N92-24404

IGGy - AN INTERACTIVE ENVIRONMENT FOR SURFACE GRID GENERATION*

Nathan C. Prewitt
Sverdrup Technology Inc.
Eglin AFB, FL

SUMMARY

A graphically interactive derivative of the EAGLE boundary code is presented. This code allows the user to interactively build and execute commands and immediately see the results. Strong ties with a batch oriented script language are maintained. A generalized treatment of grid definition parameters allows a more generic definition of the grid generation process and allows the generation of command scripts which can be applied to topologically similar configurations. The use of the graphical user interface is outlined and example applications are presented.

INTRODUCTION

The EAGLE code [1-3] was the first large-scale, widely accepted, multi-block, structured grid generation code for general CFD applications. It is broken into two pieces: the EAGLE boundary code [2] and the EAGLE grid code [3]. With the grid generation problem approached as the solution to a boundary-value problem, the boundary code is used to generate surface grids which define the boundaries and the grid code is used to generate the volume grid from these boundaries. Both codes operate off of a series of batch oriented commands constructed from reading a FORTRAN *namelist*. Together, these commands define a primitive language for grid generation. Although it has been surpassed by more user-friendly interactive grid generation systems, EAGLE still displays some advantages over these systems via its batch oriented command structure.

One of the advantages of the EAGLE code is its ability to build scripts of the grid generation process using its language-like commands. The scripts not only serve as a convenient way of storing or exchanging grids but also force the user to obtain a good mental picture of the grid generation process. Command scripts are most powerful through the use of indirectly addressed parameters. These parameters are in the form of three component vectors representing points in space, integer values representing numbers of points on boundary segments, and real values representing grid point spacings along segments. The use of these parameters allows the user to redimension the grid, change the grid point distribution, or even alter the geometry of the physical boundaries by editing only the commands which define these parameters. Once altered, the script is reprocessed, in a

*Work done under contract with the Air Force Development Test Center, Eglin AFB, Florida batch mode, with the alterations propagating through both the surface and grid codes to produce the final grid.

When given a new configuration about which a grid is to be generated, the first step is to plan the grid topology. From the standpoint of block structured grids, this includes the number, location, and connectivity of the grid blocks [4]. This topology definition is the basic input to the grid code. In the construction of the boundaries of each block, it is necessary to specify certain points in space, the number of points along the boundaries, and the distribution of these points [1]. The specification of these parameters is the initial step in setting up input for the boundary code. References [1] and [5] give several example applications and stress the advantages of using indirectly addressed parameters in the efficient application of the EAGLE boundary code. In general, however, the specification of grid parameters is not flexible enough to allow the generation of generic scripts which can be applied to topologically similar configurations.

IGGy (Interactive Grid Generation sYstem) is a graphically interactive derivative of the EAGLE boundary code which provides a more user-friendly, intuitive interface. It is an interactive front end to the basic geometry engine that is found in the EAGLE boundary code and allows you to interactively build and execute commands and immediately see the results. The definition of the grid parameters has been generalized from point coordinates, numbers of points, and spacings to include any vector, integer, or real values needed. This extension of the indirectly addressed parameters allows a more generic definition of the grid generation process; thus, scripts can more easily be applied to topologically similar configurations.

DESIGN PHILOSOPHY

IGGy is designed for the novice user who is familiar with the applicability of different grid generation methods, but is not an expert at using the code. He/she may be new to grid generation, or may be someone whose expertise lies in another area of computational simulation and who uses grid generation only as a prerequisite to accomplishing another task. Thus the user interface is designed to be easy to use and very intuitive.

IGGy uses a namelist-like command syntax which identifies each command with a descriptive name. The commands may be entered at the console or selected from a menu system. When building a command from the menu system, the user is prompted for all necessary input. To reduce the amount of input required to accomplish a single task, IGGy commands have been patterned after the operation of the UNIX operating system: commands do a single, well defined task and do it well. It is up to the user to chain these basic commands together to create the desired effect. If an additional capability is needed, a new command is created rather than adding additional input variables to an existing command. This simplifies the command input, gives the user a stronger sense of the command capabilities, and hopefully builds a more robust, more easily maintained software system.

IGGy operates like a grid microprocessor with only one register. Many segments can be held in memory, but only one can be operated on at a time. This segment is said to be in *current* position. Most commands either generate a new *current* segment or operate on the *current* segment. These commands fall into the two categories of geometry definition and geometry manipulation. The geometry definition commands consist of the most basic Computer Aided Design (CAD) capabilities. More complex curve and surface shapes need to be imported from an outside source. The geometry manipulation commands provide the capability to distribute points on curves and surfaces in a manner appropriate for CFD analysis. This formal separation of tasks simplifies the command operation and reduces the input required for a given command.

COMMAND STRUCTURE

The IGGy input commands are of the following format:

E$INPUT ITEM = "operation", variable = value, ... $

where the string assigned to ITEM identifies the command being executed, and the list of comma separated assignments specify relevant quantities for the command. Arrays are assigned using a comma separated list of values; while particular elements of an array are assigned using the array name followed by the array element within parentheses. Since only one namelist (INPUT) is used and each command is identified by a string assigned to the variable ITEM, the following shorthand notation is available:

$"operation", variable = value, ... $

where the string immediately following the opening $ is assigned to ITEM.

To implement the more general definition of the parameters, a new parser has been written for IGGy using the standard UNIX utilities *yacc* [6] and *lex* [7]. Yacc is a general purpose parser generator; while lex performs low level lexical analysis. Lex identifies low level constructs, called tokens, by matching user specified regular expressions. Yacc groups these tokens, according to user specified context-free grammar rules, to build higher constructs. The output from these utilities is C code which can be linked with the main routines and called whenever an input command is to be parsed.

In IGGy's parser, lex recognizes variable names, integer values, floating points values, quoted strings, parameter references, intrinsic functions, and punctuation symbols, and returns appropriate tokens. Yacc uses these tokens to build integer expressions, real expressions, vector expressions, and assignment constructs which conform to the syntax specified by the grammar rules. To complete the parsing of a command, yacc evaluates all expressions and performs the assignments.

The syntax used to represent references to stored parameters are shown in Table 1, where *index* is any integer value. Any variable can access a parameter stored at the specified index if the variable and stored parameter are of the same type. After parsing, the variable will contain the value stored at the specified index. This eliminates the need for special coding to recognize parameter references.

