NASA Technical Memorandum 4543

NLEdit—A Generic Graphical User Interface for Fortran Programs

Brian P. Curlett
Lewis Research Center
Cleveland, Ohio

National Aeronautics and
Space Administration

Office of Management

Scientific and Technical
Information Program

1994

Summary

NLEdit is a generic graphical user interface for the preprocessing of Fortran namelist input files. The interface consists of a menu system, a message window, a help system, and data entry forms. A form is generated for each namelist. The form has an input field for each namelist variable along with a one-line description of that variable. Detailed help information, default values, and minimum and maximum allowable values can all be displayed via menu picks. Inputs are processed through a scientific calculator program that allows complex equations to be used instead of simple numeric inputs.

A custom user interface is generated simply by entering information about the namelist input variables into an ASCII file. There is no need to learn a new graphics system or programming language. NLEdit can be used as a stand-alone program or as part of a larger graphical user interface. Although NLEdit is intended for files using namelist format, it can be easily modified to handle other file formats.

1 Introduction

Many computer programs used by engineers are still batch oriented; that is, they read an input file, perform computations, and produce an output file. Many of these programs are written in Fortran and use a namelist format for input data. A namelist file consists of a list of variable names, each followed by a value. When a namelist read command is executed, the values of the variables in the computer code are set to the values of the variables in the namelist input file. Although this is a convenient way of entering data into a program (compared with fixed-format input), it can be difficult for users to memorize all the possible inputs and permissible values.

Graphical user interfaces are now used to simplify the execution of programs. A large program, with many input parameters, will benefit most from a graphical user interface (or GUI). Unfortunately, a GUI for a large program requires a large development effort by trained software engineers. Even interfaces for smaller programs can take a significant programming effort. The code required for the interface could easily become larger and more complex than the code required for the analysis! Worse, most engineers are not trained in the programming of graphical user interfaces and, therefore, cannot maintain the interface as changes to the analysis code are made. These problems make it difficult to justify the development of a GUI for smaller or less frequently used programs. This results in a mixed environment of graphical and text-based codes.

The program presented here is a compromise between developing custom interfaces and using text-based input. The program is called NLEdit (for namelist edit). NLEdit is not intended to be a programming language for user interfaces. It is intended is to give some of the conveniences of a graphical user interface without the user having to generate and maintain a large graphics program. It can be thought of as a fancy, highly customized text editor. NLEdit was programmed in C using X-windows and the OSF/Motif widget set.

Although NLEdit does not provide graphical representation of data nor processing of output data, it does provide facilities such as a list of all input variables, help information, a list of permissible values, default inputs, and error checking. Having this information on-line can save the engineer from time-consuming searches through manuals. Besides being difficult to access, hard copy manuals are often out of date. The on-line information in NLEdit can be easily updated to reflect the latest changes to the corresponding analysis code.

NLEdit is customized for a particular application using a *data definition file.* The data definition file is an ASCII file containing information about program inputs such as variable name, type, dimensions, default, limits, and help information. NLEdit reads this information into a database and then uses it to produce an appropriate interface. The user interface changes only in appearance for a particular data definition file; no recompiling of code is necessary.

The NLEdit program is composed of three main parts: the calculator, the data dictionary (or database), and the graphics modules. The calculator module is used to convert an equation, in the form of a character string, into a numerical value. The data dictionary allows the other modules to store and retrieve information about specific items defined in the data definition file. The graphics module is the Motif code for displaying windows and processing input events.

This paper first gives a detailed description of what NLEdit does and how to use it. Then, each of the three main program modules is discussed for the readers interested in the internal workings of NLEdit.

2 User Interface Overview

Commonly, an engineer running a Fortran computer code will edit a namelist input file, execute the Fortran code, and view the code's output. This process is repeated until the engineer arrives at the desired results. As mentioned above, the NLEdit program can be thought of as taking the place of the text editor. NLEdit can also provide menu picks for executing the Fortran program and viewing files. Perhaps the best way to explain what NLEdit does is by giving an example.

2.1 Sample Interface

One of the codes that NLEdit has been used with is a turbomachinery map-plotting code called `mapplot`. A typical namelist input file for `mapplot` is shown in figure 1. Normally, the engineer would have to read a reference manual, determine what inputs are required, and type in the namelist file using a text editor. However, with NLEdit the engineer would just invoke `mapplot`, an easy-to-use input screen (fig. 2). The programming to produce such an interface requires only typing information from the reference manual into a data definition file. A sample data definition file needed to produce the `mapplot` interface is given in the appendix. This file contains only information that describes the input data; no graphics programming information is required. Details of the format of this file are given in section 3. In the following discussion, the namelist file (e.g., fig. 1) is referred to as the "input file." The modified version of this file, produced by NLEdit, is referred to as the "output file."

2.2 Main Window

The main window (shown at the top of fig. 2) is composed of a menu system and a message area. Menu picks are used for saving files, selecting the namelist to edit, setting options, calling help, and executing the analysis code. Details on using all the menu picks are available

```
&INPUT
 IOPT=2, AREA=.5, FONT='6x12', PSIZE=1, MAPFILE="test.maps",
 FACFLO=1.0, FACEFF=1.0, MAPFLO=5112, MAPEFF=5113,
&END
&INPUT
 IOPT=1, ANGLE=0, NDAT=8, DATSYM=1,  DATLIN=1,
 FONT='Rom14.500', PSIZE=0, GRID=0, GRAY=0,
 IRVAL=0, ICOLOR=14, MAPFLO=1004, MAPEFF=1005,MAPPR=1006,
 FACFLO=24.7547, FACEFF=0.890229, FACPR=0.769657,
 FLOWDAT(1)=25.3715,25.321,25.3681,25.4933,25.3175,24.7832,23.6231,
 PRDAT(1)=6,5.97566,5.99843,6.05716,5.97477,5.72539,5.26105,4.913,
&END
```

Figure 1: Namelist file for `mapplot`.

from the **Help** menu and are, therefore, not included in this report. The message area is used to display system messages and warnings to the user.

