User Interface Tutorial

How to use the Input class and interface with Khoros

Peter Teuben and Timothy P. P. Roberts

1 Introduction

During the prototype stage a basic command line user interface was developed. This attempt at easing the trouble of passing information to an executable program resulted in a set of C++ classes called `Param` and `Input`. The programmer may simply include the `Input` class into their code and have immediate Command Line User Interface (CLUI) capabilities. The programmer's AIPS++ application is run from the unix level prompt by invoking its name and listing linearly on the same command line the **`keyword=values`** or **`-keyword values`** associated with proper execution. The `Input` and `Param` classes will successfully parse the command line into the executable program and check for appropriateness.

The CLUI capabilities are further extended to a Graphical User Interface through the use of the Khoros Cantata environment. The user starts up Cantata from the unix prompt and, by utilizing a series of pull down windows, invisibly creates an X-based window for visual display of all parameters associated with the AIPS++ application's need for external input.

As more AIPS++ applications develop, the Canata environment will allow the user to connect the output of programs to the input of other programs. The connections between applications are formed by simple pointing and clicking. The entire workspace may then be saved to a file where default values are stored. By clicking on the **run** button the chosen chain of applications is executed under the parameters previously saved.

Finally, the more sophisticated programmer may utilize the Khoros Preview environment to invent more creative windows for their application. Sub-window dependence, thorough help windows, and more esthetically pleasing lay-out are all possible. All windows will hook seamlessly onto the `Input` and `Param` CLUI written by the programmer.

This document contains working examples, as well as hypothetical examples to get a discussion going on future extensions. These latter ones will be clearly marked as such.

Some time has also been spend in showing that both an AIPS interpreter and a Graphical User Interface (GUI) can be plug-in compatible user interfaces. For example, a functional GUI for Khoros[1] is available for demo purposes. An AIPS shell interpreter can be thought of in terms of the MIRIAD shell interpreter.

[1] (c) University of New Mexico

The `Input` and `Param` classes and their associated user interfaces should be viewed as AIPS++ development tools only. The limitations imposed by the current Khoros GUI techniques dictate the need for a more flexible system. It is believed that as the C++ environment matures an obvious candidate for the final AIPS++ Graphic User Interface will emerge. Until that time, the methods outlined here will, hopefully, reduce the programmer's worry list.

2 Parameters

2.1 Program Keywords

The basic command line user interface is an ordered series of `keyword=value` pairs, which we call **parameters**

The `class Param` (see `Param.h`) implements one single such **parameter**. Values may be Int, Block<Int>, double, Block<double>, Bool, or Strings. In addition to a name and a value, a `Param` parameter has a variety of other attributes, such as a one-line help string (useful when being prompted for input or with hypertext identifiers, etc...), a type, a range and optional units. All of these attributes are character strings; parsing and error checking is done at a different (hidden) level. The programmer, however, will never interact with a parameter through it's `Param` class interface. Interaction is done with the `class Input`, which is a container of `Param`'s, with a variety of user interface attributes (help-level, debug-level, etc...).

Although the programmer must supply the user interface with a number of predefined **program parameters**, the user interface itself will create a small number of **system parameters** (help=, debug=). The purpose of these is to tell the task how to communicate with the user and it's environment, and give the user control over these items. For example, the user may want to see (debug) messages above a certain threshold level. The programmer simply adds debug levels to their code and allows the user to specify how deeply they wish the debugging to progress.

For the benefit of the Programmer, the user interface also defines a number of standard parameters ("templates"), which can be copied and bound to a program parameter. This predefined group of parameters (shown below) will allow the novice programmer to skip even the process of parameter creation if the application's interface needs are meet by the standard templates.

```
std.Create("infile",   "",            "Input file");
std.Create("outfile",  "",            "Output file");
std.Create("lstyle",   "solid",       "Line style for current object");
std.Create("lwidth",   "thin",        "Line thickness");
std.Create("xyzselect","",            "Image Region of Interest");
std.Create("uvwselect","",            "Visibility selection of Yegs");
std.Create("font",     "Helvetica12", "Font selection");
std.Create("device",   "",            "Graphics device");
```

The advanced programmer may ignore these definitions altogether and create their own parameters meeting the exact needs of their application. Most programs are probably happy with a simple set of parameters, like a linear list.