Table 1: Syntactic Constructs for Indirectly Addressing Parameters

~*index*	vector storage reference
!*index*	integer storage reference
#*index*	real storage reference

Table 2 lists the valid syntactic constructs for generating integer expressions. Here, *ival* represents any integer value, *iexp* represents any valid integer expression, and *rexp* represents any valid real expression. The capital letters and punctuation marks are part of the syntax. The use of *iexp* in the definition of integer expressions denotes that this definition is recursive. The order of precedence of the arithmetic operators is fashioned after FORTRAN or C; however, parentheses can be used to alter the order of evaluation of integer subexpressions. Any integer variable or array can be assigned a value using any of these constructs.

Table 2: Valid Integer Expressions

!*ival*	integer parameter reference
ival	integer value
ival:*ival*	range of integers
iexp + *iexp*	addition
iexp − *iexp*	subtraction
iexp * *iexp*	multiplication
iexp/*iexp*	division
MOD(*iexp*,*iexp*)	modulus value
iexp ** *iexp*	exponentiation
ABS(*iexp*)	absolute value
MIN(*iexp*,*iexp*)	minimum value
MAX(*iexp*,*iexp*)	maximum value
−*iexp*	unary minus
(*iexp*)	subexpression grouping
INT(*rexp*)	type conversion

Table 3 lists the valid syntactic constructs for generating real expressions. Here, *ival* represents any integer value, *rval* represents any real value, *iexp* represents any valid integer expression, *rexp* represents any valid real expression, and *vexp* represents any valid vector expression. The capital letters and punctuation marks are part of the syntax. This definition is also recursive and parentheses are available for subexpression grouping. Any real variable or array can be assigned a value using any of these constructs.

Table 4 lists the valid syntactic constructs for generating vector expressions. Again, *ival* represents any integer value, *rexp* represents any real expression, and *vexp* represents any vector

Table 3: Valid Real Expressions

#*ival*	real parameter reference
rval	real value
PI	3.14159265...
rexp + *rexp*	addition
rexp − *rexp*	subtraction
rexp * *rexp*	multiplication
rexp/*rexp*	division
rexp ** *rexp*	exponentiation
SIN(*rexp*)	sine
COS(*rexp*)	cosine
TAN(*rexp*)	tangent
ASIN(*rexp*)	inverse sine
ACOS(*rexp*)	inverse cosine
ATAN(*rexp*)	inverse tangent
LOG(*rexp*)	natural logarithm
LOG10(*rexp*)	common logarithm
SQRT(*rexp*)	square root
ABS(*rexp*)	absolute value
MIN(*rexp*,*rexp*)	minimum value
MAX(*rexp*,*rexp*)	maximum value
vexp.X	x component
vexp.Y	y component
vexp.Z	z component
|*vexp*|	vector magnitude
vexp.*vexp*	dot product
−*rexp*	unary minus
(*rexp*)	subexpression grouping
REAL(*iexp*)	type conversion

expression. All other punctuation is part of the syntax. The definition of vector expressions is also recursive; however, braces and brackets are used to alter the evaluation of vector subexpressions. A vector variable is any real array with three elements which correspond to the three Cartesian coordinate directions. The most basic form of specifying a vector value is three, comma separated real values enclosed in curly braces. If the curly braces are replaced with square brackets, the specified vector will be normalized. The use of braces or brackets is required to obviate a context sensitive ambiguity that would exist otherwise.

Table 4: Valid Vector Expressions

~*ival*	vector parameter reference
{*rexp*,*rexp*}	2d vector
{*rexp*,*rexp*,*rexp*}	3d vector
[*rexp*,*rexp*]	2d normalized vector
[*rexp*,*rexp*,*rexp*]	3d normalized vector
rexp * *vexp*	scalar/vector multiplication
vexp/*rexp*	vector/scalar division
vexp + *vexp*	vector addition
vexp − *vexp*	vector subtraction
vexp^*vexp*	vector cross product
vexp * *vexp*	component by component multiplication
vexp/*vexp*	component by component division
−*vexp*	unary minus
{*vexp*}	vector grouping
[*vexp*]	normalized vector grouping

