CFD Research Corporation
215 Wynn Dr. • Huntsville, Alabama 35805 • Tel.: (256) 726-4800 • FAX: (256) 726-4806 • info@cfdrc.com

AUTOMATED, PARAMETRIC GEOMETRY MODELING AND GRID GENERATION FOR TURBOMACHINERY APPLICATIONS

Final Phase I Report

prepared by

Vincent J. Harrand, Vadim Uchitel, John Whitmire

April 2000

CFDRC Project: 8229/3

prepared for

Glenn Research Center
Cleveland, OH

Technical Representative: Russ W. Claus
Contract Representative: Carol Sharp

Contract Number: NAS3-00001

TABLE OF CONTENTS

	Page
1. INTRODUCTION	1
1.1 Identification and Significance of the Technology	1
1.2 Phase I Technical Objectives	3
1.3 Accomplishments of Phase I Research	3
1.4 Outline of the Report	4
2. TECHNICAL APPROACH	5
2.1 Conceptual Design Process	5
2.2 System Architecture	7
2.3 Python High Level Scripting Language	9
3. PARAMETRIC DESIGN EXAMPLE: 2D AIRFOIL AND WING	16
4. TURBOMACHINERY DESIGN EXAMPLES	19
4.1 Introduction	19
4.2 Impeller Based on Axial and Radial Coordinate Data	19
4.3 Impeller Based on Blade Profile Data	21
4.4 Rotor & Stator Combination Based on IGES Surface Definitions	24
5. COMMERCIALIZATION OF TECHNOLOGY	25
6. CONCLUDING REMARKS AND RECOMMENDATIONS FOR PHASE II	27
6.1 Conclusions	27
6.2 Recommendations for Phase II	28
7. REFERENCES	29
APPENDICES	

LIST OF ILLUSTRATIONS

	Page
Figure 1. Conceptual Overview of Proposed System	2
Figure 2. Conceptual Overview of Steps Necessary for Generating Impeller Grids	6
Figure 3. Conceptual Overview of Turbomachinery Geometry/Grid Design System	8
Figure 4. Simple Python Script and Geometric Results	12
Figure 5. Structured Grid on Geometry	13
Figure 6. Structured Grid with Higher Density on Geometry	13
Figure 7. Unstructured Grid on Geometry	15
Figure 8. NACA Airfoils as Generated by the Various Scripts Appendix A and B	17
Figure 9. 3D NACA Airfoil Wing (geometry and unstructured tetrahedral grid)	18
Figure 10. UTRC 17DK Impeller Generated with Script in Appendix E	21
Figure 11. Two Views with Impeller Design Output from BLADEGEN. In yellow the 5 cross-sections of the impeller blade are given. The shroud curves are orange (downstream) and green (upstream). The hub curves are red (downstream) and magenta (upstream)	22
Figure 12. Resulting Geometry and Grid from Running Script in Appendix F with the Input Data Points in Figure 11	23
Figure 13. Rotor and Stator Geometry and Grid as Generated by Script in Appendix G and H	24

1. INTRODUCTION

1.1 Identification and Significance of the Technology

Over the past two decades the computational fluid dynamics (CFD) codes as well as computer power has progressed to a point where 3D simulations of complex turbomachinery flows can be undertaken relatively easily in a design setting with an acceptable turnaround rate. One of the key components in the CFD analysis is the model/geometry and mesh generation of the flow domain, and can significantly affect the turnaround time of the overall simulation effort. The progress in this area has, however, not kept pace with other components, and often this step is a bottleneck. The proposed project is aimed at development of a software system that can:

1. Simplify the geometry building stage via graphical and script based tools,
2. Automate the mesh generation process for quality computational grids,
3. Speed up design optimization process through parametrics and customization,
4. Interface CAD/CAM systems for direct data exchange with design and manufacturing, and
5. Provide meshes for CFD and CAE in appropriate formats to increase the utility of the software.

Turbomachine geometries are some of the most complex in shape, and may require tedious efforts in mesh generation. Although some progress has been made towards semi-automatic grid generation for certain classes of machines, e.g. centrifugal compressor, the range of geometry changes in turbomachines is very large and automated grid generators that cover the entire spectrum are still not available. Focus of the present project is on developing a software tool that will establish a uniform geometry and grid generation process, in which an entire range of geometries can be handled which are likely to be encountered during the design of a turbomachine, from axial to centrifugal to impulse machines.

An essential functionality in the designer's workbench is the capability to create detailed 3D geometry from the various analytical and parametric geometry data generated during the earlier stages of the design process. The availability of this detailed geometry is necessary for the mesh generation process, which in turn feeds the high-fidelity analysis codes (FEA, CFD, etc.). For most modern manufacturing organizations, having the detailed geometry available within a commercial CAD system is of paramount importance. From such a system further analysis of the entire system (e.g. jet engine) can be performed (CAD, CAE), management of product data can be maintained (PDM), and the manufacturing process can be controlled (CAM). This CAD/PDM interface will be an important focus of Phase II.

Traditionally this detailed geometry creation process has been a very awkward and labor intensive step. The analytical and parametric geometry data is not very suitable to be directly inputted into a CAD or grid generation system. Typical scenarios for creating geometry models within a CAD system include:

1. Create a 2D profile and either extrude this profile into the third dimension or revolve the profile into a 3D shape.

2. Use a solid modeling approach in which complex shapes are formed from basic geometric primitives such as box, cylinder, cone, etc.

None of these standard geometry creation methods applies to turbomachinery designs, and the turbomachinery designer has to somehow define the various geometric shapes, i.e. set of free-form surfaces. In the CAD world all free-form surface modeling is performed based on trimmed NURBS patches. In other words, the turbomachinery designer must input the data in CAD terminology and functions, and not in the much more desirable turbomachinery terms.

The proposed effort is targeted to solve this problem. **The system provides an efficient path from the output of various turbomachinery design codes, i.e. analytical and parametric geometry definitions, to NURBS based CAD models, to computational meshes, and to direct access of commercial CAD systems.** For example in Phase II, a bi-directional path to a commercial CAD system, such as PRO/Engineer or Unigraphics, will be developed.

A Conceptual Overview of the proposed system is given in Figure 1.

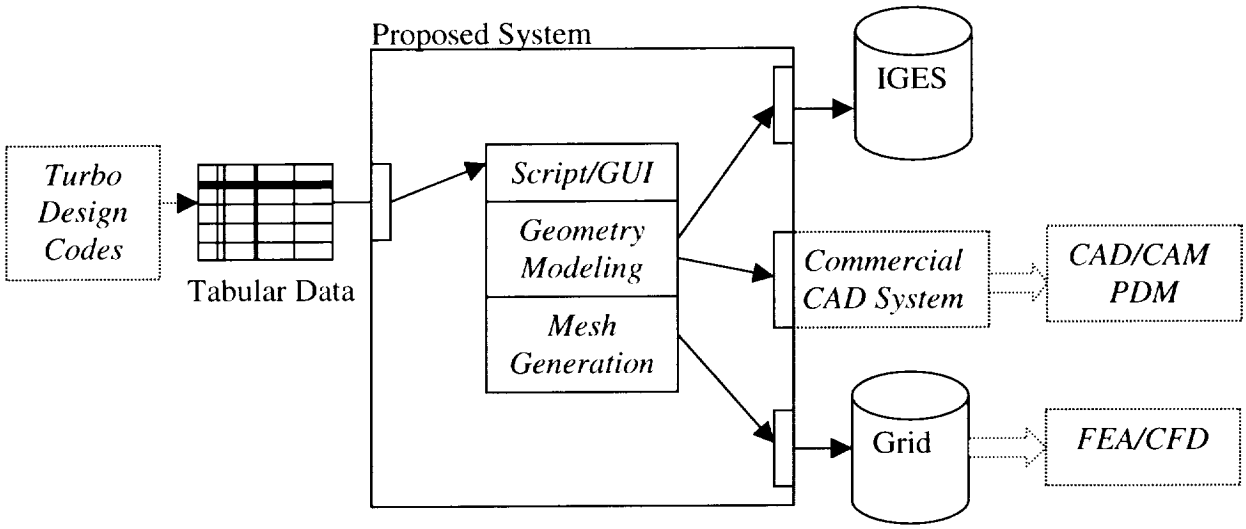

Figure 1. Conceptual Overview of Proposed System

In order to be able to put full focus on the described problem, we will use existing pieces for certain parts of the system (e.g. script interpreter, geometry kernel, IGES writer, and mesh generation). This Phase I effort focused on establishing the feasibility of the system, i.e. in an automated fashion create detailed turbomachinery geometry and meshes from analytical and parametric geometry descriptions.

1.2 Phase I Technical Objectives

The overall objective of the proposed effort is to develop a highly automated software system for geometry and grid generation for a variety of turbomachinery configurations. The focus will be primarily on establishing the technology to create detailed 3D geometry and computational grids from the output of turbomachinery design codes in a highly automated fashion. The proposed system will facilitate detailed geometry descriptions based on trimmed NURBS modeling methods. Both multi-block structured and unstructured grid generation technology will be supported by the system

Some additional objectives of the Phase I research effort include:

1. It is important to provide for a high level of end-user efficiency and effectiveness, this is accomplished by providing individual templates for the various turbomachinery components.
2. Tool must be easily customizable by the end-user. Each template is in the form of a script, which is easily modifiable, and new ones can be easily created by the end-user.

The focus of Phase I effort is to demonstrate the feasibility of the entire effort. Algorithms will be developed which convert analytic and parametric geometry descriptions into NURBS CAD definitions, the corresponding scripts/templates will be developed, the scripting facility will be integrated with the various existing pieces (geometry kernel, meshing libraries, and IGES writer), and the entire system will be tested and demonstrated on a turbomachinery geometry of interest, e.g. a centrifugal compressor. A successful Phase I demonstration implements this system, utilizes the system for the selected turbomachinery geometry, and demonstrates the high level of end-user effectiveness, efficiency, and customization.

1.3 Accomplishments of Phase I Research

Overall, we feel that the Phase I feasibility study has been very successful. We have developed a prototype of the automated geometry modeling and grid generation system. The Python scripting facility has been integrated with the geometry and grid kernel. For end-user convenience a specialized Python editor has been created and integrated with the system. The various geometry modeling and grid generation functions within those kernels have been made part of the new scripting language. In addition, several new functions have been added to those kernels (to be invoked from the Python script by the end-user) to facilitate more powerful functions and automation in the script. For more details see section 2.

We have applied this prototype system on a variety of turbomachinery components including the centrifugal compressor. We have demonstrated how the scripts can be used for parametric design, and on how a variety of different input formats (radial/axial information vs. blade profiles vs. blade surface descriptions) can be accommodated. The time to create a typical script from scratch is in the order of hours. The time necessary to modify an existing script is in the order of half an hour. Once a script or template has been created for a particular configuration, a typical design organization may never change the script. The running time of a typical script is in the order of seconds. For organizations that analyze many similar design configurations as part of a parametric analysis or design optimization system, a significant efficiency gain can be

obtained. This needs to be compared to the traditional manual process of geometry modeling and grid generation, which is in the order of days for one configuration. As a prelude to Phase II we have started to model multi-stage turbomachinery components such as a rotor-stator combination. For more information see Section 4.

An important aspect of the Phase I effort is the preparation for the commercialization activity. During Phase I we have had exploratory conversations with Concepts ETI Inc. This company specializes in developing and marketing turbomachinery design software. They are very interested in the results of our STTR program and plan to evaluate our tool as an addition to their own product line. As part of our marketing strategy, we have approached approximately 85 engineers in the US turbomachinery design industry, we have made them aware of this effort and requested feedback from them. For more information see Section 5.

1.4 Outline of the Report

The technical approach for this project is outlined in Section 2 of this report. Section 3 gives an overview on how this technology can be used for parametric design of airfoils, while elaborate examples of script templates for turbomachinery components are given in Section 4. The corresponding scripts for the various sample geometries are given in the appendices. In Section 5, an overview is given of our commercialization strategy for this STTR project.

Finally, conclusions and lessons learned from Phase I effort and recommendations for future Phase II research are included in Section 6.

2. TECHNICAL APPROACH

2.1 Conceptual Design Process

The typical output of an impeller design code is a set of tabular data consisting of axial and radial coordinates, blade angles, and thickness distributions. This data is generated using the organization's proprietary design codes after optimizing the one-dimensional model, and as such, the format can vary somewhat. It is the job of the turbomachinery Python script to convert this set of numbers into a full-blown three-dimensional CAD model.

Name	*Symbol*	*Description*
Axial Position	z	The coordinate at each axial slice
Radius	r	The radial coordinate
Wrap Angle	Θ	The angle from the radial position
Blade Angle	β	The angle that the blade impart on fluid
Thickness	t	The thickness of the blade
Blade Count	n	Number of blades in the impeller
Blade Offset	α	Angle occupied by a single blade passage

For example, a table consisting of radial and axial coordinates, along with the blade angle and thickness can be used to generate the 12 curves defining the passage through a shrouded impeller:

$$x_{hub,pressure} = r \cos \Theta$$
$$y_{hub,pressure} = r \sin \Theta$$
$$z_{hub,pressure} = z$$

$$x'_{hub,suction} = x_{hub,pressure} + t \cos \beta$$
$$y'_{hub,suction} = y_{hub,pressure} + t \sin \beta$$
$$z'_{hub,suction} = z_{hub,pressure}$$

$$x_{hub,suction} = x'_{hub,pressure} \cos \alpha - y'_{hub,pressure} \sin \alpha$$
$$y_{hub,suction} = x'_{hub,pressure} \sin \alpha + y'_{hub,pressure} \cos \alpha$$
$$z_{hub,suction} = z_{hub,suction}$$

Of course, other formats have corresponding formulae for obtaining the absolute geometric coordinates of the blade curves. After the formulae are applied to a number of discrete sampling points, a curve can be fitted through the points, resulting in a wireframe model. Curves are then interpolated into NURBS surfaces, and surfaces into the impeller passage volume.

Alternatively, a blade may be specified by the design system as a set of profile curves or as an IGES surface. In case of blade profile curves, a ruled surface needs to be created to interpolate

the final blade surface. And similar to the previous case, the surfaces can be used to model the impeller passage volume.