All input as well as output should be controlled by the user interface. The Astronomer has a varying degree of control over how and where input and output occurs. In the command line interface system control occurs through a small number of system parameters on the command line. In a GUI there will be a different mechanism visible to the user to do this, but control to the task always occurs through a set of system keywords.

For example, a interactive UNIX shell session may look like:

```
1% MyProgram key1=val1 key3=val3
2% MyProgram key1=val1 key2=val3 debug=5
3% MyProgram help=prompt
4% MyProgram help=pane > prog.pane
```

In command `1%` the user has set several parameters for the program MyProgram to applicable values. The `2%` command line invokes the executable and sets the level of displayed debugging to the programmer specified 5th level. Command `3%`: the user is prompted, and parameter default values are restored. Command `4%` gives an example of the self-describing mode of programs, where a pane description file for Khoros has been constructed. (See Chapter 5 [khoros], page 14). The latter is the first step toward building a Khoros Graphic User Interface.

2.2 System Keywords

Next to **program parameters** there exists a set of **system parameters**, which form the control between the program and upper level user interfaces.

The system keywords are:

`help` Various options can be given here, some of them may have obvious conflicting interests.

- '`pane`' The user interface is the program is described in `KHOROS` pane format. The output must be redirected to a pane file, and placed in the appropriate directory for `KHOROS` to pick it up. Program then exits normally. (See Chapter 5 [khoros], page 14).

'`miriad`'	The user interface is the program is described in **`miriad`** doc format. The output must be redirected to a doc file, and placed in the appropriate directory for **`miriad`** to pick it up. Program then exits normally.
'`prompt`'	Run-time prompting of keyword values.
'`debug`'	It makes sense to have separate levels of output, which may only be of interest to an **`aips2usr`**, **`aips2prg`** and **`aips2mgr`**, in increasing complexity. The value of this keyword is numeric, the larger the number, the more output. 0 would produce next to nothing, 1 more, 2 even more etc. Adding 10 to the number would activate the **`aips2prg`** level, and adding 100 the **`aips2mgr`** level.

3 Input's Class Member Functions

3.1 Param Class

The class `Param` is a single element of a linked list of parameters known collectively as an `Input`. The class Param **will not be visibly used by the programmer**. The following itemization is to assist the programmer in understanding `Input`'s utilization of `Params`.

```
Param MyParameter(key, value, help, type, range, units);
```

- A `Param` substantiation has six ordered arguments. The first is a `key`. The programmer may choose any word as the key as long as no other predefined system parameter keys are of the same name. Currently, the keys "help", "error", "debug" and "sound" are predefined.
- The second argument is the `value` for the parameter. Like all other `Param` arguments, the value is passed by a String type. Within that limitation, values may be Integers, Block<Int>, Doubles, Block<double>, Boolean, and String.
- The third argument is the `help` string. This allows prompting of the user when the CLUI is put in help=prompt mode. The help argument also gets used when constructing the Khoros GUI. The programmer should not feel the need to be terse when writing the help argument.
- The fourth through sixth arguments to `Param` are optional. They are the `type`, the `range`, and the `units` of the parameter. All will be parsed invisibly and used to throw errors should the value not meet an obvious restriction imposed by the above.

3.2 Input Class

The `Input` class is a means for building a linked list of parameters and gaining access to them once created. `Input` takes care of system/environment variables and assigns their values within the programmer's code. The linked list of parameters is limited only by the number of names the programmer can dream up. The programmer need not think hard on the order of definition of parameters in Input. The list of `key=values` given on the command line by the user need not be in any specific order.

The definition of parameters is by simply creating an `Input` and then using the appropriate `Input` member function. The programmer always starts by creating a `Version` parameter (for identification of application programs by revision) and a `Usage` parameter (for identification of the application's correct methodology.) Then the programmer adds to the list of parameters as necessary.

Since storage is in string form, the Input member functions may be tricked into returning innovative values if the "type" argument is not initially coded into the parameter. The Input linked list of Params may be "closed" when no further parameters are needed by the application.