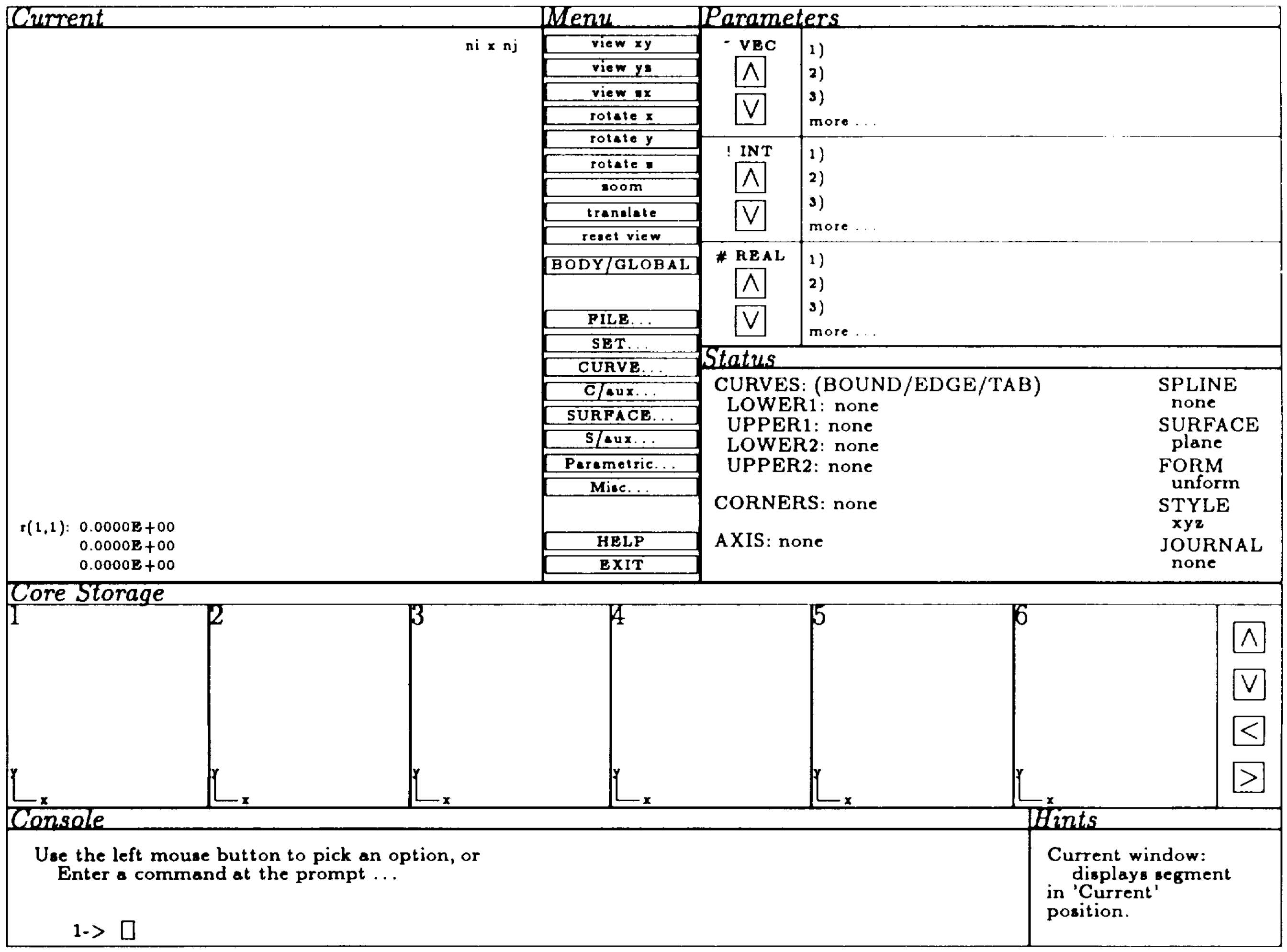
USER INTERFACE

IGGy is written for use on Silicon Graphics IRIS workstations using IRIS Graphics Language (GL) calls. The software development was started on an IRIS 3000 series workstation, and its use has been extended to the line of IRIS 4D workstations. Compatibility with the older hardware platform has been maintained through this version of the software.

Upon executing IGGy, a single graphics window is opened. This window is 1024 by 768 pixels in dimensions, and therefore takes up the full screen on the lower resolution 3000 workstation and most of the screen on the higher resolution 4D workstations. On the 4D machines, this window may be expanded to fill the entire screen, or may be left at the default size to provide easier access to other application windows.

Figure 1 shows the layout of the main window. To make the software easier to use, most of its capabilities are displayed on the screen. The buttons, which are displayed, may be *picked* using the cursor and the left mouse button. This is done by positioning the cursor over the button using the mouse, and pressing and releasing the left mouse button. Some regions of the screen, which do not appear as buttons, may also be picked. These regions are discussed where applicable.

As can be seen in Figure 1, the main window is broken into seven pseudo-windows. These are not true windows with respect to the window manager but are merely reserved sections (viewports) of the main window. They can not be resized independently, nor can they be moved relative to each other. Each of these windows is described in greater detail in the following sections.



r(1,1): 0.0000E+00
0.0000E+00
0.0000E+00

CURVES: (BOUND/EDGE/TAB)
LOWER1: none
UPPER1: none
LOWER2: none
UPPER2: none

CORNERS: none

AXIS: none

SPLINE none
SURFACE plane
FORM unform
STYLE xyz
JOURNAL none

Use the left mouse button to pick an option, or
Enter a command at the prompt ...

1-> ▯

Current window:
displays segment in 'Current' position.

Figure 1: Layout of IGGy's main screen.

Current Window

The Current window displays a wireframe drawing of the segment in *current* position. The curvilinear coordinate lines which construct this segment are color coded, with green lines being in the first coordinate direction (lines of constant η, or ξ lines) and yellow lines being in the second coordinate direction (lines of constant ξ, or η lines). A cross hatch of red line segments is used to highlight a particular node on the segment. This highlight can be moved with the arrow keys. The left and right arrow keys decrease and increase, respectively, the first index of the highlighted point; while the up and down arrow keys increase and decrease, respectively, the second index. The coordinates of the node that is highlighted are displayed in the lower left corner of the Current window; and in the upper right corner are displayed the dimensions of the *current* segment.

Menu Window

The menu window contains an array of buttons broken into three sections. The top section is used to control the view of the *current* segment. These viewing manipulation buttons allow the user to rotate the segment, translate the segment, scale the segment, and change viewing direction. The last button in this section, labeled "BODY/GLOBAL", is used to toggle between different axes of rotation. When BODY is highlighted in red, rotations are defined about the body axes. When GLOBAL is highlighted, rotations take place about a global set of axes defined relative to the screen. To use the rotation, translation, or zoom buttons, pick the appropriate button. The button chosen will be highlighted in cyan and the cursor will disappear. Move the mouse horizontally for rotating, vertically for zooming, and in both directions for translating. Once a new view is set, click (press and release) any mouse button or keyboard key and the cursor will reappear.

The bottom section contains two buttons: HELP and EXIT. To access the help facility, pick the HELP button. A prompt will request that a second button be picked. The HELP facility will then display, in the Current window, all available information relating to the function represented by the second button. Clicking a mouse button or pressing a keyboard key will close the HELP facility. To exit execution of IGGy, pick the EXIT button. A prompt will request a Continue or Abort signal. Pressing the Enter key represents Continue and will cause execution to cease; while the Esc key represents Abort and will cause normal operations to resume.

The middle section displays the menu system. If a menu item represents a submenu, the submenu name with following ellipsis is displayed. Any menu item may be selected by picking the appropriate button. Picking a submenu displays the items of the submenu. To traverse back up one level in the menu system, you may either press Esc or pick the background area above or below the presently displayed menu options. If a submenu contains more options than can be displayed at once, the last menu button will display "more..." and the remaining options are displayed as a submenu of this button.

The menu system is organized into eight primary menus: "File..." contains all of the i/o commands; "Set..." contains all commands for setting parameters; "Curve..." contains all commands which generate curves; "C/aux..." contains all commands which operate on or alter curves; "Surface..." contains all commands which generate surfaces; "S/aux..." contains all commands which operate on or alter surfaces; "Parametric..." contains all commands which deal with parametric space; and "Misc..." contains all other commands or IGGy specific functions which do not fit into one of the other menus. Some commands operate on both curves and surfaces, and thus have been duplicated in the menu system. If a curve is displayed in *current* position, picking C/aux... will display all of the commands appropriate for altering it; likewise, if a surface is in *current* position, picking S/aux... will display all of the commands appropriate for altering it.

