A Geometry Based Infra-structure for Computational Analysis and Design

Robert Haimes
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
haimes@orville.mit.edu

Introduction

The computational steps traditionally taken for most engineering analysis suites (computational fluid dynamics -- CFD, structural analysis, heat transfer and etc.) are:

- Surface Generation -- usually by employing a CAD system
- Grid Generation -- preparing the volume for the simulation
- Flow Solver -- producing the results at the specified operational point
- Post-processing Visualization -- interactively attempting to understand the results

For structural analysis, integrated systems can be obtained from a number of commercial vendors. These vendors couple directly to a number of CAD systems and are executed from within the CAD GUI. It should be noted that the structural analysis problem is more tractable than CFD; there are fewer mesh topologies used and the grids are not as fine (this problem space does not have the length scaling issues of fluids).

For CFD, these steps have worked well in the past for simple steady-state simulations at the expense of much user interaction. The data was transmitted between phases via files. In most cases, the output from a CAD system could go to IGES or STEP files. The output from Grid Generators and Solvers do not really have standards though there are a couple of file formats that can be used for a subset of the gridding (i.e. PLOT3D data formats). The user would have to patch up the data or translate from one format to another to move to the next step. Sometimes this could take days. Specifically the problems with this procedure are:

- File based. Information flows from one step to the next via data files with formats specified for that procedure. File standards, when they exist, are wholly inadequate. For example, geometry from CAD systems (transmitted via IGES files) is defined as disjoint surfaces and curves (as well as masses of other information of no interest for the Grid Generator). This is particularly onerous for modern CAD systems based on solid modeling. The part was a proper solid and in the translation to IGES has lost this important characteristic. STEP is another standard for CAD data that exists and supports the concept of a solid. The problem with STEP is that a solid modeling geometry kernel is required to query and manipulate the data within this type of file.

- 'Good' Geometry. A bottleneck in getting results from a solver is the construction of proper geometry to be fed to the grid generator. With 'good' geometry a grid can be constructed in tens of minutes (even with a complex configuration) using unstructured techniques. Adroit multi-block methods are not far behind. This means that a million node steady-state solution can be computed on the order of hours (using current high performance computers) starting from this 'good' geometry. Unfortunately, the geometry usually transmitted from the CAD system is not 'good' in the grid generator sense. The grid generator needs smooth closed solid

geometry. It can take a week (or more) of interaction with the CAD output (sometimes by hand) before the process can begin.

- One-way Comminucation. All information travels on from one phase to the next. This makes procedures like node adaptation difficult when attempting to add or move nodes that sit on bounding surfaces (when the actual surface data has been lost after the grid generation phase).

Until this process can be automated, more complex problems such as multi-disciplinary analysis or using the above procedure for design becomes prohibitive. There is also no way to easily deal with this system in a modular manner. One can only replace the grid generator, for example, if the software reads and writes the same files.

Instead of the serial approach to analysis as described above, **CAPRI** takes a geometry centric approach. This makes the actual geometry (not a discretized version) accessible to all phases of the analysis. The connection to the geometry is made through an Application Programming Interface (API) and NOT a file system. This API isolates the top-level applications (grid generators, solvers and visualization components) from the geometry engine. Also this allows the replacement of one geometry kernel with another, without effecting these top-level applications. For example, if UniGraphics is used as the CAD package then Parasolid (UG's own geometry engine) can be used for all geometric queries so that no solid geometry information is lost in a translation. This is much better than STEP because when the data is queried, the same software is executed as used in the CAD system. Therefore, one analyzes the exact part that is in the CAD system.

CAPRI uses the same idea as the commercial structural analysis codes but does not specify control. Software components of the CAD system are used, but the analysis suite, not the CAD operator, specifies the control of the software session. This also means that the license issues (may be) minimized and individuals need not have to know how to operate a CAD system in order to run the suite.

The **CAPRI** API

CAPRI is a software building tool-kit that refers to two ideas; (1) A simplified hierarchical view of a solid part integrating both geometry and topology definitions, and (2) Programming access to this part and any attached data.

A complete definition of the geometry and application-programming interface can be found in the attached document "**CAPRI**: Computational Analysis PRogramming Interface". In summary the interface is sub-divided into the following functional components:

1. Utility routines -- These routines include the initialization of **CAPRI**, loading CAD parts and querying the operational status as well as closing the system down.

2. Geometry database queries -- This group of functions allow all top-level applications to figure out and get detailed information on any geometric component in the Volume definition.

3. Point queries -- These calls allow grid generators, or solvers doing node adaptation, to snap points directly onto geometric entities.

4. Calculated or geometrically derived queries -- These calls calculate data from the geometry to aid in grid generation.

5. Boundary data routines -- This part of **CAPRI** allows general data to be attached to Boundaries so that the boundary conditions can be specified and stored within **CAPRI**'s database.

6. Tag based routines -- This part of the API allows the specification of properties associated with either the Volume (material properties) or Boundary (surface properties) entities.

7. Geometry based interpolation routines -- This part of the API facilitates Multi-disciplinary coupling and allows zooming through Boundary Attachments.

8. Geometric modification -- This will be used for an automated design system where the goal of the application is to change the geometry. Routines that allow this have the advantage that if the data is kept consistent with the CAD package, then the design can be incorporated directly and therefore is manufacturable.

Status of This Years Work

1. Component 7 above, zooming and multi-disciplinary coupling has been and tested for geometric interpolation. Though simple in concept (mapping data to and extracting interpolated data from the actual geometry), the generalization is complex. To properly support turbomachinery applications it was necessary to include the idea of "replication" and movement of Volumes. This allows matching of the stator and rotor passages when the entire wheel is not simulated. Common visualization techniques like "mirroring" become a simple form of replication.

2. The use of **CAPRI**
   Clearly for this work to be successful, it must be used. The easiest way to convince industry of its value is to work on a topical problem of interest that current methods do not provide timely results. Originally it was thought that a problem of interest to GE Aircraft Engines would be tackled, but GE could not decide on a tasks of mutual interest. Instead the following projects were completed with NASA Lewis personnel:

   APNASA – John Adamchyk's Average Passage turbomachinery simulation code.
   The grid generator (APG) of APNASA was converted to use **CAPRI** as the input for geometry. This meshing system was a good test in that it has some very stringent requirements (all resultant grid planes in the circumferencial direction must be on surfaces of revolution). This exercise brought out the importance of **CAPRI**'s provided tessellation and that is must be defined in a more rigorous manner. This prompted a redefinition of the API associated with the tessellation so that the surface triangles match at edges and all of the connectivity is supplied. UniGraphics' parts were used in this exercise. But this change in the API required a major effort for Pro/ENGINEER because the triangulation given from the CAD system was open. A surface tessellation was therefore written that holds "water".

   LAPIN – This 1-D nozzle code was used in a Pro/ENGINEER exercise. ProE was used to sketch the nozzle's geometry. This was revolved to create a 3D solid part. CAPRI was used to query the geometry and generate the input to LAPIN.

3. Native Geometry Kernel
   In an attempt to provide a working version of **CAPRI** that does not require any licensed software a simple kernel was ported. This geometry system is based on the FELISA suite by Prof. Jaime Peraire at MIT. Having this geometry system also allows the investigation into support of non-manifold objects and mixing geometry kernels.

4. SDRC's I-DEAS
   By the end of this contract period a **CAPRI** port to the CAD system I-DEAS will be complete. This requires licenses to Open I-DEAS and Orbix for the CORBA based communication.

Presentations

Seminars were given through out the year at Pratt & Whitney, GE Aircraft Engines and Stanford University. The paper "**C**omputational **A**nalysis **PR**ogramming **I**nterface" was given at the 6th International Conference on Numerical Grid Generation in Computational Field Simulations held at the University of Greenwich. A copy has been attached.

Statement of Work

The effort for moving **CAPRI** forward within the next year follows many fronts. The following list of tasks will be addressed:

1. CATIA
   The CAD package CATIA will be the next to be integrated into the CAPRI framework. The proper licenses have been obtained to allow access to CATIA parts and the geometry kernel. CATIA is the only other major CAD vendor used by the aerospace industry currently not covered. This will allow Boeing access to **CAPRI** and therefore use and contribute to this work.

2. The NPARC Consortium's WIND code & CGNS
   Efforts will be made this year to integrate **CAPRI** into the WIND code. WIND uses the data file format CGNS for the storage of the CFD grids and results. It has been commercialized by ICEM-CFD. This hierarchical format contains a placeholder for geometry but no mechanism to operate on this information. The combination of this data file format and the geometry API perfectly complement each other. It is the goal of this task to display this functionality so that both **CAPRI** and CGNS can attain greater acceptance.

3. Geometry Creation and Modification
   Once CATIA has been ported, the entire issue of modification and creation of solid-based geometry can be addressed. At a minimum, functions like scribing and splitting existing surface are required for grid generation of structured blocks as well as being able to bound and invert existing solids to create the fluid volume.

   With the knowledge of the internals of four major CAD systems (and a native geometry kernel), a group of functions can be specified so that these operations are feasible across these CAD packages. The goal is to produce an API that is both conceptually simple and very powerful. Boolean operations on solids may be the foundation for this part of **CAPRI**.

4. Commercial Grid Generators
   The turbomachinery industry is beginning to use commercial grid generators. This causes a problem for **CAPRI** specifically and automated analysis and design systems in general. The work here requires that the grid generator use the solid model during the meshing and update the information in **CAPRI** with the surface discretization when complete. Wrappers can be written to merge the operations but a more complete integration is desirable. Attempts will be made to convince the vendors of these CFD grid generators to hook up to **CAPRI** (the packages include ICEM-Hexa, ICEM-Tetra, GridGen and GridPro). The connection between CGNS and **CAPRI** should help in the case of ICEM.

5. Geometry Interface Standards and CORBA's OMG
   A proposal has been initiated by NASA Lewis personnel to the PPE group within OMG to use **CAPRI** as the basis for an analysis IDL. This will be supported in this task. A trip is planned to a future meeting of the PPE group.

6. Concurrent Support of Multiple Geometry Kernels
   There have been some requests that a mixing of parts from various CAD systems (and the native modeler) may be advantageous. This is not fundamentally difficult, but does require some recasting of the code so that when all of the systems supported are not present, applications can still be built. This may also allow the use of non-manifold objects (from within the native system).

CAPRI: Computational Analysis PRogramming Interface

Revision 0.85

Robert Haimes
Massachusetts Institute of Technology
haimes@orville.mit.edu

September 3, 1998

A Solid Modeling Based Infra-structure for Engineering Analysis and Design

Abstract

CAPRI is a CAD-vendor neutral application programming interface designed for the construction of analysis suites and design systems. By allowing access to the geometry from within all modules (grid generators, solvers and post-processors) such tasks as meshing on the actual surfaces, node enrichment by solvers and defining which mesh faces are boundaries (for the solver and visualization system) become simpler. The overall reliance on file 'standards' is minimized.

This 'Geometry Centric' approach makes multi-physics (multi-disciplinary) analysis codes much easier to build. By using the shared (coupled) surface as the foundation, CAPRI provides a single call to interpolate grid-node based data from the surface discretization in one volume to another. Finally, design systems are possible where the results can be brought back into the CAD system (and therefore manufactured) because all geometry construction and modification are performed using the CAD system's geometry kernel.

License

Contents

1 Introduction

The computational steps traditionally taken for Computational Fluid Dynamics (CFD), Structural Analysis, and other simulation disciplines (or when these are used in design) are:

- Surface Generation
  The surfaces of the object(s) are generated usually from a CAD system. This creates the starting point for the analysis and is what is used for manufacturing.

- Grid Generation
  These surfaces are used (with possibly a bounded outer domain) to create the volume of interest. Usually for the analysis of external aerodynamics, the aircraft is surrounded by a domain that extends many body lengths away from the surfaces. This enclosed volume is then discretized (subdivided) in one of many different ways. Unstructured meshes are built by having the subdivisions usually comprised of tetrahedral elements. Another technique breaks up the domain into sub-domains that are hexahedral. These sub-domains are further divided in a regular manner so that individual cells in the block can be indexed via an integer triad.

