REQUIREMENTS FOR A GEOMETRY PROGRAMMING LANGUAGE FOR CFD APPLICATIONS

Arvel E. Gentry
Aerodynamics Research, Engineering Division
Boeing Commercial Airplane Group
Seattle, WA

SUMMARY

A number of typical problems faced by the aerodynamicist in using computational fluid dynamics are presented to illustrate the need for a geometry programming language. The overall requirements for such a language are illustrated by examples from the Boeing Aero Grid and Paneling System (AGPS). Some of the problems in building such a system are also reviewed along with suggestions as to what to look for when evaluating new software products.

INTRODUCTION

In this workshop we have an opportunity to hear about a number of new and exciting software efforts as developers are responding to our needs. However, we can sometimes also learn from history. The material presented in this paper is part history in that it explains what was done and why. It also contains some ideas as to what could be done if starting all over. But first, some history.

Existing CAD/CAM systems are devoted to the creation of computer definitions of hardware parts for manufacturing purposes. These systems vary from the small PC-based programs to giant corporate systems that are approaching a paperless design/build process. Although these types of systems might do what they were designed to do very well, they do not generally contain the capabilities needed for the efficient production use of computational fluid dynamics (CFD) applications. They also usually cannot respond to the immediate and changing needs of the aerodynamicist. CFD work requires much more than just having the surfaces of a vehicle described in the computer. A great variety of problems are faced in the aerodynamic design process. Some of these problems can be handled by a CAD system. Many others cannot. The aerodynamicist has to solve similar problems over and over again as aerodynamic design is refined. We have also found that it is very difficult to communicate our CFD needs to CAD/CAM people. This is understandable since a corporation may live or die by its CAD/CAM system, and the needs of the CFD community are very small when compared with the bigger airplane design and manufacturing picture.

In the late '70s at Boeing we could see that the newer CFD codes would soon outstrip the capabilities of our older geometry tools. Even the surface paneling of complex configurations was still a very time-consuming task. Examination of emerging CAD/CAM systems did not show much promise. After several years of mathematical research, a careful study of user needs, and several experimental pilot codes, we decided to build a new system that we initially called the Aero Surface Geometry System (ASGS); the name was later changed to the Aero Grid and Paneling System (AGPS).

The first version of AGPS was running in late 1980 and used a DEC VAX-780 with Vector General graphics devices. After a period of low level development and testing, the program went into production use near the end of 1983. The system has been greatly expanded in the years since. Additions to the

system were almost entirely driven by user needs. The system was first ported to workstations in 1984-86 and to various Unix platforms in 1987-91. A version also runs on the Cray Y-MP. Today there are several hundred active AGPS users on a variety of computer platforms.

AGPS is not available for general use outside of Boeing. Two previous papers have been published on AGPS (1,2). Several other presentations at this workshop discuss different aspects of the AGPS system.

The author of this paper was responsible for the creation of the AGPS program. He was also responsible for the geometry portion of a system called the AXXYZ Integrated CAD/CAM Software. This paper reviews the experiences gained over the past eleven years with these and other related projects and suggests functionality and system design aspects that might be useful to those developing new systems.

This paper first presents a number of examples to show just why a geometry programming language is needed. This is followed by a presentation of the high-level requirements for each aspect of the task. These include the user interface, basic mathematics, surface lofting tools, data structure, support functionality, and input/output systems. Some comments are provided relative to code portability, system maintenance, training and user support. During the workshop many of the ideas discussed here will be illustrated by live demonstrations of the current version of the AGPS program.

WHY A GEOMETRY PROGRAMMING LANGUAGE?

Early in the development of AGPS it was recognized that it was impossible to anticipate all of the types of problems that the system would have to eventually solve. The solution was to build a system that the user could extend and adapt to solve new problems. The only way that we could see to accomplish this was to devise a geometry programming language accessible by the end user. This approach allowed us to build a tool that would have useful functionality in a short period of time. By collecting the user written applications, refining them, and making them available to all users as an application library, we believed that we could greatly expand the functionality of the system at minimum cost.

The validity of this decision can best be demonstrated by reviewing a few examples that have been solved over the more than ten years that the program has been in use. Some of these problems are obviously geometry problems. Many are examples of how a geometry programming language can be used in place of FORTRAN to efficiently solve other problems. The following examples might also serve as test problems for new geometry software systems.

Geometry Problems

Wing Lofting Problem (Figure 1) - A wing loft is to be generated from basic shape defining information. The loft process is to be repeated a great number of times with different input data as the design progresses. Defining information includes:

- Wing planform information (plan view of leading edge and trailing edge lines).
- A stack of airfoils to be positioned at given span locations. (the airfoil data is normalized by chord length and thickness.)
- Wing twist distribution plus the definition of the line about which the airfoils are to be twisted.
- Wing thickness distribution as a function of span location.
- Shear distribution (how each airfoil is to be displaced vertically). After the basic wing is lofted, investigate the changes necessary in the shear distribution to make the trailing edges of the slats straight lines in the jig loft position.

Working With Digitized Data (Figure 2) - A wind tunnel model has been measured using a surface digitizing process. Convert the data to mathematical surfaces so that the configuration can be analyzed in our CFD codes. The slight discrepancies in measurement and in model alignment must be corrected in the process. Some areas of the model were not covered in the digitizing process and will have to be created with the limited available information.

Working With Airplane Measurement Data (Figure 3) - Photogrammetry and theodolite methods have been used to measure a research aircraft. In addition, large sheets of clear acetate were laid over the 3-D wing and the location of the photogrammetry points marked on the acetate along with the location of all screws and access ports. Digitize the data on the acetate sheets and convert to the real 3-D coordinates on the aircraft and compare with the original airplane drawings.

Develop Design Goal Cp Distribution for HLFC Glove (Figure 4) - An inverse CFD code is to be used to design the shape for a hybrid laminar flow glove (HLFC) on a research aircraft. Develop a method for examining the initial analysis pressure distribution and then generate the new design goal Cp input data.

Surface Comparisons (Figure 5) - An HLFC glove for a research aircraft has been designed. Determine the clearance between the glove and the basic hard wing.

Paneling Problems

A great variety of paneling problems must be handled by a geometry system for CFD applications. Rather than discussing individual examples, the point can be made by looking at the paneling examples in Figure 6. Each of these paneling problems was solved by procedures written with the AGPS geometry programming language. The important thing to note here is that, once the required surfaces were available, all of these paneling operations were accomplished automatically in a few minutes. The results were paneled data sets, or in many cases actual final data files ready for submittal to a panel code. Of course, the basic problem is that each paneling procedure written in the geometry programming language handles a given set and arrangement of surface components. If the arrangement of components is different from that provided for in the paneling procedure, then the procedure will have to be modified or rewritten. With a geometry programming language, the end user can do the modifications or rewrite the procedure in a short period of time. Without a geometry programming language system, you would have to do this by writing separate FORTRAN codes for each application, or in the worst case go back to the original code developer and request the required new functionality.

Grid Generation Problems

The use of a geometry programming language to attack field grid generation problems is described by another paper presented at this workshop so I will not dwell on this subject here * (see "An Interactive Multi-Block Grid Generation System, by T.J. Kao, T.Y. Su, and R. Appleby). Only a few comments on grid generation will be made at this time.

The generation of field grids is a major problem in the CFD community. Grid generation is still a very time-consuming process. As can be seen by the many grid generation papers presented in this workshop, we have about as many approaches as we have developers. Although the word GRID is prominent in the title of the AGPS program, we do not claim that AGPS is the last word in this area. We at Boeing use a variety of grid generation tools in addition to AGPS. However, we feel that a geometry programming language can play an important role in grid generation. Accurate grids must start with an accurate representation of the surfaces themselves. This is lacking in some of the early grid systems that we have seen. Many of the new grid generation programs are highly interactive. This often means that to

*See page 333.

solve a similar problem but with a new set of surfaces, you have to go through the whole laborious interactive process all over again. This may be fine for the CFD researcher or even for the user who only has one problem to solve. However, in a real aircraft design situation this is not acceptable. Many design iterations have to be made. Some grid generation systems have recognized this problem but their journaling and language capabilities are still rather crude.

The grid generation process must be reduced to an almost transparent part of the design process before the new advanced CFD tools really become productive. Many CFD researchers and code developers have included geometry operations and grid generation methods within the CFD codes themselves. This seems to achieve the goal of having the gridding task be transparent to the user. However, experience has shown that this approach severely limits the applicability of the program. And at the risk of offending a few people, frequently the CFD code developers have clearly been lacking in knowledge of the latest curve and surface mathematics technology.

Post-Processing Problems

The plotting and visualization of CFD results is an important part of the aerodynamic design process. We started to build a separate system to do this soon after AGPS became operational. We called it the Aero Flow Imaging System (AFIS). At first AFIS was developed along the same command and program structure concepts as had been used for AGPS. This would minimize the user training and give the same flexibility as was available in AGPS. As the development of AFIS proceeded we found that we were copying more and more from the AGPS program. When we realized this, we stopped the development of AFIS and simply included the necessary display commands and functions in the AGPS system itself. Since then other programs such as PLOT3D and FAST have been built specifically for the post-processing task. However, we still find that many post-processing tasks are best handled with the capabilities provided within a geometry programming language such as AGPS. Several of these are illustrated below.

Combined Data Display (Figure 7) - On a single plot display the geometry, paneling, panel method pressures, and surface streamlines for a complex configuration.

Surface Porosity Visualization (Figure 8) - A porosity measuring device has been used to measure the skin of an HLFC glove on a flight test aircraft. Rows of flutes lie under the porous skin and provide channels for the air that is removed from the boundary layer. The porosity measuring device was placed at intervals along these flutes and the porosity of the skin measured. Prepare a display showing the rectangular area for each measurement point and fill the rectangles with different colors to represent the surface porosity.

Vehicle Trajectory Visualization (Figure 9) - Output from a trajectory optimization program must be visualized. The user would like to see the trajectory flight path in 3-D space, the orientation of the vehicle along the flight path, and the variation of selected flight parameters. Many plots are to be routinely produced automatically for a variety of flight simulations.

Flight Path Clearance Problem (Figure 10) - Visualize the takeoff flight path necessary to clear nearby mountainous terrain.

Geometry Programming Example

We have found that a geometry programming language can be used for a very wide range of what might be thought of as non-geometry problems. The example below serves as a good illustration of these types of applications.