3.3 Input Member Functions

- `Input();`

  `Input(int);`

  The use of a integer argument to the constructor for `Input` is mandatory. It enables the creation of parameters. The `Input(0)` constructor is to create a minimal parameter list without environment variable initialization. The use of `Input()` (default constructor value = 1) or `Input(1)` explicitly pre-defines a "standard list" of program parameters. These standard parameters may or may not be used but they allow the programmer to "skip" creation of parameters when the application code has trivial need for user interface. Additionally, `Input(1)` initializes any environment/system variables and puts the program in no-prompt mode unless environment variable HELP is defined with value "prompt". The output debug level is set according to the value of the environment variable DEBUG. The maximum number of error messages to be outputed is set according to the value of the environment variable ERROR.
- `void Version(String);`

  `void Usage(String);`

  These are the first of the parameters the user defines. Their use is mandatory since they set the version and usage string for announcements.

  ```
  Input myparams(1);
  myparams.Version("MyProgram: v1.23 by Me!");
  myparams.Usage("MyProgram takes the besmurch of a doo-hickey");
  ```
- `void Create(String, String, String);`

  `void Create(String, String, String, String);`

  `void Create(String, String, String, String, String);`

  `void Create(String, String, String, String, String, String);`

  The `Create` functions make a new program parameter, either from scratch or looking it up from an internal list of templates. The function also checks whether parameters can still be created, and whether the `keyword` is unique for the program. The `value`, `help` and remaining arguments are all optional.

```
String key("infile"), value("/u/myHome/good.data");
String help("The infile is the source of everything beautiful");
myparams.Create(key, value, help);
```

- `void SysCreate(String, String, String);`

  This function is used to create a dedicated **system parameter**. It is possible to create a system parameter that does not have an equivilent external environment variable.

- `void Close();`

  Disable the creation of parameters. Highly recommended, but not required.

- `void ReadArguments(Int argc, char *argv[]);`

  This function is used only after an `Input` type has been constructed and the appropriate parameters created. `ReadArguments` reads in the command line `keyword=values` and parses them to the appropriate parameters.

- `double GetDouble(String key);`

  `Block<double> GetDoubleArray(String key);`

  Get the double value of the parameter (or 0.0 if unknown key). If the program is in prompt mode, the stdout will ask the user for the value.

  ```
  double offset = myparams.getdouble(uvw_offset);
  ```

- `int GetInt(String key);`

  `Block<Int> GetIntArray(String key);`

  Get the Int value of the parameter (or 0 if unknown key). If the program is in prompt mode, ask the user for the value.

- `String GetString(String key);`

  Get the string-type value of the parameter (or "" if unknown key). If the program is in prompt mode, ask the user for the value.

- `Bool GetBool(String key);`

  Get the Bool value of the parameter (or FALSE if unknown key). If the program is in prompt mode, ask the user for the value.

- `String GetRC(String name);`

  Get a resource or global environment variable value, returns empty string if nothing found. **NOT Currently Implemented**

- `int Count();`

  Get the total number of parameters of this program.

- `Bool Debug(int l);`

  See if the current debug level is thresholded.

- `Bool Put(String key, String value);`

  `Bool Put(String keyval);`

Set a new value for an existing named parameter Returns FALSE if key is an unknown parameter name. The function can also be called with a single argument of the form **`key=value`**, where **`key`** is a valid new parameter name, and where **`value`** may be empty (the '=' is required though). In this case a new parameter will be created (provided that creation is still allowed).

```
myparams.Put("uvw_offset=15.0");
```

- **`void Announce();`**

  Announce program and version. This may also terminate execution if certain self-describing features were requested.

- **`Bool More();`**

  If program is interactive, this will halt execution, and prompt user to continue. If not, it always returns FALSE.

- **`void Exit(int level);`**

  Emergency exit from the user interface, while still saving the environment as long as level > 0

4 Implementation of `main(int argc, char **argv)`

The programmer uses **main(int argc, char **argv)** to initially pass the command line parameters. Within main, the use of the member functions of the **Input** class create the appropriate parameters and pass their values to the internal variables.