2.3 Namelist Editor

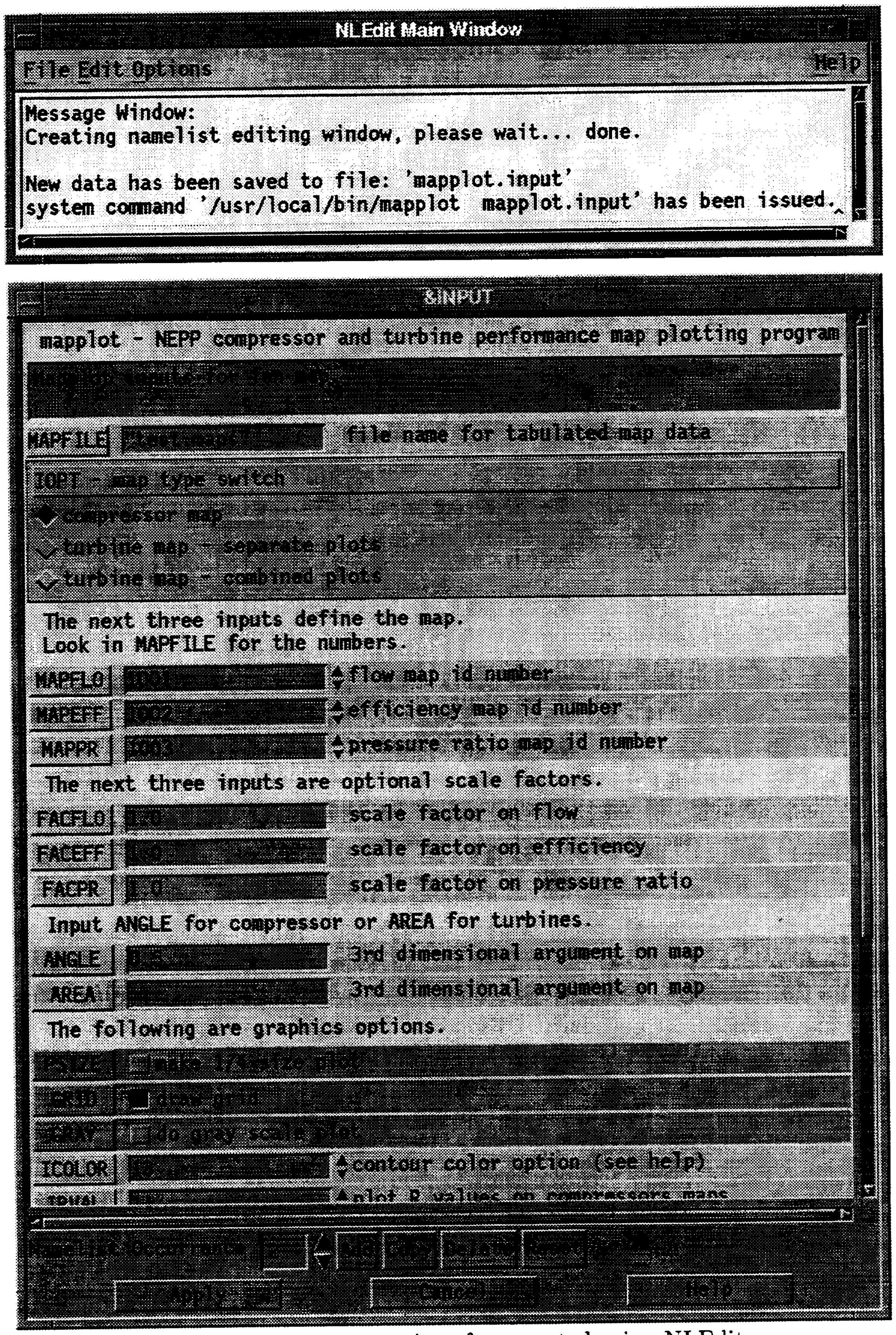
A window, like the one shown on the bottom of figure 2, is generated when the user selects a namelist name from the **Edit** menu. The form shown in this window consists of input fields for each variable in the namelist and comment lines. Under the first comment line is an input field for user comments. These comments will be included in NLEdit's namelist output file preceding the corresponding namelist.

There are three types of input fields for namelist variables: toggle fields, multiple choice fields, and text fields. In the example, the variable `GRID` uses a toggle field; the variable `IOPT` uses a multiple choice field; and the variable `MAPFILE` uses a text field (see fig. 2). The type of field used depends on the type of data represented. Toggle fields are used for variables that can have one of two values (usually true or false). Multiple choice fields are used for variables that can have one of several values. Text fields are used for all other inputs.

Each input field has a push button displaying the variable name and a label line describing the variable. Clicking on (pressing the left mouse button while the mouse pointer is on) the variable name invokes more detailed help information for that variable. Pressing the function key `F1` after selecting a field brings up display a window containing all information known about that variable (see fig. 3).

Using toggle and multiple choice fields to choose the value of a variable is straightforward. Screen highlighting is used to give the buttons a three-dimensional appearance; *in* is on or true and *out* is off or false. Multiple choice buttons have a "radio box" behavior, that is, only one button can be pressed in at a time.

The text fields can be used simply by typing values into the small text window provided. If the type of the variable is an integer (like `MAPFLO` in the example), then up and down arrow buttons are provided to increment and decrement the value, respectively.



```
Message Window:
Creating namelist editing window, please wait... done.

New data has been saved to file: 'mapplot.input'
system command '/usr/local/bin/mapplot  mapplot.input' has been issued.
```

mapplot - NEPP compressor and turbine performance map plotting program

MAPFILE [illegible] file name for tabulated map data

IOPT - map type switch
[✓ compressor map]
turbine map - separate plots
turbine map - combined plots

The next three inputs define the map.
Look in MAPFILE for the numbers.

MAPFLO 1001 flow map id number
MAPEFF 1002 efficiency map id number
MAPPR 1003 pressure ratio map id number

The next three inputs are optional scale factors.

FACFLO 1.0 scale factor on flow
FACEFF 1.0 scale factor on efficiency
FACPR 1.0 scale factor on pressure ratio

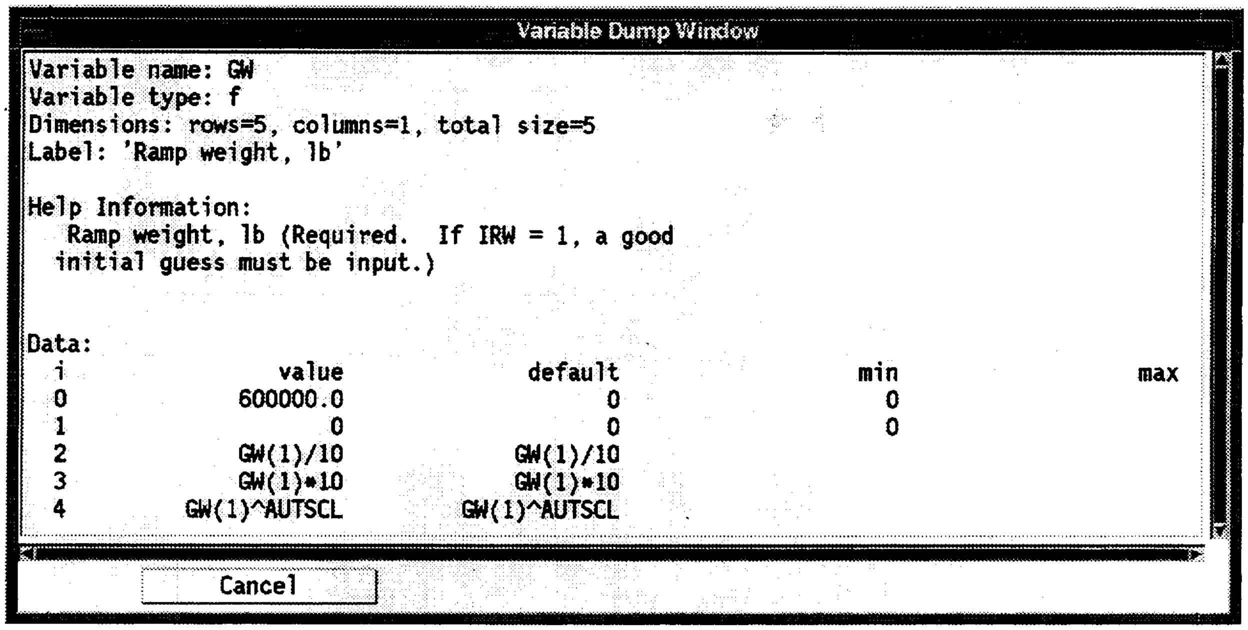
Input ANGLE for compressor or AREA for turbines.

ANGLE [illegible] 3rd dimensional argument on map
AREA 3rd dimensional argument on map

The following are graphics options.

