions are local to that function. In other words, when the function execution completes the variable goes away. A variable which doesn't disappear may be created by declaring it to be "global". Likewise, an externally defined variable may be assigned a value within a function by declaring that variable to be global within the function. For example, look at the results of the execution of two functions defined below.

```
x := 10
bill := function () {
    x := 5
}
fred := function () {
```

```
    global x := 6
}

- x
10
- bill()
F
- x
10
- fred()
F
- x
6
```

The F that appears after you execute the functions is the glish boolean, False. This is the default return value for functions which do not explicitly return a value.

An unusual feature of glish is that global variables can be implicitly referenced from within a function. However, by default, they become local to the function upon assignment, for example,

```
- gvar := 0.841471
- function x() {
+    print gvar
+    gvar := 90
warning, scope of "gvar" goes from global to local
+    print gvar
+ }
- x()
0.841471
90
- print gvar
0.841471
```

Glish provides a warning whenever the scope of a variable goes from global to local. If you want the global variable to be affected, you must explicitly declare it with the 'global' keyword, as in

```
- gvar := 0.841471
- function x() {
```

```
+    global gvar := 90
+ }
- x()
- print gvar
90
```

Finally, if you don't like the name of a function, or it's too long to type conveniently, you can give it an alias by assigning the function name to another name, just like assigning one variable to another.

```
- ps := plot_scan # note no parenthesis
- ps(t1, 48)
```

Be careful, however! You can wipe out an entire function definition by reassigning it. If you make the assignment,

```
- plot_scan := 5
```

the function plot_scan() is gone until you redefine it by including its script file again. However, the variable ps still contains the copy of plot_scan that you assigned to it earlier.

As with variables, you can protect a function name and its arguments with the 'const' keyword, for example,

```
- const myplot := function (const ary) { ... }
```

After you have finished debugging a function, it is usually a good idea to protect it with 'const'.

6 Loops and Conditionals

Three familiar program flow control statements are available in glish, "if/else", "while", and "for" loop statements. The general syntax for each is

```
if (<boolean condition>) {
    statements
} else { # there is no 'else if'
    other statements
}

while (<boolean condition>) {
```

```
    execute all of this
}

for ( <variable> in <vector> ) {
    execute all of this
}
```

The “for” loop is a bit different to its equivalents in Fortran and C in the sense that the running variable doesn't have to make equal steps. It just takes on the values of the elements of the vector in succession, whatever those values may be. A garden variety loop of i from 1 to 10 in steps of 1 is

```
for (i in 1:10) {
}
```

Every method you can think of for generating a vector can be used to create the vector argument in the “for” statement. For example, you can use the 'seq()' built-in function to step from 3.66 to 7.35 in steps of 0.0123.

```
for (x in seq(3.66, 7.35, 0.0123)) {
}
```

Even a boolean vector is OK.

```
for (i in [F, T, T, F, T]) {
    print i;
}
```

7 Records and Attributes

Scalars and arrays of the standard data types are not the only variables allowed by glish. Heterogeneous data structures, called records, can be treated as a single variable. These can be passed as arguments to a function and returned as a unit in a function return value. A record may be created all at once or added to piece by piece.

```
- header := [name='3C273', ra=12.12345, dec=2.3318]
- header.flux := 27.6
- header.type := 'quasar'
- print header.ra
12.12345
```

Record members may be strings, any of the numeric data types including arrays, as well as other records.

Any glish variable may be assigned attributes. For example,

```
- velocity::definition := 'relativistic'
- header.flux::units := 'Jy'
```

The distinction between record members and variable attributes is a subtle one, and it is seldom clear which is best suited to a particular application. Both record members and attributes may be accessed by name or by number. All of the following are acceptable:

```
- header.ra
- header[2]
- header['type']
- velocity::[1]
- velocity::definition
- velocity::['definition']
```

Numeric access lends itself to stepping through the fields or attributes with a loop index. Record field names may be retrieved with the built-in function 'field_names()', which returns a string.

```
- field_names(header)
name ra dec flux type
```

The empty attribute construct returns the attributes of a variable as a record, so you can use the 'field_names()' function on attributes, too.

```
- velocity::
[definition=relativistic]
- field_names(velocity::)
definition
```

8 Strings

There are two types of string constants in glish. Single quoted strings are treated as a single entity and are most useful in print statements where some control of placement of characters on the screen is important. Double quoted strings are treated as arrays of strings with white space separating the array members. The output of 'field_names()' in the example above is an instance of a string array. The difference is best illustrated with some examples.

```
- s1 := 'This is a single quoted      string'
- s2 := "This is a double quoted      string"
- s1
This is a single quoted      string
- s2
This is a double quoted string
- s1[1]
This is a single quoted      string
- s2[1]
This
- s1[2]
<fail>: index (= 2) out of range, array length = 1
- s2[2]
is
```

Notice that all white spaces have been reduced to single spaces when the double quoted string array is printed.