When using the menu system to build commands and a character value is being prompted for, the legal responses are displayed in a menu. Some non-character variables may also be assigned character values. In this case, a menu of the legal responses is displayed. Picking any of these menu choices causes the appropriate response to be entered into the command being built.

Parameters Window

The Parameters window displays lists of all the indirectly addressable parameters that have been set by the user. The window is separated into three sections corresponding to the three types of parameters: vector, integer, and real. Along the left side of each section is a pair of up and down arrow buttons. These buttons are used to scroll through the parameter lists. When building commands using the menu system, and a vector, integer, or real value is being prompted for, picking a value of the appropriate type from the parameter lists causes the corresponding storage index to be referenced using the proper syntax.

Status Window

The Status window displays the status of various variables and parameters which affect the operation of IGGy. Displayed are the current value of SURFACE, which controls the operation of parametric mode, the current values of FORM and STYLE, which specify the format for file i/o, and the name of the journal file currently being processed. Also displayed are flags to denote that edge curves, bounding curves, or tab curves have been set; corner points have been set; and an axis curve has been set.

When building commands using the menu system, and an integer value is being prompted for, the Status window is cleared and a slider for specifying integer values is displayed as shown in Figure 2. Likewise, whenever a real or vector value is being prompted for, the Status window is cleared and a calculator is displayed as shown in Figure 3. These two facilities allow commands to be entered without switching between using the mouse and using the keyboard.

To operate the slider, pick the slider knob and move the mouse horizontally, while keeping the left mouse button depressed. The present value of the slider is displayed in the small outline, that is centered near the bottom of the window. The knob will wrap around the ends of the slider to allow specification of integers beyond the range displayed. Pressing the Enter key, after moving the slider knob, will cause the value of the slider to be entered into the command being built.

To operate the calculator, numbers are entered by picking the buttons of the numeric keypad, located on the right side of the window, and picking the Enter button. Since the calculator is based on Reverse Polish Notation (RPN), two values must be entered before a binary operator is specified. Values are entered from the bottom of the stack, which is displayed on the left side of the window and which contains three vector registers (only two of which are displayed) and thus nine scalar registers (three components for each vector). The center section of the calculator window displays the function keys. The tilde button in the upper left corner of the function keys toggles between the scalar functions and the vector functions and also designates a vector Enter from a scalar Enter. Values and vectors can be entered into the calculator by picking an index from the parameter lists or by picking the coordinate display section of the Current window. To designate that these values should be entered into the calculator rather than into the command which is currently being built, this picking should be done using the right mouse button instead of the left mouse button. Once the desired value is calculated, pressing the Enter key will cause the value to be entered into the command being built.

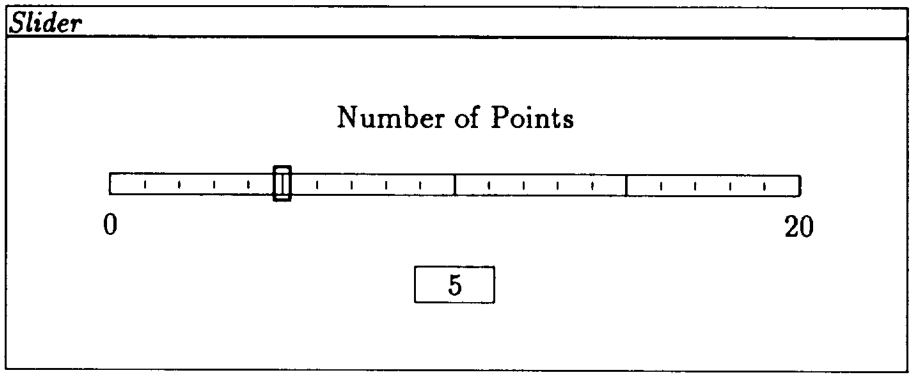


Figure 2: Integer slider.

Core Storage Window

The Core storage window displays wire frame drawings of the segments in core storage. This window is divided into six smaller viewports representing a small range of the core storage indices. The core storage index corresponding to each segment is displayed in the upper left corner of the appropriate viewport. When using the menu system, picking the appropriate viewport causes the corresponding core storage index to be inserted into the command being built. Since only six segments are displayed at once, four buttons are displayed along the right side of the Core storage window: up arrow, down arrow, left arrow, and right arrow. The left and right arrows step through core storage, one index at a time. The up and down arrows shift through core storage, six indices at a time.

The segments displayed in the Core storage window are scaled globally. This ensures that all segments are visible, but causes some segments to become indistinguishable by sight if there exists a great disparity among the size of the segments. In the upper right corner of each small viewport are displayed the dimensions of the segment stored at that index. This can be helpful in identifying smaller segments.

The only viewing manipulations available for the core storage are rotations of 90 degrees. Picking a core segment using the middle mouse button causes a horizontal rotation. Picking a core segment using the right mouse button causes a vertical rotation. The symbol in the lower left corner

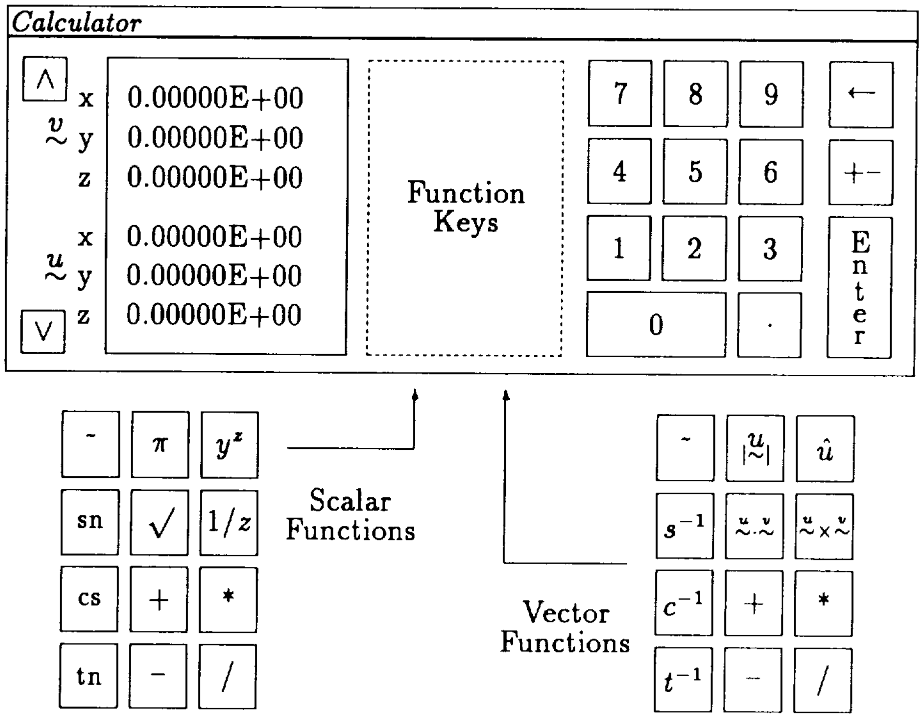


Figure 3: RPN vector/scalar calculator.

of each viewport displays the orientation of the axis of each viewport.