PSIZE make 1/4 size plot
GRID [✓ draw grid]
GRAY do gray scale plot
ICOLOR [illegible] contour color option (see help)
IDUAL [illegible] plot R values on compressors maps

Figure 2: mapplot user interface created using NLEdit.



```
Variable name: GW
Variable type: f
Dimensions: rows=5, columns=1, total size=5
Label: 'Ramp weight, lb'

Help Information:
   Ramp weight, lb (Required.  If IRW = 1, a good
  initial guess must be input.)


Data:
  i              value            default              min               max
  0           600000.0                  0                0
  1                  0                  0                0
  2           GW(1)/10           GW(1)/10
  3           GW(1)*10           GW(1)*10
  4       GW(1)^AUTSCL       GW(1)^AUTSCL
```

Figure 3: Variable dump window.

2.4 Popup Menus

Many additional capabilities are provided for text field inputs. Most are accessed via a popup menu, which displays when the right mouse button is pressed while the mouse pointer is in a text field. Some of the following options may be available from the popup menu depending on the data type of the variable:

Help — Pops up a help window.

Edit — Pops up a larger editing window.

Array Edit — Pops up an array editor.

File Selection Box — Pops up a file selection box.

Edit Contents — Pops up a file editor.

Show Calc — Displays a numerical value.

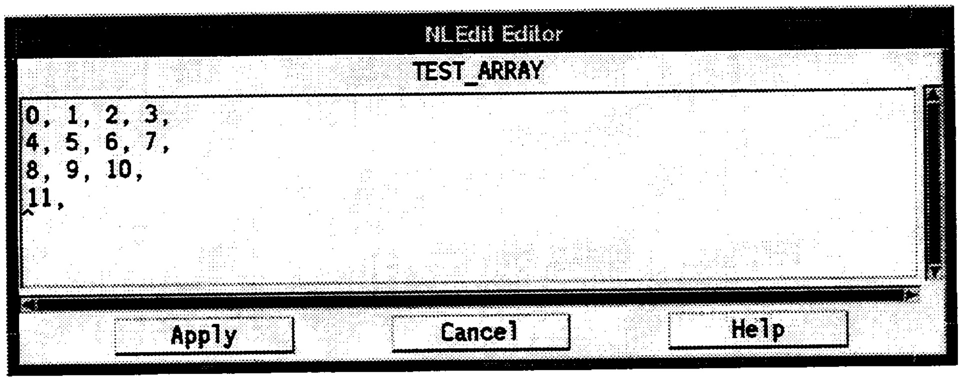
Show Default — Displays the default value.

Show Min — Displays the minimum allowable value.

Show Max — Displays the maximum allowable value.

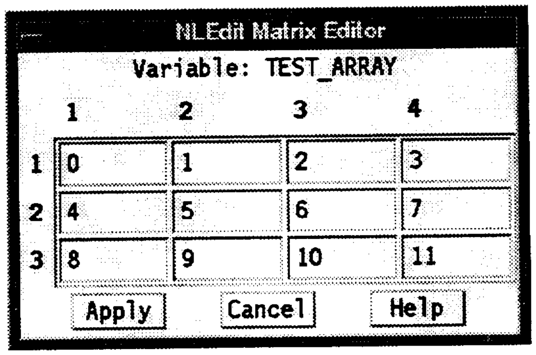
Refresh — Displays the actual value.

The **Edit** option opens a larger editing window (fig. 4.) to ease the entering of long strings of data. The **Array Edit** option opens a spreadsheet-like window containing a matrix of text fields. Each text field holds the value of one array element. This allows the user to easily find an array element without having to count commas between numbers. Figure 5



```
0, 1, 2, 3,
4, 5, 6, 7,
8, 9, 10,
11,
```

Figure 4: Editing window.



	1	2	3	4
1	0	1	2	3
2	4	5	6	7
3	8	9	10	11

Figure 5: Matrix editor.

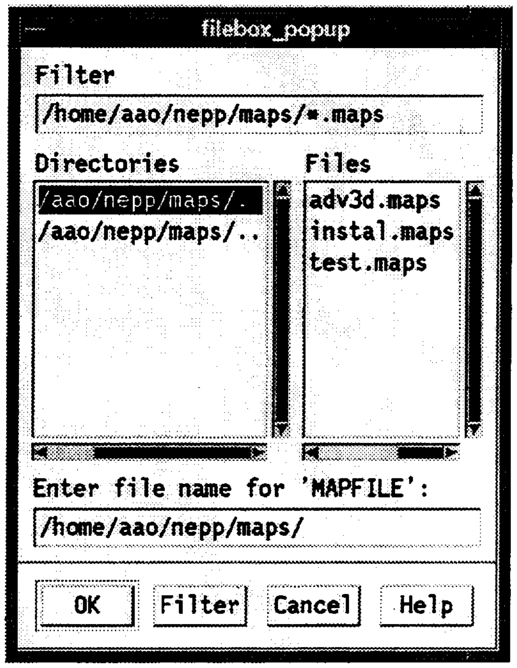
shows the array editor for a two-dimensional array. Arrays with more than two dimensions are not supported. The options **File Selection Box** and **Edit Contents** apply to variables that hold file names. The **File Selection Box** option opens a window containing a list of file names to help in choosing a file name for the input field (see fig. 6). The **Edit Contents** option opens a text editor displaying the contents of the file named in the input field (see fig. 7). This editor is a full function editor with Cut, Paste, and Search functions.

Another feature of the text input fields is the calculator. Equations may be typed into the input fields. The equations are converted to numerical values using a built-in scientific calculator program before the namelist output file is written. The result can be computed beforehand using the **Show Calc** option. The calculator (or `calc`) module is described in detail in section 4.1.

The **Show**... options all behave similarly. If, for example, **Show Default** is selected, the default value is displayed in the text field. The actual value of the variable is not changed to the default value unless the **Enter** key or the **Apply** button is pressed. This allows the users to review the default value or to actually change the value to the default value. The other options are all self-explanatory.

2.5 Error Checking

The user interface also provides built-in error checking. Errors are caused by a type mismatch, a value outside the range, or an equation that `calc` cannot evaluate (due to an undefined variable, for example). Normally, if an error occurs, a warning is displayed in the



/home/aao/nepp/maps/*.maps

/aao/nepp/maps/.
/aao/nepp/maps/..

adv3d.maps
instal.maps
test.maps

/home/aao/nepp/maps/

Figure 6: File selection box.