String handling is fairly primitive in glish. There are no string operators, except possibly for the cautious use of the comparison operators.

```
- twenty := '20'
- three := '3'
- twenty < three
T
```

If one of the variables, twenty or three, had not been a string, an error would have been reported. When in doubt, use the is_string() test first.

There are a few string manipulation functions, including paste(), spaste(), and split(). Paste() and spaste() act like print statements except that the resulting string is returned as a single string instead of being printed.

```
- r := 53.6
- sx := paste('R =', r)
- sx
R = 53.6
```

The paste() function can take a named argument, sep, to change the separator character(s) from the space default.

```
- paste('R', r, sep=' = ')
R = 53.6
```

The function spaste() is the same as paste() with no separator. The string function split() simply turns a single string into a string array or breaks a string array differently. It takes two arguments, the string to be split and a list of characters to be used as separators. The default separator is a space.

```
- str1 := 'This is a single quoted string'
- str2 := "This is a double quoted string"
- sx := split(str1)
- sx[1]
This
- sx := split(str2, "u")
- sx
This is a do ble q oted string
- sx[3]
oted string
```

9 Printing

The print statement is quite primitive so output formatting in glish is minimal. The output of print is simply the concatenation of the quantities listed converted to strings and separated by spaces. You can add spaces by printing spaces in single quoted strings, and you can control the number of decimal places shown for floating point numbers with one of two print precision variables.

```
- r := 3; s := 5.6; j := "spa   ces"
- print '     show', r, s, j
     show 3 5.6 spa ces
```

The printed precision of all floating point variables is set by the value of a member of the system record variable.

```
- old_prec := system.print.precision
- system.print.precision := 3
- x := 234.12345
- print x
234
- system.print.precision := old_prec
```

The printed precision of one variable may be set by the member of the variable's print attribute.

```
- x::print.precision := 4
- print x
234.1
```

To avoid the possibility of inadvertently printing a huge array to the screen, you can set a limit on the number of printed array elements with

```
- system.print.limit := 50
or
- any_array::print.limit := 50
```

The effects of print.precision or print.limit may be cancelled by setting them to a value less than zero. You may want to set a system print limit in your .glishrc file. See Appendix A and Getting Started in AIPS++ for more details on the role and uses of your .glishrc file.

10 Clients, Events, and Agents

From a software designer's point of view, the important part of glish is its ability to link to any number of independent processes running anywhere on a computer network. To glish these independent processes are seen as "clients", but the client-server distinction is ambiguous since services can flow either way. Communication between glish and its clients is by way of "events", rather analogous to hardware interrupts but with the added feature that events can carry any amount of data to or from a client.

As an AIPS++ user you may seldom encounter the use of events in your glish programming. Events are considerably different in concept from usual linear scientific programming. Hence, they are usually wrapped in more familiar function calls to avoid the need to learn events to use the services of AIPS++ clients. In fact, in the examples above we have used a number of clients, including the table browser and the plot client, without necessarily being aware of events passing back and forth. It's not that clients and events are all that obscure. It is just that they are one more concept to learn that might not be of any direct interest.

Clients can do anything that a normal computer program can do from performing FFT's to driving a telescope to position. Most of the application-specific facilities of AIPS++ are in the clients that glish initiates.

To see how the events work try the following commands associated with the Tk widgets attached to glish.

```
- mp := pgplotter()
- fm := frame()
- b1 := button(fm, 'plot')
- whenever b1->press do {
-  mp.eras()
-  mp.plotxy([1:100],sin(as_float(1:100) / 20.0), title='Button plot')
- }
```

The first statement puts a small window frame on the screen. The second puts a button labeled 'plot' in the window. In the whenever statement, "press" is the name of an event sent by the Tk widgets client when the button is pressed with the mouse. The statements within the whenever block are executed when the 'press' event is received. Try pressing the button.

Clients are started from glish with the client() function. A more complex example than the button client might be a hypothetical FFT engine. Let's say that we have an executable file on disk, called "fft_engine", that performs Fourier transforms and knows how to send and receive glish events. Start it up with

```
- fft := client('fft_engine', host='arcturus')
```

The host argument is not needed if "fft_engine" is on the same machine that is running glish. The variable, fft, is of a special type called an agent.

Suppose that the documentation for fft_engine says that it recognizes an event called "do_fft", and , when the FFT is done, fft_engine will send an event called "fft_done" with a new array of transformed data. First, set up to receive the "fft_done" event and do something with it. Then send the fft request with the data array. Probably in a script called 'fft.g' we'd have

```
fft := client('fft_engine')
whenever fft->fft_done do {
   mp.eras()
   mp.plotxy([1:len($value)], $value, title='FFT Result')
}
```

Then from the command line do the following:

```
- include 'fft.g'
- data := sin(as_float(1:128) / 3.0)
- send fft->do_fft(data)
- await fft->fft_done
```

In this case we have elected to wait for the "fft_done" event to return and be processed, but we could have continued to issue glish statements for other purposes. The FFT result would be returned and plotted whenever the fft_engine client was finished with it.