- Flow Solver or Simulation
  The solver takes as input the grid generated by the second step (and information about how to apply conditions at the bounds of the domain). Because of the different styles of gridding, the solver is usually written with ability to use only one of the volume discretization methods. In fact there are no standard file types, so most solvers are written in close cooperation with the grid generator. For fluids, the solver usually simulates either the Euler or Navier-Stokes equations in an iterative manner, storing the results either at the nodes in the mesh or in the element centers. The output of the solver is a file that contains the solution.

- Post-processing Visualization
  After the solution procedure has successfully completed, the output from the grid generator and the simulation are displayed and examined in a graphical manner by the fourth step in this process. Usually a workstation with a 3D graphics adapter is used to quickly render the output from data extraction techniques. The tools (such as iso-surfacing, geometric cuts and streamlines) allow the examination of the volumetric data produced by the solver. Even for steady-state solutions, much time is usually required to scan, poke and probe the data in order to understand the physics in the flow field.

These steps have worked well in the past for simple steady-state simulations at the expense of much user interaction. The data was transmitted between phases via files (the

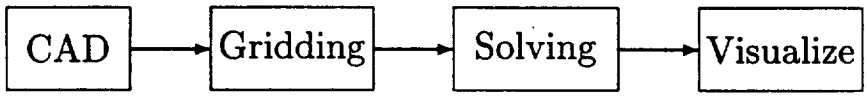

Figure 1: The Traditional Computational Analysis Suite

arrows in Figure 1). In most cases, the output from a CAD system could go to IGES files. The output from Grid Generators and solvers do not really have standards though there are a couple of file formats that can be used for a subset of the problem space (i.e. PLOT3D data formats for CFD). The user would have to patch up the data or translate from one format to another to move to the next step. Sometimes this could take days. Specifically, the problems with this procedure are:

- File based
  Information flows from one step to the next via data files with formats specified for that procedure. Historically, this allows individuals or groups to work in isolation on the construction of one of these components; unfortunately the user (or team) suffers greatly because of the lack of integration. In many cases the files that get used do not contain all the information required to couple all components so that the user can be removed from the mechanics of running the suite.

- 'Good' Geometry
  A bottleneck in getting results from a solver is the construction of proper geometry to be fed to the grid generator. With 'good' geometry a grid can be constructed in tens of minutes (even with a complex configuration) using unstructured techniques. Adroit multi-block methods are not far behind. This means that a million node CFD steady-state solution can be computed on the order of hours (using current high performance computers) starting from this 'good' geometry. Unfortunately, geometry from CAD systems (especially when transmitted via IGES files) is not 'good' in the grid generator sense. The data is usually defined as disjoint and unconnected surfaces and curves (as well as masses of other information of no interest for the mesh). The grid generator needs smooth closed solid geometry. It can take a week (or more) of interaction with the CAD output (sometimes by hand) before the process can begin. This is particularly onerous if the CAD system is based on solid modeling. The part was a proper solid with topology and in the translation to IGES has lost these important characteristics.

- One-Way Comminucation
  All information travels on from one phase to the next. This makes procedures like

node adaptation difficult when attempting to add or move nodes that sit on bounding surfaces (when the actual surface data has been lost after the grid generation phase). In fact, the information passed from phase to phase is not enriched but is filtered.

Until this process can be automated, more complex problems such as Multi-disciplinary analysis or using the above procedure for design becomes prohibitive. There is also no way to easily deal with this system in a modular manner. One can only replace the grid generator, for example, if the software reads and writes the same files.

Procedures like zooming, defined within the Numerical Propulsion System Simulation (NPSS), are difficult to acheive when the surface definition for the coupling between the simulations is lost.

2 CAPRI

Instead of the serial approach to analysis as described above, CAPRI uses a geometry centric approach. This makes the actual geometry (not a discretized version) accessible to all phases of the analysis. The connection to the geometry is made through an Application Programming Interface (API) and NOT a file system. This API isolates the top level applications (grid generators, solvers and visualization components) from the geometry engine. Also this allows the replacement of one geometry kernel with another, without effecting the top level applications. For example, if UniGraphics is used as the CAD package then **Parasolid** (UG's geometry engine) can be used for all geometric queries so that no solid geometry information is lost in a translation. If **ProE** is used then Pro/Toolkit is accessed when geometric information is required. See Figure 2.

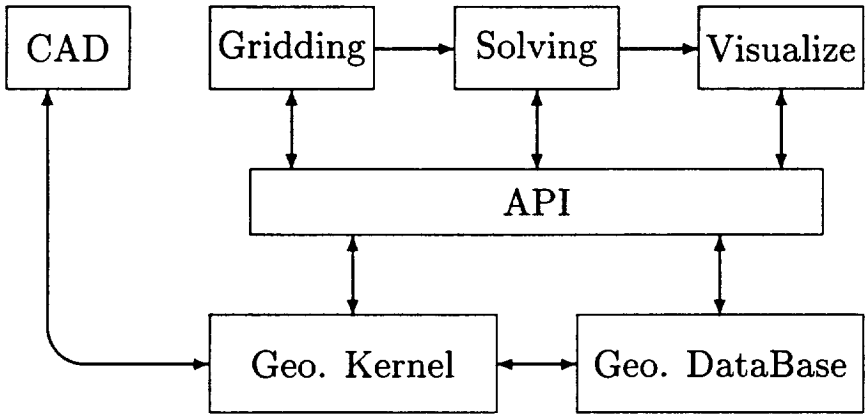

Figure 2: The CAPRI based Computational Analysis Suite

It is very important to consider the design goals when building a new software architecture. Without properly setting a broad foundation, the system may not be able to function as desired. The goals for CAPRI are:

- Modular
  The system must support a modular or building-block method for construction. This facilitates a plug and play approach at the top level as well as the underlying geometry kernel.

- Multiple languages
  It is important to support FORTRAN, C and C++. Many CFD codes are currently written in FORTRAN. On many machines, the FORTRAN compiler produces more highly optimized code, giving much better performance. Forcing the core of these

algorithms to another language, just because the rest of the system is in that language, is not be part of the philosophy found in CAPRI.

- Transient solutions
  This system must support unsteady simulations as well as steady-state, which include the underlying geometry changing with time.

- Allow Multi-discipline coupling and zooming
  This system must be general enough to allow coupling from codes of other disciplines (including but not limited to – structural analysis, heat transfer, acoustic codes). In fact the coupling could be close, in that the analysis code could be made a part of the overall design system.

2.1 Geometry and Topology

To insure that the design goals can be met and the resulting interface is not overly complex, it is crucial that the geometry description be uncomplicated (but not too simple as to impair functionality). Most systems that deal with CAD data make the distinction between geometry (points, curves and surfaces) and topology (the hierarchical connections between geometric entities). CAPRI mixes these in a simple geometry data definition.

The geometry and topology are defined in CAPRI in the following manner:

- Nodes
  These are the simplest entities and are just points in 3 space.

- Edges
  Edges are curves. Each Edge is bounded by two unique Nodes. The Edge is parameterized with t, where the first Node has a value at t_{min} and the second bounding Node has a value of t_{max}. The value of t_{min} is always less than t_{max}.

  To aid in plotting, there is an attached tesselation of the curve. This is defined as a poly-line with a specified length. The line is defined starting at the first Node and terminates with the second.

  Note: Circles, ellipses and other closed loops found in the original CAD definition are broken up by CAPRI so that there is no parameterization that is periodic. Any closed loop will be broken in two and therefore may have two Nodes added so that the Edge can be properly bounded.

- Faces
  Faces are parameterized (u, v) surfaces. The parameter range for u is u_{min} to u_{max} and v ranges from v_{min} to v_{max}, but the relationship between (u, v) and the bounding

Edges is not as simple as the Edge-Node definitions. This is because Faces may be bounded by more than 4 Edges. In fact, a Face can be a very complex surface where the ranges of the parameterization are only limits and should not be used throughout its entirety (i.e. there may be a hole or the result of some trimming).

The bounds of the Face are defined by closed set(s) of Edges. There may be one or more of these loops for each Face. Stored with each defining Edge is an orientation so that it is known whether to look at the Edge as specified or in the opposite sense. The loop is an ordered suite that defines the orientation of the Face. The outer loop(s), specify the boundary of the surface, and traverse the Face in a right-handed manner – defining the outward pointing normal (out of the volume). Any holes are specified by a left-handed traversal of Edges. See Figure 3.

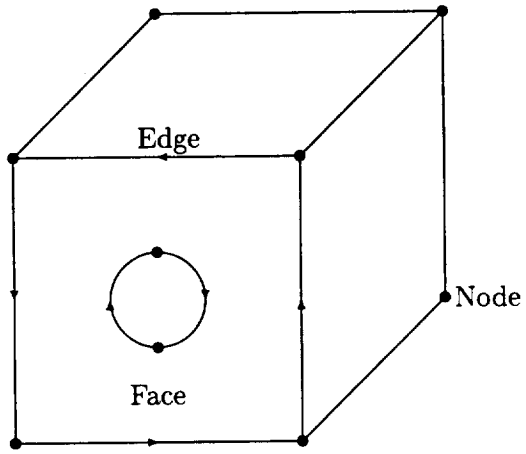

Figure 3: A simple Volume with a cylinder cutout – Edges marked for front Face

Each Edge can be found bounding two Faces, one in the forward and one in the opposite sense.

Again, to aid in plotting and to have a complete representation of this (possibly complex) Face, there is an attached tesselation. This is defined as suite of disjoint triangles of a specified length. Each triangle is right-handed with the normal pointing out of the volume.

Note: Cylinders, and other periodic surfaces found in the original CAD definition are broken up by CAPRI so that the parameterization is not periodic. Any periodic surface will be broken in two and therefore may have two Edges added so that the Face's parameter space is simple.

- Boundaries

  Boundaries are simply collections of one or more Faces. These entities are the connection between the geometry and the rest of the analysis suite, as described above. The Faces need not couple together (i.e., a periodic boundary upstream and downstream from a turbine or compressor blade) but are used to insure that the grid generation knows that these surfaces could be treated in special ways. And, the solver knows which boundary condition to apply to what section of the resultant mesh.

  Boundaries have an associated name (i.e., far-field, body, wing and etc).

- Volumes

  Volumes are completely closed regions of 3 space. Volumes are bounded by the sum of all of the Faces found in the Boundaries. These Faces match up at the shared Edges, that terminate at the Nodes. CAPRI can handle one or more Volumes at a time.

  Each Volume can be named with strings like; 'Fluids passage', 'Blade', and etc.

  Volumes may have a number of associated Tags to indicate global conditions for the discipline. Each Tag has an assigned value string. For example; the Volume 'Fluids passage' may have the Tags 'gamma' (with the value string of '1.4') and 'smoothing' (with the associated string '0.2 0.02').

The geometric entities described above are referenced within CAPRI with integer handles or indices. Each Volume is assigned a handle when loaded. All entities contained within that volume (Nodes, Edges, Faces and Boundaries) are given indices ranging from 1 to the total number of entities in that class. Therefore, it usually requires 2 handles to describe an entity, the volume and the entity indices.

There is a special Boundary index (zero) which refers to all currently unassigned Faces. When a Volume (or number of Volumes) is first read from the CAD system, this Boundary is fully populated with all Faces. As Faces become assigned, they are pulled from this Boundary and put in the appropriate place.

2.1.1 Solid Model Accuracy

Face tesselations match at the Edge tesselations. This produces a complete triangulation of the Volume that is manifold regardless of the underlying accuracy, surface matching and closure. If the accuracy is lax, it is CAPRIs responsibility to provide a closed model.

This can have some implications when using either the Edge or Face tesselations. A sense of the models accuracy can be gotten by taking the parameter value supplied for the bounds of the entity and evaluating (calls to gi_qPointOnEdge or gi_qPointOnFace) to get the coordinates. This needs to be compared to the values returned for the tesselation to

get the deviation (lack of accuracy) for the entity. Therefore, when using this data near or at the bounds (for enriching) care must be taken so that the Edge or Face data ends up appearing smooth.