Because an impeller geometry is fully characterized by the number of blades and the blade profile and the appropriate formulae, one can construct a **generalized topological model** of the impeller, consisting of 12 curves: the hub pressure, hub suction, outlet pressure, inlet pressure, outlet suction, inlet suction, outlet hub, inlet hub, shroud pressure, and shroud suction curves. This **topological template** can be implemented as a Python script, which in turn computes the true geometric location of each point. This modularity will lead to important benefits in the future. For example, if a blade coordinate is modified, the **overall geometry of the impeller is updated automatically**. Note that this is precisely the manner in which the turbomachinery designer works-- **the blade angle or thickness is modified, rather than the true geometric coordinate**. Optionally the solids (i.e. 'metal') can be generated or the 'air' passage ways. The geometric module can be used to duplicate and rotate the geometries to create the full circular geometry.

The grid generation process is similar. Here, the **topological template** can automatically determine an appropriate node distribution to better resolve the leading and trailing edges. Surface grids are generated using transfinite interpolation (TFI) and then the volume grid is created. Grid smoothing can also be enabled. Boundary and volume conditions can also be applied to the impeller, which can then be exported to an FEA or CFD code for analysis.

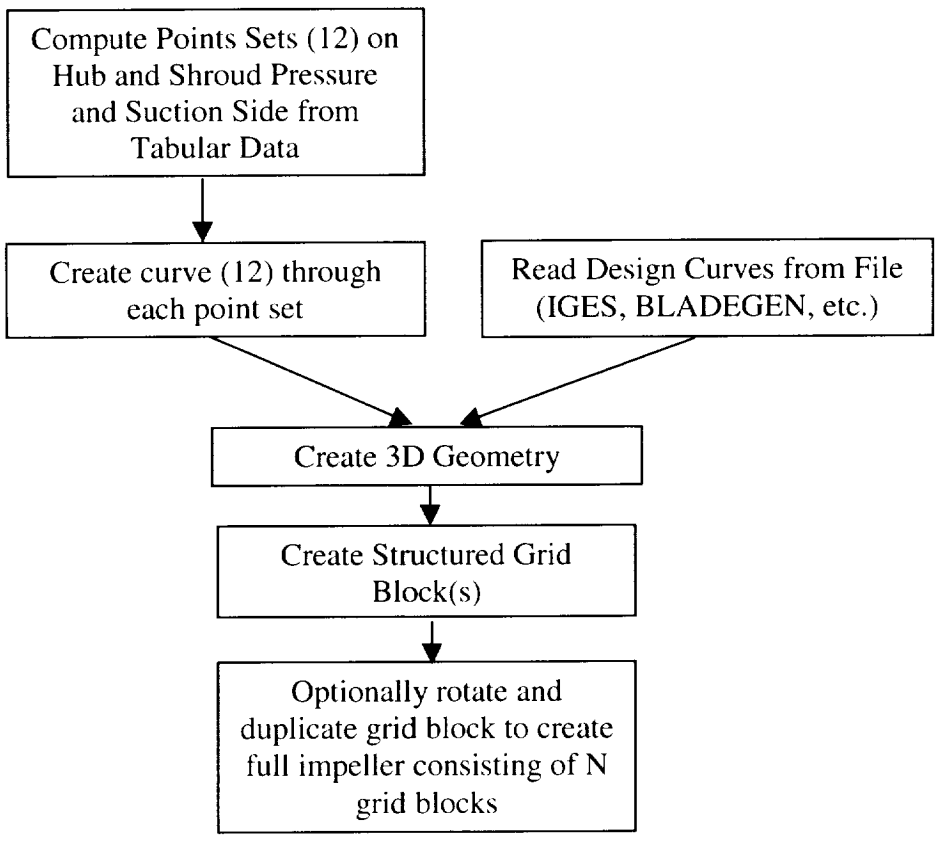

Figure 2. Conceptual Overview of Steps Necessary for Generating Impeller Grids

A conceptual overview of this process is given in Figure 2. This process can be captured in an impeller template or script, and the whole process becomes automatic. One of the important benefits of using a scripting system as described here is that **composition of scripts or templates** becomes relatively straightforward. An example of composition was the development of the impeller topological template independent of blade geometry. When run, the template is composed with the blade profile data. This system will be extended in Phase II with the development of for example vaneless diffuser and volute scripts, which allow analysis of a much larger turbomachinery system.

2.2 System Architecture

As aforementioned, the most difficult problem for turbomachinery designers today is having to interact with a CAD system which is designed to be application-neutral. In other words, designers cannot use concepts like blade angles and thickness, but must instead specify the geometry in terms of points, curves, and surfaces. Because of this, converting a set of blade coordinates into a form that is readable by the CAD system can be somewhat difficult. On the other hand, a system tailor-made for turbomachines lacks the flexibility of a full-blown CAD system.

To solve this problem, one needs a method to extend an existing CAD system into a particular application-domain space, thus addressing both of the earlier concerns. The approach taken in this project involves the use of a scripting subsystem in order to extend the CAD system in useful and meaningful ways. For example, several scripts could be developed to read in the tabular data generated from turbomachinery design codes. In fact, if the scripting system is powerful enough, it may be possible to write the design rules in the script itself. Another benefit of the scripting system is that it allows the turbomachinery designer to interact with the CAD system using turbomachinery concept and terms. For example, the user is able to refer and modify the blade profile on the suction surface at the hub, perhaps even graphically. At all times, though, the user has full access to the low-level functionality of the CAD system.

The implementation of the scripting subsystem makes use of Python, a freely available library suitable for embedding within an application. Unlike Tcl/Tk and other scripting languages, Python refers to objects through memory handles, rather than converting the data into a single representation (string, for example). Furthermore, it supports object-oriented programming and dynamic runtime binding, which lends itself to an extremely flexible system. Strictly speaking, Python is an interpreted language and source files consist of C/C++-like language constructs stored in a plain ASCII format; the code is compiled at runtime into platform-neutral byte code, akin to the process used in Java. The result is that code runs at nearly the same speed as it would had it be implemented in pure C or C++. Finally, once embedded, Python is able to invoke all the functions inside of the application being scripted.

The geometry modeling and grid generation module used for this effort is CFD-GEOM. Other geometry modeling and/or grid generation modules could later be linked in as well. In order for Python to have access to the internal functions and data inside CFD-GEOM, it is first necessary to create C/C++ interfaces that Python can call which then in turn call the appropriate method function inside of CFD-GEOM. This two-step resolution and invocation of the function is

necessary to achieve Python's type-independent late-binding in the absence of this capability in C/C++.

The process of automatically generating the interfaces is done by SWIG, the Python interface generator, which parses the CFD-GEOM header files and creates a Python source file declaring all of the parsed functions. The interface file is then compiled and linked to create a shared object library and updated executable. When CFD-GEOM is executed, Python scripting is available. To extend CFD-GEOM's capabilities for turbomachines, it necessary to write Python scripts that can access the CFD-GEOM database in a higher-level manner. For example, a script could be written to read tabular impeller blade coordinates, create the geometry, and generate the mesh in a single command. Furthermore, because Python supports arguments to functions-- it is not simply a glorified macro language-- it is possible to parameterize the geometry and grid generation process with dimensions, grid distribution options and the like.

Overall, the proposed design satisfies a number of important criteria. Firstly, because Python scripts do not need compilation, development of scripts can be done efficiently and can be built up interactively. Second, the system is flexible enough that users can customize the package to their specific domain of focus.

Our experience in Phase I was that a direct translation of the geometry/grid kernel functions into Python functions by SWIG is not sufficient (we have called this the 'low-level' script functionality). The functions provided by the kernel may not be appropriate for 'end-user usage' in the Python script. The functions were low level and resulted in long and laborious scripts. In addition, certain parameters of those functions may be awkward and unclear to the end user. Furthermore, the resulting scripting language does not fully conform to powerful Python concepts such as lists, overloading and optional parameters. To assist in the creation of powerful script functions even further, we have added new functions to the geometry/grid kernel. During the course of Phase I, this has resulted in significantly shorter scripts for each of the progress reports. This High Level Script layer is build on top of the SWIG generated Low Level Script layer, see Figure 3. This High Level script is written in Python, and invokes the functions on the Low Level Script layer. The end user writes the application scripts entirely in the High Level script language. For Phase I we have focused on creating the script on the Geometry and Grid kernel. For Phase II this will be extended with interfaces to Turbomachinery Design Software and Parametric CAD/PDM systems as indicated in Figure 3.

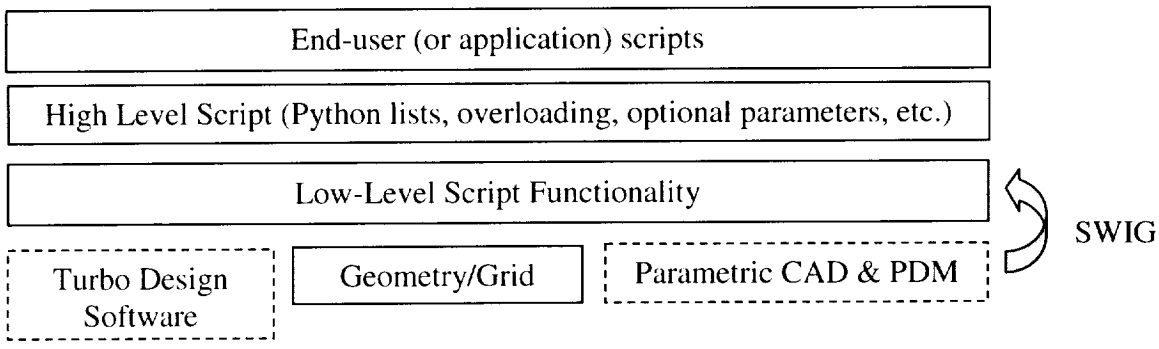

Figure 3. Conceptual Overview of Turbomachinery Geometry/Grid Design System

In the next section, an introduction to the High Level scripting language is given. During Phase I we have introduced a set of core functions sufficient to model e.g. impeller geometries and grids. In Phase II more geometry/grid kernel functionality will be introduced into the scripting language.

2.3 Python High Level Scripting Language

The Python scripting language has been integrated with the CFD-GEOM geometry and grid kernel. A two-level system has been created. The low-level functionality is a direct result of SWIG translation of the kernel functions, while the high level layer has been created to facilitate an easier script with higher level abstractions. The result is that we have created an object oriented scripting language (based on the Python) enriched with advanced geometry and grid primitives.

A special python editor window with script keyword highlighting and various options for saving, reading, and executing scripts has been developed for as part of this project. A special window with all available modules and script functions has been developed to assist the user in what functions are available. Figure 4 shows a picture with this script editor window on the right hand side. For Phase II, an on-line help widget needs to be developed as part of the commercialization strategy.

In the remainder of this section an overview of the capabilities of the developed scripting language is given. The purpose of this section is to give the reader a good introduction to the script language so that the examples in the next chapters are clear. During Phase II, the scripting language will evolve, i.e. more geometry/grid kernel functions will be added, and different functions and parameters may be chosen.

A first sample script is shown here:

```
1.  import GPoint
2.  import GCurve
3.  import GManip
4.  import GBlock

5.  p1 = GPoint.Create (0, 10, 0)
6.  p2 = GPoint.Create (10, 10, 0)
7.  p3 = GPoint.Create (10, 20, 0)
8.  p4 = GPoint.Create (0, 20, 0)

9.  l1 = GCurve.CreateThroughPoints (p1, p2)
10. l2 = GCurve.CreateThroughPoints (p2, p3)
11. l3 = GCurve.CreateThroughPoints (p3, p4)
12. l4 = GCurve.CreateThroughPoints (p4, p1)

13. rot_angle = 45
14. rev_crv1 = GCurve.CreatePtRevolvedCurve (p1, [0, 0, 0], [1, 0, 0], rot_angle)
15. rev_crv2 = GCurve.CreatePtRevolvedCurve (p2, [0, 0, 0], [1, 0, 0], rot_angle)
16. rev_crv3 = GCurve.CreatePtRevolvedCurve (p3, [0, 0, 0], [1, 0, 0], rot_angle)
17. rev_crv4 = GCurve.CreatePtRevolvedCurve (p4, [0, 0, 0], [1, 0, 0], rot_angle)

18. r_list = GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 1, l1, l2, l3, l4)
```

This is the script shown in Figure 4, and is intended for drawing an outline of a box. In lines 1 through 4 of the script we are defining which of the high level modules we are going to need in the script. The GPoint module contains all Geometry Point (creation) related functions. All functions in this module start with 'GPoint'. Similarly for the GCurve module. This module contains all functions which operate on Geometry Curves, and so on.

Lines 5 through 8 show the process of point creation. First word in the function call (GPoint) indicates that we are going to invoke function from the module GPoint. Therefore, GPoint.Create (0, 10, 0) means that we are calling function Create from the module GPoint with parameters (0, 10, 0). This function call will create point in geometry kernel at the location (0, 10, 0) in world coordinates (since the kernel is dimensionless, we do not need to specify dimension units).

Lines 9 through 17 show the different ways of line and curve creation. This is a real advantage of the developed scripting language that created functions can accept different parameters and still produce correct and desirable result. For example, script line GCurve.CreateThroughPoints (p1, p2) will call function CreateThroughPoints from the module GCurve passing parameters p1 and p2. This function call will create a line in the kernel, which goes from p1 to p2. But if we call the same function with parameters p1, p2, p3, i.e. GCurve.CreateThroughPoints (p1, p2, p3) then it will build NURBS (Non-Uniform Rational Bspline) curve, which interpolates points p1, p2 and p3.

Therefore lines 9 to 12 generates four geometric lines, while line 14 to 17 generates four arcs.

Line 14 of the script provides another way of the curve creation. In this line we are calling function CreatePtRevolvedCurve of the module GCurve. This function takes a point as an input parameter, two vectors, which define start and end points of rotation axis and rotation angle and creates a revolved curve around passed axis of rotation. By passing rotation angle as a variable rather than a value to this function, we are using another powerful feature of the scripting language – parameters reuse. This feature allows specifying a parameter as an input to a function, which leads to future reuse of the script since then by changing value of a parameter (rot_angle in this example) we are modifying inputs to several curve creation routines.