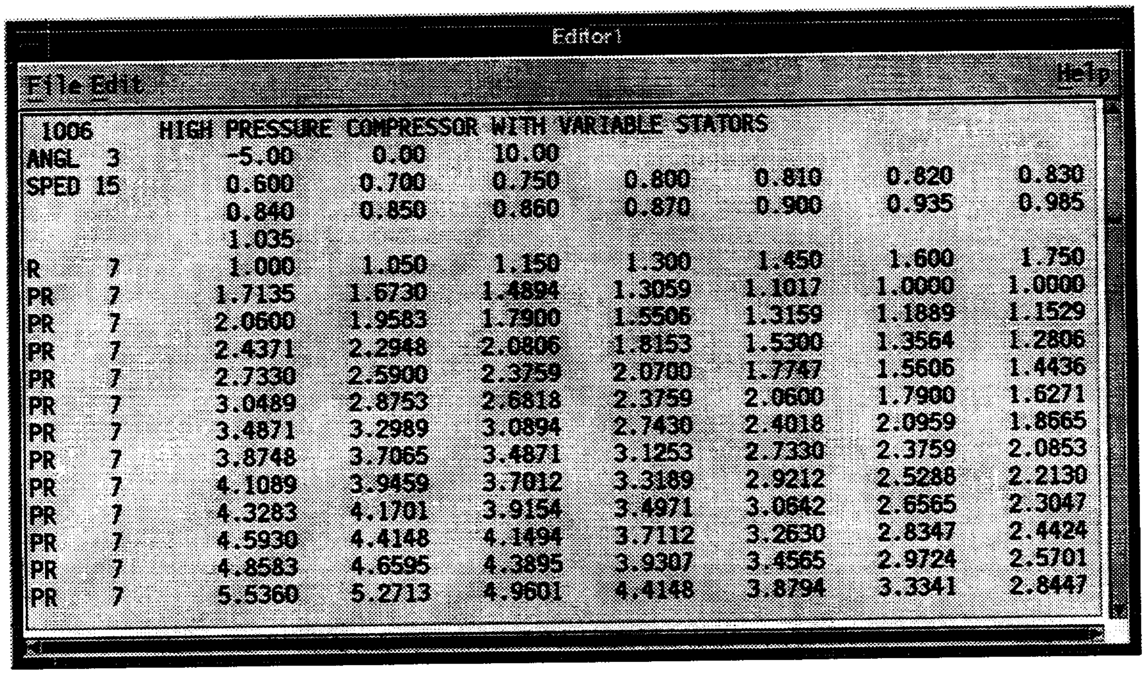
message window, and the text field is displayed in reverse video. (It is not possible to get an error from a toggle or multiple choice field.) The reverse video is hard to overlook. The user can ignore the message and proceed, or fix the error. An option is available to turn error checking off. This option should be used with care. Inputs are checked for errors when the contents of the text field are entered into the data dictionary, that is, when the **Enter** key or the **Apply** button is pressed.

2.6 Saving Changes

The three buttons on the bottom of the namelist editing window, **Apply**, **Cancel**, and **Help**, are for saving changes in the text input fields (changes to toggle fields and multiple choice fields are automatic), closing the form (without saving changes), and invoking help on using the form, respectively. The **Apply** button saves changes in the text input fields to the data dictionary; it does not create a new namelist output file. In order to update the namelist output file, you must select the **Save** or **Execute** option from the **File** menu.

2.7 Multiple Namelists

Some namelists may occur in a file more then once. The text field just above the **Apply** button in figure 2 is for changing the occurrence number of the namelist. For example, if the occurrence number is 2, then the values shown in the fields will be printed the second time the namelist occurs in the output file. Buttons are provided just after the occurrence number text field to allow the user to insert a new occurrence of the namelist, copy the current values of the namelist to a new occurrence, delete an occurrence of the namelist, and reset all values in the current occurrence of the namelist to their default values. How the namelists are ordered in the output file is explained in section 3.3.



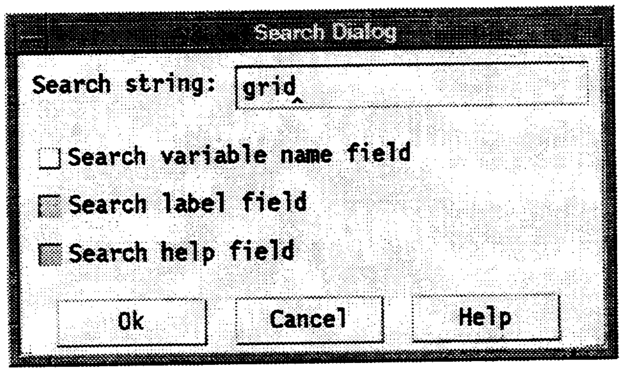
```
 1006     HIGH PRESSURE COMPRESSOR WITH VARIABLE STATORS
ANGL  3       -5.00      0.00     10.00
SPED 15       0.600     0.700     0.750     0.800     0.810     0.820     0.830
              0.840     0.850     0.860     0.870     0.900     0.935     0.985
              1.035
R     7       1.000     1.050     1.150     1.300     1.450     1.600     1.750
PR    7      1.7135    1.6730    1.4894    1.3059    1.1017    1.0000    1.0000
PR    7      2.0600    1.9583    1.7900    1.5506    1.3159    1.1889    1.1529
PR    7      2.4371    2.2948    2.0806    1.8153    1.5300    1.3564    1.2806
PR    7      2.7330    2.5900    2.3759    2.0700    1.7747    1.5606    1.4436
PR    7      3.0489    2.8753    2.6818    2.3759    2.0600    1.7900    1.6271
PR    7      3.4871    3.2989    3.0894    2.7430    2.4018    2.0959    1.8665
PR    7      3.8748    3.7065    3.4871    3.1253    2.7330    2.3759    2.0853
PR    7      4.1089    3.9459    3.7012    3.3189    2.9212    2.5288    2.2130
PR    7      4.3283    4.1701    3.9154    3.4971    3.0642    2.6565    2.3047
PR    7      4.5930    4.4148    4.1494    3.7112    3.2630    2.8347    2.4424
PR    7      4.8583    4.6595    4.3895    3.9307    3.4565    2.9724    2.5701
PR    7      5.5360    5.2713    4.9601    4.4148    3.8794    3.3341    2.8447
```

Figure 7: File editor.

2.8 Finding Variables

Some programs will have many namelists with a large number of parameters in each namelist. NLEdit has some options to help the user cope with these many parameters.

The **Show Only Used Variables** option causes the namelist editing window to display only the input fields that are currently being used (i.e., the value of the variable was found in the input file, or the value has been set in the input window). This greatly reduces the amount of extraneous material the engineer has to view.



Search string: grid

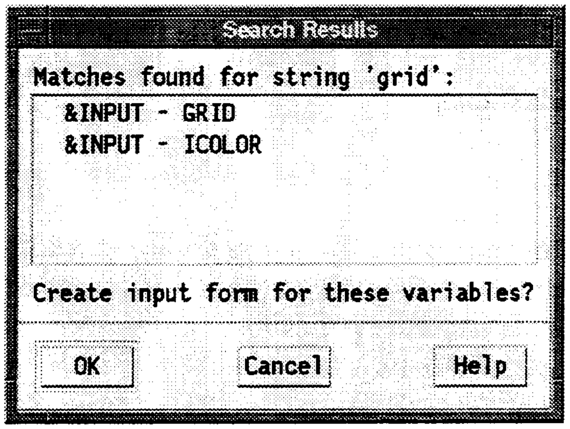
☐ Search variable name field

▨ Search label field

▨ Search help field

Figure 8: Search window.

The data dictionary search function allows searches on variable names, label strings, and help strings. The search window is shown in figure 8. The user inputs the search string into the top of this form and selects the fields to search. When the **Ok** button is pressed, a window like the one shown in figure 9 appears. This window lists all of the variables that match the search string and their corresponding namelists. The user may either cancel the search and edit the variables from the namelist windows or generate a new form containing just the



Matches found for string 'grid':

&INPUT - GRID
&INPUT - ICOLOR

Create input form for these variables?

Figure 9: Search results.

variables found in the search. The new form is used exactly like the namelist editing window, but it may contain variables from many different namelists. If a parameter is changed in the new window, it is automatically changed in the namelist window and vice versa.