Console Window

The Console window is a text port which allows the user to enter any legal command from the keyboard. When executing commands from the menu system, the Console window is also used to prompt the user for input. For any input variable, the user may use the mouse to access the slider, calculator, or menu choices, or may type the appropriate response from the keyboard.

A command history facility is accessible using Ctrl-up arrow and Ctrl-down arrow. Holding down the Ctrl key and pressing the up arrow key displays the command line previously entered. Successive presses of the up arrow displays commands from earlier in the execution process. Holding down the Ctrl key and pressing the down arrow key displays commands from later in the execution process.

Table 5 shows the line editor commands available from the Console window. All of the commands are implemented as single key control sequences.

Table 5: Command Line Editor Keystrokes

Ctrl-a	move to beginning of line
Ctrl-b	move back one character
Ctrl-d	delete character below the cursor
Ctrl-e	move to end of line
Ctrl-f	move forward one character
Ctrl-h	delete character to left of cursor
Ctrl-k	kill from character to end of line
Ctrl-u	delete the entire line

Hints Window

The Hints window is used to display hints about the operation of IGGy. The text displayed is controlled by the placement of the cursor on the screen. As the cursor is moved around the screen, the message displayed describes the button or window beneath the cursor.

Whenever the slider or calculator is displayed in the Status window, the Hints window is also cleared and two buttons are displayed as shown in Figure 4. These buttons are the graphical equivalents to the Enter and Esc keys. The Abort button can be picked at any time that it is displayed to abort the command currently being built and to force a return to normal operations.

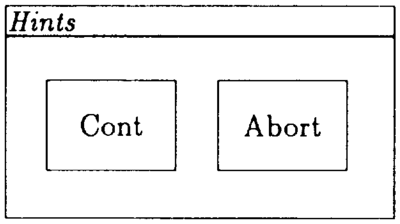


Figure 4: Continue or abort buttons.

EXAMPLE APPLICATIONS

NACA 0012 Airfoil

The following example script generates a simple two-dimensional C-type grid about a symmetric airfoil. This example corresponds to the first example given in reference [1] and is presented to highlight the advantages of using the generalized parameters. The script closely imitates the original example for comparison purposes.

Figure 5 shows the boundaries of the configuration used. The vector parameters that are used as point locations are designated by the appropriate indices inscribed in circles. Each pair of points delineate a boundary segment that is created in the script. Throughout the script, integer parameters are used to define the number of points along the boundary segments and the point distributions are defined using spacings stored in real parameters. In addition to these parameters, two real parameters are used to define the distances from the airfoil to the outer boundary.

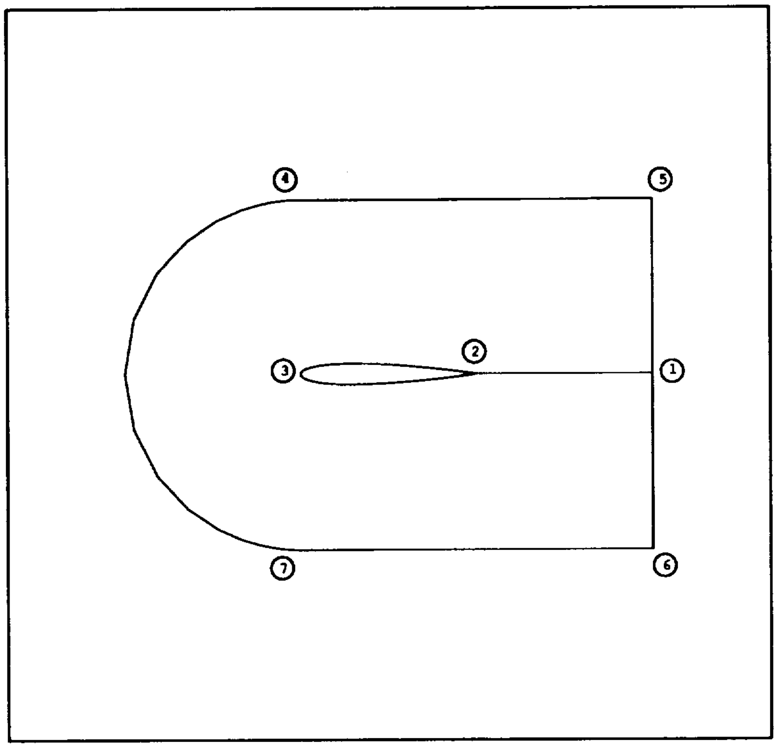


Figure 5: Boundaries of C grid about symmetric airfoil.

Looking at Figure 6, the comments at the beginning of the script show that the real value stored at index 5 defines the radius of the circular arc segment used in defining the outer boundary, and the real value stored at index 6 defines the distance from the trailing edge of the airfoil to the exit plane. The use of these parameters allows the user to alter the distance to the outer boundary. The remaining parameter setting commands allow the grid to be redimensioned and allow grid point clustering to be altered as with EAGLE.

After the initial parameters setup, the shape of the symmetric airfoil is read in. The first and last points on this airfoil shape are then extracted to define points number 2 and 3. The locations of points 1, 5, and 6 are then calculated relative to the trailing edge of the airfoil. Note, the distance stored in real index 6 is used to define the offset of point 1 in the x direction, and the radius stored in real index 5 is used to define the offset of points 5 and 6 in the +y and −y directions, respectively. Without the use of vector expressions, points 1, 5, and 6 would have been defined explicitly.