Line 18, the last line of the script, demonstrates how curves can be created by duplicating existing ones and then rotating them around specified axis on the required angle. Module GManip of the developed system currently contains functionality for rotating and translating elements, as well as, for prior duplicating them if it is required. Scaling will be added in a future development. Function DuplAndRotate of this module can take any number of elements (in the example we are passing lines l1, l2, l3, l4 to this routine), duplicate them 1 time and then rotate on the specified angle (in the example we are passing variable rot_angle, which has value 45) around rotation axis, which is determined by two endpoints (in the script - [0, 0, 0] and [1, 0, 0]). This function returns Python list of elements, which contains handles to the rotated entities (i.e. four geometric lines).

Results of this script are shown on Figure 4.

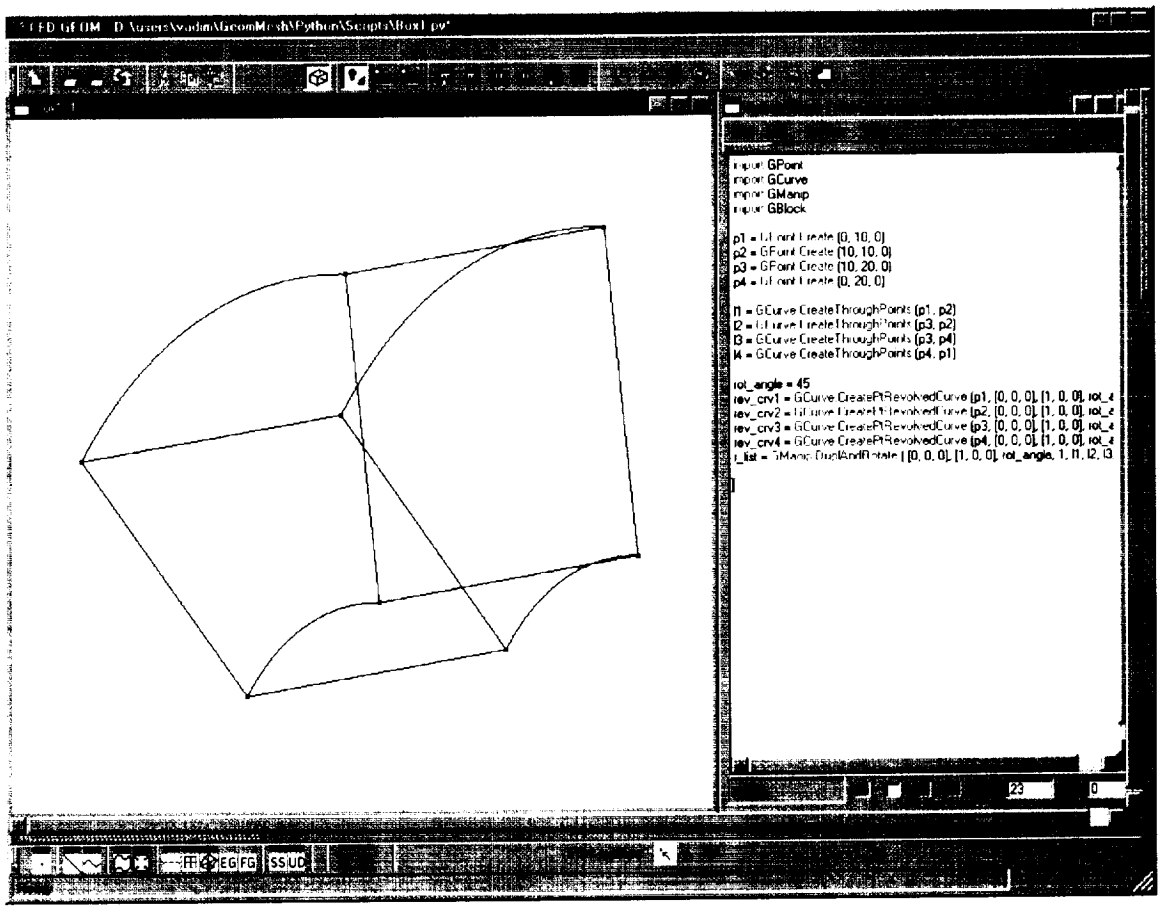

Figure 4. Simple Python Script and Geometric Results

Using the script, shown in Figure 4, we described the process of building geometries. Since we have a geometry and grid generation kernel we can generate grids (structured or unstructured) for the specified geometry. To build the structured mesh on the geometry, generated using Script in Figure 4 the following commands are needed:

```
19. list1 = [l1, l2, l3, l4]
20. list2 = [rev_crv1, rev_crv2, rev_crv3, rev_crv4]
21. GBlock.CreateFromCurves1 (r_list, list1, list2, [10, 10, 10])
```

Adding those three lines to the previous script allows us to generate a structured 3D grid (shown on Figure 5), which can be saved in any grid format and passed to the solver. Lines 19 and 20 combine previously created curves to the Python list in order to be passed to the block creation routine. This block creation routine on line 21 has been specifically added to the geometry kernel to facilitate easy grid domain set up for the Pyhton script end-user. This function accepts collections of curves in the I, J and K direction (separate Python list for each direction) and number of grid points for each I, J, and K direction. This function creates a structured block as output, which could be saved into any supported grid file format. Those 3 extra lines generate a model as shown in Figure 5.

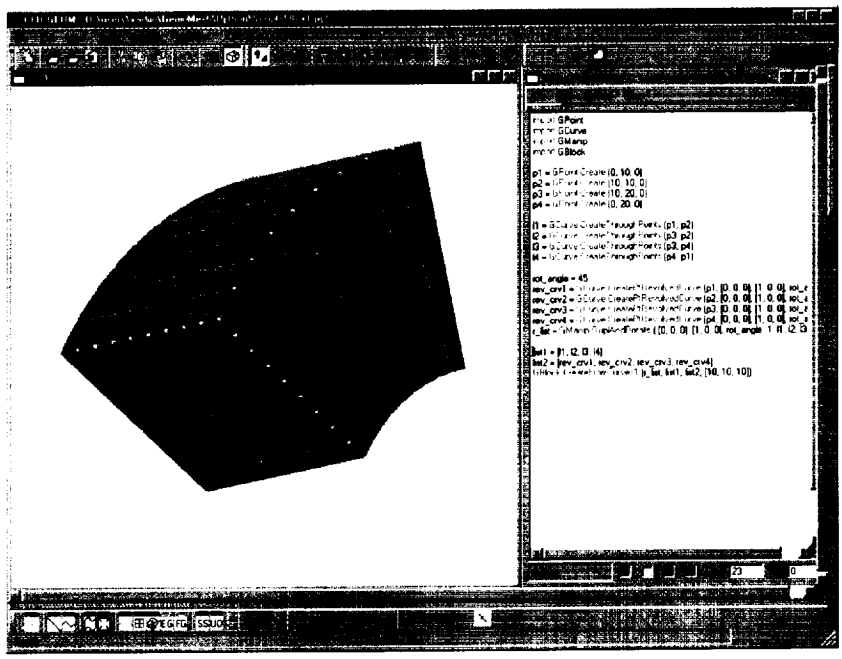

Figure 5. Structured Grid on Geometry

If we want to create 40 points on the I-side of the block, 30 points on the J-side and 50 points on the K-side then all what we need to do is to rewrite function for block creation as follows:

```
21. GBlock.CreateFromCurves1 (list1, list2, list3, [40, 30, 50])
```

Results of these changes are shown on Figure 6.

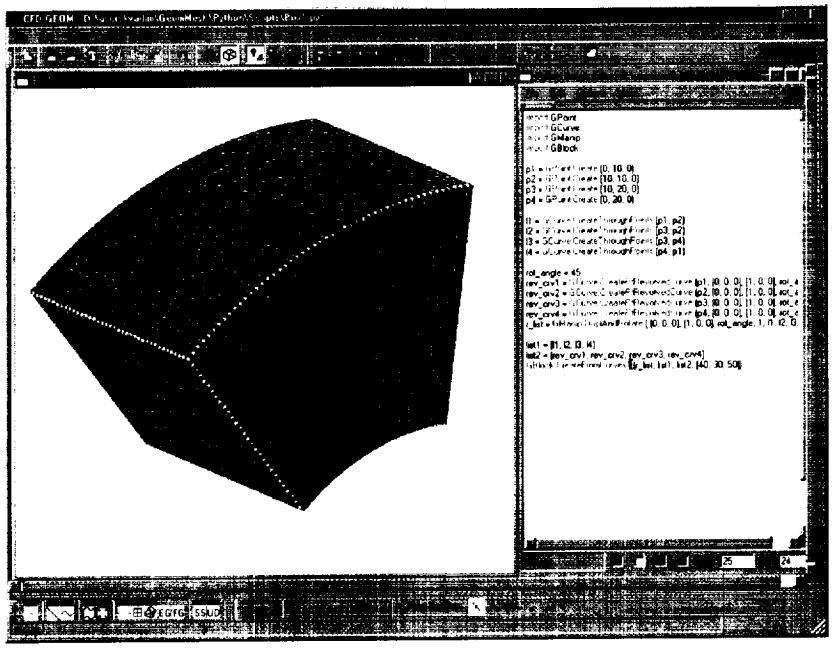

Figure 6. Structured Grid with Higher Density on Geometry

Besides structured meshes for the created geometry, we can also generate unstructured meshes. In order to do that we need first to create surfaces. Then build closed surface set (stitch all surfaces together to form one closed surface), create unstructured domain by specifying one or more surface sets (in this case 1), and generate unstructured surface and volume mesh.

```
19.  surf1 = GSurface.CreateCoons (l1, l2, l3, l4)
20.  surf2 = GSurface.CreateCoons (r_list[0], r_list[1], r_list[2], r_list[3])
21.  surf3 = GSurface.CreateCoons  (rev_crv1, rev_crv2, l1, r_list[0])
22.  surf4 = GSurface.CreateCoons  (rev_crv2, rev_crv3, l2, r_list[1])
23.  surf5 = GSurface.CreateCoons  (rev_crv3, rev_crv4, l3, r_list[2])
24.  surf6 = GSurface.CreateCoons  (rev_crv4, rev_crv1, l4, r_list[3])

25.  GSurface.TrimAll ()
26.  dShell = GUnstruct.CreateClosedSurfaceSet (surf1, surf2, surf3, surf4, surf5, surf6)
27.  domain = GUnstruct.CreateDomain (dShell)
28.  GUnstruct.GenerateTriangularMesh (domain)
29.  GUnstruct.GenerateTetMesh (domain)
```

Before calling this part of the script, we need to add lines "import GSurface" and "import GUnstruct" to our module's import section. This will tell Python that we need to call functions from the modules GSurface and GUnstruct.

Lines 19..24 are required for surface creation out of previously built box outline. Again, we can see here general approach for the developed system – we are calling function Create of a specified type of geometry (function CreateCoons – creates Coons (4-sided) surface) from a module (module GSurface in this example). Therefore, as a result we will get coons surfaces, which interpolates a surface through specified input boundary curves (lines l1, l2, l3 and l4 for the function call GSurface.CreateCoons (l1,l2, l3, l4)). In addition to creating Coons surfaces, other surface creation methods, like Ruled, Revolved, Swept, Extrude are available to the user.

Line 25 of the script is necessary for trimming created surfaces, but can be omitted after further development of the system will be done (again this is a high level vs. low level library issue).

In line 26 of the shown fragment of the script we are creating Closed Surface Set, by passing created surface in any order. Line 27 is required for building unstructured domain. Lines 28-29 are necessarily for generating unstructured surface and volume mesh respectively. Results of this script are shown in Figure 7.

In the future development we plan to simplify the unstructured grid generation process by building one convenient routine, as shown for generating structured meshes.

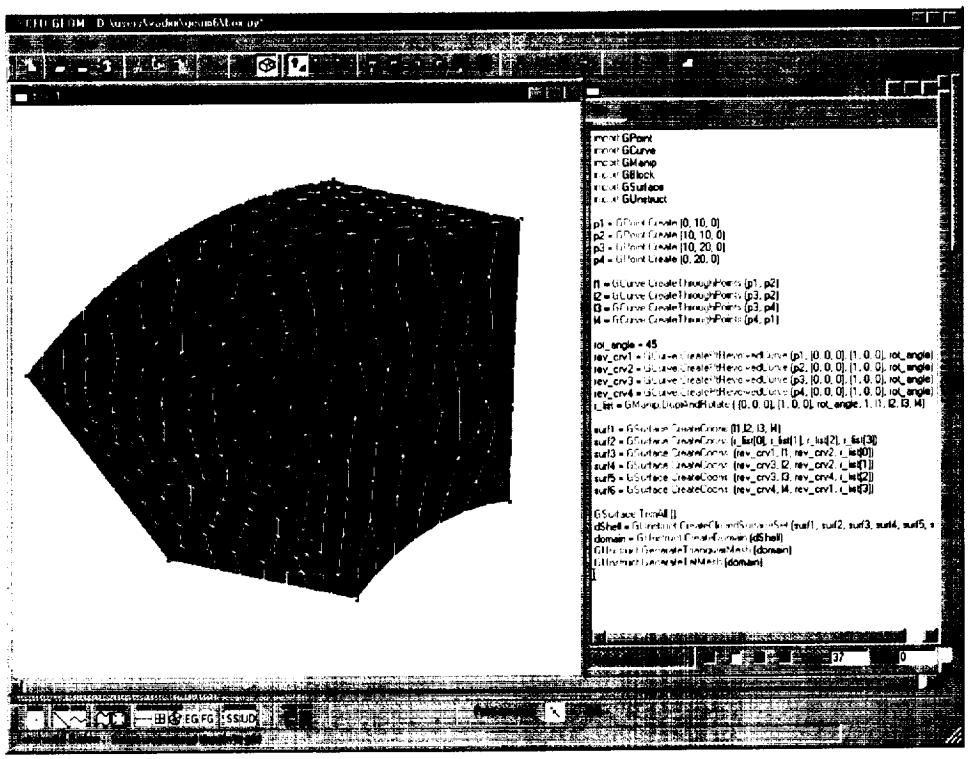

Figure 7. Unstructured Grid on Geometry

3. PARAMETRIC DESIGN EXAMPLE: 2D AIRFOIL AND WING

It is now time to show a first real application of the Python scripting/template facility. In Appendix A, a script is given for parametric definition of 2D airfoils. In Appendix B this script has been extended for multiple element airfoils configurations. In Appendix C, the script in Appendix A has been used to create a new parametric template for 3D wing applications.