2.2 Volumes

All coordinates for a Volume reported back through CAPRI are in the CADs native coordinate system, scale and units for that individual component part.

2.2.1 Replication

There is a Replication matrix and count associated with each Volume. The count refers to the number of times to apply the matrix to produce the complete representation of the object. The matrix is a column-major matrix that is 4 columns by 3 rows. This matrix is used to multiply all Volume coordinates in order to produce additional instances. This is only referenced within CAPRI for the interpolation functions.

Mirroring can be considered a simple form of this type of replication, where the count is one and the matrix is all zero except for the diagonal. The diagonal will contain 1.0 for 2 of the 3 components and -1.0 for the other.

2.2.2 Displacement

There is a Displacement matrix associated with each Volume. This matrix is also column-major matrix that is 4 columns by 3 rows. The matrix is used to multiply all Volume coordinates in order to specify movement of the Volume. This allows the support of any combination of translation, scaling and rotation. This may be used for transient problems where Volumes move by other Volumes. For the interpolation routines, Replication is applied before Displacement.

During the loading of Volumes (from assemblies – collections of parts) the Displacement matrix will properly be filled with the appropriate values so that the Volumes fit together as seen within the CAD system.

2.2.3 Volume Tags

Tags are character strings associated with the Volume. Each Tag has an attached value string. These Tag entities are useful for specifying conditions or material information for the entire Volume. For example; the Volume may have a Tag 'gamma' with the associated value '1.4'.

2.3 Boundaries

Boundaries are the pivotal data objects used within CAPRI. Boundaries are the entities that the grid generators should build the exposed parts of the mesh about. Different solver functions (boundary conditions) are then applied across these facets of the volume. When Multi-disciplinary analysis are run, boundaries are where these different physical models share information to drive the coupled solution.

The data that comes from CAD systems does not always provide a proper separation of surfaces (Edges, as specified above) that coincide with what is required by the analysis suite. This is for two reasons; (1) the CAD operator, by the order of construction, may produce artifacts (such as sliver surfaces) or detail at a level more complex than the analysis suite requires. (2) Curved surfaces such as fillets have breaks, on where these surfaces mate with other surfaces, usually not at the center of curvature where the analysis suite would require the edge of the boundary.

The first of these problems is resolved in CAPRI by allowing the collection of CAD surfaces. The analysis suite can query this collection and get to the detailed CAD surfaces if required. This has the advantage over what is done in automated techniques used for grid generation in that the CAD artifacts can be meshed through as opposed to becoming features in the grid. For example, a sliver surface would end up completely resolved, in an automated surface gridding procedure, requiring potentially large numbers of small cells in those regions.

Scribing and splitting CAD surfaces so that the analysis boundaries can be defined is a function of CAPRI. Initially this is done interactively or through program control (if the analysis suite can determine where to break the surfaces). In the future, work will be done to attempt to automate this procedure.

Interactive functions are also provided within the CAPRI framework to collect these CAD surfaces and produce the boundaries as well as setting up the information to run the entire suite.

2.3.1 Boundary Discretization

Each Boundary can have an attached discretization. This discretization can be from different mesh topologies that touch the Boundary. There are 3 types of cell faces that build this structure:

- Disjoint Triangles – 3 bnodes per entity
- Disjoint Quadrangles – 4 bnodes per entity
- Quad-Meshes – these are produced from grid 'planes' of structured blocks

These entities are supported via Boundary nodes (bnodes). The bnode numbering used is local within the Boundary. For Disjoint Quads, the bnodes must be defined as it goes around the quad in a right-handed manner pointing out of the volume. The node numbering scheme used differentiates between the nodes in the non-block regions (formed by the disjoint faces) and the structured blocks. Figure 4 shows a schematic of the bnode space. ndnode is the number of nodes for the non-block (disjoint primitive) grid. Each Quad-Mesh (m) adds $NI_m * NJ_m * NK_m$ nodes to the node space (where NI, NJ and NK are the number of nodes in each direction). The node numbering within the block follows the memory storage, that is, (i,j,k) in FORTRAN and [k][j][i] in C. The bnode number $= base + i + (j-1) * NI_m + (k-1) * NI_m * NJ_m$.
Notes:
1) All indices start at 1.
2) Either NI, NJ or NK must be 1 for each Quad-Mesh.
3) Disjoint Tri and Quad definitions may contain nodes defined within the Quad-Meshes.

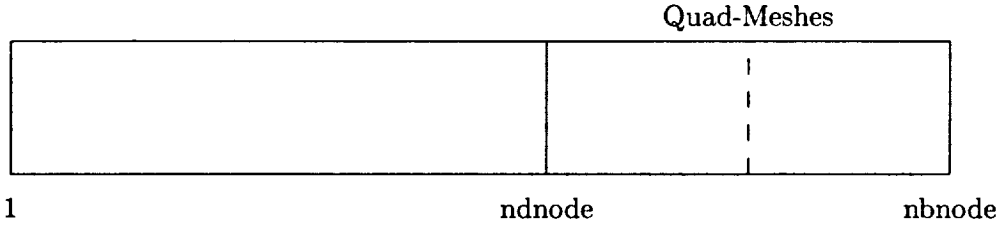

Figure 4: Boundary Node Space

2.3.2 Special Groupings

Special groupings are simply lists of bnodes that may be required by the solver's boundary condition routines. This is to flag "special" nodes. For example, if *IBlanking* is used,

there could be a list that contains the *IBlanked* nodes. If nodes along the Edges between Boundaries need to be treated differently from those interior nodes, then these edge nodes can be placed in a Special group.

2.3.3 Boundary Attachments

Boundary attachments are collections of data that are associated with the bnodes of the Boundary discretization. The attachments are identified by a name and can have an additional string that can indicate information on how and/or when the attachment was created. These attachments can be used to communicate boundary level data between modules (i.e., heat transfer to the visualization module), perform Zooming or otherwise couple like simulations at boundaries and perform multi-disciplinary coupling between Volumes.

2.3.4 Boundary Tags

Tags are character strings associated with the Boundary. Each string has an attached value string. These Tag entities are useful for specifying conditions or material information for the application of boundary conditions by the solver. For example; the Boundary named 'Wall' may have a Tag 'temperature' with the associated value '300K'.

2.4 Interpolation Handles

An integer *hook* is used in CAPRI to simplify the specification of Boundary to Boundary interpolation required by single and multi-disciplinary coupling. This Handle is an index to internal storage that contains information such as the indices and weights used in the source Boundary node-space to create Attachments to the destination Boundary discretization. Therefore each Handle is associated with four CAPRI indices. These indices are the Volume and Boundary indices for the source and the destination Volume and Boundary indices.

When a new Handle is created the current state of the Displacement and Replication data for the destination Volume is used as well as the Displacement matrix for the source when calculating the interpolation. Therefore, for transient simulations, the Handles associated with Volumes moving (with respect to the coupled volume) must be deleted and recreated every iteration.

All Volume data in CAPRI, when created or modified, is written out to files when the Volume (part) is saved. Because the data pointed to by the Handle is potentially cross Volume, this data is not written and must be recomputed when another CAPRI session is started.

2.5 The CAPRI API

The CAPRI API is sub-divided into the following components:

1. Utility routines
   These routines include initialization of CAPRI, loading CAD parts and querying status as well as closing the system down:

   - gi_uStart – Initialize CAPRI
   - gi_uLoadPart – Loads a Volume or number of Volumes from a CAD part
   - gi_uSavePart – Save away the CAD part
   - gi_uNumVolumes – Returns the number of active Volumes
   - gi_uGetModeller – Returns the Geometry Kernel in use
   - gi_uDelVolume – Removes a Volume from CAPRI
   - gi_uStop – Terminates CAPRI

2. Geometry data-base queries
   This allows all top level applications to figure out and get detailed information on any geometric component in the Volume definition:

   - gi_dGetNode – Returns the data for a Node
   - gi_dGetEdge – Returns the data for an Edge
   - gi_dGetFace – Returns the data for a Face
   - gi_dGetBoundary – Returns the data for a Boundary
   - gi_dNewBoundary – Creates a new Boundary for the Volume
   - gi_dMoveFace – Moves a Face from one Boundary to another
   - gi_dNameVolume – Assigns a string to a Volume
   - gi_dGetVolume – Returns the name of the Volume and the number of Nodes, Edges, Faces and Boundaries attached
   - gi_dBox – Returns the min and max coordinates for the Volume
   - gi_dTesselEdge – Returne the tesselation of an Edge
   - gi_dTesselFace – Returne the tesselation of a Face

3. Point queries

   These calls allow grid generators, or solvers doing node adaptation, to snap points directly on geometric entities:

   - gi_qPointOnEdge – Returns the point at the t parameter and optionally derivitives
   - gi_qNearestOnEdge – Returns the t parameter given a point
   - gi_qPointOnFace – Returns the point at the (u, v) parameters and optionally derivitives
   - gi_qNearestOnFace – Returns the (u, v) parameters given a point
   - gi_qNormalToFace – Returns the normal to the given (u, v) parameters
   - gi_qInEdge – Returns whether the given point is on the Edge.
   - gi_qInFace – Returns whether the given point is in the Face or not (in some hole or trimmed-off region)
   - gi_dInBoundary – Returns whether the given point is in the Boundary and associated Face index if it is.
   - gi_qInVolume – Returns whether the given point is contained within the specified Volume

4. Calculated or geometrically derived queries

   These calls calculate data from the geometry to aid in grid generation:

   - gi_cLengthOfEdge – Returns the arc-length of the Edge
   - gi_cCurvOfEdge – Returns the curvature at a point on the Edge
   - gi_cMaxCurvOfEdge – Returns the maximum Face curvature of a point on the Edge
   - gi_cCurvOfFace – Returns the curvatures and principal directions at a point on the Face
   - gi_cMaxCurvOfFace – Returns the maximum curature of a point on the Face
   - gi_cOrderLoops – Returns a Faces loop order and orientation

5. Boundary data routines

   This part of CAPRI allows general data to be attached to Boundaries so that the boundary conditions can be specified and stored within CAPRIs data-base:

   - gi_bSetDiscret – Sets the discretization for the Boundary
   - gi_bGetDiscret – Returns the discretization for the Boundary
   - gi_bGetCoord – Returns the discretization coordinates
   - gi_bGetTris – Returns the disjoint triangle discretization
   - gi_bGetQuads – Returns the disjoint quad discretization
   - gi_bGetQMesh – Returns the quad-mesh discretization
   - gi_bGet3DNode – Translate boundary node index to 3D mesh node
   - gi_bSetSpecial – Set/Update a Special grouping
   - gi_bGetSpecial – Return data about a Special grouping
   - gi_bGetISpecial – Return data about a Special grouping (by index)
   - gi_bDelSpecial – Removes a Special grouping

6. Geometry based interpolation routines

   This part of the API facilitates Multi-disciplinary coupling and allows zooming through Boundary Attachments:

   - gi_iSetAttach – Set/Update a Boundary Attachment
   - gi_iGetAttach – Return data about a Boundary Attachment
   - gi_iGetIAttach – Return data about a Boundary Attachment (by index)
   - gi_iDelAttach – Removes a Boundary Attachment
   - gi_iGetDisplace – Returns Volume displacement matrix used for interpolation
   - gi_iSetDisplace – Specifies Volume displacement matrix
   - gi_iGetReplicate – Returns Volume replication used for interpolation
   - gi_iSetReplicate – Specifies Volume replication for interpolation
   - gi_iNewHandle – Creates a new interpolation Handle
   - gi_iGetHandle – Returns data about an interpolation Handle
   - gi_iDelHandle – Removes a Handle
   - gi_iInterAttach – Interpolate to produce/update Boundary Attachment

7. Tag based routines

   This part of the API allows the specification of properties associated with either the Volume or Boundary entities:

   - gi_tGetNumVolume – Returns the number of Volume Tags
   - gi_tGetVolume – Return associated string for the specified Tag
   - gi_tGetIVolume – Return data for the specified Tag (by index)
   - gi_tSetVolume – Set/Update a Tag
   - gi_tGetNumBoundary – Returns the number of Boundary Tags
   - gi_tGetBoundary – Return associated string for the specified Tag
   - gi_tGetIBoundary – Return data for the specified Tag (by index)
   - gi_tSetBoundary – Set/Update a Tag

8. Geometric modification

   This will be used for an automated design system where the goal of the application is to change the geometry. Routines that allow this have the advantage that if the data is kept consistent with the CAD package, then the design can be incorporated directly and therefore is manufacturable.