The circular arc segment used in defining the outer boundary is generated at the beginning of Figure 7. Without the use of the generalized parameters, an explicited value would have been given for 'RADIUS'. With this key parameter buried in the interior of the script, the possibilities for producing errors, when altering the grid, are increased.

```
*
*  set radius of circular arc
*
E$INPUT ITEM='SETREAL', INDEX=    5, RVAL=30. $
*
*  set distance to downstream boundary
*
E$INPUT ITEM='SETREAL', INDEX=    6, RVAL=29. $
*
*  set appropriate integer values for use as numbers of points
*
E$INPUT ITEM='SETINT', INDEX=    9, IVAL=81 $
E$INPUT ITEM='SETINT', INDEX=    1, IVAL=!9+!9-1 $
E$INPUT ITEM='SETINT', INDEX=    2, IVAL=31 $
E$INPUT ITEM='SETINT', INDEX=    3, IVAL=20 $
E$INPUT ITEM='SETINT', INDEX=    7, IVAL=141 $
E$INPUT ITEM='SETINT', INDEX=    6, IVAL=41 $
E$INPUT ITEM='SETINT', INDEX=    5, IVAL=!2+!1+!2-2 $
E$INPUT ITEM='SETINT', INDEX=    8, IVAL=!5-!6-!7+2 $
*
*  set appropriate real values for use as spacings
*
E$INPUT ITEM='SETREAL', INDEX=    1, RVAL=.001 $
E$INPUT ITEM='SETREAL', INDEX=    2, RVAL=.01 $
E$INPUT ITEM='SETREAL', INDEX=    3, RVAL=.0001 $
E$INPUT ITEM='SETREAL', INDEX=    4, RVAL=.0015 $
*
*  read airfoil shape
*
E$INPUT ITEM='CURRENT', FILEIN=12, FORM='LIST', STYLE='XYZ', POINTS=100 $
*
*  extract points 2 and 3 from the airfoil definition
*
E$INPUT ITEM='GETVEC', POINT='FIRST', INDEX=    3,
   VECTYP='POINT' $
E$INPUT ITEM='GETVEC', POINT='LAST', INDEX=    2,
   VECTYP='POINT' $
*
*  calculate location of points 1, 5, and 6
*
E$INPUT ITEM='SETVEC', INDEX=    1, VVAL=¯2+{#6,0.,0.} $
E$INPUT ITEM='SETVEC', INDEX=    5, VVAL=¯1+{0.,#5,0.} $
E$INPUT ITEM='SETVEC', INDEX=    6, VVAL=¯1-{0.,#5,0.} $
*
*  distribute points on the airfoil
*
E$INPUT ITEM='CURDIST', POINTS=!  9, DISTYP='BOTH', SPLTYP='QUAD',
   SPACE=#   4,#   2, RELATIV='NO','NO' $
E$INPUT ITEM='OUTPUT', COREOUT=    9 $
*
E$INPUT ITEM='SWITCH', REORDER='REVERSE1' $
E$INPUT ITEM='SCALE', SCALE={1.,-1.,1.} $
E$INPUT ITEM='INSERT', COREIN=    9 $
E$INPUT ITEM='OUTPUT', COREOUT=    1 $
```

Figure 6: IGGy script for C grid about symmetic airfoil.

```
*
*  generate circular arc segment
*
E$INPUT ITEM='CONICUR', TYPE='CIRCLE', POINTS=100, RADIUS=#   5,
   ANGLE=270.,90. $
E$INPUT ITEM='CURDIST', POINTS=!   7, DISTYP='BOTH', SPLTYP='QUAD',
   SPACE=#   3,#   3, RELATIV='YES','YES' $
E$INPUT ITEM='OUTPUT', COREOUT=    7 $
*
*  extract points 4 and 7 from ends of circular arc segment
*
E$INPUT ITEM='GETVEC', POINT='FIRST', INDEX=    7,
   VECTYP='POINT' $
E$INPUT ITEM='GETVEC', POINT='LAST', INDEX=    4,
   VECTYP='POINT' $
*
*  extract the spacing generated at the ends of the circular are segment
*
E$INPUT ITEM='GETSPA', END='FIRST', INDEX=    7 $
*
E$INPUT ITEM='LINE', POINTS=!   6, R1=^   7, R2=^   6 $
E$INPUT ITEM='CURDIST', POINTS=!   6, DISTYP='TANH', SPLTYP='LINEAR',
   SPACE=#   7, RELATIV='NO' $
E$INPUT ITEM='SWITCH', REORDER='REVERSE1' $
E$INPUT ITEM='OUTPUT', COREOUT=    6 $
*
E$INPUT ITEM='LINE', POINTS=10, R1=^   4, R2=^   5 $
E$INPUT ITEM='CURDIST', POINTS=!   8, DISTYP='TANH', SPLTYP='LINEAR',
   SPACE=#   7, RELATIV='NO' $
E$INPUT ITEM='OUTPUT', COREOUT=    8 $
*
E$INPUT ITEM='CURRENT', COREIN=    6 $
E$INPUT ITEM='INSERT', COREIN=    7 $
E$INPUT ITEM='INSERT', COREIN=    8 $
E$INPUT ITEM='OUTPUT', COREOUT=    5 $
*
E$INPUT ITEM='LINE', POINTS=10, R1=^   2, R2=^   1 $
E$INPUT ITEM='CURDIST', POINTS=!   2, DISTYP='TANH', SPLTYP='LINEAR',
   SPACE=#   2, RELATIV='NO' $
E$INPUT ITEM='SWITCH', REORDER='REVERSE1' $
E$INPUT ITEM='OUTPUT', COREOUT=    2 $
*
E$INPUT ITEM='LINE', POINTS=10, R1=^   1, R2=^   6 $
E$INPUT ITEM='CURDIST', POINTS=!   3, DISTYP='TANH', SPLTYP='LINEAR',
   SPACE=#   1, RELATIV='NO' $
E$INPUT ITEM='OUTPUT', COREOUT=    3 $
*
E$INPUT ITEM='LINE', POINTS=10, R1=^   1, R2=^   5 $
E$INPUT ITEM='CURDIST', POINTS=!   3, DISTYP='TANH', SPLTYP='LINEAR',
   SPACE=#   1, RELATIV='NO' $
E$INPUT ITEM='OUTPUT', COREOUT=    4 $
*
E$INPUT ITEM='COMBINE', FILEOUT=1, COREIN=1,2,3,4,5, FORM='E',
   STYLE='CONTENT' $
E$INPUT ITEM='END' $
```

Figure 7: Continuation of IGGy script above.