In Appendix A, the core of the script template contains the definition of the AirfoilGeometry() function. This function computes the shape of the airfoil given the four digit NACA specification and, optionally, a list specifying the [x,y,z] position offset and a scaling factor.

The computation of the airfoil geometry has been put in a separate function so it can be re-used. Therefore, with some minimal modifications, the script for a 2-element airfoil can be generated easily. This script is shown in Appendix B. This script uses the airfoil scaling and [x,y,z] position offset optional parameters to size and position the additional airfoils.

Figure 8 shows several plots from the various scripts. In the top left window, the NACA 0012 airfoil is shown (Appendix A script). In the bottom left window a multiple element airfoil is shown with its unsuitable grid around the trailing edge. In the bottom right corner window, a point source has been used to fix the trailing edge grid. This window corresponds to the script as shown in Appendix B.

The NACA airfoil function in Appendix A can be used in other scripts for 3D wing applications. In appendix C, a script has been given for a 3D wing consisting of a set of NACA airfoil cross-sections.

This function has the following input parameters:

```
1. NACAAirfoil3D (par = [[0,0,12],[0,0,12],[0,0,12]],
2.                 twist = [0,0,0],
3.                 length = [0,2,4],
4.                 taperratio = [1,1,1],
5.                 sweepangle = 7)
```

On line 1, a python list of lists is used to specify an arbitrary number of NACA airfoil definitions for each cross-section. The example shows 3 sections with the NACA0012 airfoil. The twist allows the user to specify a rotation angle for each corresponding airfoil. The length list specifies in chord-lengths where each airfoil is located (in the 'z' or length direction). The taper ratio is a scaling factor for each airfoil. And finally in line 5, the sweep angle specifies the sweeping angle for the entire wing. Invoking this function with the parameter values mentioned in line 1-5, results in the figure as shown in Figure 9.

The entire script for the wing is given in Appendix C. This script illustrates the capabilities of the design system. The user can create a relatively short script and automatically generate a

geometry and grid definition for a wing. Any design parameters can be build into the system and the corresponding script. It is an open-ended system and the user has full control, i.e. any parameters could be added by modifying the script. A set of parameter values could be imported from any design or optimization software, and the script will create the corresponding geometries and grids automatically. For parametric analysis the various design parameters can be varied in the Python script by repeatedly invoking the function with slightly different parameters. This capability could be a crucial part of any design system because for complex cases the design of experiments phase could require generating up to hundreds of cases.

It is interesting to note here that we believe that even commercial parametric CAD systems will have a difficult time, if not impossible, to match the functionality of our script-based parametric templates. Very high level concepts for a particular application area, such as NACA 4 digit or NACA 5 digit airfoil series, can be modeled with the Python templates and made available to the end user.

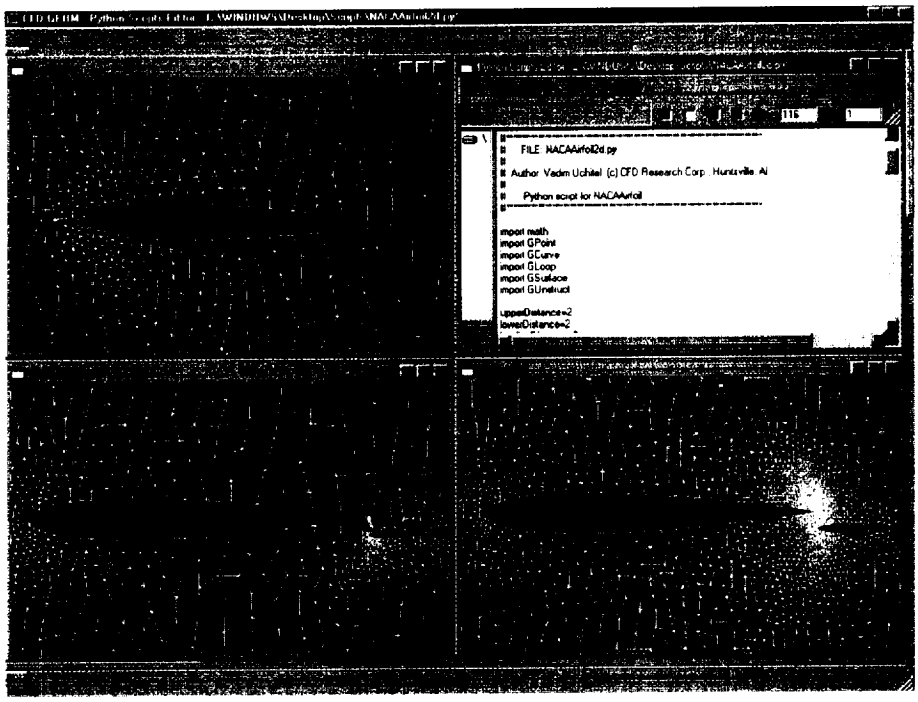

Figure 8. NACA Airfoils as Generated by the Various Scripts Appendix A and B

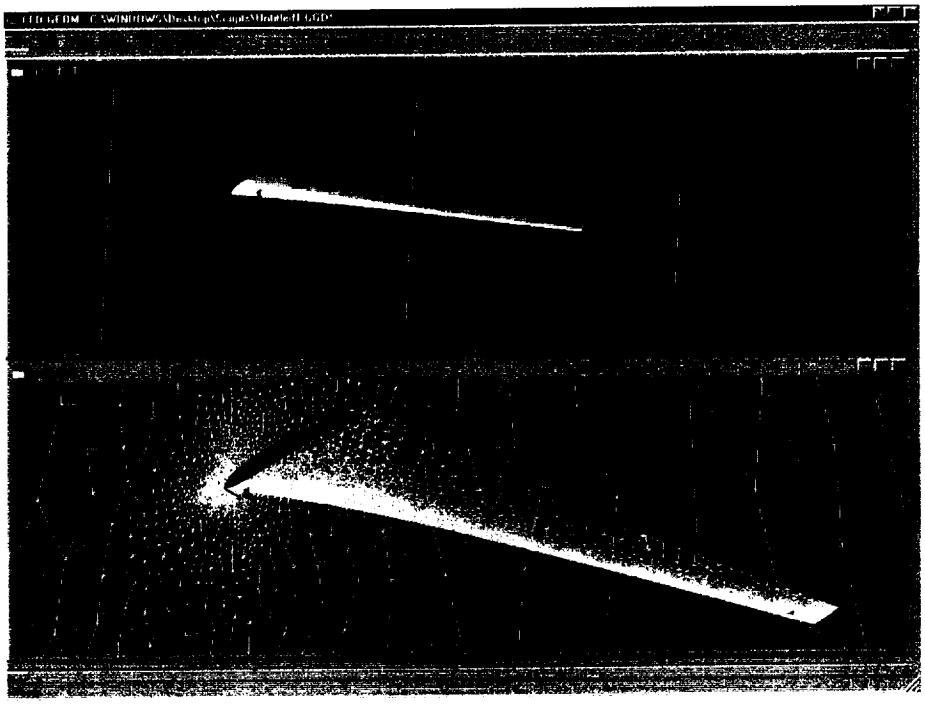

Figure 9. 3D NACA Airfoil Wing (geometry and unstructured tetrahedral grid)

4. TURBOMACHINERY DESIGN EXAMPLES

In this chapter, three turbomachinery component examples will be described in more detail. A script for an impeller is described which uses axial and radial coordinates as input. Second, an impeller based on blade profile curves is given. And finally, a multi-stage rotor/stator combination is described which uses IGES surface blade surface data as input. For all three examples we will use the input data (from a design code) and compute actual 3D geometry from the data and perform the grid generation.

4.1 Impeller based on Axial and Radial Coordinate Input

In chapter two we discussed an impeller design given by the following table and formulas:

Name	*Symbol*	*Description*
Axial Position	z	The coordinate at each axial slice
Radius	r	The radial coordinate
Wrap Angle	Θ	The angle from the radial position
Blade Angle	β	The angle that the blade impart on fluid
Thickness	t	The thickness of the blade
Blade Count	n	Number of blades in the impeller
Blade Offset	α	Angle occupied by a single blade passage

For example, a table consisting of radial and axial coordinates, along with the blade angle and thickness can be used to generate the 12 curves defining the passage through a shrouded impeller:

$$x_{hub,pressure} = r \cos \Theta$$
$$y_{hub,pressure} = r \sin \Theta$$
$$z_{hub,pressure} = z$$

$$x'_{hub,suction} = x_{hub,pressure} + t \cos \beta$$
$$y'_{hub,suction} = y_{hub,pressure} + t \sin \beta$$
$$z'_{hub,suction} = z_{hub,pressure}$$

$$x_{hub,suction} = x'_{hub,pressure} \cos \alpha - y'_{hub,pressure} \sin \alpha$$
$$y_{hub,suction} = x'_{hub,pressure} \sin \alpha + y'_{hub,pressure} \cos \alpha$$
$$z_{hub,suction} = z_{hub,suction}$$

And so on. A typical set of design data has been included in Appendix D. This example represents the UTRC 17DK impeller and the data is taken from reference [1]. Appendix E contains a script which implements the above formulas and computes the 12 characteristic curves making up the flow passage way. This impeller geometry definition corresponds to the left hand side of Figure 2. The example in the next section, see section 4.2, corresponds to the right hand side of Figure 2.

The curves are the hub and shroud pressure and suction side (4 curves), and the inlet curves (4), and the exit curves (4). The following script fragment takes the 12 curve definitions and generates a structured grid block for the passage way. This segment taken from Appendix E is very similar to the examples in the section 2.3.

```
hubList = [hubSuctionCurve,  outletHubCurve, hubPressureCurve,  inletHubCurve]
shroudList = [shroudSuctionCurve, outletShroudCurve, shroudPressureCurve,  inletShroudCurve]
connectList = [inletSuctionCurve, outletSuctionCurve, outletPressureCurve, inletPressureCurve]
impellerBlock = GBlock.CreateFromCurves (hubList, shroudList, connectList, [n, n, n])

GManip.DuplAndRotate ([0,0,0], [0,0,1], 30, 11, impellerBlock)
```

Python lists are created for the curves on the hub, the shroud, and the curves which connect the hub to the shroud. Those curve lists are then passed on the block creation routine, which will in turn generate a structured grid based on TFI's (Trans Finite Interpolants). For this reason no actual surface geometry is necessary as input to this routine, i.e. the surfaces are generated as part of the grid generation process. The grid density can be controlled from the script. In this case 'n' points are generated in all (I, J, and K) directions. The grid distribution could be controlled from the script as well with some extra parameters. This will be a Phase II extension. Similarly, an algebraic or elliptic grid smoothing function needs to be invoked as well. Again, all this functionality is already part of the grid generation kernel but just needs to be incorporated into the script in Phase II. Optionally, this grid domain can be rotated to create a multi-block structured grid for the full impeller by invoking the GManip.DuplandRotate() function. In this case the grid block gets rotated 30 degrees and 11 times duplicated (making a total of 12 grid blocks), see Figure 10.

This script segment is very powerful and illustrates many ideas behind the High Level Layer of the architecture. The shape of each curve is computed by a separate subroutine based on the radial and axial input data, see Appendix E. Once the shape has been computed it gets assigned a name, e.g. 'hubPressureCurve', which can be used in subsequent computations. Python lists are used to aggregate the various geometric entities and refer to them with one new name, e.g. 'hubList'. In principle, any geometry primitive can be made part of the list. This is a good example of the **'late binding'** feature of Python. For example, the curve creation routine allows Python lists of points as input, while the block creation routine needs lists of curves as input. The checking that the proper input data types are passed to the function is part of the High Level Layer, and proper error messages are given to the end user. A good example of **overloading** is given by the 'GManip.DuplandRotate()' function. In section 2.3 we were duplicating and rotating a set of 4 curves to create the actual geometry. For this example we are duplicating and rotating a volumetric grid block entity. This functionality is achieved with the same function call, and is transparent to the end user. **The aforementioned computer science aspects of the system facilitate an intuitive system with a high level of expressiveness.**

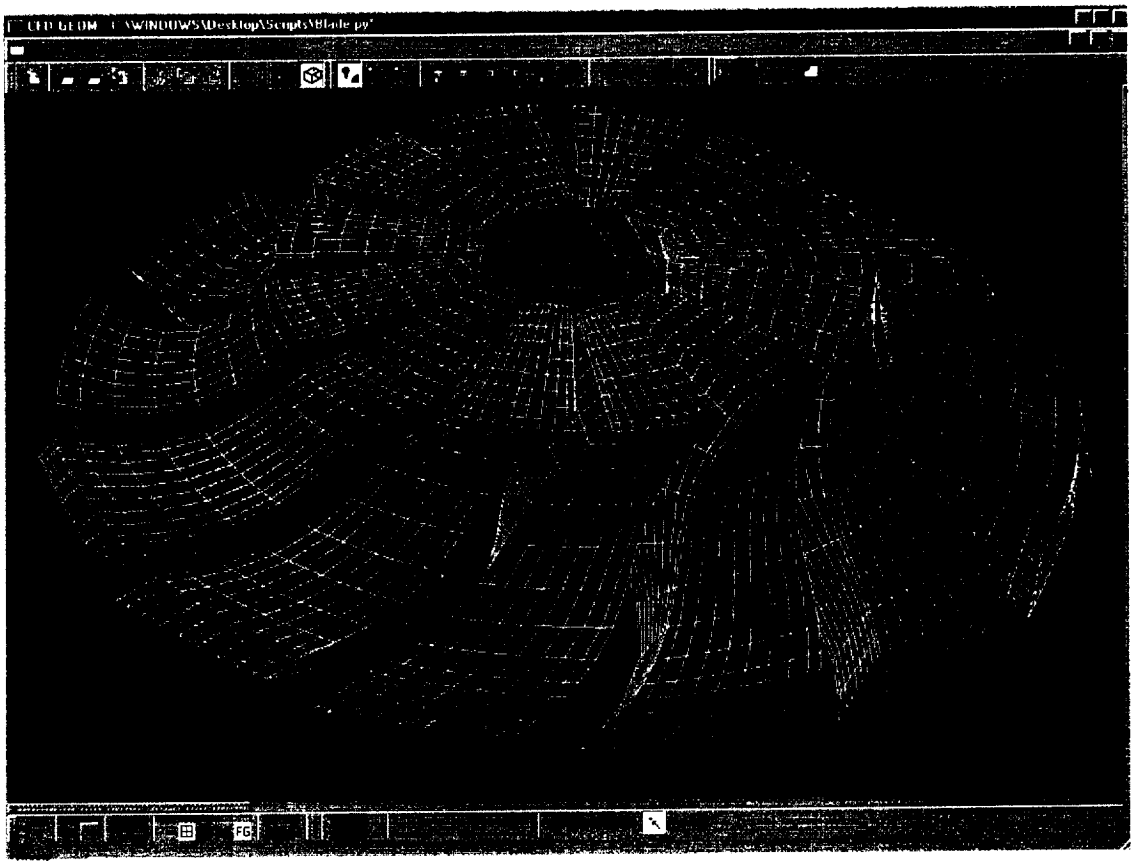

Figure 10. UTRC 17DK Impeller Generated with Script in Appendix E

When the script in Appendix E is executed with the data input deck in Appendix D (i.e. UTRC 17DK impeller) the following configuration appears on the screen, see Figure 10.