Momentum Thickness Reynolds Number Problem (Figure 11) - An Euler code has been used to calculate the flow about a configuration. Hybrid Laminar Flow Control (HLFC) studies need a quick

estimate of the momentum thickness Reynolds number along the wing leading edge attachment line. The CFD code output includes the inviscid flow properties. The momentum thickness Reynolds Number equation needs the flow velocity along the attachment line and a velocity derivative term calculated in a plane perpendicular to the attachment line. Figure 12 is a partial listing of the solution of this problem and can be studied to better understand and appreciate the great power of a geometry programming language such as AGPS.

GEOMETRY PROGRAMMING LANGUAGE REQUIREMENTS

The requirements for a geometry programming language would obviously fill a large volume. This paper is only an outline of some of the most significant issues. Again, many of these ideas have been gained not only in the over ten years of experience with AGPS but also in the development of a Boeing Computer Services CAD/CAM product called AXXYZ Integrated Software.

Early in the development of AGPS we recognized that a geometry programming language would face many of the same problems of other computer languages including:

- Training
- Language help systems
- Debugging procedures
- Language stability over time
- Code portability
- User support
- Application maintenance

The key feature to our approach to these problems was the decision to allow the user to either type in a command line or enter a command using a menu system, which then executed the command immediately. All of the system's graphics and language features could then be used to verify that the command or group of commands did what you wanted. The ability to write the commands entered during an interactive session to a file provides a means of saving part or all of an interactive session. This procedure we called journaling. The file could be edited to remove errors made in the interactive session. We then provided several ways to play back the file for repeated execution. The program could be operated purely by interactive operations, by running previously recorded procedures, or by a mixture of the two. These approaches were not new since this is the way that most computer operating systems work. In the initial design of the program, it was decided that not only should the code be highly interactive, but it should also run in the batch mode.

A few general requirements were a result of the computing environment that existed at the beginning of the program design. Initial pilots considered the use of a big mainframe computer. This was not possible at the time because of slow communications and lack of mainframe virtual memory capability. Workstations were not available when AGPS was initially developed. AGPS was originally developed by borrowing time on someone else's VAX-780 since our company division did not own one. We knew that computer hardware and graphics devices would probably change several times over the life of the program. The code had to be made as portable as possible, but the limited performance on the initial computer platform forced us to give up some portability for performance. We had to pay for this later on.

User Interface

Many aspects of the user interface determine the usability of a program. These include the workstation window system, graphics display and control functions, the use of the keyboard and mouse, etc. This discussion will only touch on those areas that are peculiar to the programming language aspects of the system.

Modern interactive programs tend to be menu driven where the developer has figured out or has been told about all of the options that the menus must hold. Some of these systems have been expanded to include a limited macro capability. The AGPS system was developed from the beginning as a command-line language system. Menu capabilities were added to improve user access to the system capabilities.

One criticism sometimes heard from first-time viewers of AGPS is that "it does not have a modern user interface". They apparently expect to see pop-up/pull-down menus with lots of icons, and with little if any text input. We have looked at this kind of user interface but have found it not suitable for a system that is essentially a programming language.

The heart of AGPS is the more than 160 commands. User access to these commands is an important part of the user interface. Most commands have a number of input parameters called keywords. Each keyword may have one or more options. Providing a convenient menu access to such a large number of commands and command options is no easy problem. In the current version of AGPS this is accomplished by providing a three level menu system. The user can switch between text input and menu mode whenever desired. The options in the highest level menu indicate the groupings of the different types of AGPS command functionality. Typical groupings are DATA STRUCTURE, DRAWING OBJECTS, and CREATE SURFACE.

Selecting the CREATE SURFACE option brings up the second level menu containing the names of all commands that create surfaces. Selecting a given method then brings up the third level menu that actually contains the command itself with its keywords and current defaults. The third level is not really a menu but an editing window where you type in or modify the keyword option inputs. Hitting the on-line help key at this stage brings up a separate help display that describes the command options along with several pages of more detailed description. Figure 13 is a screen dump of what the user sees. If users know the 3-character abbreviation of a command they can bring up the third-level command editing window without having to go through the first- and second-level menu selections. A potential enhancement to the AGPS third-level command window would be to allow the users to type in the object names required by the command and allow them to pick the other options from a menu. The text-input mode is frequently used for commands that the user is most familiar with, and it is faster. Menu mode is used for commands that the user might not be too familiar with.

A typical use of the Fit-Surface command in the text-input mode might appear as follows:

```
FIT-SURFACE SURF=SURF1 ARRAY=ARR1 DEGREE=CUBIC-B-SPLINE
```

Or, using the standard 3-character abbreviation (FSU in this case), and omitting the optional keywords by following the standard keyword order, this command could be typed in as:

```
FSU SURF1 ARR1 CUBIC-B-SPLINE
```

One of the main requirements for a geometry programming language is to make it as flexible as possible, including the menu system mechanics. We solved this in AGPS by having the contents of the menus driven from a text file. The users can copy the menu control file, called the geosyn.dat file, to their own account and customize it to meet their own needs or preferences. This provides a means of tailoring the program so that it looks like it was specifically written for a different application from the standard program defaults. Figure 14 contains an example of a menu system for use in designing an America's Cup yacht. The user can switch back and forth between the standard menu and alternate command system as often as required. These tailored and specialized menu systems may also include command files (customized macros) that give the user easy access to very complex operations. We call these specialized menu/command file systems "application packages" or "Task Menus". The default AGPS menu system includes several of these packages.

There are several user interface areas for a geometry programming language that do not fit

conveniently into a menu type of operation. These are the use of program flow control constructs (do-loops, if-then-else, etc.) and calls to specialized subroutines. These are best handled by text-input mode. However, it would not make sense to try to type in all of the lines of a large do-loop. A typing mistake means you have to abort the loop and type it all in again. It is for this reason that many complex procedures are frequently developed by a combination of typed-in, menu selected commands and command file fragments executed from an external text file created with an editor. A text editor that "knows" the syntax and commands of a geometry programming language can be a very helpful tool for the less experienced users. This idea was tried out using the Language Sensitive Editor provided on DEC VAX VMS computers. However, it was not pursued further because a similar editor was not available on any of the other host computer platforms that the program ran on.

Debugging a program is a problem common to all programming languages. This problem is handled in AGPS by a number of different techniques. Several AGPS commands can be used within a command file to help in the debugging process (MENU, PAUSE, EVALUATE, DISPLAY-FAMILY-TREE, $UNWIND, $WRITE, $ABORT). For example, the AGPS MENU command switches you from the text-input mode to the menu-input mode. The MENU command can be placed anywhere in a command file where you think you might need to interrogate the data structure, to plot data, or draw objects. When MENU is executed in a command file you are switched to the menu mode and can use any of the commands to help find out what the command file has accomplished to that point. Also because AGPS uses an interpretive language, portions of a new command file can be cut from an editing screen, pasted into the AGPS text input window, and other AGPS commands can be used to interrogate the data structure or draw objects to be sure that it did what you wanted (instead of what you thought you told it to do).

The ability to save all or portions of an interactive session is a key feature of a geometry programming language. This is called "journaling" and can be done in two different ways. One way is to save each command just as it is read by the command person from either the type-in or the menu mode. Reading the command file back into the program allows you to repeat the sequence of operations just as you did in the initial live session. If a command requires that an object be picked from the screen, then on the replay you will again be required to pick an object. This approach poses a problem when you have a complicated sequence of screen picking operations as is typical in a grid generation process. This problem is solved by a second type of journaling that not only saves the command executed but also saves the name of the object picked. During an interactive session the user should be allowed to switch between these two modes of journaling.

Many CAD systems draw everything that defines the current working entity to the screen. Items that you do not want to see are selected and made invisible. The opposite approach was used in the AGPS program. We knew that the data structure would frequently be very large, involving many strings of points, surfaces, surface grids/paneling, and space grids. We elected to place the burden on the user and have that person draw only those objects to the screen that are needed for the operations being performed. It might have been useful to attach line style and color to the different entity types as the case with many CAD systems but this was not done.

Most CAD systems allow several different views of an object on the screen at the same time. Multi-views are a desirable feature for a geometry programming language system, but were not provided in AGPS due to hardware/software limitations in the early years of its development. This capability would be a desirable future enhancement.

Mathematics

The current most popular approach for working geometry problems is non-uniform rational B-splines (NURBS). NURBS-based systems have the advantage over polynominal systems in that conic curves can be represented exactly without approximation. In a large corporate environment, the selection of the curve and surface mathematics for a geometry programming language is not necessarily a technical decision. You may have to provide capabilities to handle data from other and sometimes old outdated

systems. This was the situation with AGPS. We had to provide a variety of mathematics capabilities for most of the curve and surface generation functions. These varied all the way from procedurally defined surfaces (such as tubes, bodies of revolution, conic lofted surfaces), to NURBS-defined surfaces. There is also a requirement to be able to handle the variety of objects that are read in through IGES files from CAD systems. Regardless of how a curve or surface is stored in the program data structure, it is important that the rest of the program, such as the intersection routines, be able to work with the entities as though they are fully parametric curves and surfaces.

The form and details of the mathematics for working with parametric curves and surfaces are major subjects in themselves. This single area controls what can and cannot be done with the system. For example, the use of solids seems to be a general trend. Some CAD programs create solids by joining together surfaces with all of the edges matching up exactly. For CFD applications, however, we must approach the problem with care. How are the surfaces for the solid created? For example, in aerodynamic applications we would usually create the fuselage and wing surfaces so that they intersect each other. We would then find the intersection and trim off the portion of the wing that is inside of the body. The method used to trim the wing is important since you may not want to lose the original wing loft in the process. If you trim the wing using a method that has the effect of cutting off and losing the portion that is inside of the body, then what do you do when you apply an inverse design method to the body and find that you had trimmed off too much of the wing?

We have found that an entity type that we call a "subrange object" is very powerful for both geometry building and paneling/grid-generation processes. A subrange point consists of a pointer to a curve or surface and the parameter value specifying the location of the point. Many surface paneling operations can be accomplished simply by creating arrays of surface parameter values and a pointer to the surface itself. A subrange curve on another curve, or a subrange region of another surface is accomplished through a mapping process (Reference 1). This subrange capability allows curves or surfaces to be trimmed but retains the exact mathematical definition of the original object.