As seen in the 'CURDIST' statement following the 'CONICUR' statement, relative spacings are used to define the distribution of points along this circular arc segment. To match this spacing when generating neighboring segments, a 'GETSPA' command is used to extract the resulting absolute spacing at the ends of the segment and to store the spacing at real index 7. The remainder of the boundary segments are then generated using the defined parameters and are written to a file for importing to the grid code.

Multi-Store Interference Configuration

The following example addresses a multi-block grid about generic missile shapes in the mutual interference configurations whose frontal views are shown in Figure 8. Assuming a zero angle of attack and no interference from outside sources, the flow analysis can be performed on a small segment of the entire geometry. The dotted lines of Figure 8 can be drawn due to geometric symmetry; the dashed lines can be drawn due to the assumption of zero angle of attack. This reduces the flow field to a 90 degree wedge for the two store case and a 60 degree wedge for the three store case. With such a configuration, effects of separation distance and toe in or toe out angle can be investigated.

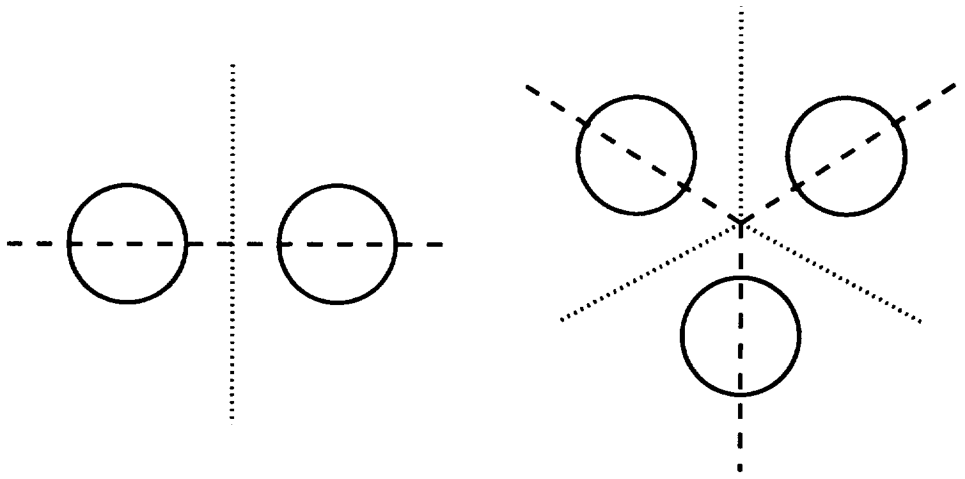


Figure 8: Two and three store configuration.

The script used to produce the boundary segments for this configuration is too long to be included in its entirety; however, Figure 9 shows the parameter definitions that are placed at the beginning of this script. The real value parameter stored at index 16 defines the angle between the planes of symmetry and thus allows the two and three store cases to be generated from the same script. As the comments imply, other real value parameters are used to control the location of the outer boundary, the separation distance between stores, and the toe in or toe out angle. No point locations are specified explicitly; rather, all necessary points are defined relative to the geometric definition of the store.

Figures 10 and 11 display perspective views of example boundary grids generated using IGGy. Figure 10 is the two store configuration; while, Figure 11 is the three store configuration. These grids consist of two blocks; and the outer boundaries are moved in close to the store for plotting

```
*
*  this runstream generates the boundary surfaces for a two block
*   grid about a ogive/cylinder/ogive/sting missile configuration.
*   two store or three store configurations are possible
*
*  length of the sting
*
E$INPUT ITEM='SETREAL', INDEX=   11, RVAL=60. $
*
*  length of the stagnation line
*
E$INPUT ITEM='SETREAL', INDEX=   12, RVAL=60. $
*
*  distance from reflection plane to centerline of missile
*
E$INPUT ITEM='SETREAL', INDEX=   13, RVAL=1.8 $
*
*  toe in or toe out angle.  toe in is positive. abs(angle) <= 5 deg.
*
E$INPUT ITEM='SETREAL', INDEX=   14, RVAL=0.*pi/180. $
*
*  distance from inner block to outer boundary
*
E$INPUT ITEM='SETREAL', INDEX=   15, RVAL=60. $
*
*  angle of rotation of vertical plane of symmetry.  this angle is
*   zero for the two store configuration and 30 deg for three stores.
*
E$INPUT ITEM='SETREAL', INDEX=   16, RVAL=0.*pi/180. $
*
*
*  set numbers of points along segments
*
E$INPUT ITEM='SETINT', INDEX=    1, IVAL=24 $       along the nose
E$INPUT ITEM='SETINT', INDEX=    2, IVAL=30 $       along the shaft
E$INPUT ITEM='SETINT', INDEX=    3, IVAL=15 $       along the tail
E$INPUT ITEM='SETINT', INDEX=    4, IVAL=23 $       along the sting
E$INPUT ITEM='SETINT', INDEX=    5, IVAL=15 $       1/3 around missile
E$INPUT ITEM='SETINT', INDEX=    6, IVAL=15 $       2/3 around missile
E$INPUT ITEM='SETINT', INDEX=    7, IVAL=!5+!6-1 $  total around missile
E$INPUT ITEM='SETINT', INDEX=    8, IVAL=32 $       normal to missile
E$INPUT ITEM='SETINT', INDEX=    9, IVAL=25 $       along stagnation line
E$INPUT ITEM='SETINT', INDEX=   10, IVAL=!9+!1+!2+!3+!4-4 $  total along x-axis
E$INPUT ITEM='SETINT', INDEX=   11, IVAL=20 $       j direction of outer blk
*
*  set spacings
*
E$INPUT ITEM='SETREAL', INDEX=    1, RVAL=.04 $    tip of nose
E$INPUT ITEM='SETREAL', INDEX=    2, RVAL=.08 $    base of nose
E$INPUT ITEM='SETREAL', INDEX=    3, RVAL=.08 $    base of shaft
E$INPUT ITEM='SETREAL', INDEX=    4, RVAL=.1 $     base of tail
E$INPUT ITEM='SETREAL', INDEX=    5, RVAL=.005 $   normal to missile
E$INPUT ITEM='SETREAL', INDEX=    6, RVAL=5. $     begn & end i dir outr bndry
E$INPUT ITEM='SETREAL', INDEX=    7, RVAL=.1 $     normal to v reflec pln
```

Figure 9: Parameter definitions for multi-store configuration.