   Not yet defined!

2.5.1 Main Program

Some CAD programming interfaces require that the CAD system start and that the modification or enhancements be performed as call-backs. This means that a CAPRI module may not actually be the *Main Program.* For C++ this may be a problem and can only be resolved by following the CAD system's recommendations. For C programming, CAPRI's include file defines CAPRI_MAIN in an appropriate manner and this can be used as the *main* declaration including the argument list.

In FORTRAN and FORTRAN/90 there is no main program (in fact for all UNIX FORTRANs a MAIN PROGRAM is actually a special subroutine). The top-level CAPRI execution starts in a subroutine called MAIN_CAPRI that has no arguments.

2.5.2 Memory Usage

The CAPRI API returns pointers to memory in many calls when using either C or C++ as the programming language. These are usually pointers to the internal CAPRI data and should NOT be **free**d. Also, the data contained within these memory blocks should only be read and never modified.

If the data is constructed and then returned for the programmer, then the description of the routine will specify that the pointer returned is **freeable** and should be **freed** by the programmer when the data is no longer required. The programmer may, in this case, change or modify the contents of the memory block(s).

Using FORTRAN/FORTRAN-90 with CAPRI functions that return data always fill the calling routines arrays. In this case the size of the array must be placed in the parameter that will return the final length. This insures that the arrays will not be over-run. If the size is insufficient to hold the data, then an error indication is set. The function may be recalled with larger arrays to get the complete information.

Character strings in FORTRAN/FORTRAN-90 are potentially truncated if the returned text information is longer than the variable. No error is set.

3 The Modules

CAPRI does not specify either the use or control of the suite. The design system or analysis software can be different programs controlled at some higher level (such as command line language scripts or another code). The software could also be built as a single integrated application. For example, an automated optimizer could be used to drive the entire suite, and in this case, it would obviously be in control. Software control could also be specified by some visually based GUI that allows plugging the modules together as envisioned by NPSS.

For this to work, and for general plug and play, there needs to be agreement on how input parameters get passed through the system(s). Within CAPRI, the Tag and Attachment concepts are used for specifying this data. Tags place simple (or integrated quantities) associated with the Volume or Boundary of interest into CAPRI so that other modules have direct access. Complex specification of input can be achieved by mapping quantities to Boundaries via Attachments. For modules to fit together agreement is required for the naming of these entities.

It is the responsibility of the controlling software to fill the required Tags and/or Attachments for the target module (or cause them to be filled by executing some other module in the system) before initiating execution.

3.1 Grid Generator

The Grid Generator task is the most straight forward. It uses CAPRI to query the geometry and topology of the part of interest and then performs the meshing. Once complete, it specifies the Boundary discretization for the Volume so that the other modules can communicate their data on the Boundary.

3.2 Solver

The simulation software gets the input data from CAPRI via Tags and Attachments but will need to read the cloud of data (both mesh point locations and solution) from outside of CAPRI. At the end of each cycle, and/or at the end of the simulation, Attachments and integrated values (as Tags) are created or updated by calls to CAPRI. This data is the information required by other modules in the system.

3.3 Visualization

The visualization software need not query the user for the surfaces of interest. These are the Boundaries of the Volume(s). The list of Attachments can be scanned for each Boundary and then these quantities (scalar, vector and/or state-vector) may be mapped and rendered on the surfaces without the graphics software needing to know how to produce these values. In this way the Viz software can be used to view and debug the coupling between simulations (either multi-discipline or within a single discipline).

4 The Geometry Viewer

The Geometry Viewer is not an integral part of the API, but is a stand-alone tool-kit that augments CAPRI. It can be thought of as another module in the software suite. The Geometry Viewer is designed to be able to either become the visual interface and/or a debugging aide for CAPRI.

The Geometry Viewer has been written to be modular and attachable to applications that deal with data which normally come from either CAD systems and/or grid generators. Unlike CAPRI, the language interfaces are only C and C++. This is due to the complex data structures that need to be exposed to the programmer.

The Geometry Viewer has a hierarchical definition of the data to be represented on the screen. At the lowest level are the Objects like; points, lines, surface facets and volume cell data. Objects of similar type can be collected as a Graphic. The plotting of Graphics can be controlled by a attribute that can be interactively or programatically adjusted. Graphics can also be picked. The highest level definition is a Family. A Family of Graphics can be of mixed Object type and the attributes can be controlled interactively in a ganged manner.

The Viewer has two execution modes; (1) normal, serial, execution where program control is passed to the graphics, the data is examined and then when the user is satisfied, execution resumes in the calling program. (2) Multi-threaded, where the data is shared between to executing threads (application and graphics) and both can be concurrently active allowing viewing as the application runs. This is particularly useful in the debugging of grid generators.

The user interface is multi-windowed and has the same look and feel as Visual3 applications and the pV3 Server and Viewer. Because the Geometry Viewer was not designed as a scientific visualization system, there is only the ability to deal with grids and geometry. More effort has been put towards lighting models and the ability to light either faceted (normals based on cell faces) as well as smoothly (normals based on nodes). The 2D window is only used for a planar cutting surface so that the interior of volumes may be examined.

The Geometry Viewer has the following features:

- OpenGL
  All 3D and 2D rendering is performed in OpenGL to achive high performance and good animation.

- 3D Viewing
  Items may be rendered in a specified color or colored via scalars that are either defined at nodes or facets. The line and surface primitives (Objects) may be either

indexed (based on a list of points) or non-indexed. The following attributes may be interactively adjusted for Graphics:

  - Points: Rendering on/off
  - Lines: Rendering on/transparent/off, Moved forward (not obscured by surfaces), Orientation (direction) on/off
  - Surfaces: Rendering on/transparent/off, Lighting faceted/smooth, Orientation (front vs. back) on/off, Mesh on/moved forward/off.

- 2D Viewing
  The intersection of the plane and lines are plotted as points in the 2D window. Intersected surfaces are displayed as curves. Any 3D mesh that is cut is displayed as the intersected cell faces (lines) within the volume.

- Picking and Locating. Picking (pointing at and selecting objects) in the 3D window is supported. This is useful in CAPRI for specifying the Boundary entities interactively. Locating (3D pointing and retrieving 3 space coordinates) is useful for interactive modification of geometry.

- Data-base This window is dedicated to the Graphics within the Geometry Viewer. This is were the interactive control of the plotting attributes is performed. Graphics not in Families (orphans) are listed on separate lines. Families are listed by name and may be opened to look at and adjust the attributes of individual members. A Family can be created in this window. Also, moving an orphan Graphic (or a Graphics in a Family) to a Family is done here.

5 Notes on Supported Systems

5.1 UniGraphics & ICAD – Parasolid

- Modes of Execution
  - Normal Mode
    This mode carries all of the used parts of the **Parasolid** library as a portion of the executable. This has the advantage that start-up is simpler, but the enormous disatvantage that any executable is HUGE.
  - Shared Library Mode
    Shared Library Mode produces modules that are small but requires that the shared library be accessable at run time.
- License Requirements
  A **Parasolid** development license is required to build CAPRI applications. A **Parasolid** run-time license is required to execute the module(s). Any UniGraphics or ICAD seat has a run-time **Parasolid** license.
- Environment Variables
  - PARASOLID
    This environment variable must be assigned to the character string that specifies the path required to find the **Parasoild** distribution.
  - P_SCHEMA
    This environment variable must point to the location that contains the **Parasoild** schema files. Usually this is assigned like:
    "setenv P_SCHEMA $PARASOLID/schema".
  - LD_LIBRARY_PATH
    This is a system level environment that must be used in Shared Library Mode. It points to the location of the shared library in the **Parasoild** distribution. Usually this is assigned like:
    "setenv LD_LIBRARY_PATH $PARASOLID/shared_object"
    as long as other shared libraries are not currently specified.

5.2 Pro/ENGINEER – Pro/TOOLKIT

- Modes of Execution

  - Simple Asynchronous Mode

    It is advised to use this mode when debugging a CAPRI application. When a CAPRI module (build for this mode) is started, it invokes **ProE** in the backgound and communicates via sockets. This has the advantage that the CAPRI application is separate from **ProE** and therefore has it's own main program and address space. The disadvantage is that the overhead in the socket communication can be overwhelming if many geometry queries are required.

  - Spawn (multi-process) Mode

    In this mode the CAPRI module becomes a secondary thread of **ProE**. In this case communication is performed via shared memory and is much faster than Simple Asynchronous Mode. A problem in debugging is that the debugger sees the address space of both the CAPRI application and **ProE**.

    The CAPRI thread is initiated by **ProE**, therefore the main program is **ProE**. The module's main should be compiled with the *-DNO_MAIN* flag so that the proper name is used to start the thread.

  - DLL Mode

    This is the mode that gives the best performance. The CAPRI application becomes an integral part of **ProE**. Again, like Spawn Mode, the module's main should be compiled with the *-DNO_MAIN* flag so that the proper name is used to connect to **ProE**.

  For either Spawn or DLL Modes special care must be taken when starting up **ProE**. CAPRI modules run during **ProE** startup and can terminate before the CAD system fully becomes operational. If no **ProE** windows are desired the command line argument "-g:no_graphics" must be used. Command line agruments to be passed on to CAPRI modules should be prefaced by a "+", to differentiate them from the **ProE** arguments. Also the file *protk.dat* is required in the current directory when **ProE** is invoked.

- License Requirements

  A license for Pro/TOOLKIT is required to do development. Once the CAPRI application is finished, the module(s) may be *unlocked* so that they can be distributed and used without the Pro/TOOLKIT license. See the Pro/TOOLKIT User's Guide for the instructions on unlocking an application.

  Each CAPRI application will consume one **ProE** license.

- Environment Variables & other Files

  - PRO_COMMAND – Environment Variable
    This variable is used in Simple Asynchronous Mode so that it is known how to start-up **ProE**. The string used to execute **ProE** at a prompt should be assigned to this varaible.
  - PRO_COMM_MSG_EXE – Environment Variable
    This variable is also used only in Simple Asynchronous Mode. This tells the CAPRI system where to find the **ProE** module used to perform the socket based communication. The path of this file must be assigned to this variable, for example:
    "setenv PRO_COMM_MSG_EXE /usr/ProE/sgi_elf2/obj/pro_comm_msg".
  - The Registry File – *protk.dat*
    This file must be in the current directory when **ProE** is invoked for either Spawn or DLL Modes. This tells **ProE** where and how to initiate the CAPRI application. The CAPRI distribution has some examples of this file. See the Pro/TOOLKIT User's Guide for a complete description of the options in the Registry File.

A Utility Calls

A.1 Start – Initialize CAPRI

icode = gi_uStart()
ICODE = IG_USTART()
This must be the first call to CAPRI. It initializes the system.

I: icode	Return code

A.2 LoadPart – Load Volume(s) from CAD part file

icode = gi_uLoadPart(char *name)
ICODE = IG_ULOADPART(NAME)
Before examining any CAD data a "solids" part must be loaded. This routine can load either a part or assembly from the CAD system. Therefore the result may be adding one or more Volumes to CAPRI.

C: name	Character string containing the file-name for the CAD data
I: icode	Return code

A.3 SavePart – Save Volume(s) to CAD part file

icode = gi_uSavePart(int vol, char *name)
ICODE = IG_USAVEPART(VOL, NAME)
This call allows the output of the part or assembly once data has been modified. Saving a Volume that is part of a assembly saves the entire assembly.