Surface Lofting Tools

The surface lofting area shares the most commonality with CAD systems. However, in CFD applications we have many strings of points representing airfoils, body cross-sections, and surface and space grids. It is cumbersome to work with this kind of data in many CAD systems. Also, most CAD systems are primarily devoted to the generation of hardware parts and are frequently rather limited in their ability to efficiently create the sculptured surfaces required in CFD applications.

A variety of surface lofting tools are needed in a geometry programming system. For curves, these should include the ability to fit splines through strings of points, the construction of classical conic curves, blending curves, and composite curves. Surface creation methods should include surface fit to a rectangular array of points (mesh), surface through curves, surface through intersecting networks of curves, a surface created by sweeping a cross-section along a control curve, plus the usual ruled and tube surfaces.

The ability to modify curves and surfaces is also important in the lofting process. This should also include the ability to convert between the different math forms (i.e., linear, polynominal based systems, NURBS).

The use of these powerful tools to create final lofts is not an easy problem. In the aircraft business, good experienced lofters are a vanishing breed. The use of CAD systems to create high quality surfaces requires a great deal of training and experience. Unfortunately, most aerodynamicists working lofting problems do not have this kind of background. The surfaces they create may have wiggles that are undesirable in the final product. These surfaces also may not meet manufacturing requirements. On the other side of the coin, surfaces generated by a designer on a CAD system may not meet aerodynamic requirements. The surfaces may look good on the screen but the arrangement and parameterization of the

surfaces may make it very difficult to work with them when you try to generate surface paneling or grids with automated tools. It is certainly possible to solve these problems entirely within the CAD environment, but the cost and flow time would be very high. Required changes to the CAD system to meet CFD requirements may take months or even years to achieve. A geometry programming language can be used effectively in attacking these problems providing the interface to and from the CAD systems is made as transparent as possible. Special tools must also be provided in the geometry programming language system for checking the quality of surfaces created by the aerodynamicist. These tools are sometimes lacking in CAD systems, and we have often found AGPS useful in checking the quality of surfaces generated in the CAD environment.

Data Structure

The data structure used in a geometry programming language will have a major impact not only on the performance of the system, but also on the users ability to learn and use the system. The data structure in AGPS was influenced greatly by the assumption that a variety of object types would be necessary and that interactive operations would involve frequently modifying the data. It was also important that certain objects in the data structure be of arbitrary dimension.

We thought that the primary building blocks would be strings of points that could be combined to form arrays to which surfaces would be fit. An array object would be defined as pointers to the locations of string objects, and a string would contain pointers to the individual points. If a point were modified, then the string and arrays would automatically be updated. Parametric surfaces fit to the arrays would involve the generation of surface patches for each set of four points in the arrays. The patches were stored without any reference to the original array. If a point was modified, the surface object would have to be deleted and a new surface fit to the array. The advantage of this approach is that the points, strings, and arrays are very easy to work with. The disadvantage of storing points, strings, and arrays in this manner is that it involves a lot of storage overhead. This became a problem when we started generating large field grids with many thousands of points. The solution was to add a data object to the system which reduced the overhead but required that the application programmer know how to find a specific point when needed.

In AGPS the user specifies the name for every object to be created. This has its advantages and disadvantages. Using logical names helps in understanding the process and in examining the data structure. An object called WING would be hard to mistake for something else. However, a special package of procedures usually has specific names assigned to objects that they create internally. This will sometimes cause conflicts when attempting to use a mixture of user generated operations and standard procedure packages. The object RENAME command becomes very useful in this situation.

The data structure of AGPS presently consists of over 30 object types. Additional object types are developed when needed. These object types can be grouped into several basic entity forms.

Geometric Representations:	Non-geometric Representations:
Points and Point-Matrices	Lists
Curves	Ostentations (show displays)
Surfaces and Solids	Text

The user of applications written with the AGPS geometry programming language does not normally have to know much about the data structure. However, the application developer must know a great deal about the data structure, and this has been one of the most difficult areas for the application programmer to learn. Commands such as the DIRECTORY, DUMP, and DISPLAY-FAMILY-TREE, are useful, but some new ideas in this area are really needed.

Support Functionality

The primary capabilities of a system are contained in the commands. AGPS has over 160 commands, some of which have previously been discussed. Identification of all of these commands is beyond the scope of this paper. A few of the other types of functions are discussed below.

Logic Control

A geometry programming language needs the usual do-loop and if-then-else constructs for controlling the solution logic. These constructs require that several lines of code be read in and stored before they are actually executed. If a mistake is made, you need some way of aborting the process and starting over. You also may have limitations on the number of commands used in a do-loop or if-then-else structured due to built-in buffer storage limitations. These problems were present in early versions of AGPS but are being removed in more recent versions.

Mathematical Expressions

The geometry programming language needs the ability to do mathematical calculations, and it is helpful if the syntax used is as close to FORTRAN as possible. The minimum set of math functions should include SIN, COS, TAN, ACOS, ASIN, ATAN, EXP, ABS, SQRT, LOG, and LOG10.

In AGPS we needed a way of separating the mathematical operations from the regular commands. This was done by placing a $-sign in front of all operations that were not to be processed by the regular command processor.

```
$X2=X*2.5
$Y2=Y*3.4/1.5
$Z2=Z*(X+Y)
REPLACE STR1.5 [(X2,Y2,Z2)]  ! Replace the string point STR1.5 with new values.
```

The variables X, Y, Z, X2, Y2, Z2 in the above examples are called symbols and their numerical values are stored in a buffer region. They are temporary variables and are not part of the regular object data structure.

Data Structure Access

The numerical values of objects stored in the data structure must frequently be retrieved so that they may be used in mathematical calculations. In AGPS this is accomplished with special $CALL directives. For example the directive GET_COORD will interrogate a curve or surface at the specified parameter location and place the coordinates in symbols.

A number of similar calls retrieve other data from the data structure or return information about the status of the system, such as computer platform identification, mode of operation, and picture viewing angles.

Miscellaneous Control Functions

Several other functions are needed to control command file processing:

- stop or continue command file processing when an error is found
- stop command file processing at this point
- exit from do-loop or if-then-else processing
- write text and symbol values to the screen

INPUT/OUTPUT CAPABILITY

A geometry programming language must have extensive capabilities to read and write data. Input data may arrive from a variety of external sources. Efficiency may also dictate that in-house standard or neutral files be readable.

The AGPS program has slowly expanded to include a number of input/output operations. Of primary importance is a flexible means of reading and writing strings of points in various formats. This input flexibility allows convenient interfaces with a variety of other codes. A number of special formats are also provided to allow direct processing of important CFD code output files such as PLOT3D.

AGPS has a WRITE-FORMATTED-DATA command with a variety of capabilities for format and output of data. Standard command files are available for the output of data in several standard Boeing formats such as WRITE-A502, WRITE-RMS, and WRITE-GGP. AGPS is frequently used in converting data file formats for other programs instead of using FORTRAN.

Every geometry or grid generation system should have the ability to read in at least a subset of IGES data that may originate from other systems. The Boeing experience, however, has shown that what is promised as conforming to the IGES standards may not always be correct. Parametric curves and surfaces from different systems may also not be parameterized in the same way. Some systems normalize parametric entities to 1.0 and others do not. The geometry system should have the ability to account for these variations. It should be able to reverse the direction of curve/surface parameterization and to refit the surface and change the mathematical representation when necessary. When this is necessary, tools should be available to check the accuracy of any surface refitting operations.

A geometry system should be able to save and restore all or part of a working session at any time. Many CAD systems automatically save data to disk at periodic intervals. Some systems also provide checkpoint restart capability. In AGPS we put the burden on the user to save data when desired. When restoring a file, object names should be checked by the system and conflicts with already existing objects automatically fixed and the user informed of all changes.

CODE PORTABILITY

The computer industry is still far from achieving standardization, particularly in the graphics programming area. Because the hardware manufacturers are always playing a game of one-upmanship, it is also hard to maintain a balance between achieving maximum performance on a given platform and maintaining portability between machines and systems. The long life of AGPS has presented a number of difficult portability problems. Work was started in 1986 to improve the portability of the system and this has paid significant dividends. Versions are now available on a variety of workstations including DEC VAXstations (VMS and Ultrix), Apollo, Hewlett Packard, Silicon Graphics, and IBM RS-6000. A version is also available on the Cray Y-MP. Currently we are heading toward the industry standard UNIX and X-windows environment.

SYSTEM MAINTENANCE

Program maintenance requirements are often not very well recognized by upper management. This is especially so in the case of a geometry programming language such as AGPS. Since the system is open-ended we are constantly seeing new applications and requirements. Company systems that we have to interface with such as CAD/CAM systems are evolving as are the CFD codes that we have to support.

Without a geometry programming language we would have to be spending a great deal more time and effort writing a myriad of computer programs and systems to keep up with the demands.

The first version of AGPS was written in nine months by a team of four people (a lead engineer, a mathematician, and two programmers). A great deal of effort has been expended over the past eleven years in system maintenance. The program has grown considerably in size although the original program structure and concepts have remained the same. Most of the program is still in FORTRAN. Some have suggested that a second generation version of AGPS be written using the C-language with object oriented programming concepts. This would be desirable but is not cost effective at the present time.

A major maintenance problem for any program is people. It is difficult for any programmer to learn and understand all aspects of a program as big and complex as AGPS. We have been fortunate in being able to keep an excellent experienced programming staff with minimum turnover.

The great success of a program also has its down side. A large number of users and applications over a long period of time imposes constraints on maintaining program stability. Because the aerodynamics community also controls the aerodynamic geometry programming language, changes are much easier to achieve than is the case with a big corporate CAD/CAM system. However, quick changes to the system are no longer as easy as they were earlier on. Capabilities may be required that cannot be achieved immediately within the AGPS system. As a result smaller special purpose programs spring up to meet specific needs. This frequently cannot be avoided since jobs have to get done. However, it does mean that overall maintenance costs for an organization will increase and the greater number of separate programs causes portability and maintenance problems.