3 Using NLEdit

3.1 Data Definition File Format

The only programming necessary to create a user interface for a Fortran program using namelist input is to type in a data definition file. The parsing routine uses a simple format. All data types use the same format, although some of the fields may have a slightly different meaning based on variable type. The format of each data item is as follows:

```
name  type  m n  default   min   max
 one line label string.
 all the help information
 you want to type
 on as many lines as you like.
.
```

The item definition starts with the variable name followed by its type, dimensions, default value, minimum value, and maximum value. The variable name is a character string of arbitrary length. The name may contain any printable characters other than the space character and parentheses. The name may not start with a **$**, **&**, or period.

The type field can be one of the following (see section 3.2 for details):

i — integer

f — floating point

d — double-precision floating point

s — character string

F — Filename

C — Comment

B — Boolean (true or false)

M — Multiple choice

Since all data are stored in the form of strings and all computations are done using double-precision, floating-point numbers, the type of the information is only important for selecting appropriate data entry fields and output formats.

The variable can be defined as an array (or matrix) by entering the dimensions `m` and `n`. If the variable holds only a single value, enter 1 for `m` and `n`. Arrays with more than two dimensions are not explicitly supported but may be used by mapping to a one- or two-dimensional representation. For example, a 2x3x3 array could be entered as a 2x9 array. All variables are treated as one-dimensional arrays internally.

The `default`, `min`, and `max` fields can contain as many as `m*n` values. Each value is followed by a comma; no spaces are permitted unless contained between single or double quotes. These three fields may be mathematical equations containing other variable names. If this first line becomes too long, it may be split into multiple lines by entering the `\` character at the end of the line. The `!` or `?` character can be used as a place holder for an undefined field.

The second input line is the label string placed after the input field for the variable. After this line are as many lines of help information as you wish to insert. Help information is displayed on screen in the same format as entered here. Note that the label string and help information are indented in order to make the item definition more readable. This indentation is not required but is recommended. The help information and an item definition are ended by a line with a period in column one. The next variable follows in the same format. There is no limit to the number of variables. Variables appear in the namelist editing window in the same order that they occur in the data dictionary.

3.2 Data Types

There are four standard data types (**i**, **f**, **d**, and **s**). These data types all use text field input format (see section 2.3). The `min` and `max` fields are used for bounds checking on the numeric types (**i**, **f**, and **d**). The `min` and `max` fields have no significance for the string type (**s**); however, a programmer may take advantage of these undefined inputs for special purposes.

There are also four special data types (**F**, **C**, **B**, and **M**). Variables of these types cannot be arrays. The file type **F** specifies a character string that holds a name of a file. It uses the text field input style but has some special menu picks for using a file selection box and editing the file contents. The `min` field, if defined, holds the starting directory to be displayed in the file selection box, and the `max` field, if defined, holds the file filter for the file selection box.

Comment lines, in the namelist editing window, can be added by inserting additional variables in the namelist group. These variables are flagged as comments by entering **C** as the variable type. The help information from these variables is used as the text. The label string line is ignored but must be present.

Toggle-button input fields are used for variables of type **B**. This type is usually used for the Fortran logical type `.TRUE.` or `.FALSE.` (It is recommended that `T` and `F` be used in place of `.TRUE.` and `.FALSE.`.), but type `B` can be used any time there is a choice between two values for a variable. If the toggle button is depressed, then the value of the variable is set to `max`; otherwise, it is set to `min`. And its initial value is set to `default`, which must be identically equal to `min` or `max`.

The last available data type is **M** or multiple choice. It is used for variables that can have one of several values. The variable must be dimensioned to the number of choices possible. The `min` field holds the valid choices, and the `max` field holds label strings for each choice. If these labels contain spaces, the labels should be placed in quotes. A value identically equal to one of the values in the `min` field must be placed in the `default` field. The multiple choice field works as follows: if, for example, the third button is pressed by the user, the value of the variable will be set to the third value in the `min` array.

3.3 Ordering Namelists in Output File

When the `&` or `$` character appears in column one of the data definition file, the parser routine is flagged to start a new namelist. The namelist name follows the `&` or `$`. All items following the namelist name are considered part of that namelist.

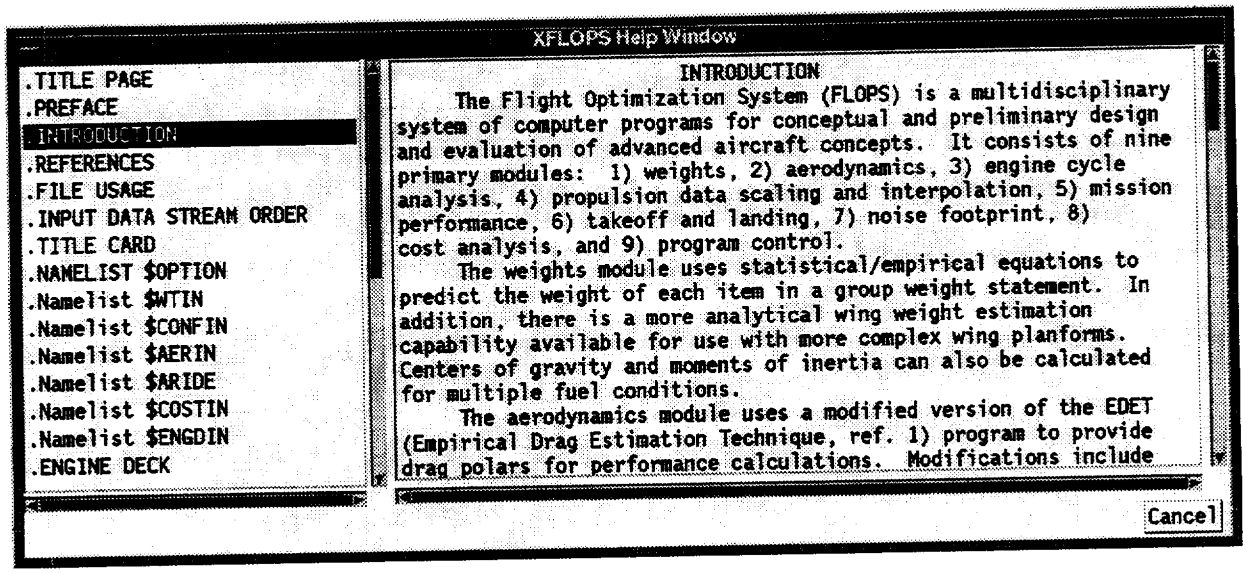
There are two ways to handle the printing of multiple occurrences of the same namelist. If the letter `s` appears after the namelist name in the data definition file, all occurrences of this namelist will be printed to the output file sequentially. If the letter `c` appears after the namelist name, all occurrences of this namelist will be collated in the output file with all other namelists marked as type `c`. If there is no indicator after the namelist name in the data definition file, the namelist can occur at most once in the output file. Namelists are printed to the output file in the order they appear in the data definition file and not necessarily in the order they appear in the namelist input file.

3.4 Documentation Files

This report and other documentation are available on-line when running NLEdit. The on-line documentation is read from a *documentation file*. The user running NLEdit as a stand-alone program has the option of including one additional on-line document. The programmer using NLEdit routines as part of a larger program may wish to include several additional on-line documents (see section 4.3). A documentation file contains ASCII text, which should be formatted to approximately 80 columns. Sections are divided by placing a period in column one on the line containing the section title. Therefore, the only required formatting for a documentation file is inserting the periods at section breaks. Figure 10 shows an on-line documentation window generated by NLEdit. On the left is the table of contents, and on the right is the text from the documentation file.