We'll finish up by looking at some code:

```
//   the code for this is available in aips++/code/aips/test/xyPlot.C
//   copyrights removed for space and dollar signs removed from RCS Id:
//   keyword to prevent substitution.

#include <aips/aips.h>
#include <aips/Vector.h>
#include <aips/String.h>
#include <aips/Input.h>      // need this if you want it to work
#include <aips/Plot.h>
#include <iostream.h>

main(int argc, char *argv[])
{
  Input inputs(1);
  // Define our input structure
  inputs.Version(
          "Id: xyPlot.C,v 1.1 1993/01/29 20:45:48 bglenden Exp");
  inputs.Usage("Plot y vs. x vectors from an AipsIO file");
  inputs.Create("xyfile",
          "/tmp/xy.aipsio",
          "File which contains xy vectors",
          "InFile");
  inputs.Create("overplot", "False", "Multiple plots?", "Bool");
  inputs.Create("lines", "True", "Plot lines or points?", "Bool");

  // and Fill them from the command line
  inputs.ReadArguments(argc, argv);

  try {
    const char *filename = inputs.GetString("xyfile");
    AipsIO xyfile(filename, AipsIO::In);
    Vector<float> x, y;
    Plot plot;

    xyfile >> x >> y; // initial vectors
    plot(x,y,inputs.GetBool("lines"));

    for (;;) { // forever
28        xyfile >> x >> y;
29        if (inputs.GetBool("overplot") == True) {
30            plot(x,y,inputs.GetBool("lines"));
31        } else {
32            plot.newPlot();
33            plot(x,y,inputs.GetBool("lines"));
34        }
35      }
36  } catch (AipsIOError x) {
37       ; // nothing - no more data
38  } catch (AllocError x) {
39       cerr << "AllocError : " << x.getMesg() << endl;
40       cerr << "Size is : " << x.size() << endl;
41  } catch (AipsError x) {
42       cerr << "aipserror: error " << x.getMesg() << endl;
43       return 1;
44  } end_try;
45
46   cout << "Any key to exit:\n";
47
48  char ch;
49  cin.get(ch);
50
51  return 0;
52}
```

Let us discuss this program line for line.

01 - This is the method of passing the command line through to the main body of code. This obviously makes it mandatory. The inclusion of the argc, argv is very well discussed in Stroustrup.[1]

03 - The instantiation of **Input** in the variable **inputs(1)** is done with an argument of (1) to indicate the constructor should build **inputs** with a standard set of program parameters and read in values for the system parameters. An argument of (0) would build an **Input** that was empty and would obligate the programmer to build the linked list of **Params** explicitly.

04-05 - The version of the code is stored within the **Input**. Note the optional use of RCS keyword substitution[2] allows the code to be automatically updated.

[1] The C++ Programming Language, page 87

06 - The `Usage` field assists the user when thrown during an error.

07-11 - The `Create` member function of `Input` builds, in this case, a parameter called `xyfile`, immediately filled with the String containing the directory that holds the data. The help String is useful for new users or prompting. The fourth argument of `InFile` is the optional type of the parameter's value. Any suitable String may be used. Missing from this example are the optional fifth and sixth arguments, the parameter's value's `range` and `units`, respectively.

12 - This is another instantiation of a `Param` inside of `input`. This parameter will be referenced by the `keyword "overplot"`. It is initialized to False and is of type Bool.

13 - This line is the third and final `Param` placed in `inputs` and is recognized by the code when accessed with `keyword "lines"`.

16 - The call of `ReadArguments(argc, argv)` should be done after the list of `Params` has been completed. This line of code fills the values from the command line. A `keyword` that doesn't match will throw an error.

19 - At this point the local variable `filename` is initialized to the String value held within the parameter accessed through the `key "xyfile"`. Recall that the value of `xyfile` was originally set to `"/tmp/xy.aipsio"` but would be replaced with the proper value at execution. The `GetString` member function returns either the default value specified during the `xyfile` parameter's instantiation or the value placed into it from the command line use of `xyfile=myfile`.

25 - Here the boolean value of the `Param` called `lines` is inserted into the call to the function `plot`.

29 - Again the `Input` interface has its parameter called `overplot` return a boolean to be used as a test for an "if". The `GetBool(key) Input` member function may be reading the default value of the appropriate parameter called `key` or using the value passed from the command line.

30 & 33 - Another call to `plot` that uses the boolean value stored in the parameter called `lines`.

Some comments:

- The **program parameters** are defined through the `Create` member function, of which only the first three arguments are required. Note the order of creating parameters is relevant, to aid

[2] See the co man page.

command line shortcuts. In addition, a `Version` and `Usage` string should be supplied to the user interface. Apart from the obvious one place where this information should be stored, they can also serve as forced execution version control (history files contain this version id, and utilities may exist that extract an executable script from this: if the current version and history file version differ, `warning` or `error` may be called.

