IN-61
114113

NASA Technical Memorandum 105696

P.28

A Graphical User-Interface for Propulsion System Analysis

Brian P. Curlett
Lewis Research Center
Cleveland, Ohio

and

Kathleen Ryall
Harvard University
Cambridge, Massachusetts

August 1992

(NASA-TM-105696) A GRAPHICAL USER-INTERFACE FOR PROPULSION SYSTEM ANALYSIS (NASA) 28 p

N92-33894

Unclas

G3/61 0114113

NASA

A GRAPHICAL USER-INTERFACE FOR PROPULSION SYSTEM ANALYSIS

Brian P. Curlett and Kathleen Ryall

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Introduction

For more than thirty years computer aided engineering has been conducted using FORTRAN programming on mainframe computers. Methods of data entry into the engineering programs have progressed from the days of punch cards, but for the most part still requires entering large volumes of information into formatted input fields or namelist formats. In the last ten years, personal computers, and many business applications have progressed towards the graphical user-interface. Its ease of use and its ability to represent information pictorially have helped to improve productivity. Personal computers are still too small and too slow to adequately handle complex engineering problems. The workstation class of computers, however, has made tremendous gains in performance over the last couple of years. Today, workstation computers rival even the fastest mainframes and supercomputers. Furthermore, today's workstation provides an ideal environment for developing graphical user-interfaces.

The X Windows System has emerged as the standard for the development of graphical user-interfaces on the workstation class of computers. The X Windows System has a unique device-independent architecture that permits X applications to display windows on any hardware that supports the X protocol, from mainframes to PCs. This and X Windows' public domain source code has lead to its popularity. Although the X Windows System provides only low-level graphics calls, there are several higher level tool kits that make programming in the X Windows System much easier. One of these tool kits is Motif from the Open Software Foundation.

X Windows with OSF/Motif running on a UNIX based workstation computer was the environment chosen for the development of the NASA Propulsion Analysis System (NPAS). NPAS is a graphical user-interface built on top of the NASA Engine Performance Program (NEPP), the Weight Analysis of Turbine Engines (WATE) and the INSTAL computer codes[1-5]. The interface consists of a method of pictorially representing and editing the engine configuration, forms for entering numerical data, on-line help and documentation, post-processing of data, and a menu system to control execution.

This paper describes the development of the NPAS graphical user-interface. This is not a users manual for NPAS. This paper does contain a description of the user-interface, the approach used to create the interface, and the rationale for this approach. In addition, other information thought to be of use in the development of similar applications is included.

Propulsion System Analysis Software

NASA uses a series of computer codes for propulsion system performance and weight analysis. This set of programs is built around the NASA Engine Performance Program (NEPP). NEPP (a.k.a. NNEP89) was derived from the Navy/NASA Engine Program (NNEP)[1]. NEPP is a one-dimensional, steady state, thermodynamic engine performance prediction program capable of modelling almost any type turbine engine. This is accomplished by allowing the user to configure a propulsion system by combining specified types of components in almost any order. The following component types are available: inlet, duct, burner, water injector, gas generator, compressor, turbine, splitter, mixer, ejector, heat-exchanger, nozzle, shaft, load and propeller. Off-design performance is calculated by using component performance maps. The compressor and turbine performance maps are scaled to match the design point of the engine being modeled. A detailed description of this code is given in Reference 2.

Thermodynamic properties of the gas flow through a propulsion system model can be calculated using one of two methods. The default thermodynamic properties routine is preset for JP4 fuel. The second is a chemical equilibrium model that can predict thermodynamic properties when chemical dissociation occurs or when using alternate fuels. This enhancement to NEPP is discussed in Reference 3.

Engine weight and flowpath dimensions are calculated using the WATE code[4]. This program predicts component weights from the thermodynamic cycle analysis and user supplied design specifications. Weight estimates are based on semi-empirical data. This code functions as a sub-program to the cycle code by linking to its FORTRAN libraries.

Inlet and nozzle installation effects and weights are calculated using the INSTAL code[5]. This code also functions as a sub-program to the cycle analysis code. The INSTAL code uses sophisticated inlet and nozzle performance maps to estimate installation losses and their effect on overall engine performance.

Together these programs enable the engineer to calculate the installed performance, dimensions and weight of a great variety engine configurations. These codes are being continually enhanced to provide more accurate results for an even greater variety of propulsion systems. Unfortunately, this flexibility adds to the complexity of the programs and to the difficulty in operating them. This is the motivation for the development of the graphical user-interface.

Background

Many interfaces for computer aided engineering applications have been bulky and difficult to use. Some only pre- or post-process part of the information. In some cases a particular interface is good for an engineer who is learning the program, but with experience the engineer begins to find the same interface limiting and slow. It is often quicker for an experienced engineer to edit a namelist input file than to access information though a deeply

nested menu system.

A few pre/post-processing programs have been developed for the NEPP program. KONFIG and REKONFIG, written in 1981, are the first two such programs[6]. KONFIG prompts the engineer line-by-line for inputs. REKONFIG allows users to modify component specification in previously created data sets. REKONFIG, however, does not allow for configuration changes as the name suggests. Both programs are written in FORTRAN IV and are therefore portable to most computers. However, because of FORTRAN's limited input/output capacity these two programs are very difficult to use. Additionally, neither code supports entry for control or optimization settings. These codes are not of much help to experienced engineers.

Another much improved pre-processor is SNAP (Simplified NEPP Automated Preprocessor) written in 1989[7]. This program uses a series of menus and forms to input design point information for NEPP. The program was written using the Xedit macros and Rexx (Restructured Extended Executor language) on a mainframe computer running VM/CMS. Although this system is helpful to new users, it is limited and includes only design point inputs for NEPP. It has not been programmed to handle the inputs required for weight analysis or installation effects. Although Rexx and Xedit are available on other computer systems, SNAP is not portable because it makes extensive use of system calls.

There are also several post-processing programs used in conjunction with NEPP. The FINDIT program provides a menu system for creating x-y plots of propulsion system output data. The NEPTOMIS program converts NEPP output into a form suitable for use with mission analysis codes. This program also produces a plot of throttle curves. MAPPLOT produces graphics of compressor and turbine maps[8]. Most of the capabilities of these programs are reproduced under the NPAS graphical user-interface.

The goal in writing the graphical user-interface was to develop a complete interface that would be capable of handling all input and output processing. An additional requirement was to present propulsion system information in a logical, easy to use manner. At the same time the interface had to be fast enough that an input change would be immediately seen in the output. This meant designing an interactive version of NEPP instead of just writing a pre/post-processor.

Having an interactive analysis code allows the engineer to explore "what if" scenarios with their propulsion system. Business applications have been using the "what if" approach to problem solving in spread sheet programs for many years. That is, a change in an input parameter will immediately cause a change in the output. This approach is very useful in the conceptual design process because of the large latitude the engineer has in varying the design parameters. Of course, numerical optimization may be used to help refine a small set of variables, but only after the engineer has developed an understanding of the critical operating characteristics of an engine. In older implementations of NEPP the iteration process for converging on a good design was lengthy and tiresome. The graphical user-interface presented here attempts to facilitate the trial-and-error approach to conceptual engine design as much as possible.

The underlying computer system has a strong influence on the form of input/output

for a program. Programs that are run in batch mode will use files for receiving input and generating output. Timesharing systems shifted the emphasis from creating programs designed to run in batch mode towards creating programs to run interactively. Workstations have strengthened this trend, and yet many programs still require strongly formatted input, usually in a form that makes the data difficult for a user to understand and interpret. When users make mistakes in these input files (which is very easy to do) they do not realize their error until the application tries to run and crashes due to the error. Even at this point it is often difficult to locate where the error was made. A graphical user-interface provides a convenient method for users to generate input to and view output from programs, and users will usually make fewer data entry mistakes. Error checking can be built into the interface to immediately notify users of incorrect data. In our interface, for example, the user is able to modify several input fields before hitting the "Apply" button to have the changes take effect. If one of these fields does not evaluate to a number, an error message is posted and the field is "highlighted" (it receives the focus) so that the user knows where something needs to be changed. In this manner, the user is immediately able to identify and correct input errors.

Text is not always the optimal media for communicating information to a user. Although today's technology does not yet support multi-media interfaces, applications in the 1990s are moving towards this goal. They will take advantage of audio and video stimuli in addition to current capabilities. Traditional character-based interfaces are by nature restricted to using text for their output. Today's hardware has the potential for more complex forms of output and displays. A schematic drawing of an engine configuration provides far more information to a user than a FORTRAN namelist containing a table of connections between components. The same holds true of output generated by a program. A graph conveys information to a user far more quickly and conveniently than a table of values. In general, visual representation of data is better provided by a graphical user-interface. This holds true even of pure "text" data in that colors and fonts can be used to increase its readability.

One of the advantages of a graphical user-interface is its user-friendliness. Most graphical interfaces today use visual cues to communicate with its users. Colors, fonts, and icons can all be used to prompt the user, to indicate options to the user. We have included a cursor that changes shape depending upon the mode and operation of the user. For example, when the user selects "Delete Component", the cursor becomes a skull and cross-bones; a hand-shaped cursor appears when the user is in edit mode and can drag objects around on the screen. We have also implemented push button and menu sensitivity in order to guide and prompt the user. When the user selects "Delete Component", for example, "Cancel" is the only button with its sensitivity on indicating that the user must actually delete a component or cancel the command. Button sensitivity is also used when a user is drawing connections between components. If an upstream component is selected first, for example, only the two downstream options remain as choices to finish a connection.