4.3 Impeller Based on Blade Profile Data

In the next script, we show an example for output data from the BLADEGEN design code, in which the blade profile curves are directly read into the script. This is shown as the right hand path in Figure 2. With those curves, the script follows a similar path in which the geometry gets created and a structured grid generated.

Figure 11 shows a visual representation of the BLADEGEN program output. In fact the output data consists of lists of (x, y, z) points which we have interpolated with curves for clarity.

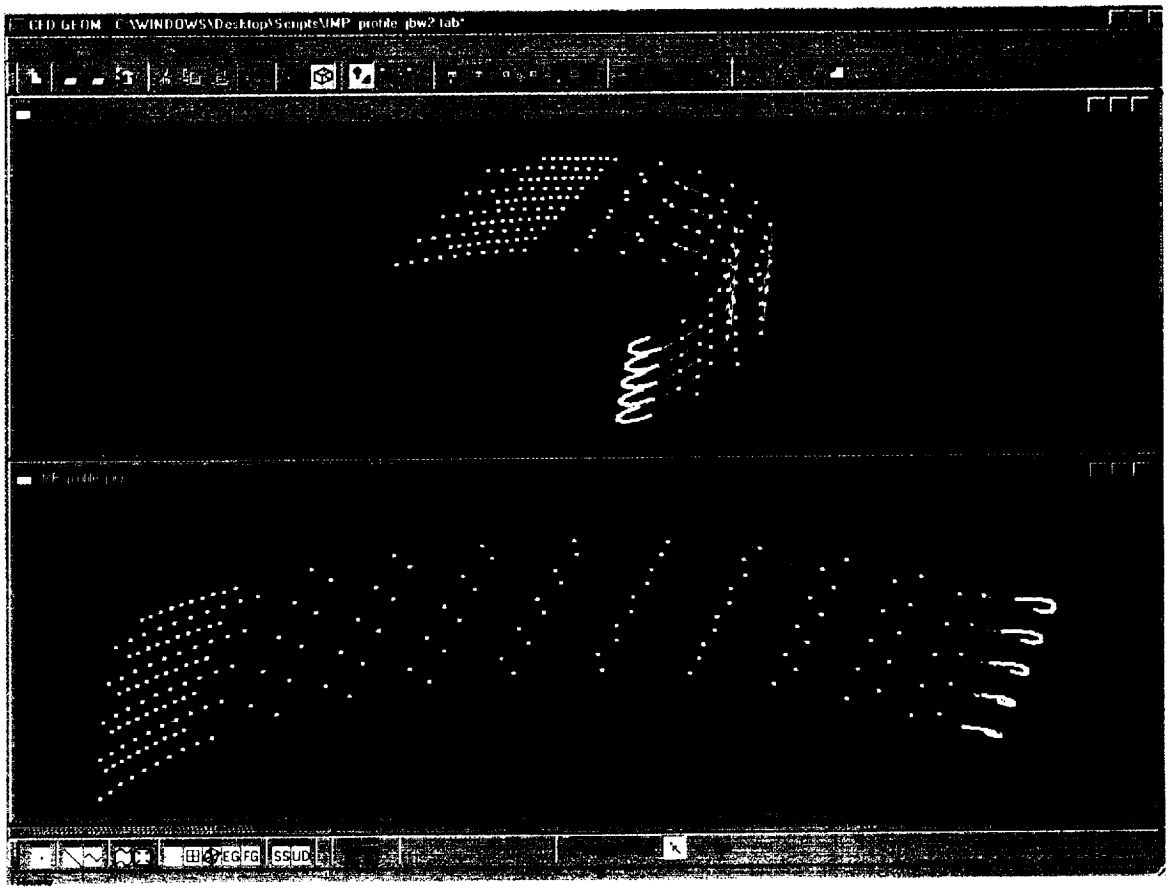

Figure 11. Two Views with Impeller Design Output from BLADEGEN. In yellow the 5 cross-sections of the impeller blade are given. The shroud curves are orange (downstream) and green (upstream). The hub curves are red (downstream) and magenta (upstream).

The script given in Appendix F has been developed to create a 3D geometry and a two-domain structured grid based on the point data presented in Figure 11. The resulting geometry and grid are shown in Figure 12. Again, no focus has been given on optimizing grid distribution and quality.

The script starts with reading the point data shown in Figure 11. The script creates separate point lists from the various data items. Curves are then interpolated through those point lists. The script is more elaborate than the previous ones in the sense that the surface geometry is created. The corresponding script segment from Appendix F shows the creation of a number of ruled surfaces through the actual blade profile curves.

```
BladeSuction = GSurface.CreateRuled (Blade[0], Blade[2], Blade[4], Blade[6], Blade[8])
BladePressure = GSurface.CreateRuled (Blade[1], Blade[3], Blade[5], Blade[7], Blade[9])
Cap = GSurface.CreateRuled (Blade[9], Blade[8])
Cap2 = GSurface.CreateRuled (c_cur1, c_cur2)
```

The above segment is a good example of the optional parameters feature in Python. For example the 'BladeSuction' surface is a ruled surface through 5 blade profile curves, while the 'cap' surface is a ruled surface between two curves. Alternatively, the 'Blade' curves could have been put in a Python list construct and passed on the 'GSurface_CreateRuled()' function.

In another script segment from Appendix F, the Python overloading feature can be illustrated again. In this invocation of 'GManip_DuplandRotate()' function, a list of surfaces is passed to this routine. Previous examples have shown curves and volumetric grid blocks.

```
GManip.DuplAndRotate([0,0,0],[0,0,1],-51.4285,11,[BladeSuction,BladePressure,Cap,Cap2])
```

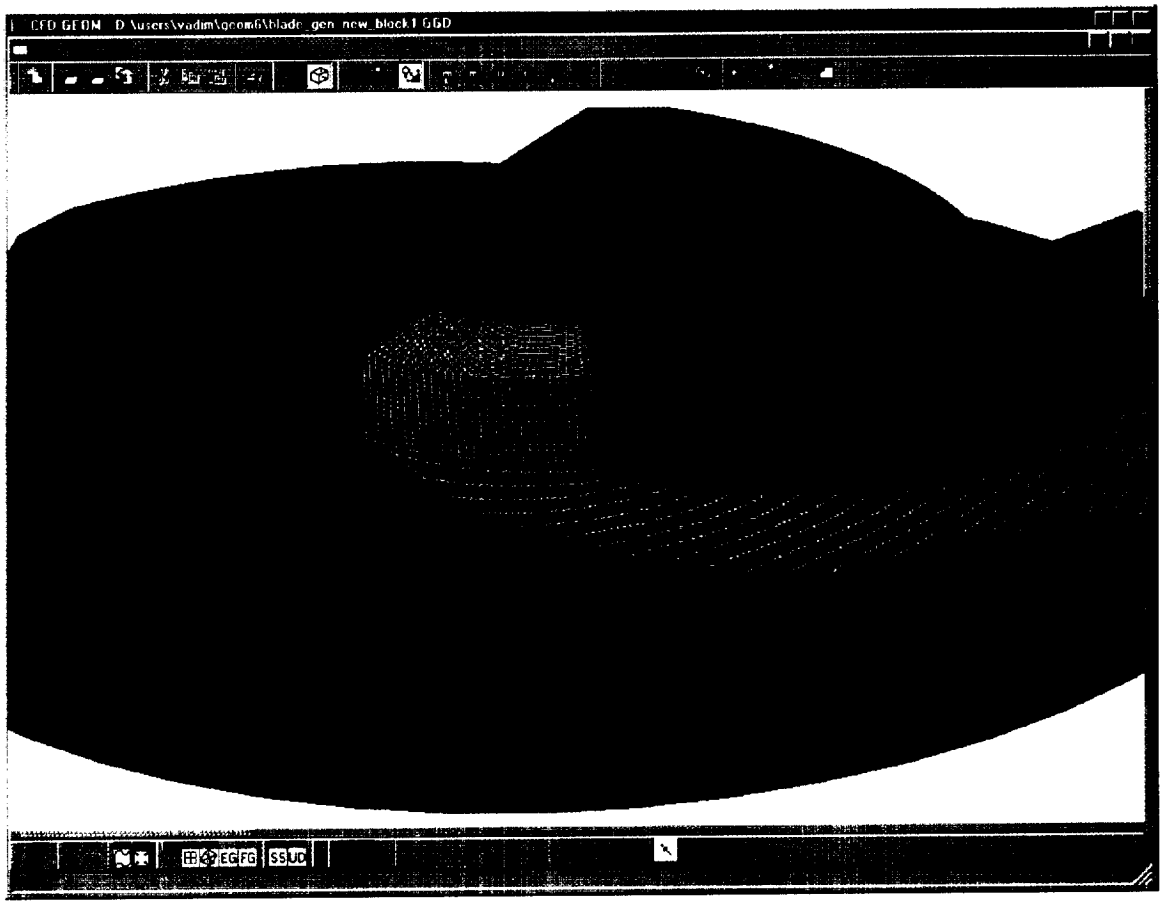

Figure 12. Resulting Geometry and Grid from Running Script in Appendix F with the Input Data Points in Figure 11

This impeller scripts run to completion in a few seconds on a 300 Mhz PC. Without the support of the scripts, the actual grid generation process may take days to complete for an average CFD user with a general purpose grid generation software. Obviously one benefits the most in

production environments in which many cases need to be analyzed for parametric analysis and/or design optimization.

4.4 Rotor and Stator Combination Based on IGES Surface Definitions

In this section a script is shown for a rotor and stator combination starting from a set of blade surface descriptions (NURBS surfaces in IGES format). The script for the stator is shown in Appendix G, while the script for the rotor is given in Appendix H. When the script is executed, the 3D geometry and grid are generated as shown in Figure 13. Both scripts are roughly in page in length.

The scripts in Appendix G and H generate the 3D geometry and the CFD grid. Optionally, the 'metal' could have been gridded for FEA. In Phase II we will show some examples of this technology. The way the grid has been built assumes that the flow solver has 'mixing plane' boundary condition, i.e. the domains don't have 1-to-1 grid match. By adjusting the geometry somewhat, a 1-to-1 grid match between the rotor and stator could be obtained.

We have included this example as a prelude for Phase II. For Phase II we want to stack the various scripts to generate multi-stage turbomachinery components.

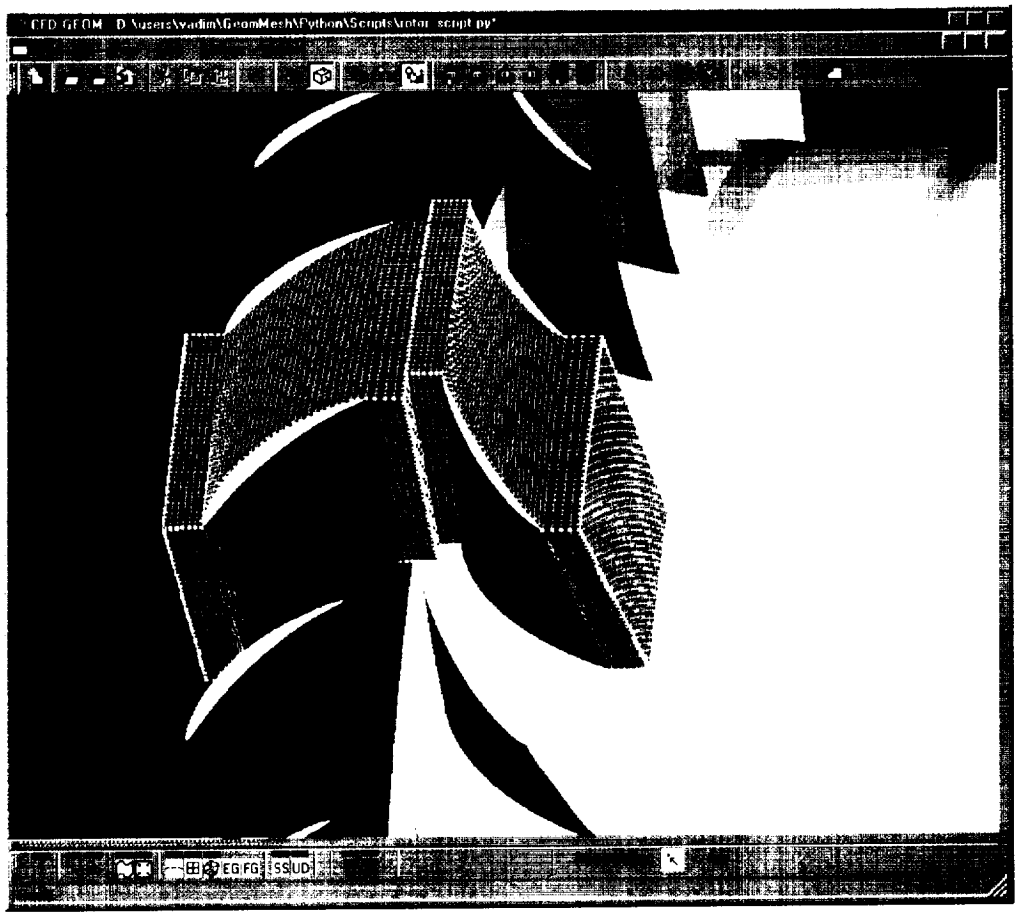

Figure 13. Rotor and Stator Geometry and Grid as Generated by Script in Appendix G and H

5. COMMERCIALIZATION OF TECHNOLOGY

The proposed tool for automated geometry modeling and grid generation for turbomachinery applications fulfills a direct need in the turbomachinery design community. One of the main bottlenecks in this area is the lack of an automated grid generation process. Traditionally this process is fairly tedious, and it seriously impacts the turnaround times. Due to the demonstrated reduction in geometry and grid development times, a significant interest from the industry is envisioned.