I: vol	Volume index
C: name	Character string containing the file-name for the part – should be a different name than used to read the part
I: icode	Return code

A.4 NumVolumes – Returns the Number of Active Volumes

numVl = gi_uNumVolumes()
NUMVL = IG_UNUMVOLUMES()
Any negative return is the indication of an error.

I: numVl	Number of Volumes/Return code

A.5 GetModeller – Returns the Geometry Kernel in use

icode = gi_uGetModeler(int vol, char **modeller)
ICODE = IG_UGETMODELLER(VOL, MODELLER)
This routine returns the string associated the CAD Kernel.

I: vol	Volume index
C: modeller	The geometry kernel string – currently either: **Parasolid** **Pro/ENGINEER**
I: icode	Return code

Note: At this point CAPRI can not mix Geometry Kernels in a single application.

A.6 DelVolume – Removes a Volume

icode = gi_uDelVolume(int vol)
ICODE = IG_UDELVOLUME(VOL)
This call deletes an active Volume from CAPRI and frees up all associated memory.

I: vol	Volume index

A.7 Stop – Terminate CAPRI

icode = gi_uStop(int exit)
ICODE = IG_USTOP(exit)
This must be the last call to CAPRI. It terminates the system and frees up all memory. CAPRI will need to be re-initialized before using any functions.

I: exit	0 - return; otherwise exit in the appropriate manner.
I: icode	Return code

B Geometry Data-Base Queries

B.1 GetNode – Returns the Data for a Node

icode = gi_dGetNode(int vol, node, double *point)
ICODE = IG_DGETNODE(VOL, NODE, POINT)
Returns the 3D coordinates associated with the Node.

I: vol	Volume index
I: node	Node index
D: point	Point – length 3 (returned)
I: icode	Return code

B.2 GetEdge – Returns the Data for an Edge

icode = gi_dGetEdge(int vol, edge, double *trange, int *nodes)
ICODE = IG_DGETEDGE(VOL, EDGE, TRANGE, NODES)
Returns the data associated with the Edge.

I: vol	Volume index
I: edge	Edge index
D: trange	t_{min} and t_{max} – length 2 (returned)
I: nodes	Node endpoint indices – length 2 (returned)
I: icode	Return code

B.3 GetFace – Returns the Data for a Face

icode = gi_dGetFace(int vol, face, double *urange, int *nloop, **loops, **edges)
ICODE = IG_DGETFACE(VOL, FACE, URANGE, NLOOP, LOOPS, EDGES)
Returns the data that defines the Face.

I: vol	Volume index
I: face	Face index
D: urange	u_{min}, v_{min}, u_{max} and v_{max} – length 4 (returned)

I: nloop	Number of Edge loops (returned)
I: loops	pointer to Edge loop lengths – nloop in total (returned)
I: edges	pointer to Edge pairs that make up all of the loops (returned) Each entry contains 2 integers, first the Edge index and second the sense (−1 or 1) – data length is the sum of all loop lengths
I: icode	Return code

FORTRAN note: The pointers is not returned. NLOOP must be set with the length of LOOPS and LOOPS(1) must be set with the size of EDGES (2 INTEGERS per) before the call is executed. LOOPS and EDGES are filled with the actual data and NLOOP is set with the number of loops for the Face. If either of the declared lengths are not long enough to store the data, then the return code CAPRI_OVERFLOW is set. Information is filled up to that limit.

B.4 GetBoundary – Returns the Data for a Boundary

icode = gi_dGetBoundary(int vol, bound, *nface, **faces, char **name)
ICODE = IG_DGETBOUNDARY(VOL, BOUND, NFACE, FACES, NAME)
Returns the data associated with the Boundary.

I: vol	Volume index
I: bound	Boundary index (0 - "UnAssigned")
I: nface	Number of faces (returned)
I: faces	pointer to the face indices (returned)
C: name	pointer to character string (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. NFACES must be set with the size of FACES at the call. It is returned with the actual length. If FACES is not declared large enough (by the calling routine) the return code CAPRI_OVERFLOW is set but all the data up to that length is correct.

B.5 NewBoundary – Creates a New Boundary for the Volume

icode = gi_dNewBoundary(int vol, char *name)
ICODE = IG_DNEWBOUNDARY(VOL, NAME)
Creates the new Boundary for the volume with the given name.

I: vol	Volume index
C: name	character string for the name of the Boundary
I: icode	Created Boundary index/Return code

B.6 MoveFace – Assigns a Face to a Boundary

icode = gi_dMoveFace(int vol, face, bound)
ICODE = IG_DMOVEFACE(VOL, FACE, BOUND)
Moves the Face from one Boundary to the assigned Boundary index. Note: All current discretizations, groupings, attachments and handles are removed from both source and destination Boundaries.

I: vol	Volume index
I: face	Face index
I: bound	Boundary index – target
I: icode	Return code

B.7 NameVolume – Assign a Title to a Volume

icode = gi_dNameVolume(int vol, char *name)
ICODE = IG_DNAMEVOLUME(VOL, NAME)
Gives the Volume a name.

I: vol	Volume index
C: name	character string assigned to the volume
I: icode	Return code

B.8 GetVolume – Returns the Data for a Volume

icode = gi_dGetVolume(int vol, *nnode, *nedge, *nface, *nbound, char **name)

ICODE = IG_DGETVOLUME(VOL, NNODE, NEDGE, NFACE, NBOUND, NAME)

Returns the number of entities associated with the Volume index.

I: vol	Volume index
I: nnode	number of Nodes associated with the volume (returned)
I: nedge	number of Edges associated with the volume (returned)
I: nface	number of Faces associated with the volume (returned)
I: nbound	number of Boundaries found within the volume (returned)
C: name	pointer to the string for the Volume's name (returned)
I: icode	Return code

B.9 Box – Return the Bounding Coordinates for the Volume

icode = gi_dBox(int vol, double *box)

ICODE = IG_DBOX(VOL, BOX)

Returns the coordinate box that contains the Volume.

I: vol	Volume index
D: box	X_{min}, Y_{min}, Z_{min}, X_{max}, Y_{max} and Z_{max} – length 6 (returned)
I: icode	Return code

B.10 TesselEdge – Returns the tesselation of the Edge

icode = gi_dTesselEdge(int vol, edge, *npt, double **pt, **t)
ICODE = IG_CTESSELEDGE(VOL, EDGE, NPT, PT, T)

Computes and returns an indexed tesselation of the Edge with the associated parameters for each point.

I: vol	Volume index
I: edge	The Edge index
I: npt	The number of points that support the tesselation (returned)
D: pt	A pointer to the points (returned) Each point requires 3 doubles (X, Y and Z) and therefore the length of pt is atleast 3*npt.
D: t	A pointer to the parameters (returned)
I: icode	Return code

FORTRAN note: The pointers are not returned. NPT must be set with the size of PT (3 DOUBLE PRECISION words per) and T (a DOUBLE PRECISION word per point) before the call. NPT is returned with the actual lengths used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

B.11 TesselFace – Returns a connected tesselation of the Face

icode = gi_dTesselFace(int vol, face, *ntri, **tris, **tric, *npt,
double **pt, int **ptype, **pindex, double **uv)
ICODE = IG_DTESSELFACE(VOL, FACE, NTRI, TRIS, TRIC,
NPT, PT, PTYPE, PINDEX, UV)

Returns the connected indexed tesselation of the Face. Data is also supplied so that matching may be done along Edges that bound the Face.

I: vol	Volume index
I: face	Face index
I: ntri	The number of triangles (returned)
I: tris	A pointer to the indexed triangle definitions (returned). Each triangle is defined by 3 indices (bias 1) into the memory pointed to by pt.

I: tric	A pointer to the triangle connections (returned). Each triangle has 3 connections as indices to tris (bias 1). Each indicates the triangle neighbor of the opposing side (of the triangle node). A negative value indicates that the side is in an Edge. The absolute value is the Edge index.
I: npt	The number of points that support the tesselation (returned)
D: pt	A pointer to the points (returned). Each point requires 3 doubles (X, Y and Z) and therefore the length of pt is atleast 3*npt.
I: ptype	A pointer to a marker that specifies the type of point (returned). -1 indicates an interior point, 0 flags a Node and any positive value points to an Edge point (where the value is the index [bias 1] into the Edge tesselation that matches)
I: pindex	A pointer that holds the indices for the type (returned). -1 flags an interior point. If ptype for the point is 0 the pindex value is the Node index, if ptype is positive then this holds the Edge index.
D: uv	A pointer to the parameters associated with the points (returned). Each point requires 2 doubles (U and V) and therefore the length of uv is 2*npt.
I: icode	Return code

FORTRAN note: The pointers are not returned. NTRI must be set with the size of TRIS (3 INTEGERS per triangle) and TRIC (3 INTEGERS per triangle) at the call, NPT must be set with the size of PT (3 DOUBLE PRECISION words per), PTYPE (one INTEGER per), PINDEX (one INTEGER per), and UV (2 DOUBLE PRECISION words per point). NTRI and NPT are returned with the actual lengths used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

C Point Queries

C.1 PointOnEdge – Returns the Coordinates On the Edge

icode = gi_qPointOnEdge(int vol, edge, double t, *point, int der,
double *d1, *d2))
ICODE = IG_QPOINTONEDGE(VOL, EDGE, T, POINT, DER, D1, D2)
Returns the Point and derivitives (optionally) at the t parameter.

I: vol	Volume index
I: edge	Edge index
D: t	t parameter
D: point	Point – length 3 (returned)
I: der	Derivative Flag:
	0 - No derivatives (only return point)
	1 - Compute and return first derivative
	2 - Compute and return first and second derivatives
D: d1	First derivative – length 3 (returned, der > 0)
D: d2	Second derivative – length 3 (returned, der > 1)
I: icode	Return code

C.2 NearestOnEdge – Finds the Nearest Position to the Edge

icode = gi_qNearestOnEdge(int vol, edge, double *coor, *t, *point)
ICODE = IG_QNEARESTONEDGE(VOL, EDGE, COOR, T, POINT)
Returns the closest coordinates to the input point on the Edge and the t parameter.

I: vol	Volume index
I: edge	Edge index
D: coor	Input point – length 3
D: t	t parameter (input & returned)
D: point	Point – length 3 (returned)
I: icode	Return code

Note: Some Geometry Kernels (currently only **Parasolid**) may require a *hint* so that the desired point can be found. This parameter value can be put in t before the call and should be a value close to desired point (if available). If t is not in the valid range then the mid-point of the parameter range is used.
Therefore, it is always recommended that either a valid t is set before the function is called (reguardless of the CAD system) or the value is set out-of-range. The resultant *point* should be checked. If it is at the bounds of the Edge or far off the target, try another *hint*.

C.3 PointOnFace – Returns the Coordinates On the Specified Face

icode = gi_qPointOnFace(int vol, face, double *uv, *point, int der,
double *du, *dv, *duu, *duv, *dvv)
ICODE = IG_QPOINTONFACE(VOL, FACE, UV, POINT, DER, DU, DV,
DUU, DUV, DVV)

Returns the Point and derivitives (optionally) at the (u, v) parameters.

I: vol	Volume index
I: face	Face index
D: uv	(u, v) parameters – length 2
D: point	Point – length 3 (returned)
I: der	Derivative Flag: **0** - No derivatives (only return point) **1** - Compute and return first derivative **2** - Compute and return first and second derivatives
D: du	First derivative of u – length 3 (returned, der > 0)
D: dv	First derivative of v – length 3 (returned, der > 0)
D: duu	Second derivative of u – length 3 (returned, der > 1)
D: duv	Cross derivative – length 3 (returned, der > 1)
D: dvv	Second derivative of v – length 3 (returned, der > 1)
I: icode	Return code

C.4 NearestOnFace – Finds the Nearest Position to the Face

icode = gi_qNearestOnFace(int vol, face, double *coor, *uv, *point)
ICODE = IG_QNEARESTONFACE(VOL, FACE, COOR, UV, POINT)
Returns the closest coordinates to the input point on the Face and the (u, v) parameters.