3.5 A Sample Data Definition File

A sample data definition file for a program with one namelist input is shown in figure 11. The first two variables, `application_help` and `program_name`, must be supplied in every data definition file. General help information about the application is given in



```
                    INTRODUCTION
     The Flight Optimization System (FLOPS) is a multidisciplinary
system of computer programs for conceptual and preliminary design
and evaluation of advanced aircraft concepts.  It consists of nine
primary modules:  1) weights, 2) aerodynamics, 3) engine cycle
analysis, 4) propulsion data scaling and interpolation, 5) mission
performance, 6) takeoff and landing, 7) noise footprint, 8)
cost analysis, and 9) program control.
     The weights module uses statistical/empirical equations to
predict the weight of each item in a group weight statement.  In
addition, there is a more analytical wing weight estimation
capability available for use with more complex wing planforms.
Centers of gravity and moments of inertia can also be calculated
for multiple fuel conditions.
     The aerodynamics module uses a modified version of the EDET
(Empirical Drag Estimation Technique, ref. 1) program to provide
drag polars for performance calculations.  Modifications include
```

Figure 10: Documentation window.

application_help. The file name of an on-line manual is set in the default field. Menu items are added for accessing this information. The item program_name is used to create a menu pick to execute a program which is set in the default field. (The example uses /bin/myprog.) The min field is used to hold any additional parameters that need to be specified when invoking the program. The label lines from these variables are displayed on the menu picks. Additional items, for internal use by NLEdit, are read from a separate data definition file.

The three variables OutputFile, XX, and YY are part of namelist &INPUT. OutputFile is an example of a file name input; XX is an array of three integers; and YY is an array of three floating-point numbers. The default and maximum values for YY are functions of the values input for XX. The maximum for YY(3) is undefined and, therefore, not limited. Note that the ! character is used as a placeholder for an undefined field but it is not needed when leaving the trailing portion of an array undefined.

The variable input_msg1 is a message line. The help field from this variable will be displayed in the namelist editing window for namelist &INPUT. The line Sample comment... shows how nondisplayed comments may be entered into the data definition file. Comments may appear between any variables in the file; they may not start in column one and are completely ignored by the NLEdit program. The appendix has a more complete example of a data definition file.

3.6 Executing NLEdit

After completing the data definition file, the user interface is invoked by entering the command:

```
nledit <filename> <dictionary>
```

where <filename> is the file name containing the namelist inputs and <dictionary> is the file containing the data dictionary.

```
application_help s  1 1  /usr/local/myprog/myprog.doc ! !
 Myprog Help...
 Help information for program myprog.
 more about myprog.
.
program_name s           1 1  /bin/myprog '-b -x' !
 Run Myprog
 !
.
 Sample comment: Namelist &INPUT starts here...
&INPUT
input_msg1 C        1 1  ! ! !
 !
 Namelist &INPUT for myprog.
.
OutputFile F        1 1  "myprog.output" /usr/local/myprog/data *.output
 Output file name
 Enter the file name for output data.
.
XX i                3 1       1,2,3    0,0,0    !
 XX array
 Help info for xx array.
.
YY f                3 1       XX(1),2.55*XX(2),4.5*XX(3)-2 \
0,0,0   6*XX(3),6*XX(3)
 YY array
 Help info for yy array.
.
```

Figure 11: Sample data definition file.

It is usually convenient to write a script for executing a program with or without NLEdit. For example, a Unix C-shell script for the mapplot example might look like this:

```
# Shell script for mapplot
if ($1 == 'i' && $2 != '') then
  nledit $2 /usr/local/mapplot/mapplot.hlp
else if ($1 == 'b' && $2 != '') then
  mapplot $2
else
  echo 'SYNTAX: mapplot i|b <filename>'
  echo 'Options:'
  echo ' i => interactive, b => batch'
endif
```

Alternatively, the script could have been written with a default for the option if it is not specified.

4 Program Architecture

NLEdit is composed of three main program parts: the calculator, the data dictionary, and the graphics modules. This section describes each module in enough detail to allow a programmer to appreciate how NLEdit works. With a basic understanding of how the program is set up and the comments provided in the source code, it is relatively easy to use NLEdit routines to develop larger graphical user interfaces.

4.1 The Calculator

The basic function of the calculator module, or `calc`, is to evaluate character strings and return the appropriate values. This entails parsing the input string (based on mathematical associativities and precedences), looking up the value of any constant or user-defined variable, and doing any necessary computations. The calculator supports predefined constants, logarithmic functions, trigonometric functions, and scientific notation. The syntax for these functions and other up-to-date information on using `calc` is given in the on-line help.

Two Unix utilities, LEX and YACC, were used to implement the calculator program [1]. LEX is a lexical analyzer program generator that can be used for simple lexical analysis of text. The user provides a set of regular expressions and actions to be executed, and LEX generates a C program. YACC is a parsing program generator. The user supplies a context-free grammar, which YACC converts into a set of tables used by an LR(1) parsing algorithm [2]. In addition, the user may specify precedences and associativities to remove any ambiguities inherent in the original grammar. YACC generates a C file, which may be incorporated into a larger program. The utilities LEX and YACC greatly simplify the programming of `calc` and make it easy to add new functionality.

The calculator program receives a string as its input. It passes this string to the lexical analyzer module, which converts it into tokens. The token stream is then passed to the parsing modules, which build the appropriate parse tree. The parse tree is stored as a linked list (with the elements of the tree in prefix order). Each tree is then evaluated (by a third module) using a stack.

Along with the traditional functions associated with a calculator, there is support for creating variables and binding them to values. This is implemented by using the data dictionary as a variable table. When a new variable is defined by the user, its name and corresponding value are added to the data dictionary. Once a variable has been added to the dictionary, it may also be redefined (i.e., have its value changed). Variable names must be unique, and, in this respect, all variables are, in effect, global in their scope.

Since the data dictionary stores character strings and not numerical values, equations will be automatically reevaluated each time the value of a variable is required. If, for example, `x=y+1`, the string "`y+1`" is stored for `x`. If the value of `y` changes, the string "`y+1`" will evaluate to a different value the next time the program requests the numerical value for `x`. Currently, processing speed is not a problem. However, if speed becomes a problem, the data dictionary could be modified to store the parse tree, eliminating the need to parse the equation each time it is evaluated.

4.2 The Data Dictionary

The data dictionary is used to simplify programming by eliminating the need for global data and by hiding the structure of data storage from the rest of the program. Each item in the data dictionary has a unique identifier. This identifier (or key) is called a *variable name* because it usually refers to one of the namelist variables. The variable name is just a character string of arbitrary length. Note that since each item must have a unique variable name, the same variable name cannot be used in two different namelists.

Information from the data dictionary is accessed by calling the appropriate function with the variable name as an argument. For example, `DbGetHelp(''input_file'')` returns a string containing help information about the variable `input_file`. This type of data handling greatly simplifies programming because the programmer does not need to know how the data are stored, how to search the data for a member, nor how to copy the desired information. Additionally, data integrity is maintained because all functions return a copy (usually a pointer to a copy) of the data and not a pointer to the actual data. This is much like data hiding in an object-oriented language.

All data in the data dictionary are stored in the form of strings (i.e., character arrays). This approach was chosen for three reasons: First, the data are read from files, edited, and written back into files all in the form of characters, so conversion to numerical types may not even be necessary. Second, all types of data can be represented in a character string. Finally, strings can hold equations in addition to basic types, such as numbers and characters (An equation is really just a combination of numbers, symbols, and letters). When a numerical value is needed, `calc` is used to make the conversion.