- The example above doesn't make use of the ability to reset a parameter to a value from within the main. If the programmer wanted a parameter to have a user inserted value (read from the command line) at certain parts in the code but reset to a hard-wired value later, he/she could use a call to the `Input` member function `Put(key)`. A short example using the variables defined in the example above may be:

```
for(Int somecounter = 0; somecounter < 5; somecounter++) {
   if(inputs.GetBool("lines") && (somecounter > 10))
      inputs.Put("lines=False");    // parameter "lines" now is false
   plot(x,y, inputs.GetBool("lines"));
   }
```

5 KHOROS User Interface

5.1 Cantata

Khoros is a programming environment, within which visual programming is implemented in a program called `cantata`. This program allows the user to visually place tasks on a drawing board, connect them with data-flow paths, fill in the parameter values, and execute single tasks or complete data-flow diagrams. What happens at the lower level is the GUI executes a Khoros executable with parameters set via command line switches. The user can monitor this in the parent window from which `cantata` was started up, as well as all normal output the program produces.[1]

Given this principle, we can let Khoros `cantata` talk to our simple user interface. The essential information needed by `cantata` to run 'foreign' programs, is a User Interface Specification (UIS) file, also referred to as a **pane** file (each basic program in Khoros typically comes with a predefined `program_name.pane` file).

Since an AIPS++ task is now self-describing (all relevant information is in the executable, not in a file on disk) and can supply the caller with internal information about it's knowledge of keywords, and defaults, it is relatively straightforward to automate this process and have each program create a `pane` file from itself. This is done by the executable itself through the `help=pane` parameter.

Utilizing the `xyPlot` program as an example,

```
xyPlot help=pane > xyPlot.pane
```

where the `xyPlot.pane` will look like:

```
-F 4.2 1 0 170x7+10+20 +35+1 'CANTATA for KHOROS' cantata
 -M 1 0 100x40+10+20 +23+1 'An AIPS++ program' aips++
  -P 1 0 80x38+22+2 +0+0 'Plot y vs. x vectors from an AipsIO file' xyPlot
```

[1] The KHOROS environment is available via anonymous FTP from pprg.eece.unm.edu or may be ordered on tape from the Khoros Group, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131. Installation of Khoros is detailed in Volume II (Khoros Programmer's Manual), Chapter 10 of the Khoros documentation, also available at the above locations.

```
      -I 1 0 1 1 0 1 50x1+2+2 +0+0 '/tmp/xy.aipsio' 'xyfile' 'File which contains
       xy vectors' xyfile
      -s 1 0 1 1 0 50x1+2+4 +0+0 'False' 'overplot' 'Overplot or multiple plots?'
       overplot
      -s 1 0 1 1 0 50x1+2+6 +0+0 'True' 'lines' 'Plot lines or points?' lines
      -H 1 13x2+1+8 'Help' 'Help for xyPlot' aips.help
      -R 1 0 1 13x2+39+8 'Run' 'RunMe' xyPlot
     -E
    -E
   -E
```

The indentation is purely done to visualize the hierarchy which can be present in more complicated pane files. Currently most parameters are string parameters (the '`-s`' line) except input (the '`-I`' line) and output files (the '`-O`' line), which need to be labeled as such in the creation phase (`Input.Create()` in our case).

A few comments and problems:

- Array or composity parameters (C structures) can only be implemented as strings, since Khoros cannot handle those. Structures can be implemented via the mutually exclusive or inclusive tags that can be part of a UIS file. This can be implemented later.
- Though AIPS programs lend themselves to the dataflow based model, they may not like their existing files to be overwritten. By default, datasets are assumed re-usable and implemented as temporary file, sockets or shared memory (user and run-time selectable). It is up to the user to select the mode, or define his own name. Selection your own name turned out to be a bit laborious. Selecting sockets or shared memory as data-model will cause problems if datasets are implemented as directories (as is for example in Miriad).
- Programs that send their output to `stdout` or `stderr`, or for that matter, reading from `stdin` will get stuck. If one sticks to the load-and-go principle, like AIPS programs normally are, there is no problem with input. Output is another story. The `stdout` goes to the screen (whichever cantata was started up from), and `stderr` will appear in the error window that the glyph will pop up.
- The pane files should be organized in the KHOROS tree, '`$KHOROS_HOME/<toolbox>/repos/cantata/`'. In addition, subform files (pane files which define a hierarchy in tasks) need to be defined too. This process is not easily automated. The Khoros program `Preview` may assist in building a more pleasent pane file.[2]
- The on-line documentation (referenced by the '`-H`' line on a pane file) comes in two options: single file and directory. If the name, referenced on the -H help line, is a directory, within

[2] See the Khoros documentation, Volume II (The Khoros Programmer's Manual), Chapters 2 and 3, for details.

that directory there can be several files which will be displayed. Khoros/Cantata comes with a program **formatdoc**, which converts an nroff-like document in the format that this help system wants (more or less ascii).