TRAINING AND USER SUPPORT

A flexible and extendible system such as AGPS imposes some tough training and user support problems. AGPS is not as easy to learn as is the usual CFD program. Geometry is not a favorite subject for the aerodynamicist. It is a necessary evil in solving design and analysis problems. As a result, geometry experts in the aerodynamic community are few and far between. Designers working on CAD/CAM systems are frequently given weeks of training followed by a closely supervised apprenticeship period. AGPS training has been usually limited to a week or less. Subsequent training is accomplished on the job. The major drawback of this approach is that more support is required from experienced full-time experts. Another aspect of the training problem is that the aerodynamics users will frequently have to do complex lofting tasks for which they may not be adequately trained. The result will be bad aerodynamic lofts that get sent to the big corporate CAD/CAM system and then need to be fixed. One solution is to increase the lofting training of those aerodynamicists who will be designing complex surfaces. Another important solution is to provide the aerodynamicist with the proper tools to evaluate the quality of the surfaces that are created. Another possibility is to train the aerodynamicist in the use of the big CAD systems. For some type of surfaces the CAD system may be a good approach. However, for many other types of surfaces the geometry programming language may do the job quicker.

A geometry programming language does a lot more things than just lofting surfaces. These include paneling, grid generation, and special post-processing problems. In AGPS several approaches were used to attack the user training and support issues for this great variety of problems. We assumed that users with limited training and experience would primarily be using applications written by other more experienced people. The more experienced users could modify existing procedures and write new ones of their own using the help of full-time experts when needed. A small group of full-time experts would prepare larger heavily used applications for a broader user community. They would also serve as consultants, direct the computer programmers developing and extending the executable code itself, and test and validate new versions of the code. The full-time experts would also maintain a standard library of applications written with the language.

CONCLUSIONS

More than ten years of development and experience with the AGPS geometry programming language has proven that it can be a powerful tool for CFD applications. The key to this has been the ability to solve problems that not only were not thought of in the beginning, but that would require a much greater amount of resources to solve using any other approach. Problems that frequently required weeks or days to complete are now run on a routine basis in hours or minutes. The major down side is that AGPS has at times been viewed as competing with corporate CAD systems. The existence of two systems that seem to be doing the same job has bothered some managers. It has sometimes been hard to convince them that both systems have their purpose and that the best approach is to use the right tools for the right job. A seamless interface between these systems should be the desired end goal.

ACKNOWLEDGMENT

Special thanks to Robyn Wittenberg for her valuable suggestions and skills with AGPS in preparing material for this paper.

REFERENCES

1. Snepp, David K., and Pomeroy, Roger C.: A Geometry System for Aerodynamic Design. AIAA-87-2902, 1987.

2. Capron, W. K., and Smit, K. L.,: Advanced Aerodynamic Applications of an Interactive Geometry and Visualization System. AIAA-91-0800, 1991.

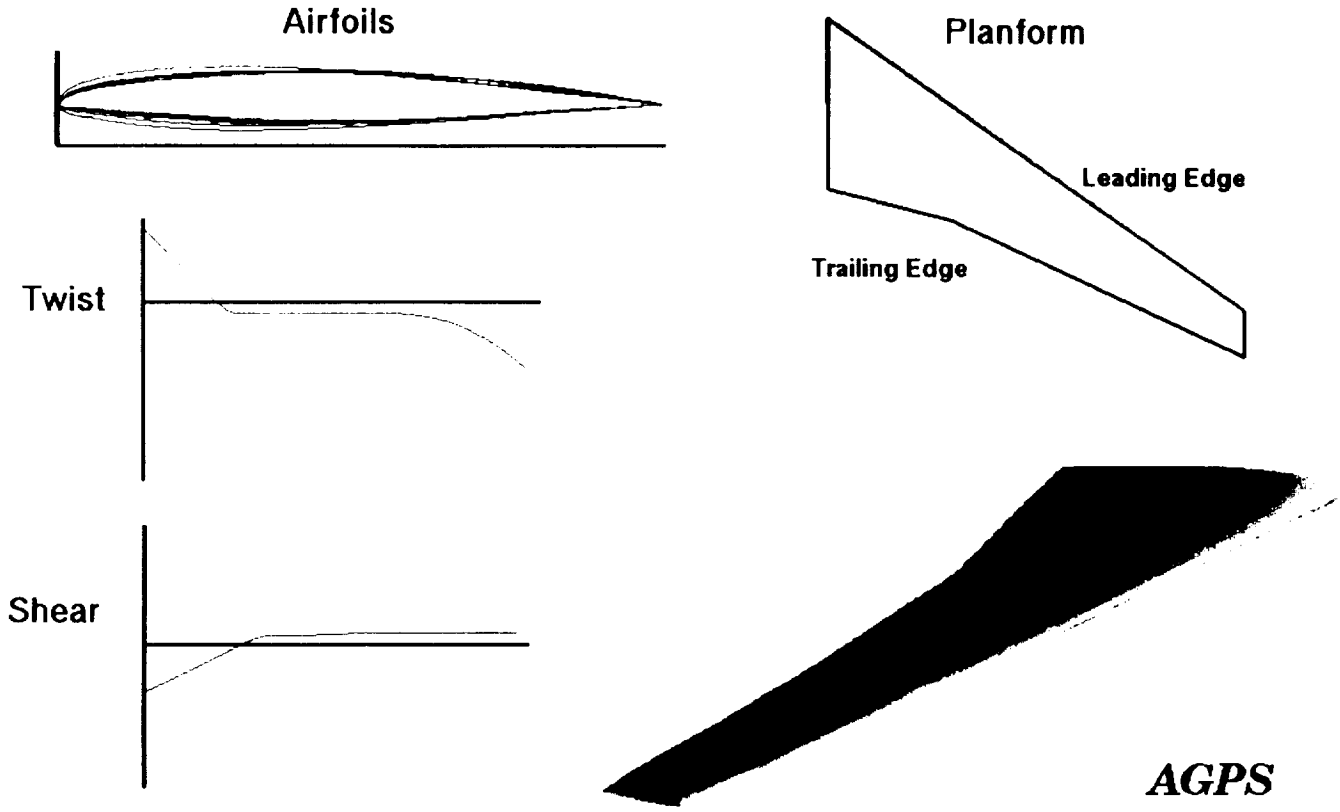


Figure 1. Typical wing loft problem.

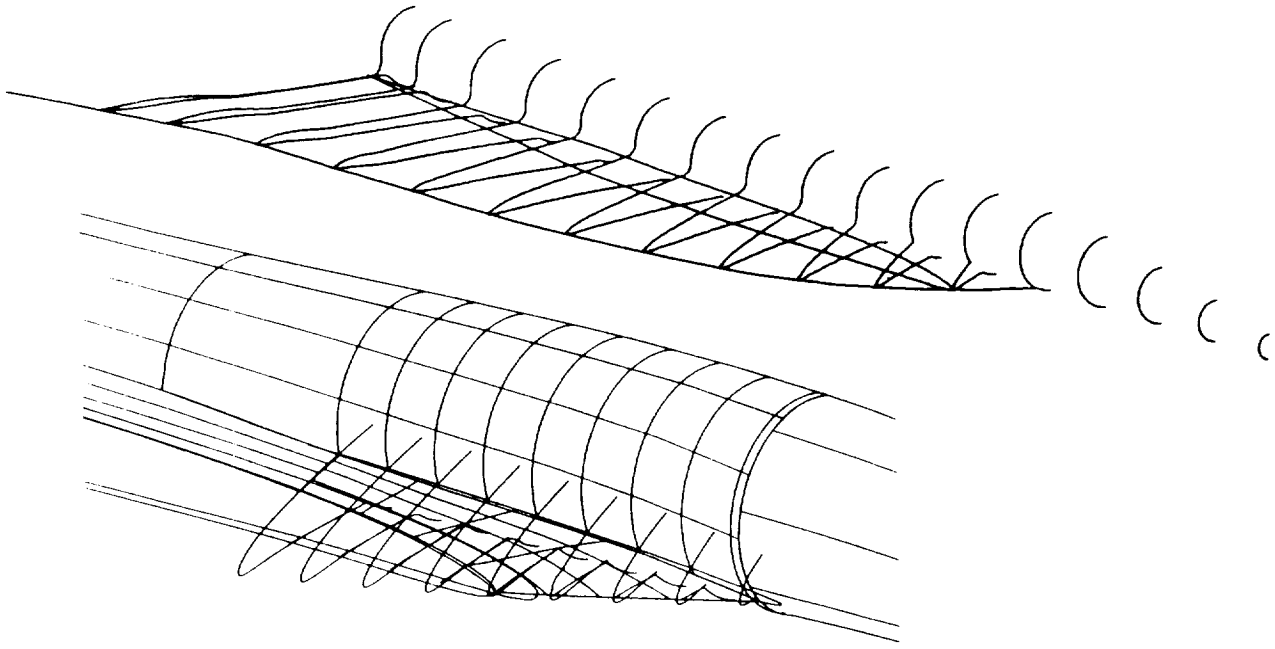
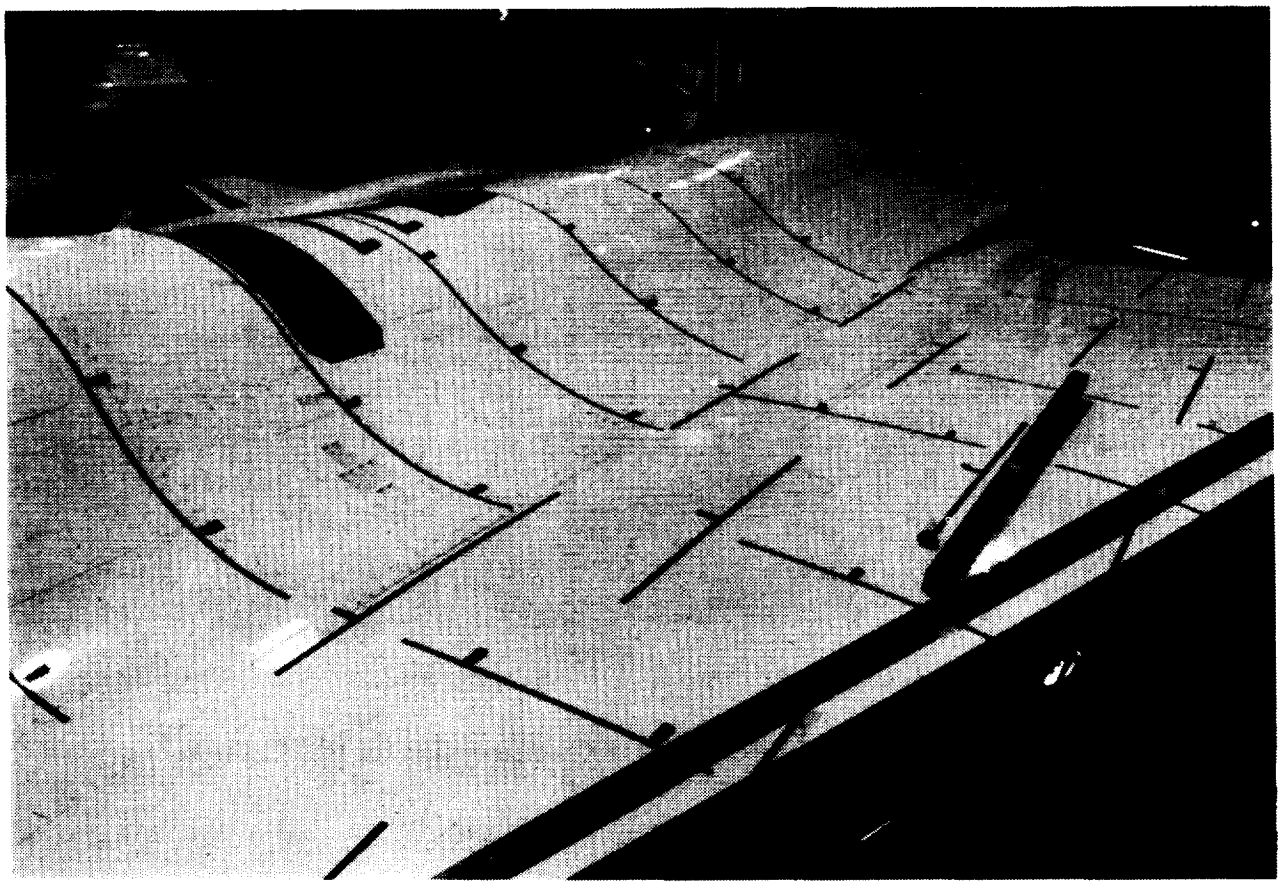