Testing of the software in design settings will also be undertaken by actively soliciting joint ventures with turbomachine manufacturers, to establish beta-testing program, in addition to testing at CFDRC and Penn State. Specific applications which will be targeted will include gas turbine engine compressor and turbine sections, centrifugal compressors (compressible flows), axial and mixed flow pumps, centrifugal pumps, and pelton wheels (liquid flows). CFDRC has working relationships with several manufacturers, e.g. GE, P&W, Dresser Rand, Sulzer Pumps and collaboration with these and other manufacturers will be developed.

For this purpose, we have contacted approximately 85 engineers from US turbomachinery design companies and send them a copy of the second Phase I progress report for their evaluation and feedback. Depending on the feedback, we plan to invite a few of the respondents for active collaboration. Other respondents will be kept as potential sales targets for commercialization. At the end of Phase II, we plan to repeat this process (now with a much more mature product) and generate opportunities for Phase III collaboration and software sales.

Another commercialization opportunity has been identified with Concepts ETI Inc. During the Phase I project we have been approached by this company based on our Phase I proposal abstract. This company specializes in axial and radial turbomachinery design codes, and over 40 organizations license their design software. Currently, they are offering their users a standard, i.e. manual, third party grid generation tool. However, they feel that grid generation is a bottleneck in their design process, and they would like to offer more automated tools to their users. During Phase II, they will evaluate the software and give us feedback and suggestions for further improvements. Upon favorable evaluation, they may become a VAR for the system in Phase III.

A subsidiary of CFDRC in Germany (CFDRC Gmbh) has a large support contract with a European manufacturer of turbomachinery components. One aspect of this contract is the large number of cases which need to be analyzed. After evaluating the second progress report, CFDRC GmbH and their customer have decided to adopt the script based template technology. This will provide valuable feedback during Phase II on actual design problems, as well as direct sales opportunities in the future.

Independent of aforementioned pure project related efforts, CFDRC has an established track record in software commercialization. It has a top-notch technical support infrastructure for commercial software, including specialized departments for software sales and marketing, technical user support, training, etc. CFDRC has sales and technical support offices throughout the world. Currently CFDRC has over 400 organizations license its software worldwide. This

successful commercialization record has been recognized by the by the Small Business Administration:

- 1995 SBA SBIR Annual Report: Success Stories of 1995; and
- 1996 Tibbetts Award Winner: U.S. SBA Report, July 1996.

Specifically with regards to NASA Glenn and this STTR, the current project is based on the CFD-GEOM geometry modeling and grid generation kernel. This product is partly the result of a previous SBIR with NASA Glenn. The commercial success of this product is evident from a variety of NASA publications:

- NASA Tech Briefs, March 1999, Vol. 23, No. 3
- NASA SBIR Success Story
  http://technology.larc.nasa.gov/scripts/nls_ax.dll/w3SuccItem(801049)
- (will be featured in) NASA Spinoff Magazine 2000

6. CONCLUDING REMARKS AND RECOMMENDATIONS FOR PHASE II

6.1 Conclusions

The overall objective of the proposed effort is to develop a highly automated software system for geometry and grid generation for a variety of turbomachinery configurations. The focus will be primarily on establishing the technology to create detailed 3D geometry and computational grids from the output of turbomachinery design codes in a highly automated fashion. The proposed system will facilitate detailed geometry descriptions based on trimmed NURBS modeling methods. Both multi-block structured and unstructured grid generation technology will be supported by the system

Therefore the overall project is aimed at development of a script based approach for automated, parametric geometry modeling and grid generation for turbomachinery applications. The final system has the following characteristics:

- simplify the geometry building stage via graphical and script based tools,
- automate the mesh generation process for quality computational grids,
- speed up design optimization process through parametrics and customization,
- interface CAD/CAM systems for direct data exchange with design and manufacturing, and
- provide meshes for CFD and CAE in appropriate formats to increase the utility of the software.

The Phase I effort has successfully demonstrated the feasibility of such a system. We have developed a prototype of the software, and we have applied this prototype system on a variety of turbomachinery components including the centrifugal compressor. We have demonstrated the system for a simple multi-stage rotor-stator combination. And finally, we have demonstrated that the script-based templates can be used in combination with several input formats, such as blades defined by radial and axial data, sets of blade profiles curves, and with blade NURBS surface definitions in IGES format.

Some important additional objectives of the Phase I research effort were identified as:

- provide for a high level of end-user efficiency and effectiveness, this is accomplished by establishing individual templates for the various turbomachinery components.
- Tool must be easily customizable by the end-user. Each template is in the form of a script, which is easily modifiable, and new ones can be easily created by the end-user.

We have demonstrated that the script based templates are relatively short (in the order of one to two pages), and that scripts are easily modified from one configuration to the other. The scripts are not compiled and can be interactively modified and built up by the end-user. We feel that the real pay-off is in situations where many similar design configurations need to be analyzed (design optimization and parametric analysis cases). This system could automatically generate tens or hundreds of geometries and grids in a matter of minutes. In the largest cases, e.g. many revolved surfaces and grid blocks, the running times are 10-20 seconds per configuration.

6.2 Recommendations for Phase II

Throughout this report we have identified many issues which will be addressed in a Phase II effort. From an overall point of view, we feel that this Phase I has proven the feasibility of one of the most critical components of the overall system. For Phase II we would like to expand the system as follows:

1. Bring more functionality from the geometry kernel into the scripting facility, allowing more elaborate designs, and construction methods in general.

2. Bring more grid generation functionality into the scripting language, such as better grid control, algebraic and elliptic grid smoothing, etc.

3. Add specification of boundary conditions.

4. Demonstrate the system on more turbomachinery components, especially for multi-stage components. Develop scripts/templates for a wide variety of turbomachinery applications, such as multi-stage turbomachinery, shrouded stator/rotor, tip clearance integration, film cooling configurations, cavity configurations, etc.

5. On the front-end of the system we would like to integrate the system with NASA design codes. In coordination with the NASA technical monitor select a NASA code, integrate with system, and demonstrate it as one coupled system. This will be the primary focus with our University Partner.

6. At the back- end of the system, we would like to integrate the system with commercial CAD/PDM software. Candidate systems available at CFDRC are Unigraphics or PRO/Engineer. In consultation with the technical contract monitor at NASA we will select one system. The idea here is to generate the geometry and the corresponding FEA grids ('metal parts') and CFD grids ('air'), and write this to a parametric CAD system.

This will result in a fully coupled system for turbomachinery design. By integrating the various existing tools, it facilitates design optimization and parametric analysis. The script based turbomachinery geometry and grid templates enable large scale problems, i.e. automation of many design configurations and multi-stage components. The proposed system will directly support a modern computing approach in which the CAD/PDM system is central in the overall analysis environment. Other analysis software such as structural analysis, computational fluid dynamics, or entire frameworks such as NPSS would directly interface with the CAD/PDM system as well, and in fact closing the analyses loop.

7. REFERENCES

1. "Centrifugal Compressor Impeller Aerodynamics: An Experimental Investigation," Joslyn, H.D., Brasz, J.J., Dring, R.P., Transactions of ASME, October 1991.

APPENDIX A

SAMPLE PYTHON SCRIPT FOR NACA 2D AIRFOIL

```
#*************************************************************************
#              (c) CFD Research Corporation
#                    Huntsville, Alabama, USA.
#                         2000.
#
#*************************************************************************
# FILE: NACAAirfoil2d.py
#
#  Author: Vadim Uchitel, (c) CFD Research Corp., Huntsville, Al.
#
#  Python script for 2D NACA Airfoil
#*************************************************************************

import math
import GPoint
import GCurve
import GLoop
import GSurface
import GUnstruct

upperDistance=2
lowerDistance=2
leadingDistance=2
trailingDistance=2
curvePoints=25

# __________Check for reasonable inputs__________

def check_input (camber, position, thickness, scale):

    InvalidValueException = 'InvalidValueException'
    #
    if ( scale <= 0.0 ):
      raise InvalidValueException
    #
    if (not((camber >= 0.0) and (camber < 10.0))):
      raise InvalidValueException
    if (not((position >= 0.0) and (position < 10.0))):
      raise InvalidValueException
    if (not((thickness > 4.0) and (thickness < 25.0))):
      raise InvalidValueException
    #
    if (not(curvePoints > 10)):
      raise InvalidValueException

#__________Compute Airfoil Geometry__________

def AirfoilGeometry(camber =0,position =0,thickness =12,twist = 0.,offset=[0.0, 0.0, 0.0],scale = 1.0):

    check_input (camber, position, thickness, scale)

    c = 0.01*camber
    p = 0.1*position
    t = 0.01*thickness
    #
    xinit = offset[0]*math.cos(GGlobal.DTOR (-twist)) + offset[1]*math.sin(GGlobal.DTOR(-twist))
    yinit = -offset[0]*math.sin(GGlobal.DTOR (-twist)) + offset[1]*math.cos(GGlobal.DTOR(-twist))
    leadingPoint = GPoint.Create(xinit,yinit,offset[2])
    #
    upperPoint = []
    lowerPoint = []
    for i in range(1,curvePoints):
      xOrdinate = 0.5 - 0.5*math.cos(i*math.pi/curvePoints)
```

```
      tValue = 0.28430*pow(xOrdinate,3)-0.10150*pow(xOrdinate,4)
      yThickness = t/0.20*(0.29690*math.sqrt(xOrdinate)-0.12600*xOrdinate-0.35160*pow(xOrdinate,2)+tValue)
      #
      if (xOrdinate < p):
        yCamber = c/p/p*(2.0*p*xOrdinate - pow(xOrdinate,2))
        yOffset = 2.0*c/p/p*(p - xOrdinate)
      else:
        yCamber = c/(1.0-p)/(1.0-p)*(1.0-2.0*p + 2.0*p*xOrdinate - pow(xOrdinate,2))
        yOffset = 2.0*c/(1.0 - p)/(1.0 - p)*(p - xOrdinate)
      #
      if (xOrdinate < 0.01):
        theta = 0.0
      else:
        theta = math.atan(yOffset)
      #
      xUpperPos = scale * (xOrdinate - yThickness*math.sin(theta)) + offset[0]
      yUpperPos = scale * (yCamber + yThickness*math.cos(theta)) + offset[1]
      xLowerPos = scale * (xOrdinate + yThickness*math.sin(theta)) + offset[0]
      yLowerPos = scale * (yCamber - yThickness*math.cos(theta)) + offset[1]

      xUpperPos = xUpperPos*math.cos(GGlobal.DTOR (-twist)) + yUpperPos*math.sin(GGlobal.DTOR(-twist))
      yUpperPos = -xUpperPos*math.sin(GGlobal.DTOR (-twist)) + yUpperPos*math.cos(GGlobal.DTOR (-twist))
      xLowerPos = xLowerPos*math.cos(GGlobal.DTOR (-twist)) + yLowerPos*math.sin(GGlobal.DTOR(-twist))
      yLowerPos = -xLowerPos*math.sin(GGlobal.DTOR (-twist)) + yLowerPos*math.cos(GGlobal.DTOR (-twist))
      #
      upperPoint.append (GPoint.Create(xUpperPos,yUpperPos,offset[2]))
      lowerPoint.append (GPoint.Create(xLowerPos,yLowerPos,offset[2]))
    #
    xlast= ((1.0*scale)+offset[0])*math.cos(GGlobal.DTOR(-twist)) + offset[1]*math.sin(GGlobal.DTOR(-twist))
    ylast= -((1.0*scale)+offset[0])*math.sin(GGlobal.DTOR(-twist)) + offset[1]*math.cos(GGlobal.DTOR(-twist))
    trailingPoint = GPoint.Create(xlast,ylast,offset[2])
    #
    upperCurve = GCurve.CreateThroughPoints(leadingPoint, upperPoint, trailingPoint)
    lowerCurve = GCurve.CreateThroughPoints(leadingPoint, lowerPoint, trailingPoint)
    airfoilCurves = [upperCurve, lowerCurve]
    return airfoilCurves


#_________________Create Unstructured Grid Around Airfoil_____________________

#Create Computational Domain
min_point = [-1.0*leadingDistance,-1.0*lowerDistance,0.0]
max_point = [1.0+trailingDistance, upperDistance, 0.0]
srf = GSurface.CreateBoundingPlane ( min_point, max_point)
GSurface.TrimAll ()

#Create Airfoil Shape NACA 0012
airfoilCurves1 = AirfoilGeometry (0, 0, 12, 0., [0.0, 0.0, 0.0], 1.0)
airfoilLoop1 = GLoop.Create  (airfoilCurves1)
GSurface.AddLoop (srf, airfoilLoop1)

dShell = GUnstruct.CreateOpenSurfaceSet (srf)
domain = GUnstruct.CreateDomain (dShell)

GUnstruct.GenerateTriangularMesh (domain)
```

APPENDIX B

SAMPLE PYTHON SCRIPT FOR MULTIPLE NACA 2D AIRFOILS

```
#_____Create Unstructured Grid Around Multiple Airfoils_____

#Create Computational Domain
min_point = [-1.0*leadingDistance,-1.0*lowerDistance,0.0]
max_point = [1.0+trailingDistance, upperDistance, 0.0]
srf = GSurface.CreateBoundingPlane ( min_point, max_point)
GSurface.TrimAll ()

#Create Main Airfoil Shape NACA 0012
airfoilCurves1 = AirfoilGeometry (0, 0, 12, 0., [0.0, 0.0, 0.0], 1.0)
airfoilLoop1 = GLoop.Create  (airfoilCurves1)
GSurface.AddLoop (srf, airfoilLoop1)

#Create Airfoil Shape NACA 0012 for flap
airfoilCurves2 = AirfoilGeometry (0, 0, 12, 0., [1.0, -0.05, 0.0], 0.2)
airfoilLoop2 = GLoop.Create  (airfoilCurves2)
GSurface.AddLoop (srf, airfoilLoop2)

dShell = GUnstruct.CreateOpenSurfaceSet (srf)
domain = GUnstruct.CreateDomain (dShell)

# Adding Source to domain to control trailing edge grid density
src_point = GCurve.GetEndPoint (airfoilCurves1[0])
GUnstruct.AddSource (domain, src_point, 0.0001)

GUnstruct.GenerateTriangularMesh (domain)
```