5.2 The KHOROS demo

In this section we will walk you through some examples within **cantata**, the Khoros graphical user interface with visual programming. It will read a lot more pleasant to be in **cantata** when you read this. You either enter it the way you want, or follow the demo route:

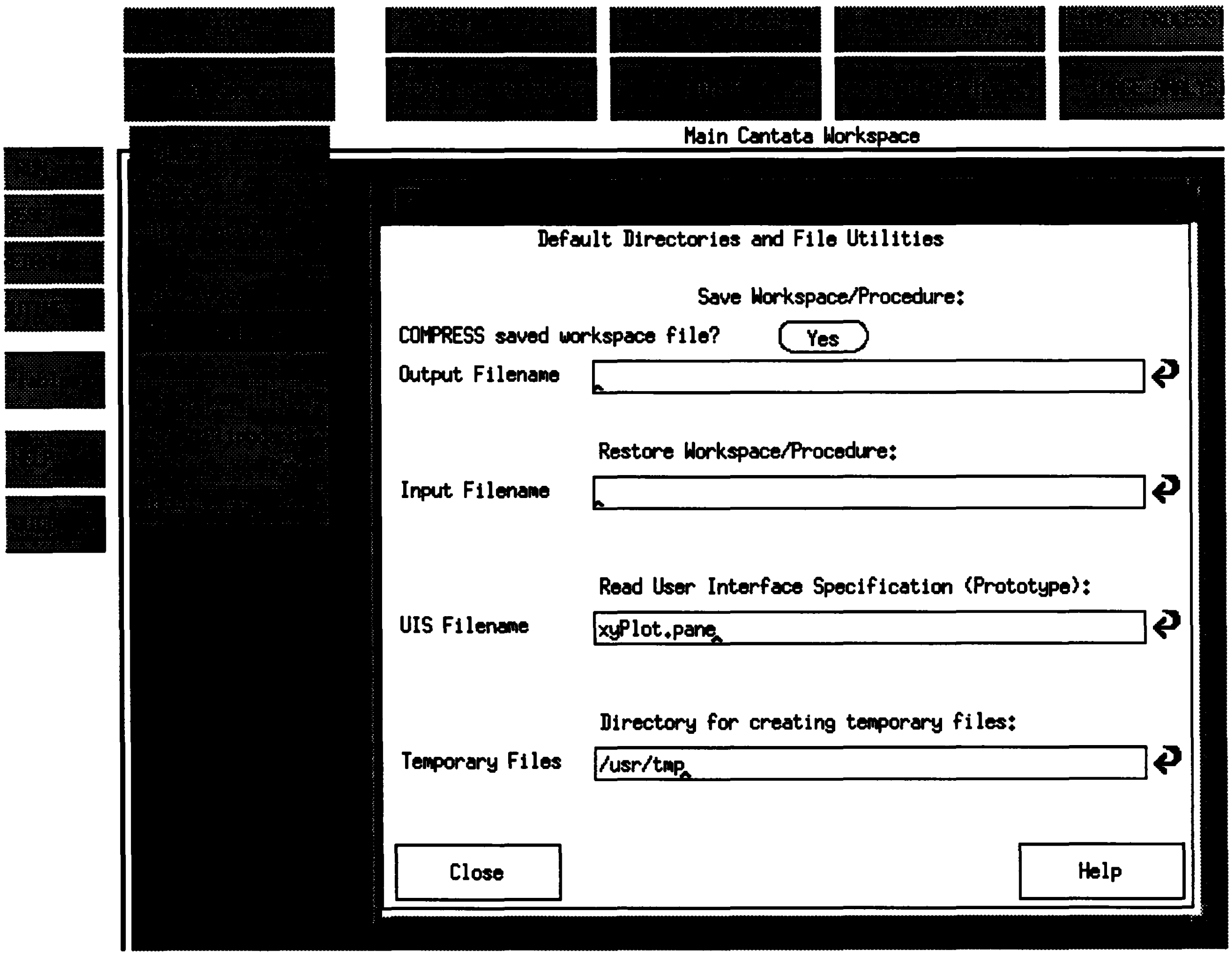
```
cd $AIPSHOME/code/aips/test
ao demo
ao demo.workspace.Z
csh demo
```

This will start up **cantata**[3] , with the demo predefined for you. The setup was saved in a file called **demo.workspace.Z**.