Figure 2. Working with digitized data.



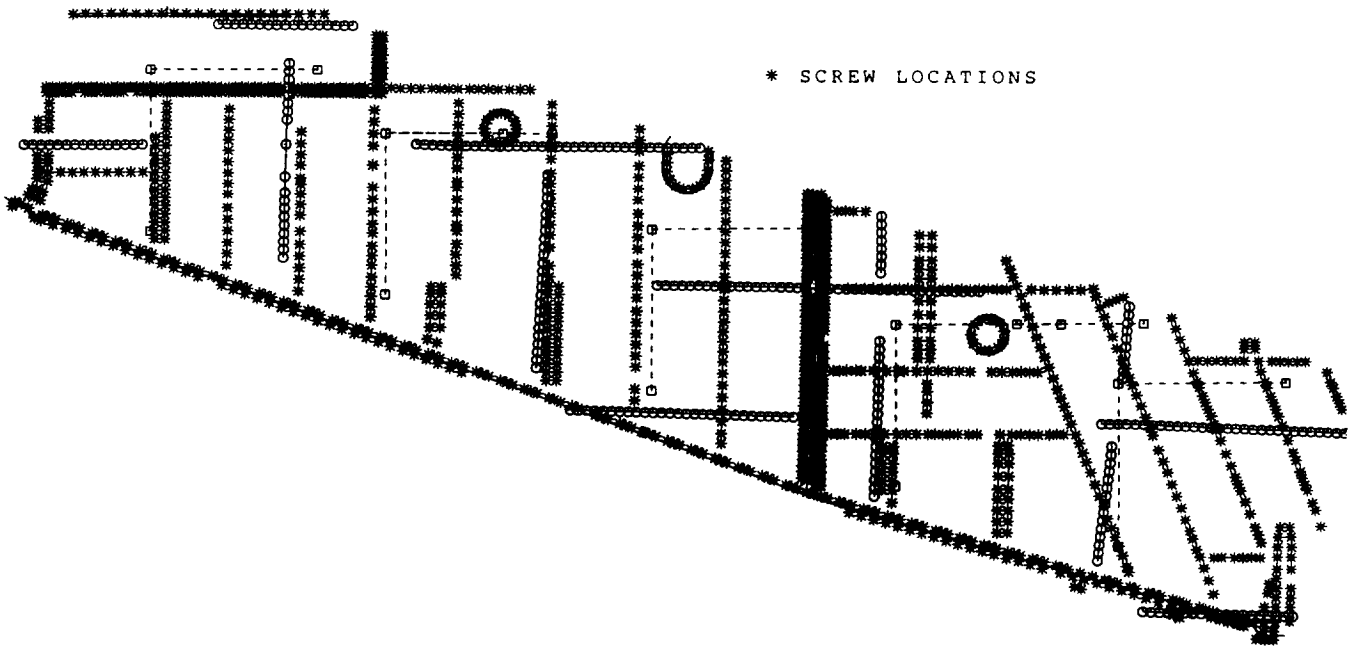


Figure 3. Working with airplane measurement data.

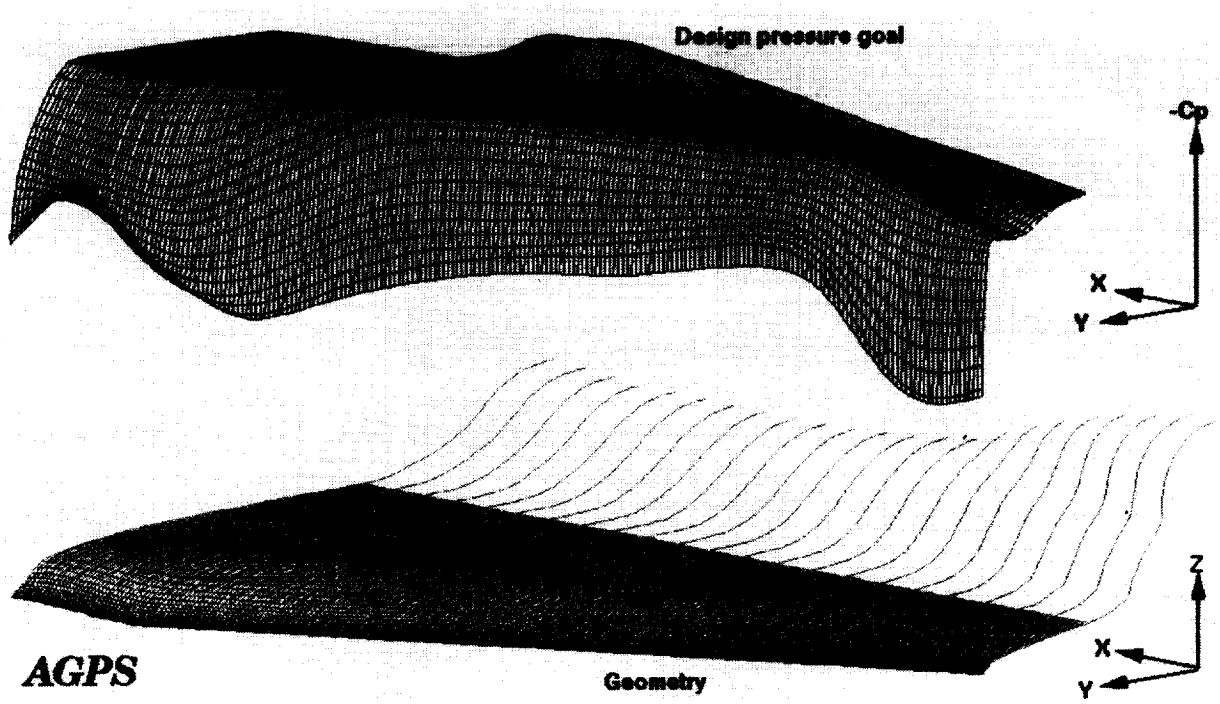


Figure 4. Developing design goal Cp distribution for HLFC glove.

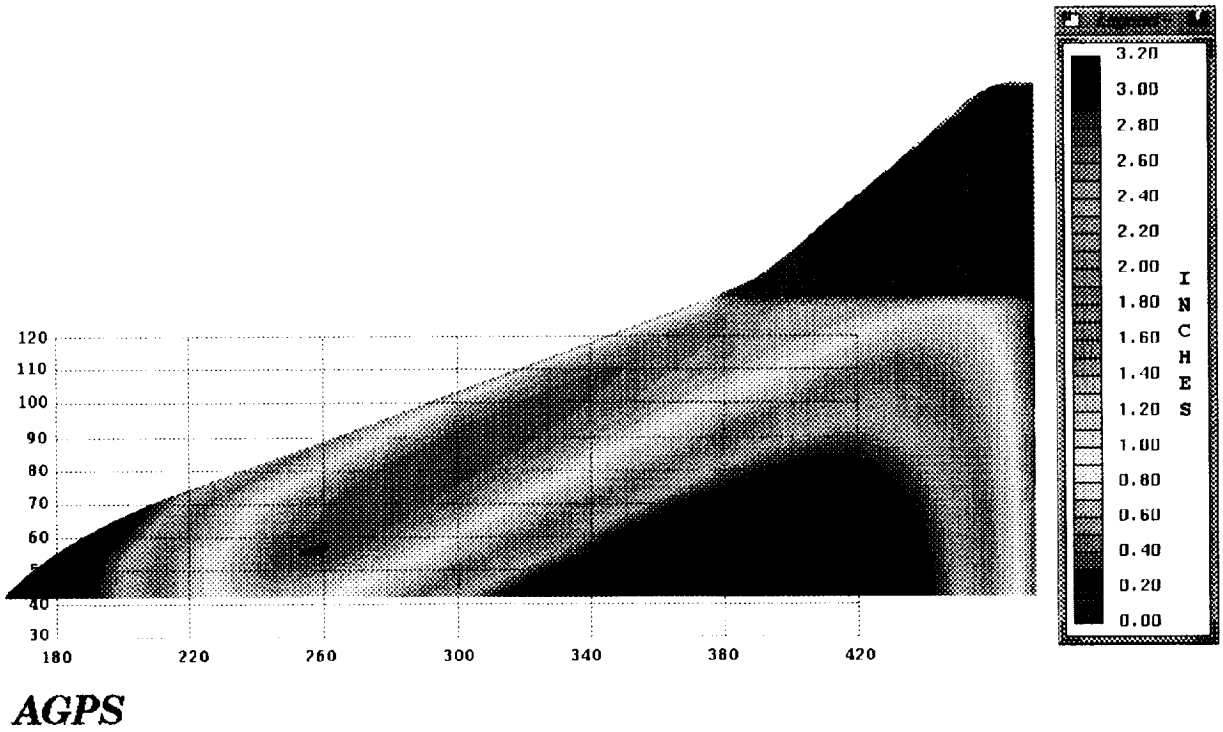


Figure 5. Comparison between two surfaces.