I: vol	Volume index
I: face	Face index
D: coor	Input point – length 3
D: uv	(u, v) parameters – length 2 (input & returned)
D: point	Point – length 3 (returned)
I: icode	Return code

Note: Some Geometry Kernels (currently only **Parasolid**) may require a *hint* so that the desired point can be found. These parameter values can be put in *[u,v]* before the call and should result in a point close (if available). If *[u,v]* is not in the valid range then the mid-point of the parameter ranges is used.
Therefore, it is always recommended that either a valid *[u,v]* is set before the function is called (reguardless of the CAD system) or a value is set out-of-range. The resultant *point* should be checked. If it is at the bounds of the Face or far off the target, try another *hint.*

C.5 NormalToFace – Finds the Normal at the Specified Parameters

icode = gi_qNormalToFace(int vol, face, double *uv, *point, *norm)
ICODE = IG_QNORMALTOFACE(VOL, FACE, UV, POINT, NORM)
Returns the normal to the Face at the (u, v) parameters.

I: vol	Volume index
I: face	Face index
D: uv	(u, v) parameters – length 2
D: point	Point – length 3 (returned)
D: norm	Normal – length 3 (returned)
I: icode	Return code

C.6 InEdge – Is the Point Contained in the Edge

icode = gi_qInEdge(int vol, edge, double *point)
ICODE = IG_QINEDGE(VOL, EDGE, POINT)
Returns a condition indicating whether the point is on the Edge.

I: vol	Volume index
I: edge	Edge index
D: point	Point – length 3
I: icode	Return code – CAPRI_OUTSIDE is returned when the point is not contained on the Edge

C.7 InFace – Is the Point Contained on the Face

icode = gi_qInFace(int vol, face, double *point)
ICODE = IG_QINFACE(VOL, FACE, POINT)
Returns a condition indicating whether the point is on the Face.

I: vol	Volume index
I: face	Face index
D: point	Point – length 3
I: icode	Return code – CAPRI_OUTSIDE is returned when the point is not contained on the Face

C.8 InBoundary – Is the Point Contained on the Boundary

icode = gi_qInBoundary(int vol, bound, double *point, int *face)
ICODE = IG_QINBOUNDARY(VOL, BOUND, POINT, FACE)
Returns a condition indicating whether the point is on the Boundary.

I: vol	Volume index
I: face	Bound index
D: point	Point – length 3
I: face	Face index for Face containing the point (returned)
I: icode	Return code – CAPRI_OUTSIDE is returned when the point is not contained on the Boundary

C.9 InVolume – Is the Point Contained within the Volume

icode = gi_qInVolume(int vol, double *point)

ICODE = IG_QINVOLUME(VOL, POINT)

Returns a condition indicating whether the point is in the Volume.

I: vol	Volume index
D: point	Point – length 3
I: icode	Return code – CAPRI_OUTSIDE is returned when the point is not contained within the Volume

D Calculated or Geometrically Derived Queries

D.1 LengthOfEdge – Returns the arc-length of the Edge

icode = gi_cLengthOfEdge(int vol, edge, double t1, t2, *len)
ICODE = IG_CLENGTHOFEDGE(VOL, EDGE, T1, T2, LEN)
Returns the length along the Edge between the parameter range t_1 and t_2.

I: vol	Volume index
I: edge	Edge index
D: t1	t parameter for the start of the calculation
D: t2	t parameter for the end of the calculation – t_1 must be less than t_2.
D: len	the resultant length (returned)
I: icode	Return code

D.2 CurvOfEdge – Gets the tangent and curvature for an Edge point

icode = gi_cCurvOfEdge(int vol, edge, double t, *tang, *curv)
ICODE = IG_CCURVOFEDGE(VOL, EDGE, T, TANG, CURV)
Returns the curvature and unit tangent found at t along the Edge.

I: vol	Volume index
I: edge	Edge index
D: t	t parameter along the Edge
D: tang	the unit tangent – length 3 (returned)
D: curv	the curvature (returned)
I: icode	Return code

D.3 MaxCurvOfEdge – Gets the maximum curvature for the attached Faces

icode = gi_cMaxCurvOfEdge(int vol, edge, double t, *curv)
ICODE = IG_CMAXCURVOFEDGE(VOL, EDGE, T, CURV)
Returns the maximum curvature found at t along the Edge for the Faces that share the Edge.

I: vol	Volume index
I: edge	Edge index
D: t	t parameter along the Edge
D: curv	the maximum curvature (returned)
I: icode	Return code

D.4 CurvOfFace – Gets the principal directions and curvature at a Face point

icode = gi_cCurvOfFace(int vol, face, double *uv, *dir1, *cur1, *dir2, *cur2)
ICODE = IG_CCURVOFFACE(VOL, FACE, UV, DIR1, CUR1, DIR2, CUR2)
Returns the curvature and principle directions at (u, v) in the Face.

I: vol	Volume index
I: face	Face index
D: uv	(u, v) parameters for the Face – length 2
D: dir1	the first principal direction – length 3 (returned)
D: cur1	the curvature for first principal direction (returned)
D: dir2	the second principal direction – length 3 (returned)
D: cur2	the curvature for second principal direction (returned)
I: icode	Return code

D.5 MaxCurvOfFace – Returns the maximum curvature of a Face point

icode = gi_cMaxCurvOfFace(int vol, face, double *uv, *curv)
ICODE = IG_CMAXCURVOFFACE(VOL, FACE, UV, CURV)
Returns the maximum curvature found at (u, v) in the Face.

I: vol	Volume index
I: face	Face index
D: uv	(u, v) parameters for the Face – length 2
D: curv	the maximum curvature (returned)
I: icode	Return code

D.6 OrderLoops – Returns a Faces loop order and orientation

icode = gi_cOrderLoops(int vol, face, *nsface, *nloop, **loop)
ICODE = IG_CORDERLOOPS(VOL, FACE, NSFACE, NLOOP, LOOP)
Returns the maximum curvature found at (u, v) in the Face.

I: vol	Volume index
I: face	Face index
I: nsface	Number of subfaces – outer loops (returned) NOTE: Some CAD systems may allow a single Face definition to be used for more than one surface. For example: a plane completely bisected by another primitive – this creates 2 planar surfaces with the same parameterization.
I: nloop	Number of loops (returned)
I: loop	pointer to loop order information – nloop in total length (returned & freeable) The data is ordered in the same manner as the loops returned from gi_dGetFace. The items returned contain absolute values from 1 to nsface. This identifys to which subface the loop belongs. Any subface has only 1 outer loop (positive) and any number of holes (negative). NOTE: If nloop is 1 then NULL is returned.
I: icode	Return code

FORTRAN note: The pointer is not returned. NLOOP must be set with the size of LOOP at the call. It is returned with the actual length used. If LOOP is not declared large enough (by the calling routine) the return code CAPRI_OVERFLOW is set but all the data up to that length is correct.

E Boundary data routines

E.1 SetDiscret – Declares the Discretization for the Boundary

icode = gi_bSetDiscret(int vol, bound, ndnode, ntris, nquads, nqmeshs, flag, *nbnode)

ICODE = IG_BSETDISCRET(VOL, BOUND, NDNODE, NTRIS, NQUADS, NQMESHS, FLAG, NBNODE)

This routine sets the grid discretization for the Boundary. This may be comprised of a homogenous or heterogenous collection of disjoint triangles, disjoint quadrangles and quad-meshes. This call implicitly defines a boundary node (bnode) numbering, where NDNODE is the number of nodes associated with the disjoint primitives, the rest of the bnodes are defined from the quad-meshes (attached to structured blocks). This routine will cause the execution of as many as 5 supplied routines, based on the arguments. These call-backs define the collection of data for the bnodes, triangles, quadrangles, quad-meshes and node coordinates.

I: vol	Volume index
I: bound	Boundary index
I: ndnode	Number of nodes associtated with the disjoint primitives
I: ntris	Number of disjoint triangles assigned to the Boundary
I: nquads	Number of disjoint quadrangles assigned to the Boundary
I: nqmeshs	Number of quad-meshes (from structured blocks)
I: flag	Update flag (if the Discretization changes): **0** - Remove all Attachments and Special groupings **1** - Remove all Special groupings, interpolate to new bnodes for Attachments
I: nbnode	Total number of bnodes for the Boundary (returned)
I: icode	Return code

NOTE: If ntris, nquads and nqmeshs are all zero, then the descritization is removed.

gibFillCoord(int vol, bound, nbnode, double *xyz)
IGBFILLCOORD(VOL, BOUND, NBNODE, XYZ)
This routine must be supplied for any call to gi_bSetDiscret. Its responsibility is to fill the coordinate data associated with the bnodes.

I: vol	Volume index
I: bound	Boundary index
I: nbnode	Number of Boundary nodes
D: xyz	The 3-space coordinates for each bnode. Length is 3*nbnode (filled)

gibFillTris(int vol, bound, ntris, *tris, *ctris)
IGBFILLTRIS(VOL, BOUND, NTRIS, TRIS, CTRIS)
This routine must be supplied if the call to gi_bSetDiscret specifies any disjount triangles ($ntris \neq 0$). gibFillTris' responsibility is to fill the data required for disjoint triangles.

I: vol	Volume index
I: bound	Boundary index
I: ntris	Number of disjoint triangles assigned to the Boundary
I: tris	3 bnode numbers are required for the definition of each triangle. The bnode numbers may come from either the set of disjoint nodes and/or the nodes defined via the quadmeshes. Length is 3*ntris (filled)
I: ctris	The mesh 3D cell number containing the trianglar face. Note: This is not used internally by CAPRI. Length is ntris (filled)

gibFillQuads(int vol, bound, nquads, *quads, *cquads)
IGBFILLQUADS(VOL, BOUND, NQUADS, QUADS, CQUADS)
This routine must be supplied if the call to gi_bSetDiscret specifies any disjount quads ($nquads \neq 0$). gibFillQuads' responsibility is to fill the data required for disjoint quadrangles.

I: vol	Volume index
I: bound	Boundary index

I: nquads	Number of disjoint quadrangles assigned to the Boundary
I: quads	4 bnode numbers are required for the definition of each quad. The bnode numbers may come from either the set of disjoint nodes and/or the nodes defined via the quad-meshes. Length is 4*nquads (filled)
I: cquads	The mesh 3D cell number containing the quad face. Note: This is not used internally by CAPRI. Length is nquads (filled)

gibFillQMesh(int vol, bound, nqmeshs, *block, *bsizes, *lims)
IGBFILLQMESH(VOL, BOUND, NQMESHS, BLOCK, BSIZES, LIMS)
This routine must be supplied if the call to gi_bSetDiscret specifies any quad-meshes (i.e., $nqmeshs \neq 0$). gibFillQMesh's responsibility is to fill the data required for faces of structured blocks mapped to the Boundary.

I: vol	Volume index
I: bound	Boundary index
I: nqmeshs	Number of quad-meshes touching the Boundary
I: block	Block number (in the complete grid) with the associated mapping. Length is nqmeshs (filled)
I: bsizes	The sizes (N_I, N_J, N_K) for the block. Length is 3*nqmeshs (filled)
I: lims	6 entries that define the extent of the exposed block. The first 2 entries are the min and max indices for the first index (usually I). The next 2 entries are the min and max for the second index (J). The last 2 entries are the min and max indices for the last index (usually K). One of the set must be the same and numbering is 1 biased. For example: 1,1, 1,23, 10,100 – specifies the first I plane, with J going from the first index up to (and including) 23 and K starting at 10 and continuing up to 100 specifying 1980 quads. Length is 6*nqmeshs (filled)

gibFillDNodes(int vol, bound, ndnode, *nodes)
IGBFILLDNODES(VOL, BOUND, NDNODE, NODES)
This routine must be supplied if the call to gi_bSetDiscret specifies any disjoint nodes (*ndnode* $\neq$ 0) and CAPRI is to be used to translate bnode numbers back to 3D mesh indices (calls to gi_bGet3DNode are used).