APPENDIX C

SAMPLE PYTHON SCRIPT FOR 3D NACA AIRFOIL DEFINITION WITH MULTIPLE CROSS-SECTIONS

```
#
def NACAAirfoil3D (par = [[0,0,12],[0,0,12],[0,0,12]],twist=[0,0,0],length=[0,2,4],taperratio=[1,1,1],sweepangle=7):

   if len (par) < 2:
      raise "There should be more than 1 airfoil"

   if (len (par) != len (twist)) or (len (par) != len (length)) or (len (par) != len (taperratio)):
      raise "Not all the required data was provided for all airfoils"

   numAirfoils = len (par)
   airfoilCurves = []
   for x in range (0, numAirfoils):
      xpos = (0.5 - 0.5/taperratio[x]) + length[x] * math.tan (GGlobal.DTOR (sweepangle))
      offset = [xpos, 0.0, length[x]]
      airfoilCurves.append (AirfoilGeometry (par[x][0], par[x][1], par[x][2], twist[x], offset, 1./taperratio[x]))

   airfoilLoop = GLoop.Create  (airfoilCurves[0])

   min_point = [-1.0*leadingDistance,-1.0*lowerDistance,length[0]]
   max_point = [1.0+trailingDistance, upperDistance, length[numAirfoils-1]*2]

   srfs = GSurface.CreateBoundingBox ( min_point, max_point)
   GSurface.AddLoop (srfs[0], airfoilLoop)

   ruledTopCurves = []
   ruledBottomCurves = []
   for x in range (0, numAirfoils):
            ruledTopCurves.append ([airfoilCurves[x][0]])
            ruledBottomCurves.append ([airfoilCurves[x][1]])
   usurf = GSurface.CreateRuled (ruledTopCurves)
   lsurf = GSurface.CreateRuled (ruledBottomCurves)
   bsurf = GSurface.CreateRuled (airfoilCurves[numAirfoils-1][0], airfoilCurves[numAirfoils-1][1])
   GSurface.TrimAll ()

   dShell = GUnstruct.CreateClosedSurfaceSet (srfs, bsurf, usurf, lsurf)
   domain = GUnstruct.CreateDomain (dShell)

   GUnstruct.GenerateTriangularMesh (domain)
   GUnstruct.GenerateTetMesh (domain)




#_____Invoke 3D airfoil with 3 cross sections_____

#First    section is NACA 4 4 12, twist angle is -2 degrees, z position is 0, taperratio = 1
#Second section is NACA 1 4 12, twist angle is  0 degrees, z position is 1, taperratio = 2
#Third    section is NACA 4 4  8, twist angle is  3 degrees, z position is 3, taperratio = 6
#Sweep angle for entire wing is 13 degrees


NACAAirfoil3D([[4,4,12], [1,4,12], [1,4,8]], [-2,0,3], [0,1,3], [1,2,6], 13)
```

APPENDIX D

IMPELLER DESIGN DATA (UTRC 17DK IMPELLER)

```
#*****************************************************************************
#
# Python File Blade_Data.py
#
# Axial and Radial Coordinates, Blade Angles, Thickness Distribution for UTRC 17DK Impeller
#
# Columns are z, r, x, y, Θ, β, t
#
#*****************************************************************************


fHubData    =       [[-5.060,1.897,0.088,1.895,2.644,34.000,0.3111],
                     [-4.567,2.003,0.420,1.959,12.111,31.113,0.3406],
                     [-4.099,2.143,0.715,2.021,19.475,26.115,0.3420],
                     [-3.656,2.316,0.967,2.105,24.674,19.999,0.3382],
                     [-3.238,2.522,1.184,2.227,27.993,13.504,0.3342],
                     [-2.846,2.762,1.373,2.397,29.812,7.183,0.3295],
                     [-2.479,3.035,1.540,2.615,30.494,1.451,0.3243],
                     [-2.138,3.341,1.689,2.883,30.356,-3.367,0.3209],
                     [-1.822,3.681,1.822,3.198,29.669,-7.007,0.3184],
                     [-1.531,4.053,1.944,3.556,28.668,-9.252,0.3167],
                     [-1.265,4.459,2.064,3.953,27.565,-9.932,0.3160],
                     [-1.025,4.899,2.189,4.382,26.546,-8.934,0.3160],
                     [-0.810,5.371,2.336,4.837,25.775,-6.230,0.3160],
                     [-0.620,5.877,2.519,5.310,25.383,-1.914,0.3160],
                     [-0.455,6.416,2.759,5.793,25.468,3.763,0.3160],
                     [-0.316,6.989,3.074,6.276,26.091,10.357,0.3160],
                     [-0.202,7.594,3.481,6.750,27.280,17.219,0.3160],
                     [-0.114,8.233,3.993,7.200,29.015,23.535,0.3160],
                     [-0.051,8.906,4.615,7.616,31.215,28.437,0.3160],
                     [-0.013,9.611,5.336,7.994,33.725,31.261,0.3160],
                     [ 0.00,10.350,6.133,8.337,36.337,32.000,0.3160]]
fShroudData  =       [[-4.774,6.094,0.000,6.094,0.000,57.900,0.0472],
                     [-4.456,6.105,0.456,6.088,4.285,52.394,0.0678],
                     [-4.153,6.137,0.817,6.082,7.654,47.038,0.0884],
                     [-3.864,6.190,1.111,6.089,10.339,41.936,0.1165],
                     [-3.589,6.264,1.356,6.116,12.505,37.194,0.1307],
                     [-3.328,6.360,1.568,6.164,14.274,32.902,0.1446],
                     [-3.082,6.477,1.757,6.234,15.739,29.130,0.1568],
                     [-2.849,6.615,1.931,6.327,16.973,25.932,0.1630],
                     [-2.631,6.775,2.097,6.442,18.035,23.354,0.1632],
                     [-2.427,6.956,2.261,6.578,18.972,21.432,0.1579],
                     [-2.237,7.158,2.428,6.734,19.829,20.195,0.1483],
                     [-2.061,7.381,2.602,6.908,20.641,19.661,0.1385],
                     [-1.899,7.626,2.788,7.098,21.445,19.826,0.1301],
                     [-1.752,7.892,2.991,7.303,22.273,20.650,0.1232],
                     [-1.619,8.179,3.216,7.521,23.155,22.043,0.1180],
                     [-1.500,8.488,3.468,7.747,24.118,23.845,0.1143],
                     [-1.395,8.818,3.751,7.980,25.178,25.823,0.1121],
                     [-1.304,9.169,4.069,8.217,26.343,27.681,0.1104],
                     [-1.228,9.541,4.421,8.455,27.601,29.101,0.1095],
                     [-1.165,9.935,4.806,8.696,28.927,29.854,0.1092],
                     [-1.117,10.350,5.220,8.937,30.285,30.000,0.1090]]
```

APPENDIX E

IMPELLER SCRIPT BASED ON AXIAL AND RADIAL COORDINATES, BLADE ANGLES AND THICKNESS DISTRIBUTIONS

```
#************************************************************************
#                    (c) CFD Research Corporation
#                          Huntsville, Alabama, USA.
#                               2000.
#
#************************************************************************
#       FILE: Blade.py
#
#  Author: Vadim Uchitel, (c) CFD Research Corp., Huntsville, Al.
#
#    Python script for impeller based on axial and radial design data.
#************************************************************************

import math
import GPoint
import GCurve
import GBlock
import GManip
import BladeData

n = number_of_grid_points = 10
fBladeCount  =  17

fHubData = BladeData.fHubData
fShroudData = BladeData.fShroudData

def GetPressure ( radius, theta, axis ):
    x = radius*math.cos(math.pi*theta/180.0)
    y = radius*math.sin(math.pi*theta/180.0)
    return [x, y, axis]

def GetSuction ( thickness, beta, coord, axis ):
    alpha = 2.0*math.pi/(fBladeCount-1)
    coord[0] = coord[0] + thickness*math.cos(math.pi*beta/180.0)
    coord[1] = coord[1] + thickness*math.sin(math.pi*beta/180.0)
    x = coord[0]*math.cos(alpha)-coord[1]*math.sin(alpha)
    y = coord[0]*math.sin(alpha)+coord[1]*math.cos(alpha)
    return GPoint.Create (x, y, axis)

def HubSuctionSideCoordinate(i):
    coord = GetPressure (fHubData[i][1], fHubData[i][4], fHubData[i][0])
    return GetSuction ( fHubData[i][6], fHubData[i][5], coord, fHubData[i][0])

def ShroudSuctionSideCoordinate(i):
    coord = GetPressure (fShroudData[i][1], fShroudData[i][4], fShroudData[i][0])
    return GetSuction ( fShroudData[i][6], fShroudData[i][5], coord, fShroudData[i][0])

def MeridonalCoordinate( type, i ):
    alpha = math.pi/(fBladeCount-1)
    if (type == 1):
        coord = GetPressure (fHubData[i][1], fHubData[i][4], fHubData[i][0])
    else:
        coord = GetPressure (fShroudData[i][1], fShroudData[i][4], fShroudData[i][0])
    x = coord[0]*math.cos(alpha)-coord[1]*math.sin(alpha)
    y = coord[0]*math.sin(alpha)+coord[1]*math.cos(alpha)
    return GPoint.Create (x,y,coord[2])

def GenerateImpeller():
    p = len(fHubData)
    hubPressurePoints = []
    hubSuctionPoints = []
    shroudPressurePoints = []
```

```
    shroudSuctionPoints = []
    for i in range(0,len(fHubData)):
      coord = GetPressure (fHubData[i][1], fHubData[i][4], fHubData[i][0])
      hubPressurePoints.append (GPoint.Create (coord[0], coord[1], coord[2]))
      hubSuctionPoints.append (HubSuctionSideCoordinate(i))
    for i in range(0,len(fShroudData)):
      coord = GetPressure (fShroudData[i][1], fShroudData[i][4], fShroudData[i][0])
      shroudPressurePoints.append(GPoint.Create (coord[0], coord[1], coord[2]))
      shroudSuctionPoints.append(ShroudSuctionSideCoordinate(i))
    #
    hubPressureCurve = GCurve.CreateThroughPoints(hubPressurePoints)
    hubSuctionCurve = GCurve.CreateThroughPoints(hubSuctionPoints)
    shroudPressureCurve = GCurve.CreateThroughPoints(shroudPressurePoints)
    shroudSuctionCurve = GCurve.CreateThroughPoints(shroudSuctionPoints)
    inletPressureCurve = GCurve.CreateThroughPoints(hubPressurePoints[0],shroudPressurePoints[0])
    inletSuctionCurve = GCurve.CreateThroughPoints(hubSuctionPoints[0],shroudSuctionPoints[0])
    inletHubCurve = GCurve.CreatePtsArc(hubPressurePoints[0],MeridonalCoordinate(1, 0),hubSuctionPoints[0])
    inletShroudCurve = GCurve.CreatePtsArc(shroudPressurePoints[0],MeridonalCoordinate(2, 0),
shroudSuctionPoints[0])
    outletPressureCurve = GCurve.CreateThroughPoints(hubPressurePoints[p-1],shroudPressurePoints[p-1])
    outletSuctionCurve = GCurve.CreateThroughPoints(hubSuctionPoints[p-1],shroudSuctionPoints[p-1])
    outletHubCurve = GCurve.CreatePtsArc(hubSuctionPoints[p-1],MeridonalCoordinate(1, -1),hubPressurePoints
[p-1])
    outletShroudCurve = GCurve.CreatePtsArc(shroudSuctionPoints[p-1],MeridonalCoordinate(2, -1),
shroudPressurePoints[p-1])
    #
    hubList = [hubSuctionCurve,  outletHubCurve, hubPressureCurve,  inletHubCurve]
    shroudList = [shroudSuctionCurve, outletShroudCurve, shroudPressureCurve,  inletShroudCurve]
    connectList = [inletSuctionCurve, outletSuctionCurve, outletPressureCurve, inletPressureCurve]
    #
    impellerBlock = GBlock.CreateFromCurves (hubList, shroudList, connectList, [n, n, n])
    GManip.DuplAndRotate ([0,0,0], [0,0,1], 30, 11, impellerBlock)
    ########## End of GenerateImpeller routine ############

impeller = GenerateImpeller()
```