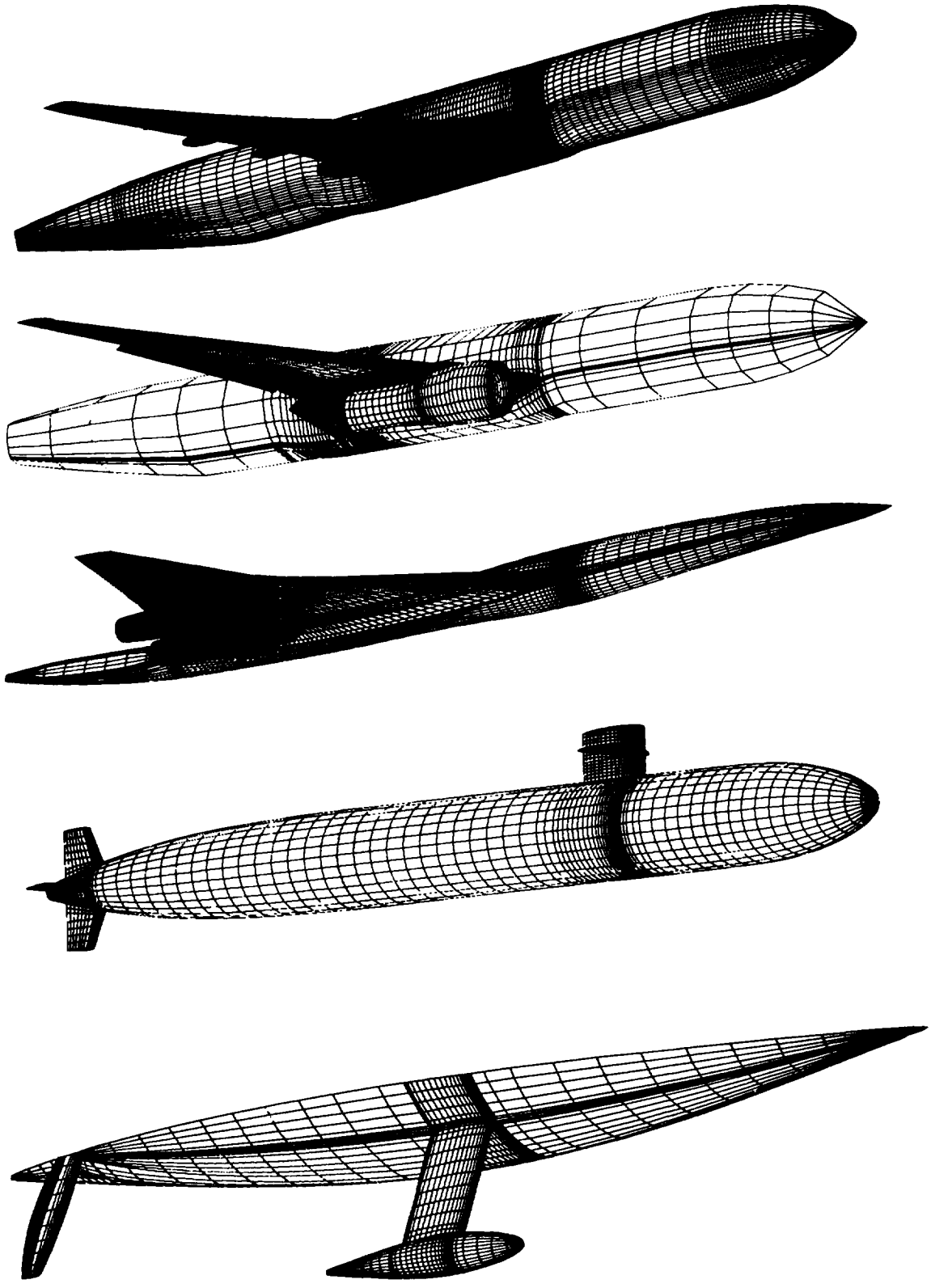


Figure 6. Examples of automatic paneling procedure results.

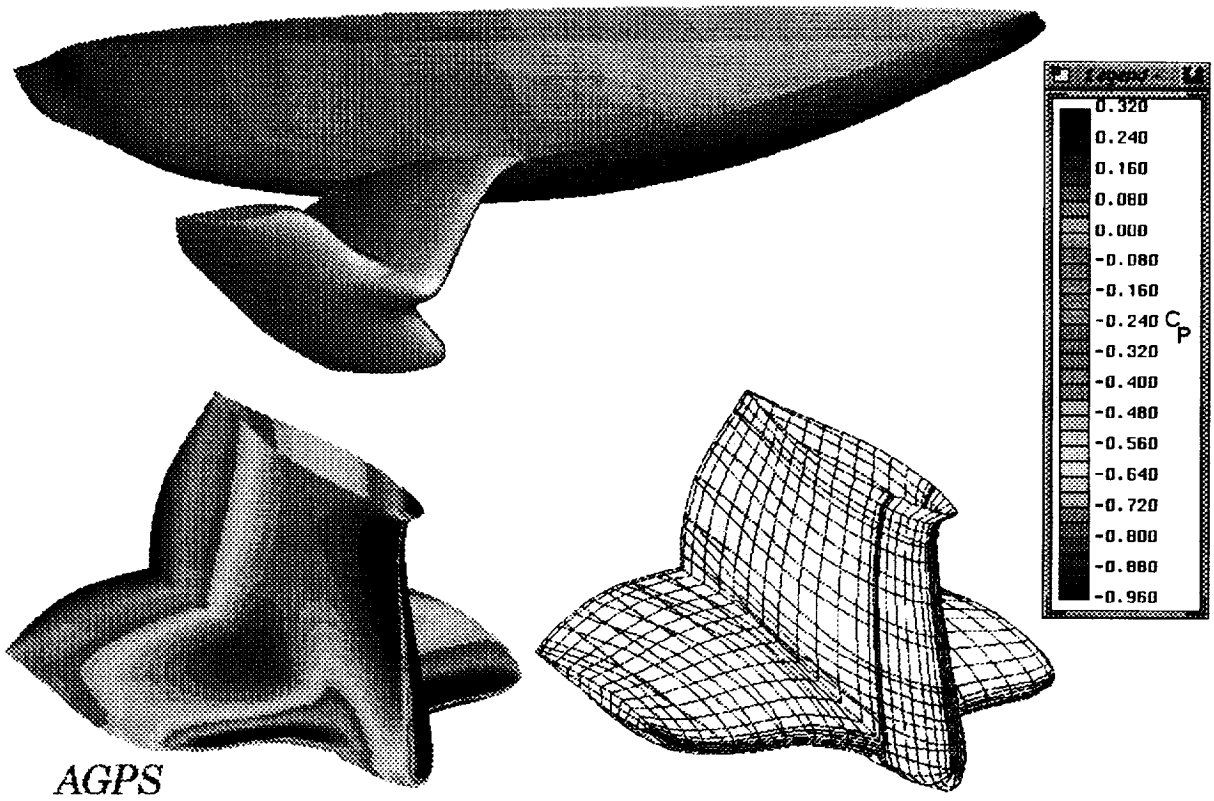


Figure 7. Display of both geometry and CFD results.

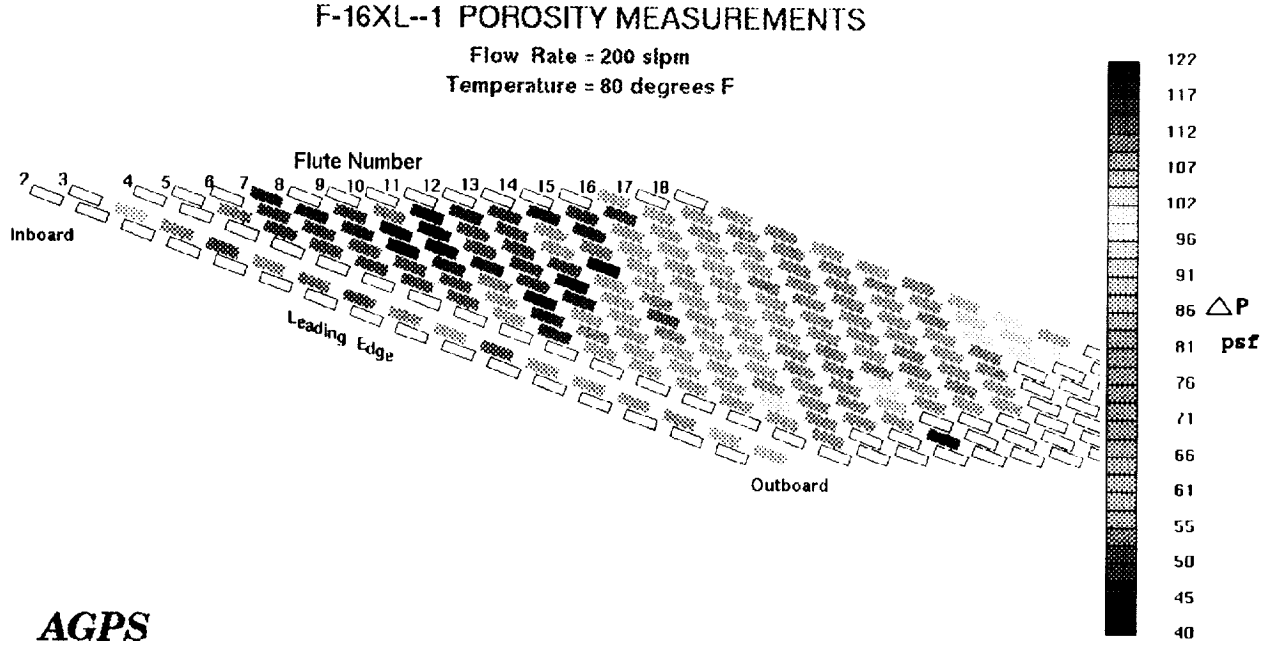


Figure 8. Display of surface porosity data measured on a flight test airplane.