I: vol	Volume index
I: bound	Boundary index
I: ndnode	Number of nodes used in the disjoint tris and quads.
I: nodes	3D node number (in the complete grid). Length is ndnode (filled)

E.2 GetDiscret – Returns data about the Discretization for the Boundary

icode = gi_bGetDiscret(int vol, bound, *nbnode, *ndnode, *ntris, *nquads, *nqmeshs, *nattach, *nspecial)
ICODE = IG_BGETDISCRET(VOL, BOUND, NBNODE, NDNODE, NTRIS, NQUADS, NQMESHS, NATTACH, NSPECIAL)
This routine gets the sizes of grid discretization and the lengths for any associated data for the Boundary.

I: vol	Volume index
I: bound	Boundary index
I: nbnode	Number of bnodes found in the Boundary (returned) A zero indicates no discretization
I: ndnode	Number of disjoint bnodes found in the Boundary (returned)
I: ntris	Number of disjoint triangles (returned)
I: nquads	Number of disjoint quadrangles (returned)
I: nqmeshs	Number of quad-meshes (returned)
I: nattach	Number of associated attachments (returned)
I: nspecial	Number of associated special groups (returned)
I: icode	Return code

E.3 GetCoord – Returns the Boundary Discretization Coordinates

icode = gi_bGetCoord(int vol, bound, *nbnode, double **xyz)
ICODE = IG_BGETCOORD(VOL, BOUND, NBNODE, XYZ)
This routine returns the coordinates associated with all of the bnodes.

I: vol	Volume index
I: bound	Boundary index
I: nbnode	Number of Boundary nodes (returned)
D: xyz	pointer to the 3-space coordinates for each bnode. Length of data is 3*nbnode (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. NBNODE must be set with the size of XYZ (3 DOUBLE PRECISON words per bnode) at the call. It is returned with the actual length used. If XYZ is not declared large enough (by the calling routine) the return code CAPRI_OVERFLOW is set but all the data up to that length is correct.

E.4 GetTris – Returns the Disjoint Triangle Discretization

icode = gi_bGetTris(int vol, bound, *ntris, **tris, **ctris)
ICODE = IG_BGETTRIS(VOL, BOUND, NTRIS, TRIS, CTRIS)
This routine returns the list of disjoint tris defining the Boundary discretization.

I: vol	Volume index
I: bound	Boundary index
I: ntris	Number of disjoint triangles (returned)
I: tris	pointer to 3 bnode numbers for the definition of each triangle. Length of data is 3*ntris (returned)
I: ctris	pointer to the mesh 3D cell number containing the trianglar face. Length of data is ntris (returned)
I: icode	Return code

FORTRAN note: The pointers are not returned. NTRIS must be set with the size of TRIS (3 INTEGERS per triangle) and CTRIS at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

E.5 GetQuads – Returns the Disjoint Quadrangle Discretization

icode = gi_bGetQuads(int vol, bound, *nquads, **quads, **cquads)
ICODE = IG_BGETQUADS(VOL, BOUND, NQUADS, QUADS, CQUADS)
This routine returns the list of disjoint quads defining the Boundary discretization.

I: vol	Volume index
I: bound	Boundary index
I: nquads	Number of disjoint quadrangles (returned)
I: quads	pointer to 4 bnode numbers for the definition of each triangle. Length of data is 4*nquads (returned)
I: cquads	pointer to the mesh 3D cell number containing the quad face. Length of data is nquads (returned)
I: icode	Return code

FORTRAN note: The pointers are not returned. NQUADS must be set with the size of QUADS (4 INTEGERS per quad) and CQUADS at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

E.6 GetQMesh – Returns the Quad-Mesh Discretization

icode = gi_bGetQMesh(int vol, bound, *nqmeshs, **block, **bsizes, **lims)
ICODE = IG_BGETQMESH(VOL, BOUND, NQMESHS, BLOCK, BSIZES, LIMS)
This routine returns the list of quad-meshes used in the Boundary discretization.

I: vol	Volume index
I: bound	Boundary index
I: nqmeshs	Number of quad-meshs in the Boundary (returned)
I: block	pointer to the block number (in the complete grid). Length of data is nqmeshs (returned)
I: bsizes	pointer to the sizes (N_I, N_J, N_K) for the block. Length of data is 3*nqmeshs (returned)
I: lims	pointer to 6 entries that define the extent of the exposed block. Length of data is 6*nqmeshs (returned)

I: icode	Return code

FORTRAN note: The pointers are not returned. NQMESHS must be set with the size of BLOCK, BSIZES (3 INTEGERS per qmesh) and LIMS (6 INTEGERS per qmesh) at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

E.7 Get3DNode – Translates the Boundary node to 3D node number

icode = gi_bGet3DNode(int vol, bound, bnode, *type, *location)
ICODE = IG_BGET3DNODE(VOL, BOUND, BNODE, TYPE, LOCATION)
This routine returns the 3D mesh index associated with the bnode.

I: vol	Volume index
I: bound	Boundary index
I: bnode	Boundary node index – starts at 1.
I: type	Node type (returned) 0 – from a node associated with disjoint primitives 1 – from a node associated with quad-meshes
I: location	Mesh location (returned) Type 0: 3D Node number Type 1: I, J, K and Block # – 4 integers
I: icode	Return code

E.8 SetSpecial – Specify/Update a Special Grouping

icode = gi_bSetSpecial(int vol, bound, char *name, int size)
ICODE = IG_BSETSPECIAL(VOL, BOUND, NAME, SIZE)
This routine specifies a Special listing (by name). These special groupings can be used to indicate lists of bnodes that may have special boundary condtions applied (such as at the Edge between two Boundaries). If the listing already exists, it is overwritten with the new data. This routine will cause a call-back (documented next) to be executed.

I: vol	Volume index
I: bound	Boundary index
C: name	Listing name (i.e., "hub edge", "wing-body edge")
I: size	The length of the list
I: icode	Grouping index/Return code

gibFillSpecial(int vol, bound, char *name, int size, *list)
IGBFILLSPECIAL(VOL, BOUND, NAME, SIZE, LIST)
This call-back will be executed after a call to gi_bSetSpecial. The routines responsibility is to fill the list requested for the grouping.

I: vol	Volume index
I: bound	Boundary index
C: name	Special grouping name
I: size	The number of entries for the list
I: list	Special list – length is size (filled)

E.9 GetSpecial – Return the info for a Special Grouping

icode = gi_bGetSpecial(int vol, bound, char *name, int *size, **list)
ICODE = IG_BGETSPECIAL(VOL, BOUND, NAME, SIZE, LIST)
This routine returns data about a Special grouping (by name).

I: vol	Volume index
I: bound	Boundary index
C: name	Special grouping name
I: size	The length of the list (returned)

I: list	pointer to the list – data length is size (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. SIZE must be set with the length of LIST at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

E.10 GetISpecial – Get a Special Grouping by Index

icode = gi_bGetISpecial(int vol, bound, index, char **name, int *size, **list)
ICODE = IG_BGETISPECIAL(VOL, BOUND, INDEX, NAME, SIZE, LIST)
This routine returns data about a Special grouping (by index).

I: vol	Volume index
I: bound	Boundary index
I: index	Grouping index – bais 1.
C: name	Grouping name (returned)
I: size	The length of the grouping (returned)
I: list	pointer to the list – data length is size (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. SIZE must be set with the length of LIST at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

E.11 DelSpecial – Remove a Special Grouping

icode = gi_bDelSpecial(int vol, bound, char *name)
ICODE = IG_BDELSPECIAL(VOL, BOUND, NAME)
This routine deletes the data associated with a Special grouping. NOTE: the indices used with the groupings will be affected.

I: vol	Volume index
I: bound	Boundary index
C: name	Special listing name
I: icode	Return code

F Geometry Based Interpolation Routines

F.1 SetAttach – Specify/Update a Boundary Attachment

icode = gi_iSetAttach(int vol, bound, char *name, int rank, char *update)
ICODE = IG_ISETATTACH(VOL, BOUND, NAME, RANK, UPDATE)
This routine specifies a Boundary attachment (by name). If the attachment already exists, it is overwritten with the new data. This routine will cause call-back (documented next) to be executed.

I: vol	Volume index
I: bound	Boundary index
C: name	Attachment name (i.e., "pressure", "heat transfer")
I: rank	The number of entries per bnode. i.e., scalars are 1, vectors are 3 (or -3 – do not apply replication/displacement).
C: update	A character string to indicate something about the attachment. For example, if the simulation is transient this could contain the solvers time when last filled.
I: icode	Attachment index/Return code

giiFillAttach(int vol, bound, char *name, int rank, char *update, int nbnode, double *data)
IGIFILLATTACH(VOL, BOUND, NAME, RANK, UPDATE, NBNODE, DATA)
This call-back will be executed after a call to gi_iSetAttach that specifies a non-zero rank. The routines responsibility is to fill the data requested for the attachment.

I: vol	Volume index
I: bound	Boundary index
C: name	Attachment name
I: rank	The number of entries per bnode
C: update	A character string to indicate something about the attachment.
I: nbnode	Number of boundary nodes
D: data	Attached data – length is rank*nbnode (filled)

F.2 GetAttach – Get a Boundary Attachment

icode = gi_iGetAttach(int vol, bound, char *name, int *rank, char **update, int *nbnode, double **data)

ICODE = IG_IGETATTACH(VOL, BOUND, NAME, RANK, UPDATE, NBNODE, DATA)

This routine returns data about a Boundary attachment (by name).

I: vol	Volume index
I: bound	Boundary index
C: name	Attachment name
I: rank	The number of entries per bnode (returned)
C: update	pointer to the update character string (returned)
I: nbnode	Number of boundary nodes (returned)
D: data	pointer to attached data – data length is rank*nbnode (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. NBNODE must be set with the size of DATA at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

F.3 GetIAttach – Get a Boundary Attachment by Index

icode = gi_iGetIAttach(int vol, bound, index, char **name, int *rank, char **update, int *nbnode, double **data)

ICODE = IG_IGETIATTACH(VOL, BOUND, INDEX, NAME, RANK, UPDATE, NBNODE, DATA)

This routine returns data about a Boundary attachment (by index).

I: vol	Volume index
I: bound	Boundary index
I: index	Attachment index – bais 1.
C: name	Attachment name (returned)
I: rank	The number of entries per bnode (returned)

C: update	pointer to the update character string (returned)
I: nbnode	Number of boundary nodes (returned)
D: data	pointer to attached data – data length is rank*nbnode (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. NBNODE must be set with the size of DATA at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

F.4 DelAttach – Remove a Boundary Attachment

icode = gi_iDelAttach(int vol, bound, char *name)
ICODE = IG_IDELATTACH(VOL, BOUND, NAME)
This routine deletes the data associated with a Boundary attachment. NOTE: the indices used with the attachments will be affected.

I: vol	Volume index
I: bound	Boundary index
C: name	Attachment name
I: icode	Return code

F.5 GetDisplace – Gets the Volume's displacement matrix

icode = gi_iGetDisplace(int vol, double *dmatrix)
ICODE = IG_IGETDISPLACE(VOL, DMATRIX)
This routine returns the displacement matrix associated with the specified volume. The displacement matrix is a column-major matrix that is 4 columns by 3 rows and declared in C as [3][4] and in FORTRAN as (4,3). This matrix is used to multiply all Volume coordinates before interpolation is performed and therfore supports any combination of translation, rotation and scaling.

I: vol	Volume index
D: dmatrix	The displacement matrix
I: icode	Return code

F.6 SetDisplace – Set the Volume's displacement matrix

icode = gi_iSetDisplace(int vol, double *dmatrix)
ICODE = IG_ISETDISPLACE(VOL, DMATRIX)

This routine specifies the displacement matrix associated with the specified volume. The displacement matrix is a column-major matrix that is 4 columns by 3 rows and declared in C as [3][4] and in FORTRAN as (4,3).