You can also manually load in new programs as follows:

- pull down the **Workspace** menu, and select the **File Utilities** option.
- Fill in the name of the relevant pane file by using the builtin file browser: click on the area of the window where the words **UIS Filename** appear. A window will open with a list of filenames. You may navigate through the directories of your machine by typing them in the small field at the bottom of the window or by clicking the mouse cursor on the appropriate directory/pane filename. The name will appear in the original **File Utilities** box to the right. A glyph will appear on the **cantata** workspace. Drop it with a left mouse click anywhere you like, you may always move it later.

[3] You must have KHOROS installed at your site.


Main Cantata Workspace

Default Directories and File Utilities

Save Workspace/Procedure:

COMPRESS saved workspace file?	Yes
Output Filename	

Restore Workspace/Procedure:

Input Filename	

Read User Interface Specification (Prototype):

UIS Filename	xyPlot.pane

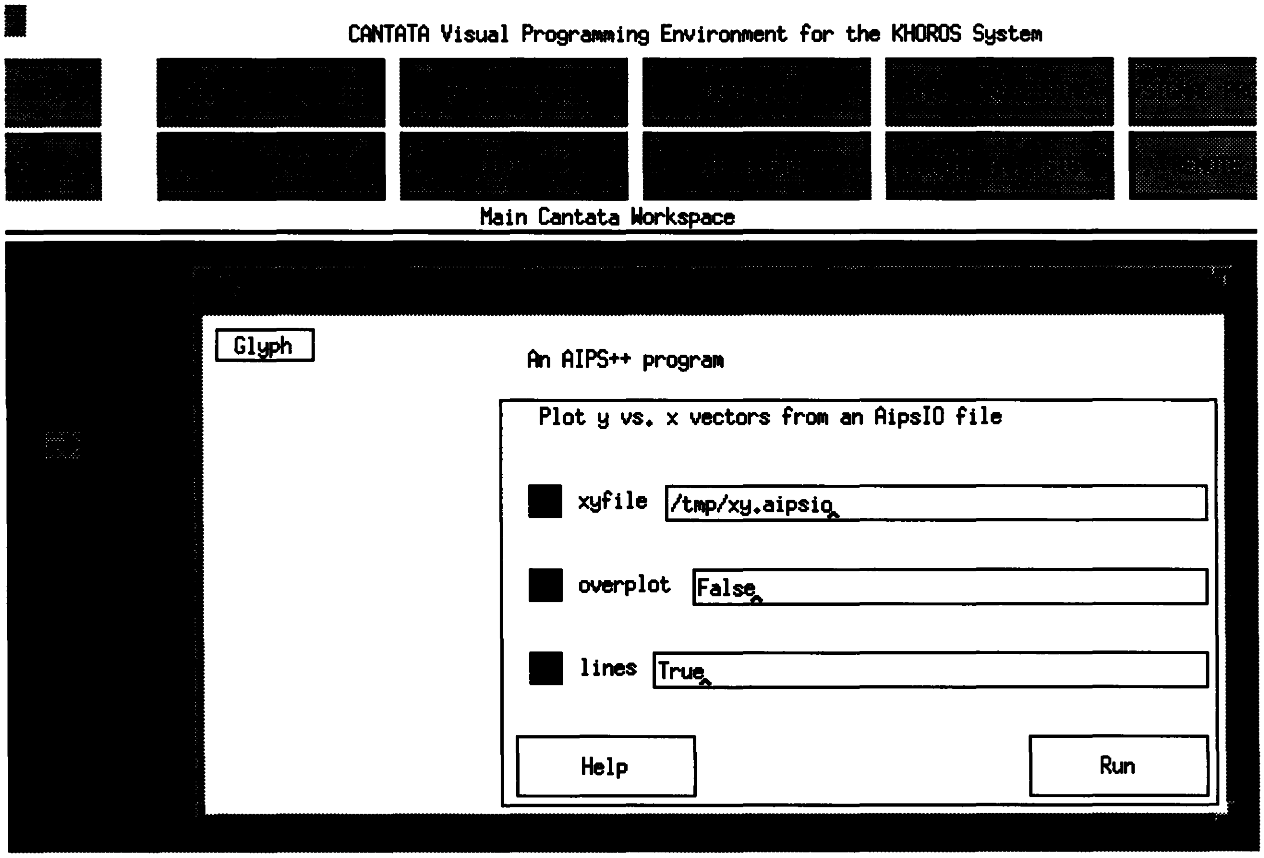
Directory for creating temporary files:

Temporary Files	/usr/tmp

Cantata - the workspace pull down menu and window.

Some comments:

- An individual application is represented by a glyph (the small white rectangle appearing on the cantata workspace.) Each glyph displays a set of red (bomb), blue (form) and green (on/off) buttons. They represent a Quit, Edit and Run functionality. Typically a user selects the programs to run, opens a glyph by clicking on the middle/ blue (form) button, and fills out the necessary parameters. Output appears in the parent window, from which **cantata** was started.


CANTATA Visual Programming Environment for the KHOROS System
Main Cantata Workspace
Glyph
Help
Run

An AIPS++ program

Plot y vs. x vectors from an AipsIO file

xyfile /tmp/xy.aipsio

overplot False

lines True

Cantata - the manually loaded **xyPlot** glyph and window.

- Input file's parameters inside the pane (visible after you've clicked on the blue button) come with a file browser to easy filename insertion.
- Parameters can use the program default by unsetting the option button. It normally comes as a filled black square, meaning the entered value will be the one used by the program.
- Dataflow lines (the green lines) can be created by first clicking on an **out** (OutFile) arrow, followed by an **in** (InFile) arrow. An OutFile can be split (i.e. goto various InFile's), but of course one cannot merge various OutFile's into one InFile. Any previous connections would then be broken.
- Dataflow lines can be files (default), sockets or shared memory. Intermediate results are typically lost, after you exit from Khoros, though can always be stored into a file at demand. Click the mouse while pointing at a dataflow line, and you'll see what they mean with this.
- With the RESET button (on the left of the drawing board) you can force the flow to be run all the way through for the next RUN. Be sure all required inputs and outputs are connected, otherwise the flow will be interrupted. An inactive in/out can be recognized as a grey arrow, as opposed to a black arrow, inside the yellow rectangles.
- Saving a workspace can be done within the WORKSPACE/File Utilities menu option. The default demo is in **demo.workspace.Z**.
- A few Khoros specific tasks may also be of interest during the first few months of the project.
  1. To view an existing FITS file:

- Pick up `Input Data File` from `INPUT SOURCES`, select `User Defined` and either fill in the name of the FITS file in the box next to `User Specified File` or use the file browser by clicking on the `User Specified File` area itself. Then click on `glyph` to close it. Put the glyph anywhere on the drawing board, it should have the `input` label now.
- Pick up `Standard File Format` from `CONVERSIONS` and select `FITS to VIFF` (the `viff` format is Khoros' native image format). Again click on `glyph` to place it on the board, preferably just right of the previous `input` gly[h you dropped; it should have the `fits2viff` label now.
- Pick up `Display Image` from `OUTPUT` and select `Interactive Display`. Click on `Glyph` to close it, and place it just to the right of the `fits2viff` glyph. Check the label is `editimage`.
- Connect appropriate output with the inputs, and run the flow.

6 Conclusion

The ease of application programming offered by the `Input` and `Param` classes well offsets their current limitations. As the C++ environment matures, the author expects a replacement that will allow much more variety on the command line. The added beauty of `Input`'s trivial `Khoros` GUI generation should not be seen as justification for `Input`'s use beyond the development stage. `Khoros` was developed with its own visualization applications in mind. This limits the graphical user interface to load&run execution.

The AIPS++ use of the `Khoros` environment is as simple or as complex as the programmer wishes to make it. Those programmer's wishing to make their GUI's more functional (e.g. interdependent sub-windows) are refered to the Khoros documentation, Volume II (The Khoros Programmer's Manual), Chapters 2 and 3, for details.