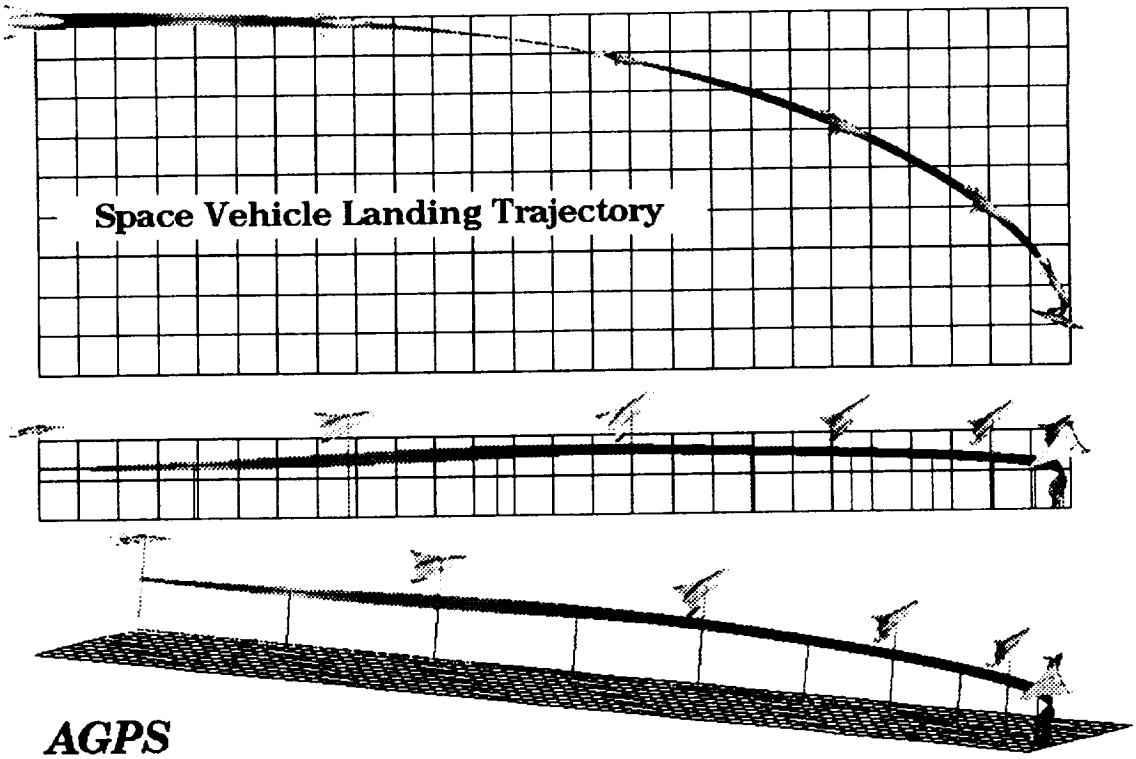


Figure 9. Using AGPS to display results from a trajectory program output.

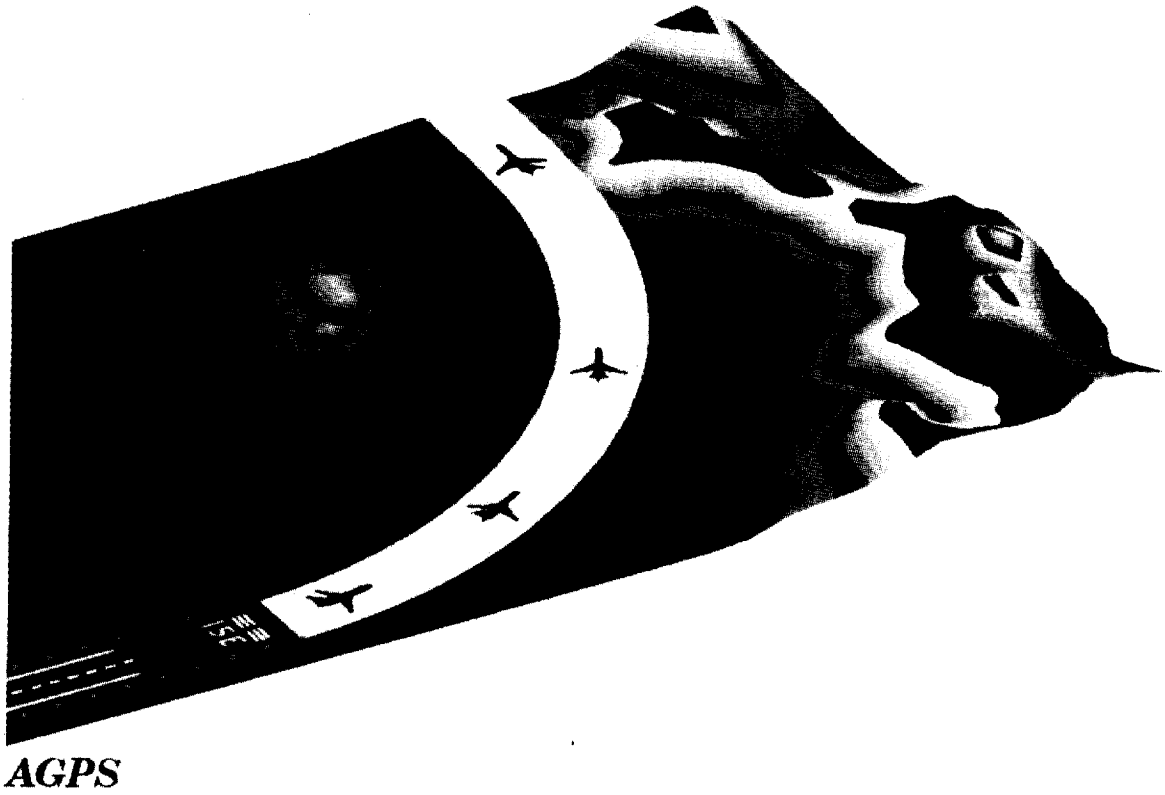


Figure 10. Using AGPS to examine takeoff clearance problem in mountainous terrain.

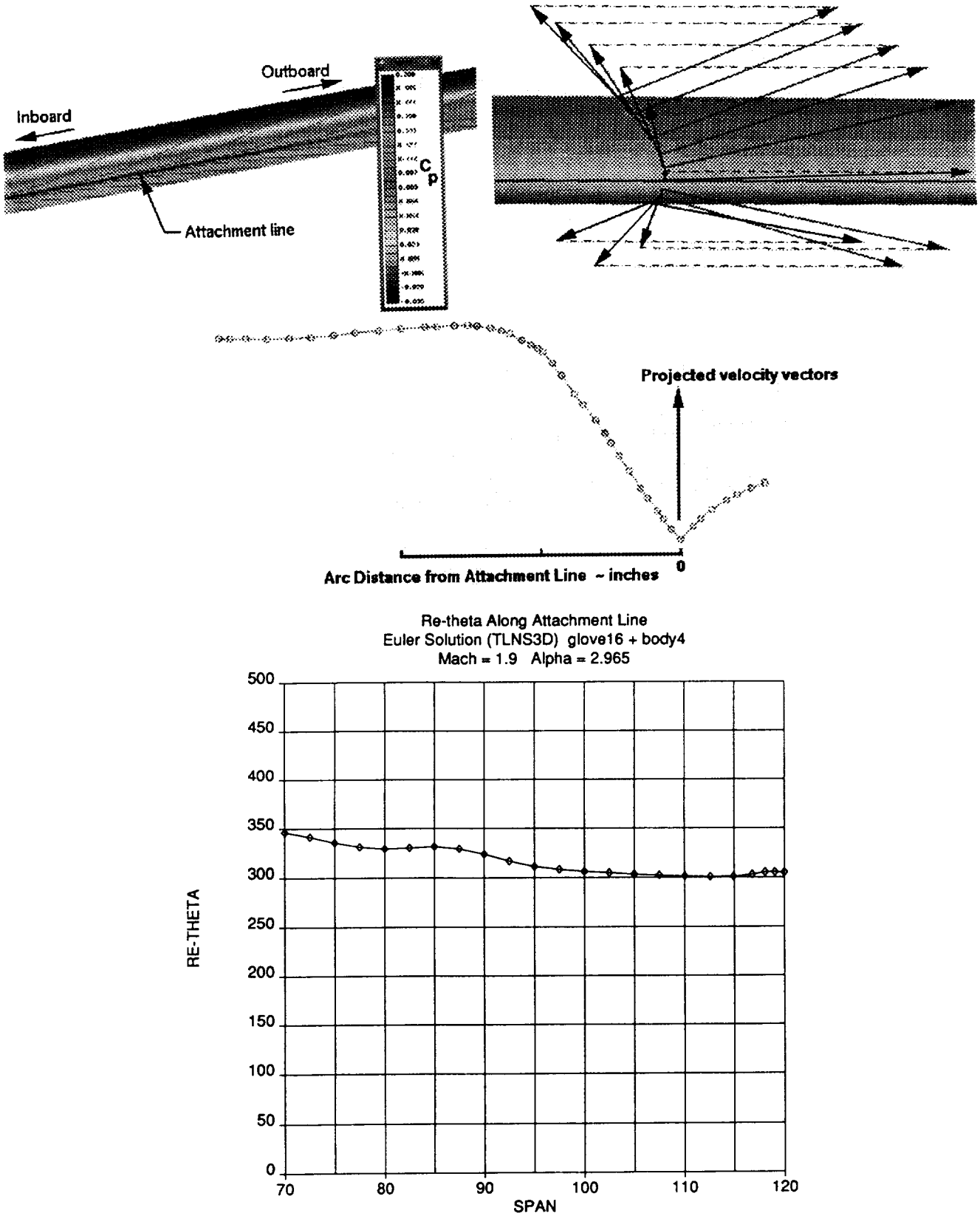


Figure 11. Using AGPS to calculate momentum thickness Reynolds number.