A graphical user-interface can provide on-line help and documentation for its users in a centralized location -- within the application itself! This makes it relatively easy for new users to just sit down and start using an application. When a question arises they need only click a button to receive more information. Menu-based help also provides information on a need to know basis; a user can locate information on a given subject without wading

through large amount of extraneous material. Through on-line help and documentation a graphical user-interface supports a friendly and convenient environment for both new and experienced users.

A graphical user-interface implemented in windows (such as ours) differs from traditional character-based interfaces in that it is an event-driven. In an event-driven interface, the user maintains control throughout the execution of the application, except in a few special cases (dialog boxes are one such example). The interface does some initialization, and then enters a loop, waiting for information from the user. Based on the action of the user, the interface will execute some routines, and then it will return to its loop to await further events. These events may result from user-action or from system side-effects. Furthermore, in a window system there are several different methods available to the user for sending input to the application. The input may be generated by keystrokes from the keyboard, from mouse motion, or by clicking any combination of buttons on the mouse. The user may also move from window to window, changing the keyboard focus from one application to another. Character-based systems, on the other hand, are very limited. Once the interface is started, it retains control throughout its execution. It has complete control over what input is permissible, and most of the time the input will come directly from the keyboard. A character-based interface restricts its users' actions, and enforces sequential execution of commands. Overall, an event-driven window-based user-interface provides the greatest flexibility for its user.

Graphics Software Selection

The X Windows System has emerged as a standard for programming graphical user-interfaces on UNIX workstations. There are several reasons for this. First, the X Windows System is available on most computers. Most computer manufacturers have ported the X Windows System to their hardware and distribute the binary code with their machines for little or no charge. If not, source code for the X Windows System is available in the public domain. Secondly, the X Windows System is network transparent; an application running on one machine can be displayed on a different machine, even in a heterogenous environment. Lastly, high-level tool kits that provide convenience functions are available with the X Windows System. These functions simplify many common graphic programming tasks, such as creating a text input field or a menu.

The X toolkit is composed of three levels. The lowest layer of software in X (the Xlib) provides the underlining communications and graphics protocols. Built on top of the Xlib is the Xt Intrinsics library. This layer of software provides a framework for creating user interface components. The third layer of software consists of a set of predefined user interface components. These components are referred to as widgets. This software level is not part of the X standard. There are several widget sets available. We chose to use the Motif widget set from the Open Software Foundation for our application because it is widely available and more robust than some of the public domain widget sets. Additionally, Motif is the native widget set on our hardware and, therefore, the look and feel of our user-interface will be consistent with the look and feel of commercial software on our system.

Another advantage to Motif is the ability for users to define much of the behavior of the user-interface. This is done through resource files. A resource file contains a list of properties associated with a widget or class of widgets. Properties include such things as location, size, color, and font. Even the functionality of a widget can be modified by the user through a resource file. For example, in our interface, a user can define which key is used for activating help. Being able to change the font or size of a window may not seem important, but this may be critical when considering the various types of display hardware. Someone running X on a personal computer with a VGA screen would not be able to use the same screen layout as someone working on a high resolution graphics workstation. In our interface we allow users to change those resources that are not critical to the program's execution. For more information on the X Windows System and Motif see References 9 through 12.

Hardware/Software Compatibility

The Aeropropulsion Analysis Office at NASA Lewis Research Center has been moving towards a distributed computing environment. Several types of workstation computers have been purchased. The primary development platform for NPAS is the IBM RS/6000. Currently, work is being done on an IBM RS/6000 Model 550. Smaller machines, for example the model 320, are more than adequate to run this code. To avoid excessive paging, 32 Mbytes of RAM are recommended. The NPAS user interface has also been ported to a Silicon Graphics Iris workstation.

NPAS should port to any UNIX machine running Motif that also has a C and FORTRAN compiler. The interface is programmed in the ANSI standard for C. There are a limited number of UNIX system calls in the user-interface. The Xlib, the Xt Intrinsics, and Motif graphic libraries are required. Some convenience functions available in Motif 1.1 are duplicated in our program to be compatible with Motif 1.0. Most of the analysis modules are written in FORTRAN and will port to other computer systems with little or no modification. Any device running an X server, such as an X terminal, workstation or PC, can be used for displaying the NPAS user-interface.

Interface/Analysis Coupling

Four types of interfacing between analysis code and the user-interface have been evaluated. These four methods are: file transfer, interprocess communication, database management, and direct coupling. The advantages and disadvantages of each of these methods are discussed briefly in this section.

One method for the interface to invoke the analysis code is to write an input file and make a system call to start the analysis as a separate program. Output can be read from another intermediate file back into the user-interface. If intermediate files are used, the analysis code may require little or no modification. This method, however, can be slow. Also, because it is difficult to read file formats designed for humans into the user-interface, input/output routines may have to be changed. By using file transfer, the analysis modules and the user-interface will execute as separate processes on the computer. In fact, the

separate processes could be executed on different computers. This is easily accomplished by writing files to a shared disk area and issuing a remote execution command. This may not be as elegant as some other forms of interprocess communication, but it requires little knowledge of system commands. The advantage of running the analysis as separate processes is that in the event that the analysis code crashes, the user-interface will continue to run. This is important because program crashes are still very common with many older, poorly written, but potentially useful engineering analysis codes. Communications to the plotting portion of our code uses intermediate files.

The second method is to have separate processes for each of the different modules, but to have these processes communicate by a more direct manners. There are several ways to do this; FIFOs, message queues, semaphores, shared memory, sockets, transport layer interface, or remote procedure calls may be used for interprocess communication[13,14]. The last three methods are intended to be used across a network, but may be used to execute code on the same machine. Any form of interprocess communication adds to the complexity of the programming. The selection of what type of interprocess communication to use depends on the programming requirements and portability. We currently do not use of any of these type of communication. Sockets or remote procedure calls may be added in the future to allow execution across the network. This is desirable because some processors on the network are significantly faster than others.

A commercial database management system was not selected for use with NPAS because of distribution considerations. Many universities and smaller organizations using NEPP could not afford the high cost of commercial database management software. There are some publicly available database managers. They are unsupported, however, and may require porting to new hardware. The benefits of using a database management system do not seem to outweigh the programming effort, the software cost, and code distribution problems involved with using such software.

The last method is to directly couple the analysis and the user-interface. That is, make the analysis code a sub-program of the user-interface. Initially, losing the advantage of running separate processes made this method appear to be a poor choice. There is no way to execute code across the network and potential crashes in the analysis will require restarting the user-interface. The ease of programming and the fast interchange of data, however, outweigh these drawbacks. The closely coupled approach was chosen for the initial version of NPAS.

Although the NPAS user-interface and analysis codes are closely coupled, an effort is made to keep the modules as separate as possible. Only the input and the main start-up routines have been changed from the batch versions of the analysis codes. The output routines have remained unchanged, in case the user prefers the old style of hard copy output.

Programming Languages and Interfacing

The C programming language was chosen for development of the graphical user-interface. The C language binds to the X Windows and Motif libraries and is the native

language on UNIX machines, making it ideally suited for systems calls. In addition, C has many advantages over FORTRAN, most notably: the ability to dynamically allocate memory, the use of data structures, and pointers. These features are essential for writing good maintainable code.

Although object oriented programming (e.g. C++) is increasing in popularity, we felt that C was better suited for this application. One reason for this is that C++ compilers are not available to all users. Also, users are more likely to be familiar with C than C++ and hence, more capable of modifying the code for their own needs. A rewrite of the code into an object oriented programming language may be considered in the future. The data and analysis modules for engines and their sub-assemblies are well-suited to this type of programming.

The analysis portion of the code is done by linking to the FORTRAN libraries. Data are passed from the FORTRAN common blocks to the C code by making C data structures with the same members (type and order) as in the common blocks. If a pointer to the structure has the same name as the common block, members of the common block can be accessed from the C code as members of the structure. Note some system append an underscore character to FORTRAN names, it is therefore necessary to append an underscore to the name of the C pointer.

The batch version of the engine cycle code reads in one case at a time, executes it, and prints the results. The user-interface needs to store input and output information for a large number of cases. It is therefore necessary to store much more information than in the FORTRAN common blocks. The C structures that overlay the FORTRAN common blocks can be copied for each case. This, however, would be a waste of memory because not all the information in every common block needs to be saved. Furthermore, the format of the data would be quite difficult to work with. Therefore, data are copied out of the common block format into other data structures that can be more easily manipulated.

Data are stored as a link list of engine design and off-design cases. Each item in this list is referred to as an engine structure. Each engine structure contains all the information necessary to run the analysis code. Structures containing information about each component and flow station as well as graphics information are pointed to by each engine structure. Memory is dynamically allocated as needed. The entire list of engine structures can be saved to a binary file. This file can later be read, enabling a user to continue working from the point where he had previously suspended execution. FORTRAN namelist reads and writes are also available from the user-interface to remain compatible with the batch version of the code.