APPENDIX F

IMPELLER SCRIPT BASED ON BLADE PROFILE CURVES (BLADEGEN DATA)

```
#*****************************************************************************
#                    (c) CFD Research Corporation
#                          Huntsville, Alabama, USA.
#                               2000.
#
#*****************************************************************************
#       FILE:BladeGen2Geom.py
#
#  Author: Vadim Uchitel, (c) CFD Research Corp., Huntsville, Al.
#
#  Python script for generating turbomachinery components from BladeGen data
#*****************************************************************************

import GInterface
import geom
import GManip
import GCurve
import GSurface
import GBlock

# Creating Hub & Shroud outlines
def  ImpellerTemplate (curve1, curve2, top_curve, bottom_curve):
    rot_list = GManip.DuplAndRotate ([0, 0, 0], [0, 0, 1], -51, 1, top_curve, bottom_curve, curve1)
    curve1.Delete ()
    new_top_curve = rot_list[0]
    new_bottom_curve = rot_list[1]
    new_curve1 = rot_list[2]
    point = GCurve.GetStartPoint (top_curve)
    etp = GCurve.GetEndPoint (top_curve)
    upstream_revolve = GCurve.CreatePtRevolvedCurve (point,  [0, 0, 0], [0, 0, 1], -51)
    s_rev_pt = GCurve.GetStartPoint (upstream_revolve)
    e_rev_pt = GCurve.GetEndPoint (upstream_revolve)
    s = spoint_downstream = GCurve.GetStartPoint (bottom_curve)
    e = epoint_downstream = GCurve.GetEndPoint (bottom_curve)
    snd = spoint_new_downstream = GCurve.GetStartPoint (new_bottom_curve)
    endp = GCurve.GetEndPoint (new_bottom_curve)
    sop = spoint_old_profile = GCurve.GetStartPoint (curve2)
    eop = GCurve.GetEndPoint (curve2)
    snp = GCurve.GetStartPoint (new_curve1)
    enp = epoint_new_profile = GCurve.GetEndPoint (new_curve1)
    points = [s, snd, e, endp, sop, snp, enp, etp, point, s_rev_pt, e_rev_pt]
    downstream_revolve = GCurve.CreatePtRevolvedCurve (epoint_downstream,  [0, 0, 0], [0, 0, 1], -51)
    line2 = GCurve.CreateThroughPoints (spoint_downstream, spoint_old_profile)
    line1 = GCurve.CreateThroughPoints (spoint_new_downstream, epoint_new_profile)
    blend_curve = GCurve.CreateBlendCurve (line1, epoint_new_profile, line2, spoint_old_profile)

    curves1 = [downstream_revolve, new_bottom_curve, [line1, blend_curve, line2], bottom_curve]
    curves2 = [[top_curve, curve2],  blend_curve, [new_curve1,new_top_curve], upstream_revolve]
    return [points, curves1, curves2]

# Read Curves Data
Blade = GInterface.TABGetData("c:/windows/desktop/scripts/IMP_profile_jbw2.tab",0,1,2,3,4,5,6,7,8,9)
IMP_hub_downstream = GInterface.TABGetData ("c:/windows/desktop/scripts/IMP_hub_downstream.tab", 10)
IMP_hub_upstream = GInterface.TABGetData ("c:/windows/desktop/scripts/IMP_hub_upstream.tab", 11)
IMP_shroud_downstream = GInterface.TABGetData ("c:/windows/desktop/scripts/IMP_shroud_downstream.tab", 12)
IMP_shroud_upstream = GInterface.TABGetData ("c:/windows/desktop/scripts/IMP_shroud_upstream.tab", 13)
IMP_hub = GInterface.TABGetData ("c:/windows/desktop/scripts/IMP_hub.tab", 14)

BladeSuction = GSurface.CreateRuled (Blade[0], Blade[2], Blade[4], Blade[6], Blade[8])
BladePressure = GSurface.CreateRuled (Blade[1], Blade[3], Blade[5], Blade[7], Blade[9])
Cap = GSurface.CreateRuled (Blade[9], Blade[8])
c_cur1 = geom.Entity_GetFromGeom (geom.CURVE_LIST, 16)
c_cur2 = geom.Entity_GetFromGeom (geom.CURVE_LIST, 17)
Cap2 = GSurface.CreateRuled (c_cur1, c_cur2)
```

```
hub_downstream = IMP_hub_downstream[0]
hub_upstream =IMP_hub_upstream[0]
shroud_downstream = IMP_shroud_downstream[0]
shroud_upstream = IMP_shroud_upstream[0]

hub_list = ImpellerTemplate  (Blade[0], Blade[1], hub_upstream, hub_downstream)
shroud_list =  ImpellerTemplate (Blade[8], Blade[9], shroud_upstream, shroud_downstream)

connections = []
for x in range (0, len (hub_list[0])):
    connections.append (GCurve.CreateThroughPoints (hub_list[0][x],  shroud_list[0][x]))

connect_list1 = [ connections[2], connections[3], connections[1], connections[0]]
block1 = GBlock.CreateFromCurves (hub_list[1], shroud_list[1], connect_list1, [31, 31, 31])

connect_list2 = [connections[9], connections[4], connections[6], connections[10]]
block2 = GBlock.CreateFromCurves (hub_list[2], shroud_list[2], connect_list2, [31, 31, 31])

GManip.DuplAndRotate ([0,0,0],[0,0,1], -51.4285, 11, [BladeSuction, BladePressure, Cap, Cap2])
GSurface.CreateRevolved ([0,0,0],[0,0,1], 360, IMP_hub[0])
```

APPENDIX G

STATOR SCRIPT BASED ON IGES CURVES AND SURFACE DATA

```
#*****************************************************************************
#                    (c) CFD Research Corporation
#                          Huntsville, Alabama, USA.
#                               2000.
#
#*****************************************************************************
#  FILE:stator_script.py
#
#  Author: Vadim Uchitel, (c) CFD Research Corp., Huntsville, Al.
#
#  Python script for generating stator from IGES data
#*****************************************************************************

import GInterface
import GManip
import GCurve
import GSurface
import GBlock
import GEntity

dist = 0.017
rot_angle = 10

GInterface.IGESRead ("d:/users/vadim/geom6/Python/stator.iges")
GInterface.IGESRead ("d:/users/vadim/geom6/Python/hub_shroud.iges")
surf1 = GEntity.GetEntity (GEntity.SURFACE_LIST, 0)
surf2 = GEntity.GetEntity (GEntity.SURFACE_LIST, 1)

curve1 = GEntity.GetEntity (GEntity.CURVE_LIST, 1)
curve2 = GEntity.GetEntity (GEntity.CURVE_LIST, 3)
curve3 = GEntity.GetEntity (GEntity.CURVE_LIST, 4)
curve4 = GEntity.GetEntity (GEntity.CURVE_LIST, 5)
curve5 = GEntity.GetEntity (GEntity.CURVE_LIST, 0)
curve6 = GEntity.GetEntity (GEntity.CURVE_LIST, 2)

Cap1 = GSurface.CreateRuled (curve3, curve6)
Cap2 = GSurface.CreateRuled (curve4, curve5)

shroud =  GEntity.GetEntity (GEntity.CURVE_LIST, 6)
hub = GEntity.GetEntity (GEntity.CURVE_LIST, 7)

GSurface.CreateRevolved ( [0, 0, 0], [1, 0, 0], 360, shroud)
GSurface.CreateRevolved ( [0, 0, 0], [1, 0, 0], 360, hub)

pt1 = GCurve.GetStartPoint (curve1)
pt2 = GCurve.GetEndPoint (curve1)
pt3 = GCurve.GetStartPoint (curve2)
pt4 = GCurve.GetEndPoint (curve2)

ln1 = GCurve.CreateExtrusionLine (pt1, [1, 0, 0], dist)
ln2 = GCurve.CreateExtrusionLine (pt2, [1, 0, 0], dist)
ln3 = GCurve.CreateExtrusionLine (pt3, [-1, 0, 0], dist)
ln4 = GCurve.CreateExtrusionLine (pt4, [-1, 0, 0], dist)

pt5 = GCurve.GetEndPoint (ln1)
pt6 = GCurve.GetEndPoint (ln2)
pt7 = GCurve.GetEndPoint (ln3)
pt8 = GCurve.GetEndPoint (ln4)

ln5 = GCurve.CreateThroughPoints (pt5, pt6)
ln6 = GCurve.CreateThroughPoints (pt7, pt8)
```

```
rev_crv1 = GCurve.CreatePtRevolvedCurve (pt5, [0, 0, 0], [1, 0, 0], rot_angle)
rev_crv2 = GCurve.CreatePtRevolvedCurve (pt6, [0, 0, 0], [1, 0, 0], rot_angle)
rev_crv3 = GCurve.CreatePtRevolvedCurve (pt7, [0, 0, 0], [1, 0, 0], rot_angle)
rev_crv4 = GCurve.CreatePtRevolvedCurve (pt8, [0, 0, 0], [1, 0, 0], rot_angle)

l1 = GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 1, ln1, ln2, ln3, ln4, ln5, ln6)
l2 = GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 1, curve1, curve2, curve3, curve4)

list1 = [ln1, ln2, ln3, ln4, ln5, ln6, curve1, curve2, curve5, curve6]
list2 = [l1[0], l1[1], l1[2], l1[3], l1[4], l1[5], l2[0], l2[1], l2[2], l2[3]]
list3 = [rev_crv1, rev_crv2, rev_crv3, rev_crv4]

block = GBlock.CreateFromCurves1 (list1, list2, list3, [41, 41, 41])
GEntity.SetLocalNames (globals ())
GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 35, surf1, surf2, Cap1, Cap2)
```

APPENDIX H

ROTOR SCRIPT BASED ON IGES CURVES AND SURFACE DATA

```
#*****************************************************************************
#                     (c) CFD Research Corporation
#                          Huntsville, Alabama, USA.
#                               2000.
#
#*****************************************************************************
#  FILE:rotor_script.py
#
#  Author: Vadim Uchitel, (c) CFD Research Corp., Huntsville, Al.
#
#  Python script for generating rotor from IGES data
#*****************************************************************************

import GInterface
import GManip
import GCurve
import GSurface
import GBlock
import GEntity

dist = 0.01643
rot_angle = -10

GInterface.IGESRead ("d:/users/vadim/geom6/Python/rotor.iges")
GInterface.IGESRead ("d:/users/vadim/geom6/Python/hub_shroud2.iges")
surf1 = GEntity.GetEntity (GEntity.SURFACE_LIST, 0)
surf2 = GEntity.GetEntity (GEntity.SURFACE_LIST, 1)

curve1 = GEntity.GetEntity (GEntity.CURVE_LIST, 0)
curve2 = GEntity.GetEntity (GEntity.CURVE_LIST, 2)
curve3 = GEntity.GetEntity (GEntity.CURVE_LIST, 4)
curve4 = GEntity.GetEntity (GEntity.CURVE_LIST, 5)
curve5 = GEntity.GetEntity (GEntity.CURVE_LIST, 1)
curve6 = GEntity.GetEntity (GEntity.CURVE_LIST, 3)

Cap1 = GSurface.CreateRuled (curve3, curve5)
Cap2 = GSurface.CreateRuled (curve4, curve6)

hub1 =  GEntity.GetEntity (GEntity.CURVE_LIST, 6)
hub2 = GEntity.GetEntity (GEntity.CURVE_LIST, 7)
shroud1 = GEntity.GetEntity (GEntity.CURVE_LIST, 8)
shroud2 = GEntity.GetEntity (GEntity.CURVE_LIST, 9)

GSurface.CreateRevolved ( [0, 0, 0], [1, 0, 0], 360, shroud1)
GSurface.CreateRevolved ( [0, 0, 0], [1, 0, 0], 360, hub1)
GSurface.CreateRevolved ( [0, 0, 0], [1, 0, 0], 360, shroud2)
GSurface.CreateRevolved ( [0, 0, 0], [1, 0, 0], 360, hub2)

pt1 = GCurve.GetStartPoint (curve1)
pt2 = GCurve.GetEndPoint (curve1)
pt3 = GCurve.GetStartPoint (curve2)
pt4 = GCurve.GetEndPoint (curve2)

ln1 = GCurve.CreateExtrusionLine (pt1, [1, 0, 0], dist)
ln2 = GCurve.CreateExtrusionLine (pt2, [1, 0, 0], dist)
ln3 = GCurve.CreateExtrusionLine (pt3, [-1, 0, 0], dist)
ln4 = GCurve.CreateExtrusionLine (pt4, [-1, 0, 0], dist)

pt5 = GCurve.GetEndPoint (ln1)
pt6 = GCurve.GetEndPoint (ln2)
pt7 = GCurve.GetEndPoint (ln3)
pt8 = GCurve.GetEndPoint (ln4)
```

```
ln5 = GCurve.CreateThroughPoints (pt5, pt6)
ln6 = GCurve.CreateThroughPoints (pt7, pt8)

rev_crv1 = GCurve.CreatePtRevolvedCurve (pt5, [0, 0, 0], [1, 0, 0], rot_angle)
rev_crv2 = GCurve.CreatePtRevolvedCurve (pt6, [0, 0, 0], [1, 0, 0], rot_angle)
rev_crv3 = GCurve.CreatePtRevolvedCurve (pt7, [0, 0, 0], [1, 0, 0], rot_angle)
rev_crv4 = GCurve.CreatePtRevolvedCurve (pt8, [0, 0, 0], [1, 0, 0], rot_angle)

l1 = GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 1, ln1, ln2, ln3, ln4, ln5, ln6)
l2 = GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 1, curve1, curve2, curve3, curve4)

list1 = [ln1, ln2, ln3, ln4, ln5, ln6, curve1, curve2, curve5, curve6]
list2 = [l1[0], l1[1], l1[2], l1[3], l1[4], l1[5], l2[0], l2[1], l2[2], l2[3]]
list3 = [rev_crv1, rev_crv2, rev_crv3, rev_crv4]

GBlock.CreateFromCurves1 (list1, list2, list3,[ 41,41, 41])
GManip.DuplAndRotate ( [0, 0, 0], [1, 0, 0], rot_angle, 35, surf1, surf2, Cap1, Cap2)
GEntity.SetLocalNames (locals())
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	19 Apr 2000	22 Oct 99 - 20 Apr 2000

4. TITLE AND SUBTITLE
Automated, Parametric Geometry Modeling and Grid Generation for Turbomachinery Applications

5. FUNDING NUMBERS
C NAS3-00001

6. AUTHOR(S)
Vincent J. Harrand, Vadim G. Uchitel, John B. Whitmire

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES(ES)
CFD Research Corporation
215 Wynn Drive
Huntsville, AL 35805

8. PERFORMING ORGANIZATION REPORT NUMBER
8229/3

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESSES(ES)
NASA Glenn Research Center
Technology Support Branch MS 500-305
21000 Brookpark Road
Cleveland, OH 44135

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
See Handbook NHB 2200.2.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The objective of this Phase I project is to develop a highly automated software system for rapid geometry modeling and grid generation for turbomachinery applications. The proposed system features a graphical user interface for interactive control, a direct interface to commercial CAD/PDM systems, support for IGES geometry output, and a scripting capability for obtaining a high level of automation and end-user customization of the tool.

The developed system is fully parametric and highly automated, and, therefore, significantly reduces the turnaround time for 3D geometry modeling, grid generation and model setup. This facilitates design environments in which a large number of cases need to be generated, such as for parametric analysis and design optimization of turbomachinery equipment.

In Phase I we have successfully demonstrated the feasibility of the approach. The system has been tested on a wide variety of turbomachinery geometries, including several impellers and a multi stage rotor-stator combination. In Phase II, we plan to integrate the developed system with turbomachinery design software and with commercial CAD/PDM software.

14. SUBJECT TERMS
Turbomachinery design, geometry modeling, grid generation, parametric analysis, design optimization

15. NUMBER OF PAGES
46

16. PRICE CODE

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