There are over 50 functions for manipulating information in the data dictionary; however, the programmer needs only to understand a couple of these to effectively use the data dictionary. Although it is tempting to do, it is not recommended to programmatically add variables to the data dictionary. Instead, the programmer should add new variables to the data definition file. This leaves basically two functions the programmer needs to know how to perform: getting information about a variable and setting the value of the variable.

The function `DbGetValue(name)` returns the value of the variable `name` as a character string (char *). The function `DbCalcValue(name)` returns the double-precision, floating-point value of `name`. If it is necessary to refer to a specific field of an array, an array subscript can be used. For example, the name `"X(2)"` refers to the second element of the array `X`. This form of indexing was selected because it is (unfortunately) the form used in Fortran namelist.

Arrays of data can be retrieved with the functions `DbGetValues` and `DbCalcValues`. `DbGetValues(name)` returns an array of character strings (char **) for the variable `name`. `DbCalcValues(name)` returns an array of numeric values. Since the conversion from double to other types (int, float) is not as straightforward for arrays, `DbCalcValues` automatically makes the conversion to the correct type. In both cases, if `name` is indexed, the array returned starts at the index; otherwise, it starts with the first value. There is only one function to set the value of a variable, it is `DbSetValue(name,data)`, where `name` is the variable name and `data` is a character string. If the variable is an array, each element (except the last one) is followed by a comma. The program parses the fields of `data` into an array of strings suitable for evaluation in `calc`.

In addition to the value of a variable, its minimum, maximum, and default values are stored in the data dictionary as, of course, strings. Arrays of these data are handled the same as arrays of values. Functions are provided to return strings or numerical values for all of these data. These functions are the same as those described above but operate on different data. The names of these functions can be found in the source code header file.

There is some additional information that the data dictionary has to keep track of, such as type and dimensions of the variable, a one-line description of the variable, and detailed help information on the variable. These data cannot be arrays. Sufficient documentation for functions that operate on these data can be found in the source code.

A generic search function is provided so the programmer can define new search functions without having to understand the storage structure. The format of the function is

```
char ** DbSearch(int (*funct)(), void *data)
```

`DbSearch` loops through all the variables in the data dictionary and returns a list of variable names that meet the criterion specified in the user-defined function `funct`. `Funct` receives a variable name and `data` as its arguments and returns 1 (true) or 0 (false), depending on whether the variable meets the search criterion.

So far, the data dictionary described here is a general-purpose piece of code that can be used for a number of applications. The code is easily modified to store other types of data without adversely affecting the current use.

4.3 The User Interface

The graphical user interface module was written such that many of the pieces could be reused to make user interfaces for more complex programs. Some users may want to make simple modifications, such as adding menu picks for plotting results. Other users may want to develop much more elaborate interfaces but still use some of the functionality of NLEdit. This section explains how this graphical user interface code can be reused.

The main window in NLEdit makes a good starting point for most applications. The function `make_main_window()` can be easily modified to add new menu picks to the interface. Before trying to add new items to the menu, it is helpful to understand how the Motif code interacts with the data dictionary. (For more information on using OSF/Motif see [3] and [4].) Let's consider adding a help button to the user interface. First a couple of definitions are necessary. A callback is a function that is invoked when a specific event (such as a mouse click) occurs in the user interface. Client data is a predefined piece of information that is passed to a callback when it is invoked. It is recommended that a variable in the data dictionary be used to store the help information. The callback for the help button is defined with the variable name (actually a pointer to a character array holding the variable name) as its client data. The help callback is then a simple function to write. It needs only to call `DbGetHelp(client_data)` and to "pop up" a help window with the resulting string. The help window could be created beforehand and have its widget identity (ID) retrieved by calling `DbGetWiget(''help_window'')`. The same help callback can be used in many places by changing the variable name in its client data. Another advantage of using the data dictionary is that the help information is easily changed just by editing the data definition file. Of course, there is already a help callback that works like this in NLEdit; it is called `HelpCB()`.

There are several other callbacks in NLEdit that can help extend the functionality of the interface. The callback `PostHelpCB()` can be used to add documentation to the interface as described in section 3.4. The name of the file containing the documentation is passed to the callback as client data; everything else is automatic. The callback `popup_editor1CB()` is used to invoke the editor shown in figure 7. If a variable of type **F** (filename) is passed as client data, the content of the file is displayed in the editor; otherwise, the current value of the variable is displayed in the editor. A filename can be read into a variable name by calling `popup_fileSelectionBox1CB()`. The callback `fileSelectionBox1OkCB2()` is used to add more functionally to the file selection box. The callback `popup_DbQuestionCB()` can be used

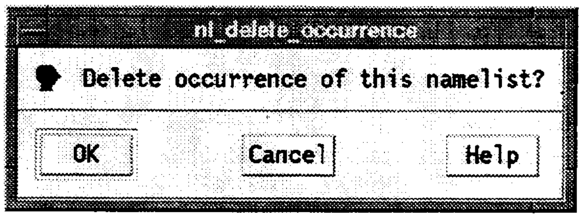


Figure 12: Question Dialog.

to pose a yes or no question to the user. An example of a question dialog box is shown in figure 12. The `label` field of a variable in the data dictionary is used to hold the question. The value of the variable is set to 1 if **Ok** is pressed and 0 if **Cancel** is pressed. The function `DbAddCallback()` can be used to add a side effect when the question is answered. That is to say, a function to be called when the value of the variable is changed can be specified using `DbAddCallback()`.

Many computer programs use formatted input data or a mixture of formatted and namelist inputs. In this case it is necessary to write a new read routine for NLEdit. However, the namelist editing window may still be used to edit the formatted data. To do this, one must add variables to the data dictionary for the formatted data the same way as for the namelist data. A dummy namelist name can be used to group variables together. The function `DbSetValue()` is called from the formatted read routine to store the data. The callback `popup_namelistCB()` is used to open a namelist editing window for a namelist name specified as its client data. The function `popup_names()` can be used to edit an arbitrary group of variables using a namelist editing form. Alternatively, the data editor shown in figure 4 or the matrix editor shown in figure 5 may be used (usually without modification) for editing formatted data.

In some cases it may be desirable to make special windows in the user interface. A convenient way to build these windows is to use the same building blocks used in making the namelist editing windows. The code for fully functional radio boxes, toggle buttons, text fields, labels, and labeled text fields can be found in the file `fields.c`. All of these functions work with variables in the data dictionary, so it is necessary to add a variable to the data definition file for each field used.

An objective of this section was to provide just enough detail to allow one to start programming with NLEdit. Additional information can be found in the source code. Several large graphical user interfaces have been successfully developed based on NLEdit. Working with the data dictionary and the higher level graphics routines in NLEdit greatly simplified the programming of these interfaces.

5 Concluding Remarks

A fast, effective way to produce graphical user interfaces for Fortran programs was developed. This was accomplished by creating a generic graphical user interface that is customized through a data definition file. The programming effort to produce an interface by this method is minimal. In many cases, no actual code modifications are necessary to either the GUI or the analysis code.

The resulting interface provides a better user environment than the traditional approach of using namelist input. In addition to providing extensive on-line help, it provides considerable error checking and increased flexibility in input fields by means of the calculator module.

The program was written in the C programming language using the OSF/Motif widget set. This program was written using small, comprehensible, well-structured functions, greatly improving its maintainability. Many of these functions could be (and have been) reused in other programs.