The parameter, $value, is a special one associated with events. It contains the event's data and can be anything from a single number or string to a very complicated record with arrays and other records as members. Normally, glish and the client know what to expect in $value, but some flexibility of contents and data types can be supported with the is_xxxx() test functions.

A fully implemented glish client is written at least partly in C++ using the classes that support events and glish data structures. However, existing programs may be turned into simple clients using standard input and output ASCII data as event values. Also, clients may be written in the glish language, but there are fewer reasons for doing it this way.

The standard-in, standard-out client is created by the shell() function rather than the client() function. A one-time communication with a client program is simply the UNIX command as a string argument to shell().

```
- file_list := shell('ls *.g')
```

The variable, file_list, is then an array of strings, one string for each file name that ends with ".g" in the directory.

If the program can handle more than one exchange of data by way of UNIX pipes, it may be started and communicated to with events. The startup glish code looks something like

```
a := shell('my_prog', async=T)
whenever a->stdout do {
    print $value
}
whenever a->stderr do {
    print 'Error !', $value
}
```

The async=T parameter assignment causes the shell() function to return an agent type quantity instead of a string. The data events that can be sent by the client started with the shell command are called "stdout", and "stderr" and their $value is always a string. The two events recognized by a shell client are "stdin", whose argument must always be a string, and "EOF". Hence, an event sent to "my_prog" is, for example,

```
- send a->stdin('23.876 14.999')
```

To avoid having an event arrive from a new client before its associated whenever statements are set up, put the client() or shell() function calls and the whenever statement in the same script file, and run them together in the script. Glish guarantees that events are held until the full script file is read.

11 What now?

To learn more about Glish, you might wish to read the "Glish 2.7 User Manual". For information on AIPS++, see Getting Started in AIPS++ and the AIPS++ User Reference Manual.

12 Appendix A - Glish Environment

Glish can be most easily run by setting up the AIPS++ environment. For information on how to do this, see Getting Started in AIPS++.

Glish, itself, reads an environment file, called .glishrc, which may be used to set a number of system variables. The most important one is system.path.include, which specifies the directories to be searched for ".g" files that are loaded with the include statement. Any ".g" file may be loaded by giving its full path name, but the system.path.include assignment saves a lot of typing. The contents of a .glishrc file might look like

```
print 'reading /hyades1/rfisher/Pulsars/P1823/.glishrc'

system.path.include :=
    ". /gbt3/aips++/sun4sol/libexec /aips2/glishScripts-2509a /";

print 'system.path.include: '
for (i in 1:len(system.path.include))
    print '    path', i, system.path.include [i];
```

The print statements are just embellishments to remind you of where the environment variables are coming from when glish starts up. Notice the '.' path name as well as the system glish directories.

When glish starts up it reads the .glishrc file in the system glish directory. It then looks for a .glishrc file in your current directory. If none is found there, it tries your home directory. Note that the system.path.include assignment shown above will override the one in the system .glishrc file. If you just want to add additional paths, you can use the following construct to expand the string array:

```
system.path.include[len(system.path.include) + 1] := '/home/rfisher'
```

Arrays are automatically expanded in an assignment statement when the index of the assigned array is greater than its current size. "len()" is another name for the length() function.

Finally, it's worth a look at the UNIX script that starts the glish/AIPS++ combination. This script is the command given at the beginning of this document, for example, "aips++". To find out where this script is, use the UNIX 'which' command. Then list the file using the full path name.

```
$ which aips++
/aips++/stable/sun4sol_gnu/bin/aips++
$ more /aips++/stable/sun4sol_gnu/bin/aips++
[ lines setting up the aips++ environment variables deleted ]
exec $BINDIR/glish -l $LIBDIR/aips++path.g -l $LIBDIR/aips++.g
```

The last line executed in this script is the command that starts glish. It automatically includes two files, `aips++path.g` and `aips++.g`. Both files contains glish code which loads the aips++ clients. The "-l" (minus ell) flag tells glish to keep running in interactive mode. Without this flag the script `aips++.g` would be executed and then glish would terminate.

If you want to start glish without any of the AIPS++ functions or clients, just issue the UNIX command

```
$ glish
```

13 Appendix B - Example Data Files

The two AIPS++ tables, t1 and t7, used in the examples in this document were created from some GBT system test data from the 140-ft telescope.

Other instruments, such as the VLA, will have a different file and directory structure for the original data from the one outlined below, but many properties of the created AIPS++ tables will be the same.

The original data are in several FITS Binary Table files written by different system modules, e.g., the weather monitor, telescope position control, and one of the data backends. The t1 table contains data from the continuum backend, and t7 is a bit of spectral processor data in spectral line mode. There will be an automated process to convert the raw FITS data into tables readable by AIPS++. For now, the data must be filled by hand. The utility to fill data from the GBT backends, for example, is called gbtmsfiller.

A step to opening tables within glish was hidden by the glishtutorial.g script. Namely, to open a table use:

```
- t7 := table('t7')
```

where the argument to the table() function is the directory produced by the filler.