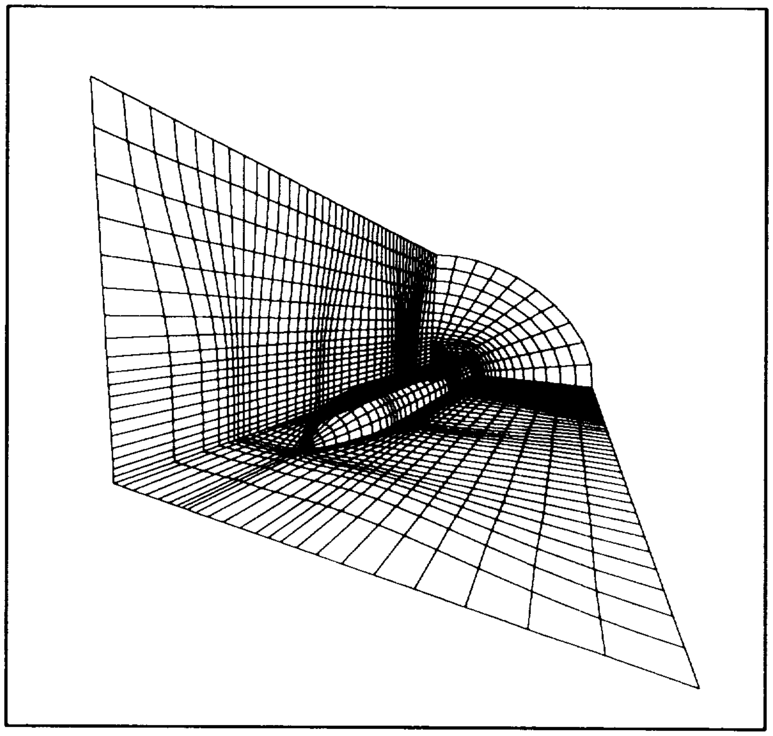


Figure 10: Example boundary grids for two store configuration.

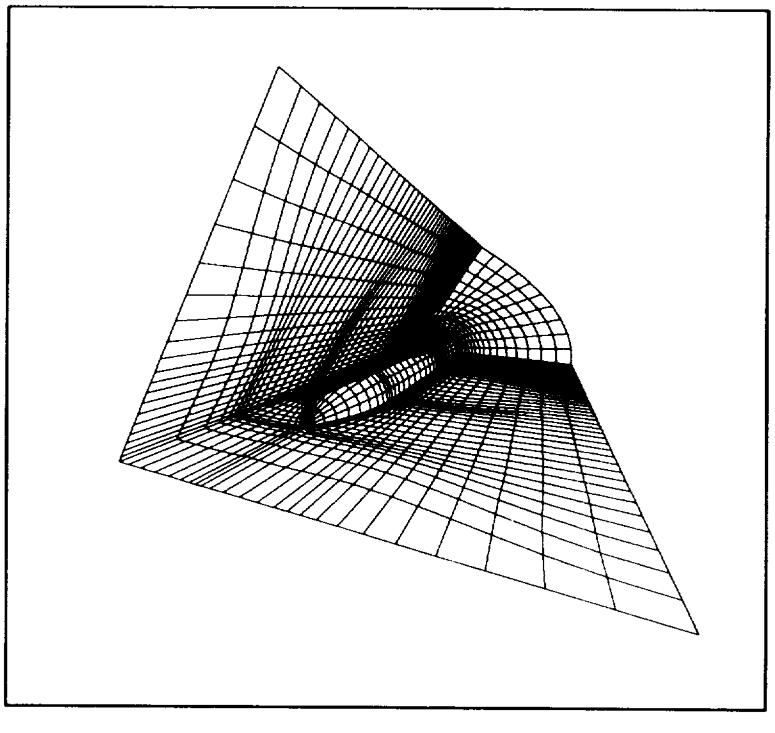


Figure 11: Example boundary grids for three store configuration.

purposes. Both of these grids were generated by altering only the parameters shown in Figure 9, with the only difference between the two being the value assigned to the real parameter stored at index 16.

CONCLUSIONS AND RECOMMENDATIONS

Many of the ideas relating to the generation of the needed parser and the syntax of the scalar and vector expressions were taken from SDL [8], a general purpose language for surface grid generation. Whereas EAGLE provides a user-friendly command structure with limited flexibility, and SDL provides the complete flexibility of a programming language (including looping constructs, conditional statements, and subroutines), IGGy has sought a compromise in flexibility and power while retaining the user-friendly command structure and providing a user-friendly graphical environment.

The recent trend in extensions to the EAGLE code has involved the generation of new commands which are built around the capabilities of the original commands. These new commands essentially automate some function which would have previously taken several commands to accomplish. This trend goes against the philosophy taken in IGGy. Rather, it would be preferable to produce a macro language which would allow the user to produce such capabilities from the commands which already exist. Such a macro language would not produce the flexibility of a programming language but would give the user the added flexibility to tailor the code to a particular application.

REFERENCES

1. Lijewski, L. E., Cipolla, J., et. al., "Program EAGLE User's Manual Volume I - Introduction and Grid Applications", AFATL-TR-88-117, September 1988.
2. Thompson, J. F. and Gatlin, B., "Program EAGLE User's Manual Volume II - Surface Generation Code", AFATL-TR-88-117, September 1988.
3. Thompson, J. F. and Gatlin, B., "Program EAGLE User's Manual Volume III - Grid Generation Code", AFATL-TR-88-117, September 1988.
4. Eiseman, Peter R., "Applications of Algebraic Grid Generation", AGARD Fluid Dynamics Panel Specialists' Meeting on Applications of Mesh Generation to Complex 3-D Configurations, Loen, Norway, May 1989.
5. Thompson, J. F., Lijewski, L. E., and Gatlin, B., "Efficient Applications Techniques of the EAGLE Grid Code to Complex Missle Configurations", AIAA-89-0361, 27th Aerospace Sciences Meeting, Reno, Nevada, January 1989.

6. Johnson, S. C., "Yacc: Yet Another Compiler Compiler", Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

7. Lesk, M. E., "Lex - A Lexical Analysis Generator", Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

8. Maple, Raymond C., "SDL - A Surface Description Language", NASA Workshop on Software Systems for Surface Modeling and Grid Generation, NASA CP- 3143 , 1992.