Schematic Representation of Engines

Perhaps the most important feature of this user-interface is its ability to represent propulsion system configurations schematically. An example of such a schematic is shown in Figure 1. The schematic of the propulsion system alone would be quite helpful to the engineer (it quickly displays important information about the connection between components

and the thermodynamic properties of the gas path), but the real advantage of this diagram is that it is also a mechanism for the engineer to communicate with the application program. Engine components and flow connections can be added and removed directly from the diagram as seen on the computer screen. Detailed input specifications and computed results for a given engine component can be called up by a simple mouse click on the diagram. Additionally, displayed information is instantly updated as new data are processed. There are many complex operations necessary to handle the simple interaction between the user, the schematic representation, and the application code. This section explains some of these interactions.

In order to take advantage of the large number of pre-existing data sets, the interface was required to automatically lay out a given propulsion system configuration. Given the old style of input (a FORTRAN namelist), we wanted to be able to generate a "standardized" schematic representation of the propulsion system. This entails logical placement of components, minimizing line crossings when depicting the flow paths and physical (shaft) connections between components, and minimizing the total area used by the layout. In addition, we acknowledge that there is not necessarily one correct schematic representation for each propulsion system configuration. Individual users each have their own preferences. For example, some prefer a horizontal orientation of a layout, while others prefer a vertical layout (see Figure 2). The user can specify this option in the interface, and may even toggle the orientation once the representation has been drawn. We have included other mechanisms (discussed below) to modify an automatically generated layout.

The window containing the schematic of the propulsion system (called the layout window) is built upon a grid, which has many roles in the interface. First, the grid is used for scaling the objects in the layout window. The basic unit size is determined by the font size used, which is not hard-coded into the interface. This is one of the "customizable attributes" that the user may specify through the resource file or select from a menu pick. All of the objects within the layout window are sized relative to each other. Second, the grid is used for alignment purposes. We wanted to leave the user as much flexibility as possible to draw and move objects around the screen. At the same time, it is very difficult for users to draw a truly straight line or to align objects within a window; they will tend to be off by a few pixels. Complete flexibility in placing components within the layout window also complicated automatically drawing connections between components. Two "snap-grids" are used for alignment purposes. The larger grid is used for components, and a finer grid (1/4 of the size of the larger grid) is used for drawing connections between components. The visibility of the grids can be toggled on and off. Finally, the grids are used as data structures to hold all of the graphical information associated with the layout window. All of the routines associated with laying out and manipulating a schematic representation can access the information from one global source.

Physical engine components are represented by push button widgets. Push buttons have built-in functionality which ease the programmers' burden. The system detects when a push button action has been taken (pressing the button, mouse motion on top of the button, releasing the button, etc.) and signals the appropriate event. The programmer may add callbacks and event handlers (i.e., functions) to a push button in order to specify what the appropriate action is for each event. Push buttons also have labels, which in this case have been used to indicate the type and number of a component (e.g. "INLT 1", "SHFT 13").

Without the Motif (or other such) widget set, a great deal more effort (and many more lines of code) would have been necessary in creating the graphics of the interface, rather than in the behavior of the interface.

The drawing area widget is used as the parent widget of the component push buttons. This permits lower level X calls to be made to do the actual drawing of connections (both air flow and physical) between the engine components. Each line segment is specified by its end points, and the pixels between these two points are drawn by the system. The line segments have no built-in functionality; unlike a push button widget, they have no built-in mechanism to detect if a mouse button has been pushed while the mouse pointer is on top of a line segment. One possible way to connect engine components would be to draw a straight line from the center of one component to the center of the other. This would, however, result in very messy drawing, and would make editing the drawing extremely difficult. We decided to represent a single connection by a series of connected line segments. The endpoint of each line segment is a small push button (in the case of the end line segments, the engine component is the endpoint, and an additional push button is not added). Figure 3 shows a schematic with the line segment endpoint push buttons on. These small push buttons act as control nodes or handles for the line. Each connection has a minimum of two control nodes. As a result, the connection lines can be reshaped and moved by the user by moving one or more of the control nodes. As a line segment is "moved", its old position is "erased" by redrawing the line segment in the same color as the background, before drawing the line segment in the new position.

A station label consists of a station number and an associated engine property (see Figure 1). Although station labels are implemented via push buttons, they do not have a grid to keep them aligned. The user is able to move station labels to any position within the layout window. The leader line will remain attached to the first line segment control node associated with the station. The push buttons are "flattened", and their background color matches that of the drawing area in order to give the illusion that only the label string is actually written directly on the drawing area. Users may also define their own labels to add information to the diagram. These user-defined labels behave the same way that station labels do with the exception that the push button label is input by the user, and is not calculated by the analysis code. The user may edit an existing label, or create a new one. These labels may also be placed anywhere within the layout window. It should be noted that rather than using push buttons for the station and user defined labels we could have simply written the strings directly on the drawing area using low-level X calls (similar to the methods used for drawing connection lines). This would have complicated many of the station-related routines, including moving the labels, updating station property values, and redefining user defined labels. For this reason, station and user-defined labels were implemented using push buttons.

One motivating fact for building a graphical user-interface was the capability to create a propulsion system on the screen. Also, since there are a wide variety of individual preferences in display, we needed to include the ability to edit a particular layout (whether it was automatically or interactively placed). Through our interface, the user has the ability to add components and connections to a configuration. All of the objects (components, stations, and line segments) may be moved or deleted by the user. For single objects, motion is achieved through event handlers, and deletion is implemented via callbacks. In addition,

the user may select a region within the layout window to delete or move. A "rubber-band" rectangle is used to mark a particular region. The user marks the upper left corner of the rectangle, and then drags the lower right hand corner of the rectangle to the desired position. As in the case of the lines used to connect the engine components, the rubber-band rectangle is drawn using Xlib calls. To get the rubber-band effect, the rectangle is "erased" (by redrawing it in the background color) before being redrawn in the new position. Once the user has finished marking the region, the rectangle resizes itself relative to the smaller grid. In this way an object must fall either completely in or out of the region. Deletions within a region are done by locating all objects that fall within the region and deleting each object individually. In the case of moving a region, an event handler is used to track the mouse motion until the user indicates the new position of the region. After checking that the new region is clear, each object within the original region is moved to its new location.

Various error-checking routines have been implemented in the graphical user-interface. When a user is placing a component or control node, the interface must ensure that the object being moved is not placed on top of an existing object. There are restrictions on what components may be connected to each other, and on how many upstream and downstream connections each component may have. We have built in error checking routines for shaft connections and for the number and order of connections on each shaft. There is also a great deal of effort made updating and maintaining the underlying data structures for the interface. To delete component "C", for example, the component structure for C must be deleted and any connections to C must be removed. The data structures used by the analysis code must be updated as well. To connect two components the user need only draw in a line between them. The interface, however, needs to keep track of the path as it is drawn, and at the same time must ensure that there are at least two control nodes on the path, and that none of the new control nodes overlap existing nodes. A station structure must be created and the appropriate values inserted, and finally both the upstream and downstream component structures must be updated to reflect the new connection. In order to simplify the interface for the user, a considerable amount of time was spent developing the logic behind the interface.

The User-interface

Some of the windows used in our graphical user-interface are shown in Figure 4. The main window, at the top of the screen, has three major components: the main menu, the message line, and the layout window. Along the top of the main window is the main menu. This menu controls the majority of the execution of the program. We made the look and feel of the menu system consistent with the one that is used for most Motif applications. The user should, therefore, be immediately familiar with the location of important menu selections (e.g. the location of help or file retrieval). Immediately below the main menu is the message line. Here messages will be posted in response to user commands. For example, a successful execution of the analysis code would be indicated on this line. The layout window is composed of a drawing area widget inside a scrolled window widget. The details of the layout window have already been discussed.

We have included two types of documentation in the NPAS program. One is the help

system. This is activated from help buttons on various forms or from pressing the help key while in a text field. Help information is then recalled from an ASCII file by searching for an identifier that is associated with the activating widget. This information is then displayed in a message window that is popped to the foreground for high visibility. Examples of this type of help are how to use a form or a detail description of an input.

The other method used for documentation uses a table of contents approach. This is shown in Figure 5. On the left is a list of sub-topics for the current documentation category. The documentation category is selected from the help menu. In order to view a topic, the user simply uses the mouse to click on the desired topic. The corresponding information is recalled into the scrolled text region to the right. Currently, documentation files exist on the following topics: user-interface, NEPP user's manual, component descriptions, and usage notes for components. The documentation file consists only of topic titles followed by the corresponding help information. The file will be searched for all topic titles and the table of contents will be generated. The complete content of the file is not read into memory. After the user makes a selection from the list, the file is searched again to find the corresponding information. Although this approach may sound slower then using a binary read on the entire file, we found the response time to be almost instantaneous. Also, because the on-line documentation is recalled from an ASCII file, there is no need to recompile code to add additional help information.

The middle window on the bottom of Figure 4 is the configuration listing. This window is a scrolled list widget that contains a list of components and their connections. This list is needed to select "components" that don't represent physical hardware and, therefore, are not shown in the layout window. An example of a non-physical component is a control. A control allows the user to vary input parameters to produce some desirable effect. Clicking on an item in this list will make that component active and refresh other windows to reflect the change. Clicking on a component in the layout window has the same effect. The information in the configuration listing is represented in a manner similar to the old namelist input, and so is familiar to experienced NEPP users.

To the left of the configuration listing in Figure 4 is the off-design window. As mentioned above, each case to be executed by the analysis module has a unique data structure referred to as an engine structure. This window contains a listing of all the engine structures that have been created. Data in these structures can be viewed or modified by selecting an item in the off-design listing with a single mouse click. A double mouse click will execute that case in the analysis code. The list contains the case number, a flag indicating if it is a design or off-design case, the Mach number and the altitude. The Mach number and altitude are displayed here because these are the variables most commonly changed during off-design operation. Although these values are normally set in the inlet component specifications, input fields for these values are placed at the bottom of the off-design form for convenience. After execution, the net thrust and specific fuel consumption are also displayed in this listing. This is a good visual cue to the user that the case has been executed.

Much of the numerical input and output in the user-interface is done using forms with a consistent look and feel. We felt that this would make learning to use the interface easier. One example of a form is shown in the bottom right corner of Figure 4. This is the input form for component specifications. The form consists of a column of text input fields -- one

for each numeric input of the component. To the right of each text field is a short description of the input. To the left of each text field is a push button. These buttons activate help for the corresponding input field. The help includes a detailed description of the input and, if applicable, a selection list of valid choices. Each time a component is selected from the layout window or the configuration listing, the fields in these forms change to reflect the newly selected component.

Each string entered into a text field is processed though a scientific calculator program. This allows inputs to be mathematical expressions instead of simple numeric values. The calculator program was written for use with NPAS but can be used in any other program, and is also able to stand alone. Details on the calculator program are given below. Equations can be recalled by typing an equal sign in the input field. This is currently the only way to re-evaluate an equation.

In many cases, when an input is modified, the user only wants to change the selected engine case. Sometimes, however, the user may need to change all cases to the same value. For example, when making a throttle curve the user probably does not want to change the turbine inlet temperature in all cases. On the other hand, when changing the design compressor pressure ratio the user probably wants the change to be reflected in all off-design cases. This is accomplished by using special flags in the input fields. Three flags are available, they are:

#x - change input for all cases to x,
+x - increase input for all cases by x, and
%x - change input for all cases by x percent.

If one of these flags is not in column one of the input field, only the currently selected case is modified. Using these types of flags in input fields may seem contrary to good mouse driven programming style. In this case, however, we felt these flags were easier to use than a mouse driven selection.

Forms for configuration inputs, component outputs, weight inputs, and inputs for the installation code are similar to those used for component specification. In general, this type of form is used for entering data that were previously input as some form of array. The calculator program is only used when entering real numbers. Integer and character inputs are not processed through the calculator, although the global change character, #, may be used. Output fields can not be edited.

Other types of data entry are done through menus or specialized dialog boxes. Although data entry fields are not standardized in these dialog boxes the locations of the Apply, Cancel and Help buttons are made consistent throughout the interface. The Open Software Foundation provides a Motif style guide to help programmers develop a consistent look and feel to user-interfaces[15].

A spreadsheet approach to numeric input was also considered. Although this would permit more data to be displayed on the screen at one time the advantage of having a description for each input/output field seemed more important. A spreadsheet approach also has the advantage of cell references in equations. Our solution to this was to allow variable

naming in the input fields. Of course, there still is no mechanism to refer to a range of data but we had little need for this in our application.

We have created a specialized text editing program for our user-interface. The primary function of this editor is to display tables of output data. The editor was created by using the multi-line scrolled text widget. This widget makes a good beginning for an editor program. Adding a menu system along with cut and paste function makes a complete editor. The text editing window is shown in Figure 6. The "Display" menu allows the user to select what information is to be displayed in the window. Each time the user executes a new case, the output information in the editor is changed. Also, selecting a previously executed case in the off-design window will change this information. An append mode is available so that the user may compare results from various cases. Cut and paste functions can be used to arrange data into a more desirable manner. Files can be selected for display from the File menu. They include normal display mode, NEPP output (old format), map tables, namelist input, and plot setup files. Of course, any changes to any of these files can be saved to disk. Help on using the editing program is also available from the menu system or by pressing the help key.

Plotting Capabilities

Another important feature of this user-interface is its graphics capability. Plots can be created for a variety of data. The user can request plots for any of NEPP's input or output variables by simply selecting the items in a dialog box. Turbine and compressor maps can also easily be plotted. Additionally, flow paths can be drawn from the output information of the WATE code. Examples of a throttle curve, a compressor map and a flow path are shown in Figures 7 through 9, respectively.

Currently, all plots are created using a modified version of the Grafic plot library from the Massachusetts Institute of Technology. Grafic is a set of FORTRAN callable subroutines for creating x-y, contour and 3-D plots. Grafic has device drivers for both X Windows and GL. The source code is public domain and is easily customized for new plotting applications. Postscript output may be generated for any plot created by NPAS.

Most of the plotting is done by writing plot files and executing a separate program to create the plot. In this way, the user-interface is not strongly tied to any set of plot routines. Another advantage is that the user may retain their plot files for later use. The creation of the plot files and the execution of the separate program is transparent to the user.

Calculator Program

A customized calculator program was created to be used with NPAS. It evaluates character strings and returns the appropriate value. This entails parsing the input string (based on mathematical associativities and precedences), looking up the value of any constants or user-defined variables, and doing any necessary computations. The calculator supports pre-defined constants, logarithmic functions, trigonometric functions, and scientific

notation. As previously mentioned, the calculator program is an independent module that can be used in other applications or by itself.

Two UNIX utilities, LEX and YACC, were used to implement the calculator program[16]. 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[17]. In addition, the user may specify precedences and associations to remove any ambiguities inherent in the original grammar. YACC generates a C file which may be incorporated into a larger program.

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 builds up 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. Although building a parse tree is more complicated than implementing immediate evaluation, we believed it provided greater flexibility for the calculator program, and would simplify future extensions.

Along with the traditional functions associated with a calculator, there is also support for creating variables and binding them to values. This is implemented by maintaining a variable table. When a variable is defined by the user, its name/identifier and corresponding value are added to the table. Once a variable has been added to the table, it may also be re-defined (i.e., have its value changed). Variable names must be unique, and in this respect all variables are, in effect, global in their scope. This enables users to define their own constants, and to define one expression based on another. The issues involved in re-evaluation of expressions are left to the modules that call the calculator program.

Concluding Remarks

A fast effective way to do propulsion system performance and weight analysis has been developed resulting in a substantial time savings for the engineer. This was accomplished by building a graphical user-interface around the NEPP, WATE and INSTAL programs. The X Windows System with OSF/Motif was found well-suited for this application. A direct coupling between the user-interface and the analysis code was used because of its simplicity and speed of execution. In addition to automatically laying out an existing propulsion system configuration, the interface enables the user to interactively create a configuration from scratch. The schematic representation of the propulsion system is not only a way for information to be displayed but is also the mechanism which the engineer uses to communicate with the application software. Forms have been created to simplify entry of numeric data. A calculator program was incorporated to allow the user to input mathematical expressions in the input fields for component and engine specifications. The graphical user-interface was designed to be user-friendly for both new and experienced engineers. As it is event-driven and window-based, the user-interface provides a great deal of flexibility for its users.

As pointed out earlier, much of the programming effort in building a user-interface is not in writing graphics code but obtaining the proper behavior of the interface. X Windows and Motif do much of the graphics work for you. The critical part in developing the user-interface is not detecting when the mouse button is pressed, but what to do with the information that a particular mouse event represents. This information may be handled in a variety of ways: it may need to be checked for validity, stored for later access, converted and transferred to the analysis program, checked for impact on other data, written to disk, etc. These tasks represent a great deal more of the programming effort than, say, drawing a line to the point where that mouse click occurred. To ease the programming effort, it is important to have a well thought-out data structure and data flow. CASE tools, which are now appearing on the market, may be helpful in this regard.

Another important issue that engineers tend to overlook while writing code is error handling. This code is no exception; initial versions of the code had little checking on user inputs. As indicated previously, it is critical to the success of the application either to prevent users from doing things incorrectly or to inform them if they have. An event-driven interface leaves control in the hands of the user, and so it must be ready to respond to any action taken by the user. It is not possible to anticipate all potential user-actions (especially errors that may be made). Thus as the program matures more error handling should be added.

Future work may also include integration of mission analysis under the user interface. It is not adequate to compare propulsion systems based solely on performance and weight; propulsion/airframe integration must be considered. This can be accomplished faster and easier with a mission analysis code integrated under a common user interface.

References

1. Fishbach, L.H.; and Caddy, M.J.: *NNEP - The Navy NASA Engine Program.* NASA TM-X-71857, 1975.

2. Plencner, R.M.; and Snyder, C.A.: *The Navy/NASA Engine Program (NNEP89) - A User's Manual.* NASA TM-105186, 1991.

3. Fishbach, L.H.; and Gordon, S.: *NNEPEQ - Chemical Equilibrium Version of the Navy/NASA Engine Program.* NASA TM-100851, 1988.

4. Onat, E.; and Klees, G.W.: *A Method to Estimate Weight and Dimensions of Large and Small Gas Turbine Engines.* NASA CR-159481, 1979.

5. Kowalski, E.J.; and Atkins, R.A., Jr.: *Computer Code for Estimating Installed Performance of Aircraft Gas Turbine Engines, Volume II User's Manual.* NASA CR-159692, 1979.

6. Fishbach, L.H.: *KONFIG and REKONFIG - Two Interactive Preprocessing Programs to the NAVY/NASA Engine Program (NNEP).* NASA TM-82636, 1981.

7. Berton, J.J.; Plencner, R.M.: *An Interactive Preprocessor for the NASA Engine Performance Program.* NASA TM (to be published).

8. Plencner, R.M.: *Plotting Component Maps in the Navy/NASA Engine Program (NNEP) - A Method and Its Usage.* NASA TM-101433, 1989.

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

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

11. Nye, A.: *The Xlib Programmer's Manual.* O'Reilly and Associates, 1988.

12. Nye, A.: *The Xlib Reference Manual.* O'Reilly and Associates, 1988.

13. Rochkind, M.J.: *Advanced UNIX Programming.* Prentice-Hall, 1985.

14. Stevens, W.R.: *UNIX Network Programming.* Prentice-Hall, 1990.

15. Open Software Foundation: *OSF/Motif Style Guide.* Prentice-Hall, 1991.

16. Kernighan, B.W.; Pike, R.: *The UNIX Programming Environment.* Prentice-Hall, 1984.

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

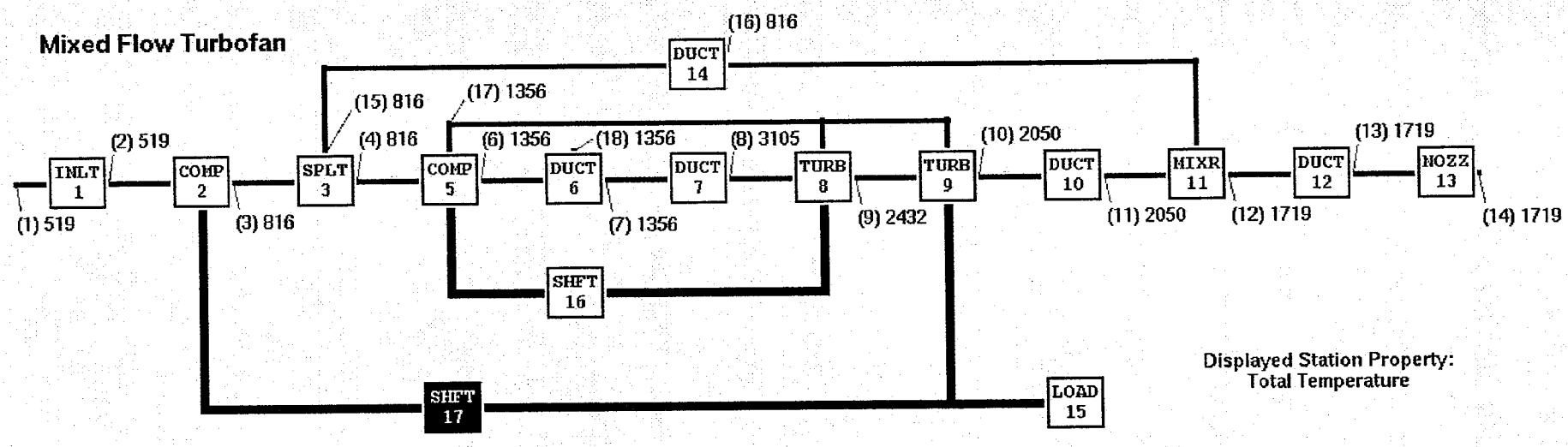


Figure 1: Schematic Representation of a Propulsion System

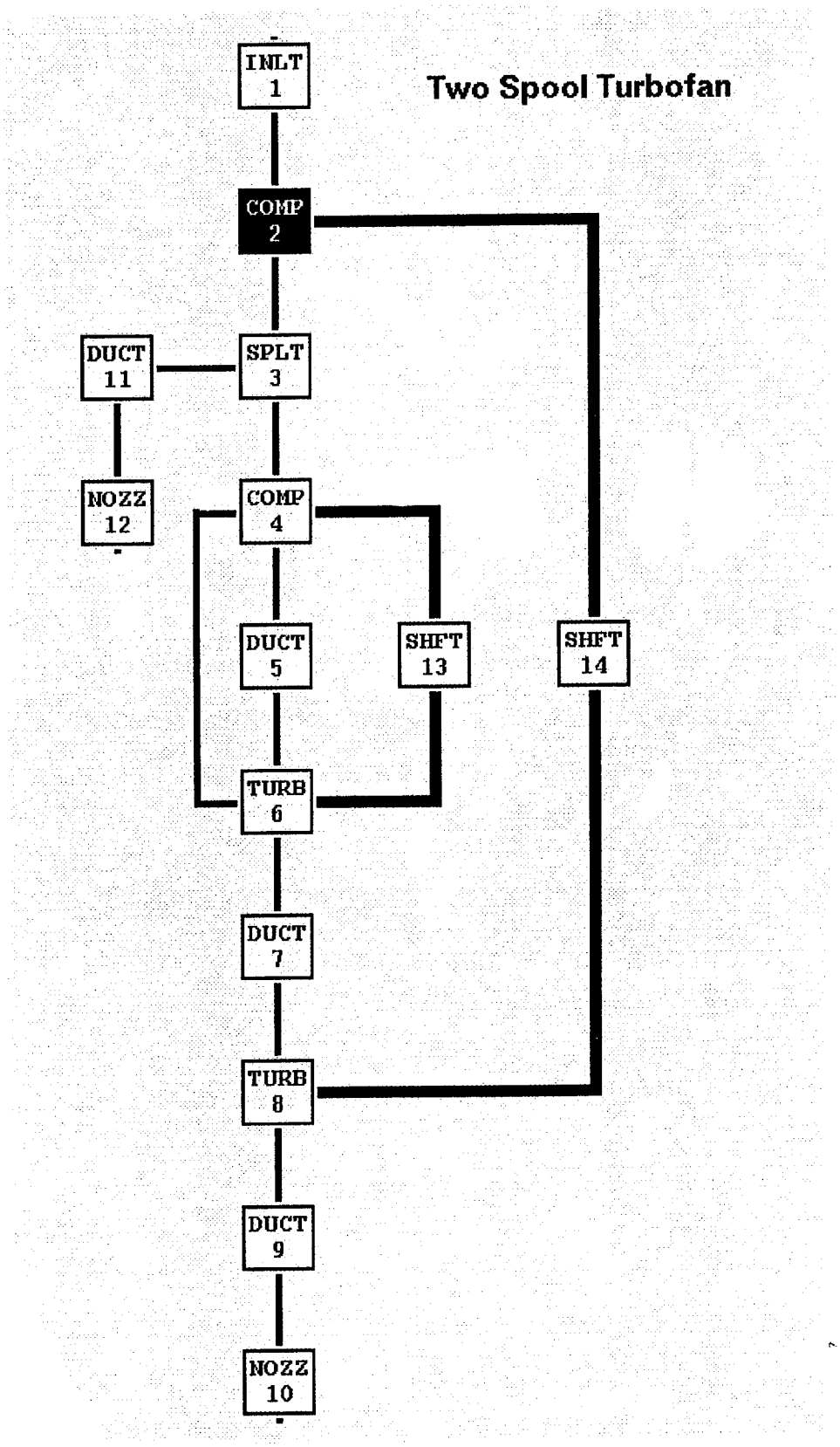
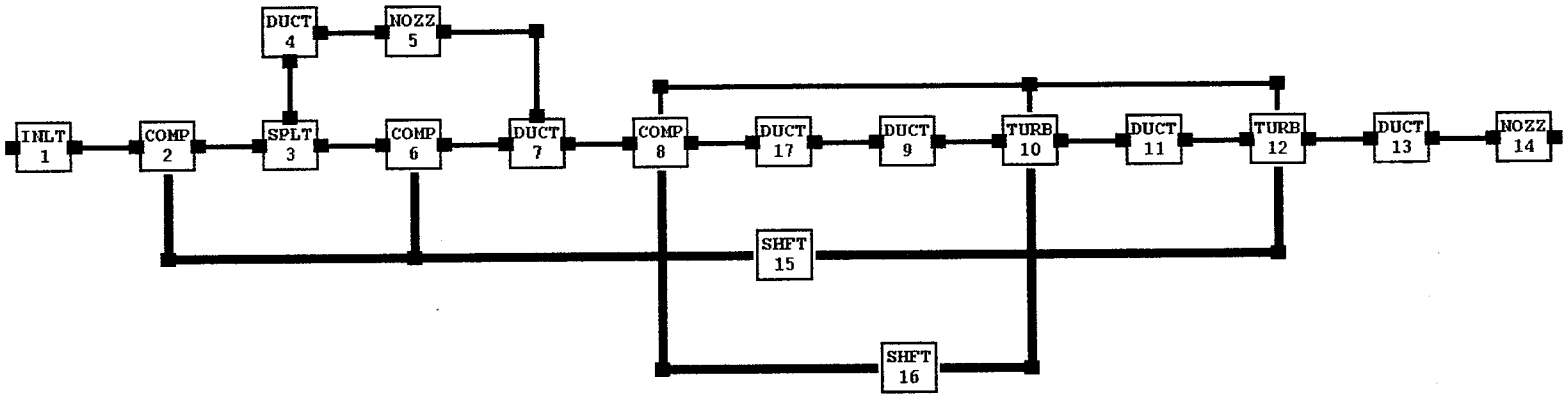
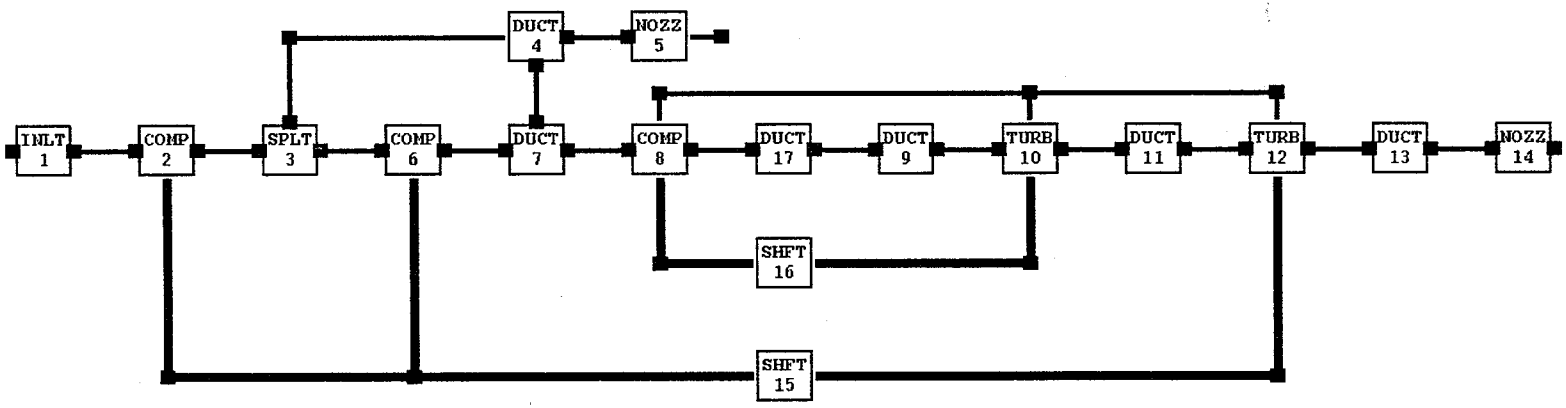