One major limitation of this program, imposed by the data dictionary, is that every item must have an unique identifier. This implies that the same variable name cannot occur in two different namelist inputs. Plans to improve the program include removing this restriction via reprogramming the data dictionary as a C++ class.

Although some extensions will be added when an application demands them, the basic premise of NLEdit will remain that it be the fastest, easiest way to develop a graphical user interface for Fortran programs and that it not become another difficult-to-use user interface programming language.

Acknowledgements

I would like to thank Kathleen Ryall for her work on the calculator module. I would also like to thank Thomas Lavelle and Louis Handler of Lewis for their help in testing this program and for their suggested improvements.

References

[1] Kernigham, B.W.; and Pike, R.: *The UNIX Programming Environment.* Prentice-Hall, 1984.

[2] Aho, A.V.; Sethi, R.; and Ullman, J.D.: *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] Young, D.A.: *The X Windows Programming and Applications with Xt — OSF/Motif Edition.* Prentice-Hall, 1990.

[4] Open Software Foundation: *OSF/Motif Programmer's Reference.* Prentice-Hall, 1991.

Appendix — Sample Data Dictionary

The following is an example of a data dictionary used to create a graphical user interface with the NLEdit program. The resulting interface is shown in figure 2.

```
 data dictionary for mapplot
 programmed by: brian curlett    12/5/92

application_help s 1 1    /usr/local/mapplot/mapplot.doc ! !
 Mapplot
 The database for the mapplot code is now loaded.
 Mapplot plots NEPP formatted compressor and turbine maps
 using the Grafic plot library. See "Mapplot Manual..." for
 more info on Grafic.
.
program_name s 1 1  /usr/local/bin/mapplot ! !
 Run Mapplot
 !
.
&INPUT s
input_msg1 C 1 1     ! ! !
 msg
 mapplot - NEPP compressor and turbine performance map plotting
program
.
MAPFILE F 1 1       "test.maps"  /home/aao/nepp/maps  *.maps
 file name for tabulated map data
 enter the file name for tabulated map data
.
IOPT M 3 1          1,2,3   "compressor map",\
"turbine map - separate plots","turbine map - combined plots"    !
 map type switch
 Map Type Switch
 1 compressor map (default)
 2 turbine map separate plots
 3 turbine map combined plot
.
input_msg2 C 1 1    ! ! !
 msg
 The next three inputs define the map.
 Look in MAPFILE for the numbers.
.
MAPFLO i 1 1            1000 0 !
 flow map id number
 flow map id number
.
MAPEFF i 1 1            1001 0 !
```

```
 efficiency map id number
 efficiency map id number
.
MAPPR i 1 1         1002 0 !
 pressure ratio map id number
 pressure ratio map id number
 used only for compressor maps
.
input_msg3 C 1 1    ! ! !
 msg
 The next three inputs are optional scale factors.
.
FACFLO f 1 0  1.0 ! !
 scale factor on flow
 scale factor on flow (default=1.0)
.
FACEFF f 1 1        1.0 ! !
 scale factor on efficiency
 scale factor on efficiency (default=1.0)
.
FACPR f 1 1         1.0 ! !
 scale factor on pressure ratio
 scale factor on pressure ratio
 used only for compressor maps (default=1.0)
.
input_msg4 C 1 1    ! ! !
 msg
 Input ANGLE for compressor or AREA for turbines.
.
ANGLE f 1 1         0.0 ! !
 3rd dimensional argument on map
 3rd dimensional argument on compressor maps (same as AREA)
 (default=0.0)
.
AREA f 1 1          0.0 ! !
 3rd dimensional argument on map
 3rd dimensional argument on turbine maps (same as ANGLE)
 (default=0.0)
.
input_msg5 C 1 1    ! ! !
 msg
 The following are graphics options.
.
PSIZE B 1 1         0 0 1
 make 1/4 size plot
 plot size
 FALSE (0) for a full size plot
 TRUE (1) for a 1/4 size plot
.
GRID B 1 1          0 0 1
 draw grid
 grid switch
 TRUE (1) draw grid lines on plot
 FALSE (0) no grid lines are drawn
.
GRAY B 1 1          0 0 1
 do gray scale plot
 gray scale switch
 TRUE (1) draw plot using gray scales
 FALSE (0) draw plot in color (or monochrome)
.
IRVAL B 1 1         1 0 1
 plot R values on compressors maps
 TRUE (1) plot R values on compressors maps
 FALSE (0) do not plot R values on compressors maps
 (default=TRUE)
.
FONT s 1 1          ! ! !
 screen font
 Screen font
 Examples: "fixed"   "Rom14.500"   "*clean-bold-r*160*"

 Note: this font is not used for postscript output.
.
input_msg6 C 1 1    ! ! !
 msg
 The following inputs are for drawing an operating
 line on the plot. All array must be the same size.
.
NDAT i 1 1          0 0 100
 size of arrays FLOWDAT, PRDAT and EFFDAT
 size of arrays FLOWDAT, PRDAT and EFFDAT
.
FLOWDAT f 100 1     ! ! !
 array of corrected flow data
 array of corrected flow data
 max size 500
.
EFFDAT f 100 1           ! ! !
 array of efficiency data
 array of efficiency data
 (not used for compressor maps)
 max size 500
.
PRDAT f 100 1        ! ! !
 array of pressure ratio data
 array of pressure ratio data
 max size 500
.
DATSYM i 1 1         1 0 15
 plot symbol (see help)
 plot symbol:
 0 none, 1 square, 2 circle , 3 triangle,
 4 +   , 5 X     , 6 diamond, 7 arrow   ,
 8 picnic table  , 9 Z      , 10 Y      ,
 11 square w/ X  , 12 *     , 13 hour-glass,
 14 vertical bar , 15 star

 This symbols are used only on the operating
 line.
 .
```

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (*Leave blank*)	**2. REPORT DATE** August 1994	**3. REPORT TYPE AND DATES COVERED** Technical Memorandum

4. TITLE AND SUBTITLE NLEdit—A Generic Graphical User Interface for Fortran Programs	**5. FUNDING NUMBERS** WU–505–69–50
6. AUTHOR(S) Brian P. Curlett	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration, Lewis Research Center, Cleveland, Ohio 44135–3191	**8. PERFORMING ORGANIZATION REPORT NUMBER** E–8049
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration, Washington, D.C. 20546–0001	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA TM–4543

11. SUPPLEMENTARY NOTES

Responsible person, Brian P. Curlett, organization code 2410, (216) 977–7041.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited, Subject Category 61	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 words)***

NLEdit is a generic graphical user interface for the preprocessing of Fortran namelist input files. The interface consists of a menu system, a message window, a help system, and data entry forms. A form is generated for each namelist. The form has an input field for each namelist variable along with a one-line description of that variable. Detailed help information, default values, and minimum and maximum allowable values can all be displayed via menu picks. Inputs are processed through a scientific calculator program that allows complex equations to be used instead of simple numeric inputs. A custom user interface is generated simply by entering information about the namelist input variables into an ASCII file. There is no need to learn a new graphics system or programming language. NLEdit can be used as a stand-alone program or as part of a larger graphical user interface. Although NLEdit is intended for files using namelist format, it can be easily modified to handle other file formats.

14. SUBJECT TERMS Interactive graphics; Fortran; Editing routines (computers); User manuals (computer programs)	**15. NUMBER OF PAGES** 24
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT**