NOTE 187
A Help Protocol for AIPS++ (Draft 3)

Paul Shannon, NRAO

January 30, 1996

Introduction

This paper proposes a simple protocol for storing and retrieving help information in AIPS++. The proposal extends to all AIPS++ "software entities": glish functions, glish clients written in C++, and distributed objects. These questions are addressed:

1. What minimum help information should be provided by every software entity?
2. How will that information be stored? How will it be retrieved?
3. What options will be available for viewing the information?
4. How will the user navigate through thousands of different help topics?
5. Some entities may benefit from carefully qualified, lengthy explanations, and elaborate demonstrations. How will the help protocol support this?

The original design and implementation of standardized help for glish functions came from Rick Fisher. Though he is in no way responsible for subsequent modifications (and drift!), the author is grateful for the insightful suggestions that got this effort started.

Overview: Self-Description and Atoms

The core of the proposal is the notion that every software entity should be self-describing, and that there are a small number of *atoms* of help information which any entity should return on demand.

We use the word "atom" because these items of help information are fundamental and irreducible in their domain, and can be assembled into more complex structures. For each atom there is a corresponding retrieval mechanism, which will often be a simple glish function call typed at the command line, or invoked by a GUI help viewer.

The software entities divide naturally into two groups – objects and functions – and each group has characteristic help atoms. Distributed objects, glish clients and (eventually) glish objects are in the first group; glish functions, client functions, and object member functions are in the second group. Here is a tenative list of the help atoms *shared* by both groups:

1. category
2. version
3. name string
4. apropos keywords
5. short description
6. long description
7. caveats
8. see also
9. example code
10. example code comments

These additional atoms are required for functions:

1. return value
2. parameter n
3. parameter n author's default
4. parameter n current default
5. parameter n comment
6. parameter n min
7. parameter n max
8. parameter n legal values

And this atom is specific to objects:

1. member functions

Design Principles

The design of the help system is based on a couple of aphorisms:

- Low threshold, high ceiling.
- Mechanism, not policy.

"Low threshold" here means that the protocol should be easy to comprehend and uncomplicated to use – both for the author of help text, and for the reader of help text. We want every code author to provide adequate help information for their code. To that end, we must select a minimum but sufficient list of help categories; if the list is a sensible one, and if using it is straightforward, then we can reasonably ask code authors to adopt it.

On the opposite end of the help system, where help the user is looking for help, a minimum but sufficient list of help categories can be easily learned, easily remembered, and easily invoked.

"High ceiling" means that we must allow for the creation and display of very sophisticated and nuanced help information. The mechanism for this is straightforward: two of the atoms ("long description" and "see also") are plausible places for the author to embed WWW URL's; any of the help viewers we develop can easily be designed to recognize a URL, and to run a WWW browser.[1] This hook provides access to an open-ended set of help tools (imagine a Java applet that uses animation to demonstrate the algorithms of a clean task), but does so without forcing any other costs upon the code author who is not so inclined, or whose code needs only simple explanation.

The protocol does not assume any particular mode of viewing. (This contrasts with AIPS++ class documentation system, which requires either a web browser, or a patient reader of heavily marked-up text.) Three quite different help viewers were developed just prior to the writing of this paper – see the section below titled **Help Viewers**.

Another important sense in which the protocol is "mechanism, not policy" is that it tries to identify the fundamental categories of information one needs to know about a software entity, and once identified, keeps the categories separate. One might argue that the typical user most often wants to know these things about (say) a glish function:

1. The short description
2. Basic parameter information
3. Function return type

If you accept this, it may seem plausible for the help protocol to provide a single operation which returns these pieces of information together – or any other packaging of the information that will be requested frequently. But the protocol intentionally avoids this: the emphasis is, instead, upon identifying the fundamental atoms of information, and supporting them. Any particular grouping of the atoms is a policy decision, and should be handled at a higher level that what we address here.

Where Does Help Text Go? An Argument for Collocation

There is nothing which can guarantee that code authors will update their help information when they update their code. But if the two kinds of information are stored separately, the likelihood of consistent updating is probably reduced. The protocol therefore includes the recommendation that help text be stored in the same file as the glish function, C++ glish client, or distributed object it describes.

[1]A particular viewer may also choose to ignore any URL's – and since the fundamental information is not stored in URL's or HTML markup, this choice is quite defensible.

Navigating through “Too Much” Help

Two of the proposed help atoms, “category” and “apropos keywords” are designed as navigation aids. The first of these allows the user to restrict the scope of a search to one or more categories: mathematics, image processing, or plotting, for example. The second allows the function author to list all of the ideas or keywords that could be associated with their function. For example, at the Glish command line:

```
set_help_category ('fitting', 'plotting');
apropos ('vector')
```

or

```
set_help_category ('storage');
apropos ('vector')
```

would produce a very different set of functions for further exploration.

Help Viewers

Three viewers were quickly assembled as part of the exploratory work behind this paper, and a handful of glish functions (see *Example* section just below) were documented in the proposed form. These viewers were:

1. Text-only access: this is simply a set of standard glish functions which cause the appropriate text to be written to stdout when called from the glish command line.
2. Web browser access: a combination of glish and perl scripts extract the text information (using the same function calls as the text-only browser), and translates them into HTML.
3. Dedicated Glish/Tk browser: like the web browser, but the text is sent to a Glish/Tk client for (arguably) easier viewing.

There will be a need very soon for a fourth “viewer”, one which creates static HTML or postscript from the glish, C++, or other source code, so that help may be obtained without actually running the code in which it is embedded. This is a job of at least moderate complexity, somewhat similar to “cxx2html”, the AIPS++ documentation extractor.

Example: Help for a Glish Function

“fitPoly” is one of the first glish functions written for the GBT integration tests in 1995. The original help for this function consisted of simple glish print statements hard-coded into a glish function called “helpFitPoly”:

```
print "fitPoly (x, y, order), linear least-squares fit of a polynomial of"
print "the specified <order> to the x and y vectors";
print "returns a vector of length <order>, containing the coefficients of"
print "the fitted polynomial"
```

In the proposed scheme, the author of fitPoly reads in a bit of boilerplate text to the glish source code, and then edits it, adding the function name, and all of the relevant explanations. This allows simple glish functions (which would become standard parts of the the standard file “help.g”) to extract the information from the glish source. (In this example, and to keep things relatively simple, help information for parameters is not shown.) Here is the boilerplate:

```
::category := '';
::nameString := '';
::help.aproposKeywords := '';
::help.shortDescription := '';
::help.longDescription := '';
::help.returnValue := '';
::help.caveats := '';
::help.seeAlso :=
::help.example := [=];
::help.example.code :=  '';
::help.example.comments := ''
```

which when edited, looks like this:

```
fitPoly::category := 'fitting';

fitPoly::nameString := 'fitPoly';
fitPoly::help.aproposKeywords := 'fit fitting polynomial linear least \
 squares llsq vector order';

fitPoly::help.shortDescription :=
'fitPoly (x, y, order): fit polynomial of specified order to x-y data';

fitPoly::help.longDescription :=
'fitPoly (x, y, order): fit polynomial of specified order to x-y data. \
This function performs a linear least-squares fit of a polynomial of \
the specified <order> to the <x> and <y> vectors.  It returns a vector of \
length <order>, containing the coefficients of the fitted polynomial.';

fitPoly::help.returnValue :=
'a vector of floating-point numbers: a, b, c, d... which are the constant \
terms in the expression   a + bx + cx^2 + dx^3 + ...';

fitPoly::help.caveats :=
'At present (Autumn, 1995), this function will return meaningless values if \
called with x or y vectors that contain numbers whose magnitude swamps their \
differences.  This is because the C++ polynomial fitting client uses a class \
which employs single precision floating point numbers, not doubles or \
complex. This will be remedied in the near future... In the meantime, here \
is a way to work around the problem: \
<show glish code that normalizes x and y before fitting, and then.\
un-normalizes them afterwards.>';

fitPoly::help.seeAlso :=
'glish client: $AIPSPATH/code/trial/apps/gfitpoly/gfitpoly.cc \n\
 AIPS++ library class: \
http://baboon.cv.nrao.edu/aips%2b%2b/docs/trial/implement/Fitting.html';

fitPoly::help.example := [=];
fitPoly::help.example.code :=  '\
  x := array (1:10,10)                         # 1\n\
  y := 2 + (3 * x) + (4 * x * x * x);          # 2\n\
  coefficients := fitPoly (x,y,2);             # 3\n\
  fittedValues := evalPoly (x,coefficients);   # 4\n\
  plotxy (x, y, "original curve");             # 5\n\
  plotxy (x, fittedValues, "fitted curve")     # 6';

fitPoly::help.example.comments := '\
This code fragment shows how to fit a second order polynomial to some data \n\
calculated with a cubic function -- so the fit will be imperfect.\n\
  line 1:  create an array of 10 elements, initialized to 1-10\n\
  line 2:  create y as a function of x\n\
  line 3:  get the three coefficents of the fitted, second-order polynomial\n\
  line 4:  evaluate those coefficients at each of the original x values\n\
  line 5:  plot the original function\n\
  line 6:  plot the fitted curve';
```

There is no doubt that the composition, typing, and proofreading of this help text is a great deal of work. The virtue of the proposed approach is that it adds as little as possible to this work: there is no markup and formatting[2] to learn (and to get wrong...). There is simply a small and comprehensible set of variables to assign.

With these function attribute definitions in place, the glish user may type, for example,

```
return_value (fitPoly)
```

and will get an appropriate[3] display of this text:

```
a vector of floating-point numbers: a, b, c, d... which are the constant
terms in the expression   a + bx + cx^2 + dx^3 + ...;
```

[2]Strictly speaking, there are two syntactic conventions the function author must understand: the glish line continuation character (the backslash) and the glish newline token (backslash n). It may be that the first of these can be eliminated in time.

[3]"appropriate" will depend on the viewer. See the **Help Viewers** section for an explanation.

For each of the fields in the help attribute attached to a glish function (these fields include *category, nameString, aproposKeywords, shortDescription, longDescription, returnValue, caveats, seeAlso, example.code, example.comment,* and several fields for each function parameter) there is a companion function for retrieving the information. These fields, and these companion functions, all have long, descriptive names; the reader should keep in mind that it is straightforward to selectively compose a particular combination of these fields (perhaps all of them at once), and to retrieve them with a simple, short command. In a GUI, of course, no names need be typed at all.

Implementing Help for Distributed Objects, C++ Clients, and OO Glish

Once the protocol for help is agreed to – possibly including some modification of the *atoms* mentioned above – this same protocol can be applied to other software entities. For example, we could sensibly decree that glish clients written for AIPS++ respond to a standard set of help events.[4] The same would apply to AIPS++ distributed objects: it looks now like there will be a fundamental base class for tasking, and this base class is the natural place to include a standard help method or methods. In the same spirit, when Glish gets objects, it will be easy to translate the current functional approach to object methods.

[4]There might be a single"help" event, with a variety of values, or there could be an event for each atom in the protocol.