Figure 12. Partial listing of command file for calculating momentum thickness Reynolds Number.

```
!. RE-THETA_TLNS(QFILE=wb8i)
! PURPOSE: Generate the attachment line velocity gradient data and the resulting Retheta data. Cp along the attachment
!          line is also generated. This version of the program is set up for processing the data from the TLNS3D code.
! PREPARED BY: A. Gentry      DATE: 11/14/91 Modified from the STUFF version.
!-----------------------------------------
! INPUT FILES:
!   wing.ggp Output ggp file obtained from the p3dtoggp program selecting only the wing leading edge data region.
!          The TLNS3D data extracted with the p3dtoggp program must be as follows:
!            i 80 to 120 at 1 interval
!            j 1 to 1 at 1 interval
!            k 1 to 30 at 1 interval
!            functions: 114,150,151,152/

! INPUT SYMBOLS:
  $I1=8        ! Station starting value for the surface grid fit.
  $J1=1        ! Upper surface starting grid point for leading edge.
  $J2=70       ! Lower surface end grid point.
  $NA=200    ! Last cross-section on aftbody.
  $SF=0.750  ! Streamline starting s-value, forebody.
  $SA=0.550  ! Streamline starting s-value, aftbody.
  $ISTP=1      ! Stepping interval on the station data.

! INPUT FOR R-THETA CALCULATIONS:
  $EMAC=1.9              ! Flight Mach number.
  $CS=968.07             ! Speed of sound at 44,000ft.
  $ENU=0.000612283     ! Kinematic viscosity (ft2/sec) at 44,000ft.
  $VELO=EMAC*CS     ! Freestream velocity.

!  Z-location of the cuts (at the leading edge point). Create string of 1-D data.
  CST ZC [(62.5),(65),(67.5),(70),(72.5),(75),(77.5),(80),(82.5),(85)]
  CST ZC [(87.5),(90),(92.5),(95),(97.5),(100),(102.5),(105),(107.5)]
  CST ZC [(110),(112.5),(115),(117),(118),(119),(120)]
!-----------------------------------------------
! OUTPUT:
!   RTHE     String of points of Rthets vs. Z (span location).  Output to file rthe_qfile.ggp.
!   ATTCP    String of points of Cp vs Z along the attachment line. Output to file att_cp_qfile.ggp)
!   SVEL     Lists of strings containing the dw/ds normal velocity/Vinfinity derivative data.
!   STRL     List of attachment line point strings.
!              STRL.1 Forebody attachment line.
!              STRL.2 Aftbody attachment line.
!   CUTS.*   Cuts normal to the attachment line at each of the normal plane stations.
!-----------------------------------------
$ON ERROR EXIT
!=========================================================
! Read the wing leading edge region geometry and surface velocity data from Boeing GGP file:
  GGP QFILE.GGP FG X,Y,Z
  GGP QFILE.GGP FUVW U,V,W
  GGP QFILE.GGP FCP CP

! Thin out the data some to reduce compute time.
      $CALL GET_LENGTH(FG,N)        ! How many strings in FG.
      $CALL GET_LENGTH(FG.1.1,M)    ! How many points per string.
$FOR I=I1 STEP ISTP TO N-3 DO
  $FOR J=J1 TO M DO
      $CALL GET_COORD(FG.<I>.1.<J>,0,0,X,Y,Z)
      CST QS [(X,Y,Z)]        ! Put X-Y-Z into a string.
```

Figure 12 (continued).

```
      $CALL GET_COORD(FUVW.<I>.1.<J>,0,0,U,V,W)
      CST QUVW [(U,V,W)]      ! Put U-V-W into a string.
      $CALL GET_COORD(FCP.<I>.1.<J>,0,0,CP)
      CST QQCP [(CP)]              ! Put Cp into a string.
   $ENDDO
      ATL QSTR QS          ! Put the QS strings into a list.
      ATL QFUVW QUVW     ! Put U-V-W strings into a list.
      ATL QCP QQCP             ! Put the Cp strings into a list.
        REN [QQCP,QS,QUVW] N=    ! Free up the names so that they can be reused.
$ENDDO
!     DEL [FG.*.*,FG,FUVW.*.*,FUVW]     ! Delete objects no longer needed.
        PURGE    ! Removed ancestors no longer needed.
      RVS QSTR.*          ! Reverse all of the strings.
      RVS QFUVW.*
      RVS QCP.*
      CAY FGA QSTR.*             ! Geometry array.
     CAY FUVWA QFUVW.*   ! Matching u-v-w array.
      CAY FCPA QCP.*             ! Matching Cp array (contains Cp only).
        REN [QCP,QSTR,QFUVW] N=

! Fit a surface to the x-y-z geometry data array.
   FSU FSURF FGA CUBIC       ! Fit surface to the geometry data.
!-------------------------------------------------
! Build the S-T-Cp array and fit a surface to it.
   ESK FSURF NAA SUBRANGE=YES      ! Extract the array data as subrange points.
       $CALL GET_LENGTH(NAA,N)
       $CALL GET_LENGTH(NAA.1,M)
 $FOR I=1 TO N DO
    $FOR J=1 TO M DO
       $CALL GET_COORD(NAA.<I>.<J>.1,0,0,S,T)
       $CALL GET_COORD(FCPA.<I>.<J>,0,0,CP)
       CST QSCP [(S,T,CP)]
    $ENDDO
       ATL QSSCP QSCP
         REN QSCP N=
 $ENDDO
   CAY AFCP QSSCP.*
   FSU FSCP  AFCP   CUBIC S-PAR=FSURF T-PAR=FSURF ! Surface to Cp.
     DRA FSURF 2 51 5 51 E=YES COL=31 VIEW=0,-75,-10,1.8

! Calculate a streamline along the attachment line.
   FSU FSUVW FUVWA CUBIC S-PAR=FSURF T-PAR=FSURF ! Surface to u-v-w.
   CST STR (SF,1) SPACE=FSURF  ! Starting point for streamline.
   SCS FSURF FSUVW STR.1 STRL -0.1  ! Get the streamline.
     DEL STR
   RVS STRL.1          ! Reverse the string to make it inboard to outboard.
   DRA STRL.* E=NO COL=RED
```

Note: This listing shows about one third of the complete command file.

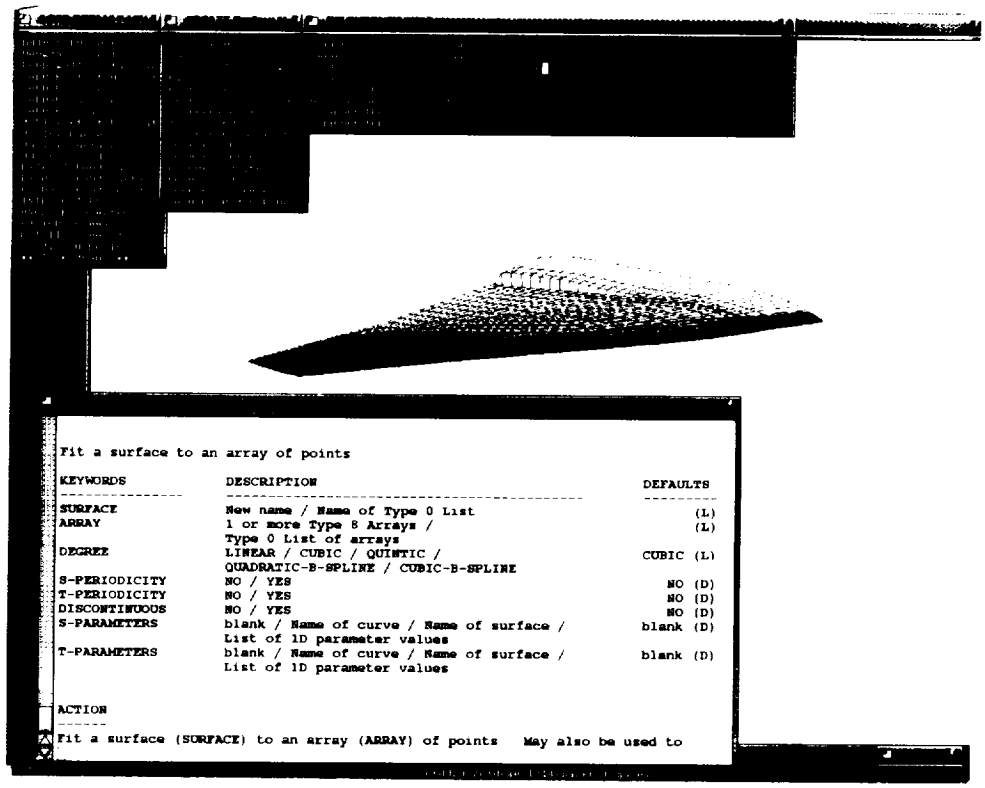


Figure 13. Screen dump of AGPS display showing menu operation.

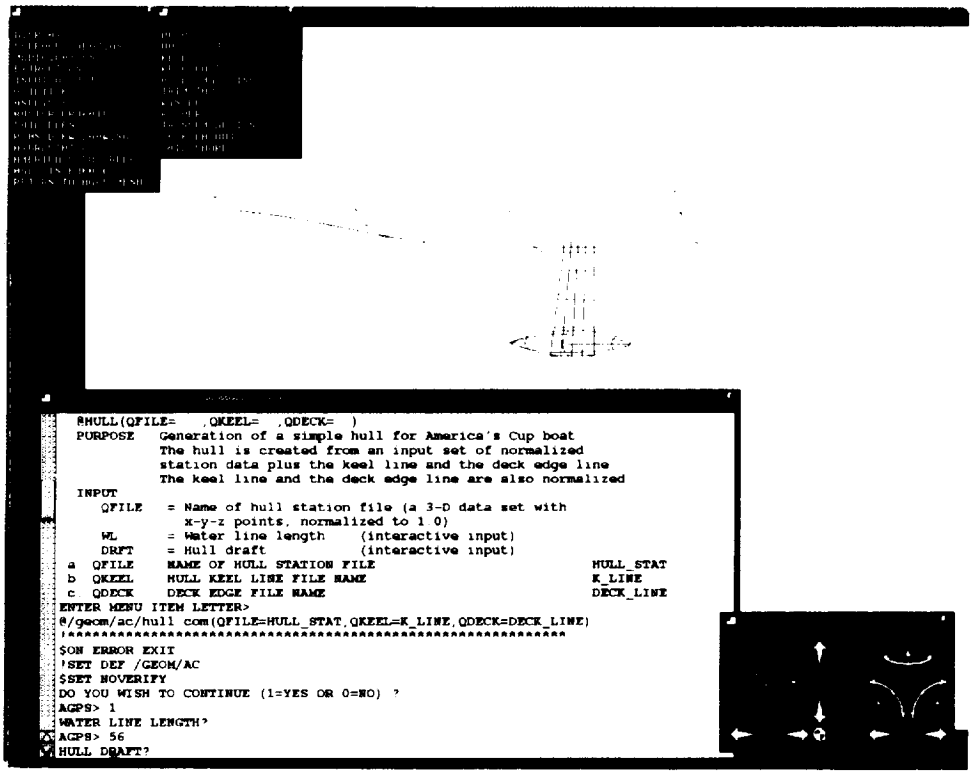


Figure 14. AGPS menu package for designing an America's Cup yacht.