Figure 2: Vertical Engine Layout



Automatic Layout



Layout After Editing

Figure 3: Layout Showing Line Segment Control Nodes

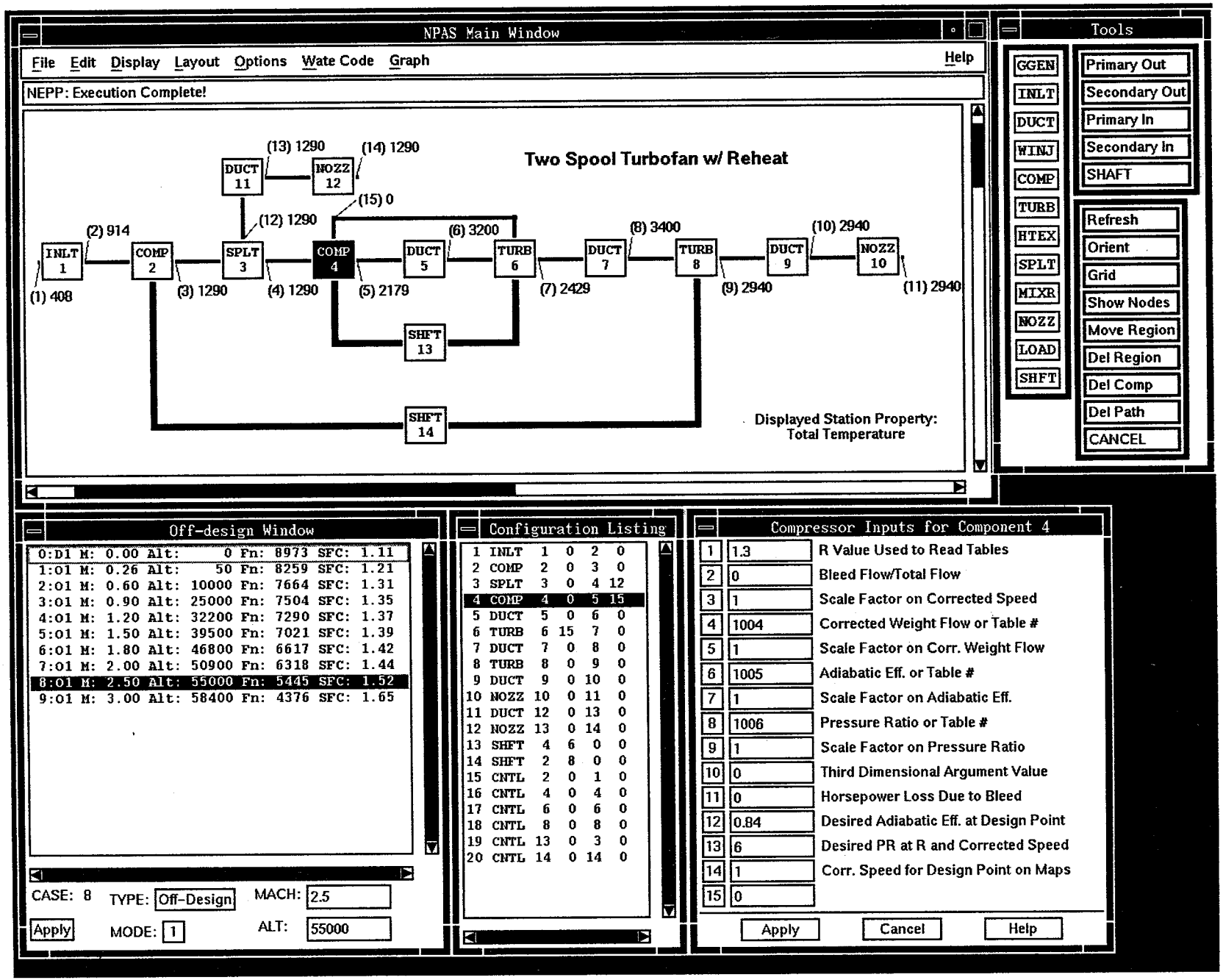


Figure 4: The User Interface

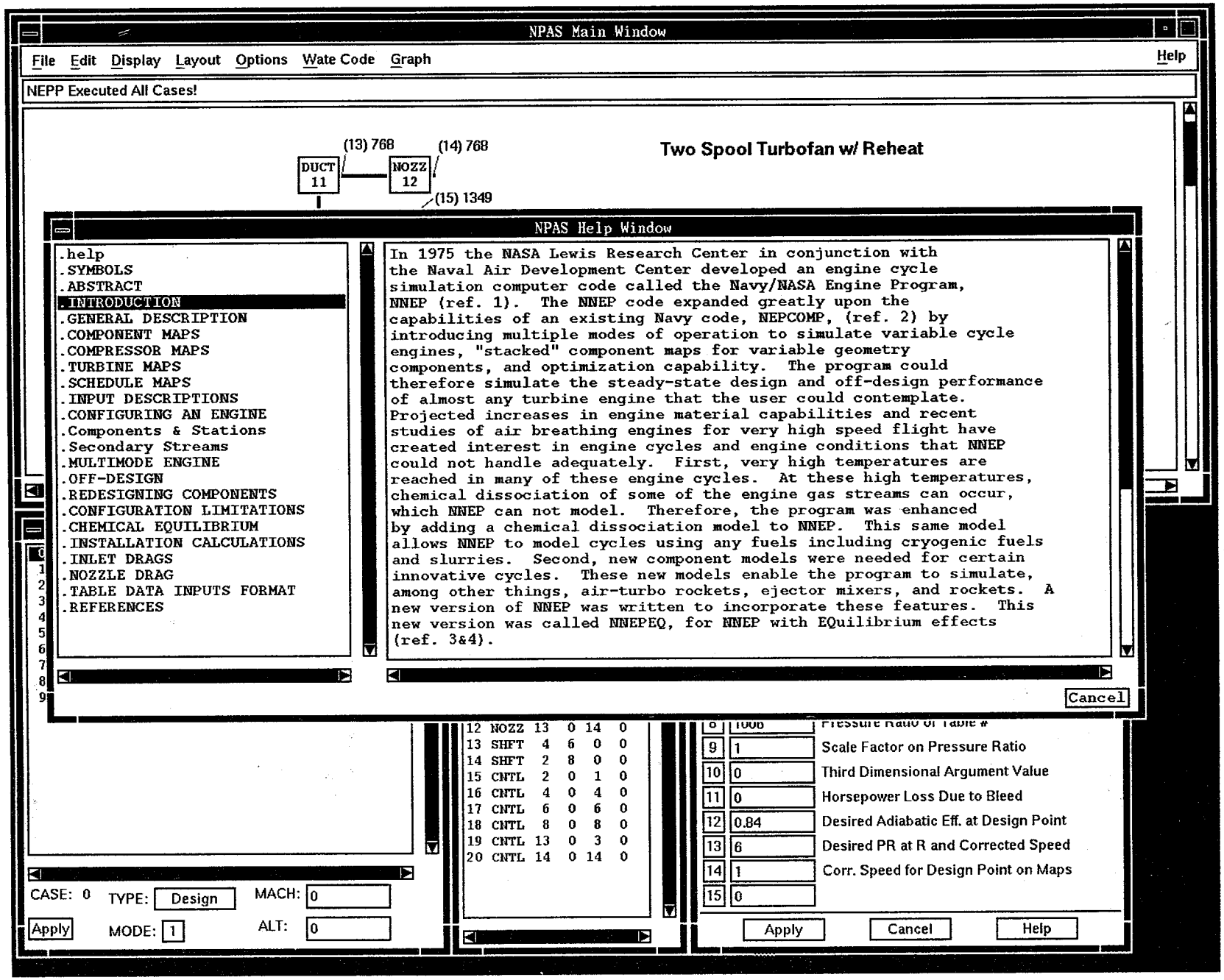


Figure 5: Help Window

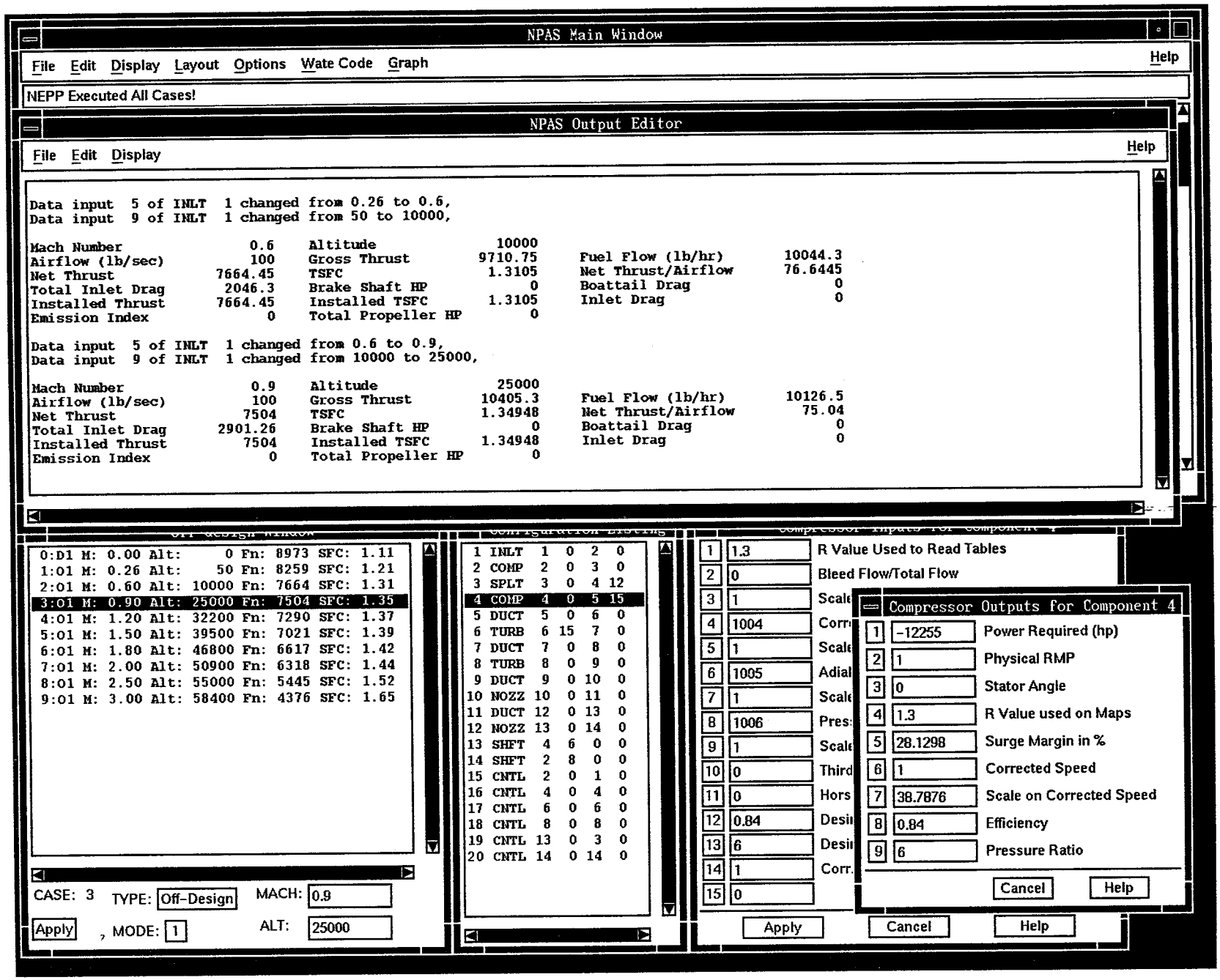


Figure 6: Output Text Editor

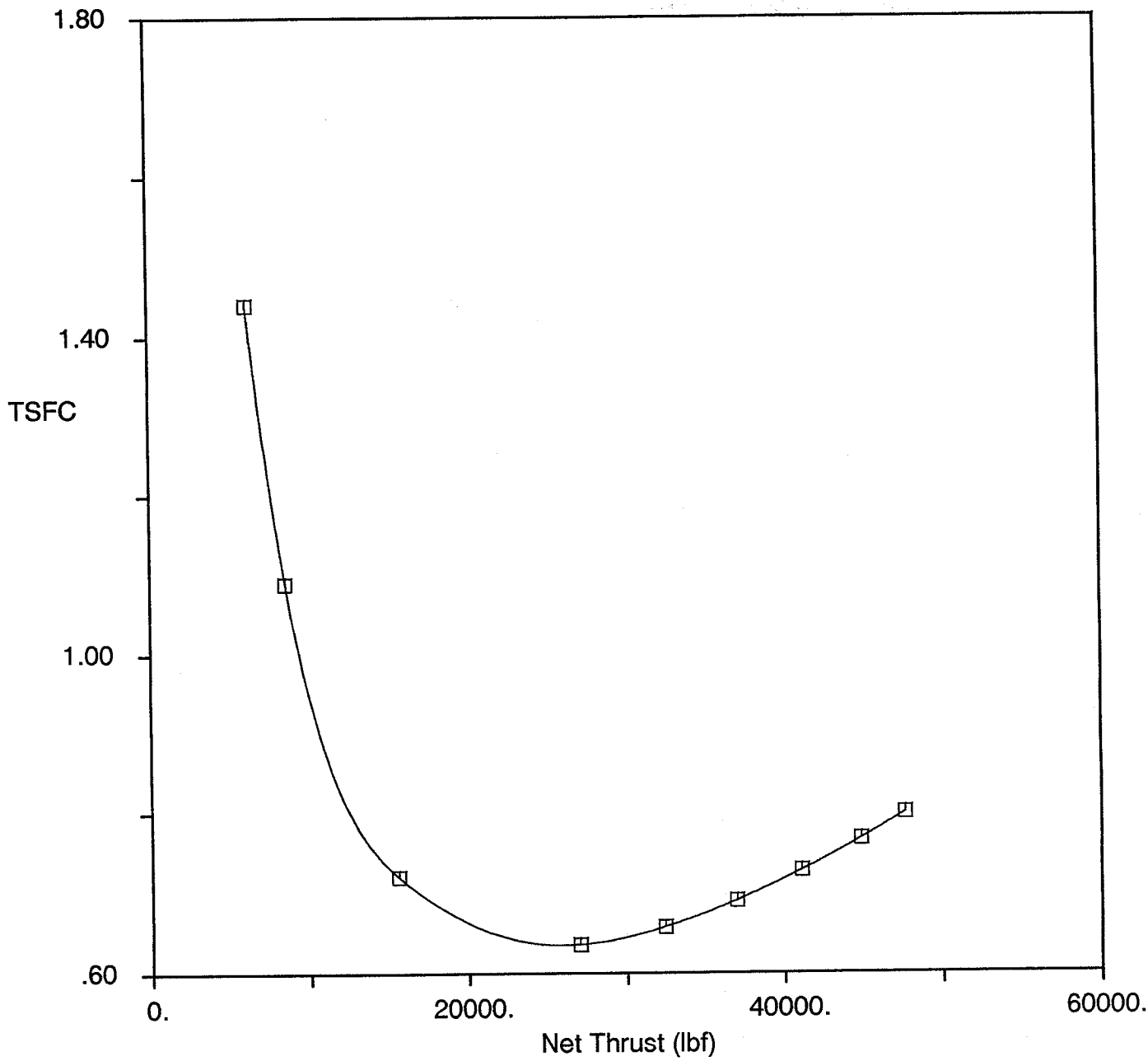


Figure 7: Sample Throttle Curve Plot

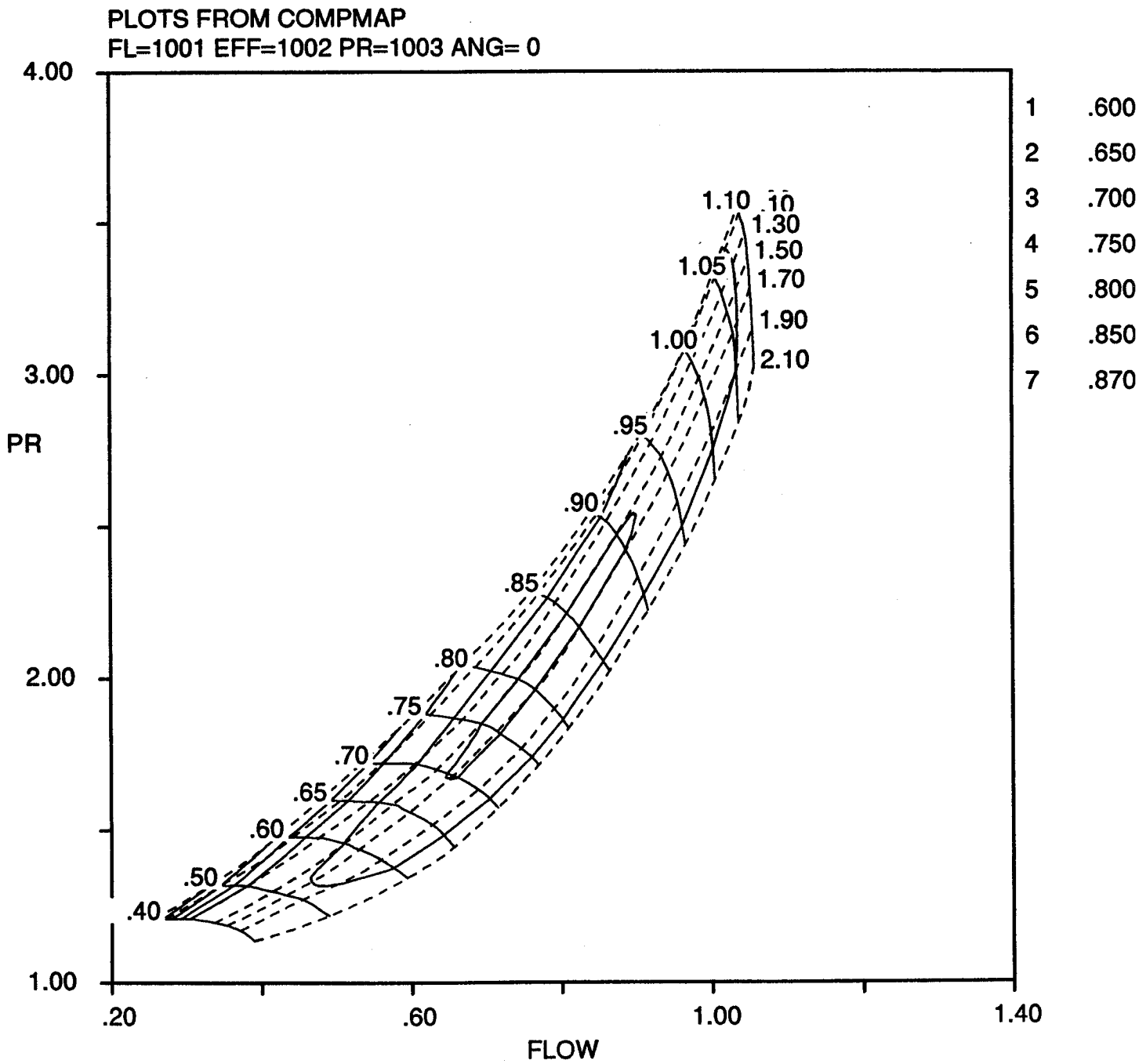


Figure 8: Sample Compressor Map Plot

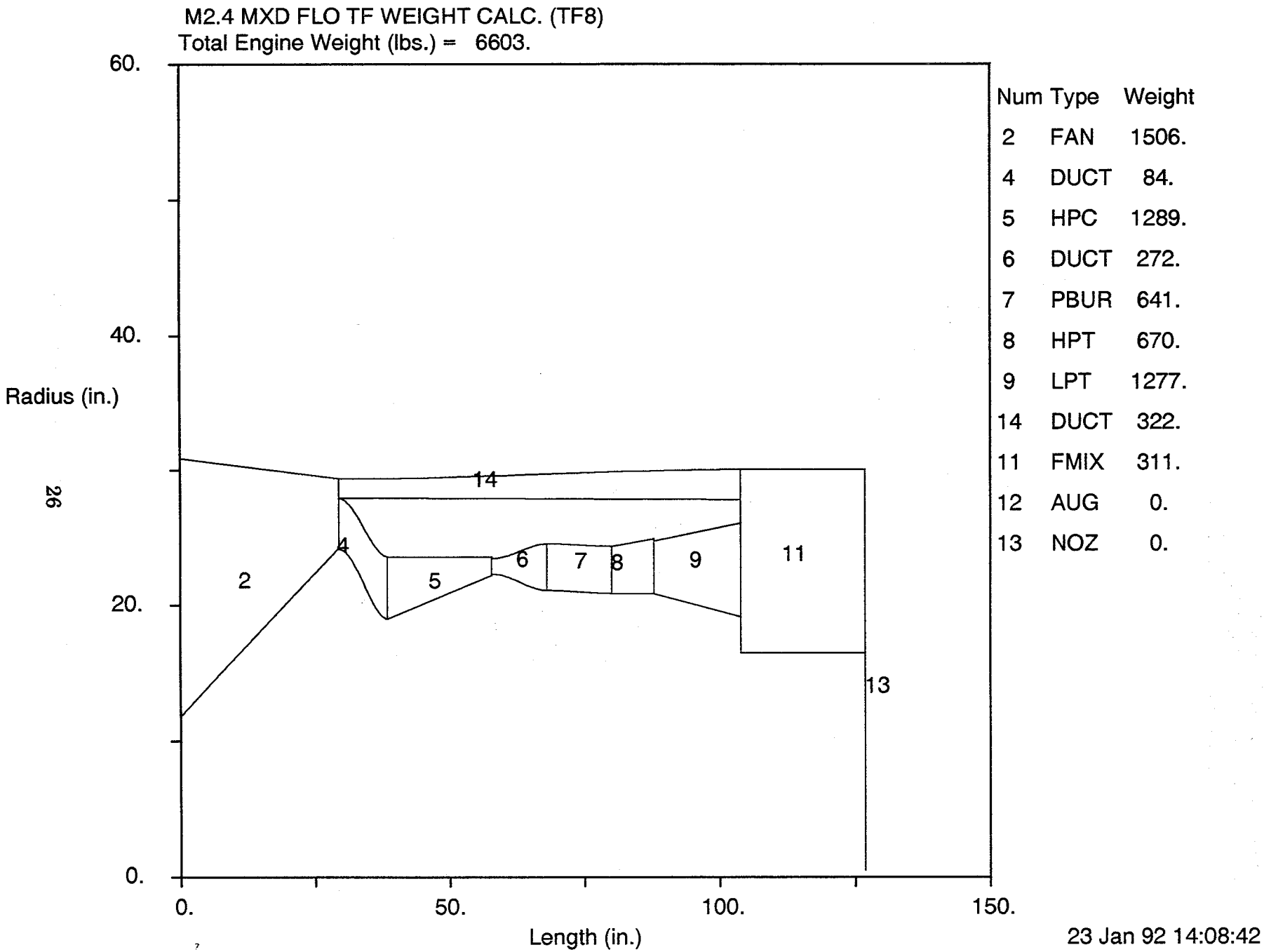


Figure 9: Sample Flow Path Plot

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (*Leave blank*)

2. REPORT DATE
August 1992

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
A Graphical User-Interface for Propulsion System Analysis

5. FUNDING NUMBERS
WU–505–69–50

6. AUTHOR(S)
Brian P. Curlett and Kathleen Ryall

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–7158

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, D.C. 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM–105696

11. SUPPLEMENTARY NOTES
Brian P. Curlett, NASA Lewis Research Center, Cleveland, Ohio, and Kathleen Ryall, Harvard University, 33 Oxford Street, Cambridge, Massachusetts 02138. Responsible person, Brian P. Curlett, (216) 433–7041.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

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

The NASA Lewis Research Center uses a series of computer codes to calculate installed propulsion system performance and weight. The need to evaluate more advanced engine concepts with a greater degree of accuracy has resulted in an increase in complexity of this analysis system. Therefore, a graphical user interface has been developed to allow the analyst to more quickly and easily apply these codes. This paper describes the development of this interface and the rationale for the approach taken. The interface consists of a method of pictorially representing and editing the propulsion system configuration, forms for entering numerical data, on-line help and documentation, post processing of data, and a menu system to control execution.

14. SUBJECT TERMS
Engine performance; Computer code; Gas turbine engines; Cycle analysis

15. NUMBER OF PAGES
28

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