I: vol	Volume index
D: dmatrix	The displacement matrix
I: icode	Return code

F.7 GetReplicate – Gets the Volume's replication data

icode = gi_iGetReplicate(int vol, *nrep, double *rmatrix)
ICODE = IG_IGETREPLICATE(VOL, NREP, RMATRIX)

This routine returns the replication data associated with the specified volume. This information is comprised of a matrix and the number of times to apply this matrix to the Volume. The replication matrix is a column-major matrix that is 4 columns by 3 rows and declared in C as [3][4] and in FORTRAN as (4,3). This matrix is used to multiply all Volume coordinates in order to produce additional instances of the Voulme (before the Displacement matrix is applied) and then the interpolation is performed. When properly used this allows mirroring and periodic volumes (like found in turbomachinery).

I: vol	Volume index
I: nrep	Number of times to apply the matrix
D: rmatrix	The replication matrix
I: icode	Return code

F.8 SetReplicate – Set the Volume's replication data

icode = gi_iSetReplicate(int vol, nrep, double *dmatrix)
ICODE = IG_ISETREPLICATE(VOL, NREP, RMATRIX)

This routine specifies the replication data associated with the specified volume. The replication matrix is a column-major matrix that is 4 columns by 3 rows and declared in C as [3][4] and in FORTRAN as (4,3). nrep set to zero turns off all replication.

I: vol	Volume index
I: nrep	Number of times to apply the matrix
D: rmatrix	The replication matrix
I: icode	Return code

F.9 NewHandle – Create new Interpolation Handle

icode = gi_iNewHandle(int vols, bounds, vold, boundd, *handle)
ICODE = IG_INEWHANDLE(VOLS, BOUNDS, VOLD, BOUNDD, HANDLE)

This routine creates a new interploation Handle and fills the associated internal data. If a Handle aready exists (that maps the source Boundary to the destination Boundary), that existing Handle is returned.

I: vols	Source Volume index
I: bounds	Source Boundary index
I: vold	Destination Volume index
I: boundd	Destination Boundary index
I: handle	Handle index (returned)
I: icode	Return code

F.10 GetHandle – Gets the Interpolation Handle data

icode = gi_iGetHandle(int handle, *vols, *bounds, *vold, *boundd, *nbnoded, **intpf, double **dist)

ICODE = IG_IGETHANDLE(HANDLE, VOLS, BOUNDS, VOLD, BOUNDD, NBNODED, INTPF, DIST)

This routine creates a new interploation Handle and fills the associated internal data. If a Handle aready exists that maps the source Boundary to the destination Boundary, that existing Handle is returned.

I: handle	Handle index
I: vols	Source Volume index (returned)
I: bounds	Source Boundary index (returned)
I: vold	Destination Volume index (returned)
I: boundd	Destination Boundary index (returned)
I: nbnoded	Number of destination Boundary Nodes (returned)
I: intpf	pointer to interpolation flags (1 - extrapolated, 0 - interpolated, -1 - not vaild) – length is nbnoded (returned)
D: dist	pointer to distance from surface – length is nbnoded (returned)
I: icode	Return code

FORTRAN note: The pointer is not returned. NBNODED must be set with the size of INTPF and DIST at the call. It is returned with the actual length used. If the length is not large enough, then the return code CAPRI_OVERFLOW is set but all the data up to the declared length is correct.

F.11 DelHandle – Remove an Interpolation Handle

icode = gi_iDelHandle(handle)

ICODE = IG_IDELHANDLE(HANDLE)

This routine deletes an interploation Handle.

I: handle	Handle index
I: icode	Return code

F.12 InterAttach – Interpolate to Produce/Update Boundary Attachment

icode = gi_iInterAttach(int handle, char *name, *named, *updated)
ICODE = IG_IINTERATTACH(HANDLE, NAME, NAMED, UPDATED)
This routine interpolates the source attachment onto the discretization for the destination boundary as defined for the Handle. A new attachment is created if NAMED does not already exist, otherwise the data is replaced. Any rank 3 Attachments have the displacement and replication matrices applied (just like the coordinates).

I: handle	Handle index
C: name	Attachment name – source
C: named	Attachment name – destination
C: updated	The update character string – destination
I: icode	Return code

G Tag Routines

G.1 GetNumVolume – Returns the number of Volume Tags

icode = gi_tGetNumVolume(int vol, *num)
ICODE = IG_TGETNUMVOLUME(VOL, NUM)
This routine returns the number of Tags for the Volume.

I: vol	Volume index
I: num	Number of Tags associated with this Volume
I: icode	Return code

G.2 GetVolume – Gets the Volume Tag

icode = gi_tGetVolume(int vol, char *tag, **val)
ICODE = IG_TGETVOLUME(VOL, TAG, VAL)
This routine returns the string associated with the Volume Tag.

I: vol	Volume index
C: tag	The Tag string
C: val	The associated string
I: icode	Return code

G.3 GetIVolume – Gets the Volume Tag by index

icode = gi_tGetIVolume(int vol, index, char **tag, **val)
ICODE = IG_TGETIVOLUME(VOL, INDEX, TAG, VAL)
This routine returns the string associated with the index for the Volume Tag.

I: vol	Volume index
I: index	Tag index – range 1 to the number of Tags.
C: tag	The Tag string
C: val	The associated string
I: icode	Return code

G.4 SetVolume – Sets the Volume Tag

icode = gi_tSetVolume(int vol, char *tag, *val)
ICODE = IG_TSETVOLUME(VOL, TAG, VAL)
This routine sets the string associated with the Volume Tag. If the Tag exists the new string is applied.

I: vol	Volume index
C: tag	The Tag string
C: val	The associated string – A NULL value deletes the Tag.
I: icode	Return code

G.5 GetNumBoundary – Returns the number of Boundary Tags

icode = gi_tGetNumBoundary(int vol, bound, *num)
ICODE = IG_TGETNUMBOUNDARY(VOL, BOUND, NUM)
This routine returns the number of Tags for the Boundary.

I: vol	Volume index
I: bound	Boundary index
I: num	Number of Tags associated with this Boundary
I: icode	Return code

G.6 GetBoundary – Returns the Boundary Tag

icode = gi_tGetBoundary(int vol, bound, char *tag, **val)
ICODE = IG_TGETBOUNDARY(VOL, BOUND, TAG, VAL)
This routine returns the string associated with the Boundary Tag.

I: vol	Volume index
I: bound	Boundary index
C: tag	The Tag string
C: val	The associated string
I: icode	Return code

G.7 GetIBoundary – Gets the Boundary Tag by index

icode = gi_tGetIBoundary(int vol, bound, index, char **tag, **val)
ICODE = IG_TGETIBOUNDARY(VOL, BOUND, INDEX, TAG, VAL)
This routine returns the string associated with the index for the Boundary Tag.

I: vol	Volume index
I: bound	Boundary index
I: index	Tag index – range 1 to the number of Tags.
C: tag	The Tag string
C: val	The associated string
I: icode	Return code

G.8 SetBoundary – Sets the Boundary Tag

icode = gi_tSetBoundary(int vol, bound, char *tag, *val)
ICODE = IG_TSETBOUNDARY(VOL, BOUND, TAG, VAL)
This routine sets the string associated with the Boundary Tag. If the Tag exists the new string is applied.

I: vol	Volume index
I: bound	Boundary index
C: tag	The Tag string
C: val	The associated string – A NULL value deletes the Tag.
I: icode	Return code

H Return Codes

-13 - CAPRI_BADHANDLE

-12 - CAPRI_NOTFOUND

-11 - CAPRI_NODISCRET

-10 - CAPRI_OVERFLOW

-9 - CAPRI_INUSE

-8 - CAPRI_RANGERR

-7 - CAPRI_MODELERR

-6 - CAPRI_NOLOAD

-5 - CAPRI_INDEX

-4 - CAPRI_UNSUPPORT

-3 - CAPRI_MALLOC

-2 - CAPRI_ALREADYON

-1 - CAPRI_NOINIT

0 - CAPRI_SUCCESS

1 - CAPRI_OUTSIDE - Not an error

Computational Analysis PRogramming Interface

Robert Haimes
Massachusetts Institute of Technology
haimes@orville.mit.edu

and

Gregory J. Follen
NASA Lewis Research Center
Gregory.J.Follen@lerc.nasa.gov

Abstract

CAPRI is a CAD-vendor neutral application programming interface designed for the construction of analysis suites and design systems. By allowing access to the geometry from within all modules (grid generators, solvers and post-processors) such tasks as meshing on the actual surfaces, node enrichment by solvers and defining which mesh faces are boundaries (for the solver and visualization system) become simpler. The overall reliance on file 'standards' is minimized.

This 'Geometry Centric' approach makes multi-disciplinary analysis codes much easier to build. By using the shared (coupled) surface as the foundation, CAPRI provides a single call to interpolate grid-node based data from the surface discretization in one volume to another. Finally, design systems are possible where the results can be brought back into the CAD system (and therefore manufactured) because all geometry construction and modification are performed using the CAD system's geometry kernel.

1 Introduction

NASA Lewis Research Center's Numerical Propulsion System Simulation (NPSS) [1] is a program focused on reducing the time and cost in developing aero-propulsion engines. This is done by addressing the multi-disciplinary nature of this problem early in the design process and by applying new computer hardware and software techniques as part of this process. NPSS has a vision: To establish an interdisciplinary "Numerical Test Cell" for propulsion systems which improves quality, and reduces development time and cost.

NPSS accomplishes this vision by beginning from an engine system view that integrates multiple disciplines such as aerodynamics, structures and heat transfer with computing and communication technologies to capture

complex physical processes in a timely and cost effective manner. In order to conduct complex multi-discipline simulations overnight, solving the problem of providing common geometry representations or common access across all pieces of the simulation or design space is required. There are other obstacles that need to be overcome to fully implement an NPSS Environment but a common geometric model has such a great potential to save time in design, improve accuracy and consistency from concept of the engine through manufacturing that it deserves special attention. NPSS has conservatively estimated that a common geometry model could save 33% in development time in building new engines to the Aeropropulsion Industry.

As in the past, NPSS has built it's environment through a series of ever increasingly complex prototypes. Therefore, before attempting to tackle the multi-disciplinary problem, it is important to look at a single discipline in detail to attempt to understand the problems in automation first. The computational steps traditionally taken for Computational Fluid Dynamics (CFD), Structural Analysis, and other simulation disciplines (or when these are used in design) are:

- Surface Generation

The surfaces of the object(s) are generated usually from a CAD system. This creates the starting point for the analysis and is what is used for manufacturing.

- Grid Generation

These surfaces are used (with possibly a bounded outer domain) to create the volume of interest. Usually for the analysis of external aerodynamics, the aircraft is surrounded by a domain that extends many body lengths away from the surfaces.

- Flow Solver or Simulation

The solver takes as input the grid generated by the second step (and information about how to apply conditions at the bounds of the domain). For fluids, the solver usually simulates either the Euler or Navier-Stokes equations in an iterative manner, storing the results either at the nodes in the mesh or in the element centers. The output of the solver is a file that contains the solution.

- Post-processing Visualization

After the solution procedure has successfully completed, the output from the grid generator and the simulation are displayed and examined in a graphical manner.

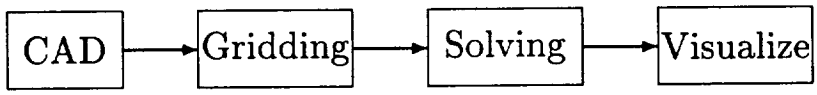

Figure 1: The Traditional Computational Analysis Suite

These steps have worked well in the past for simple steady-state simulations at the expense of much user interaction. The data was transmitted

between phases via files (the arrows in Figure 1). In most cases, the output from a CAD system could go to IGES files. The output from Grid Generators and solvers do not really have standards though there are a couple of file formats that can be used for a subset of the problem space (i.e. PLOT3D data formats for CFD). The user would have to patch up the data or translate from one format to another to move to the next step. Sometimes this could take days.

The disciplines of Structural and Mechanical Analysis are now more closely coupled than CFD. This is for two reasons; (1) the grid generation is simpler (fewer gridding schemes are employed) and the resultant meshes tend to be much smaller, and (2) the market-place is much larger. This has resulted, for most commercial packages, in the analysis being run from within the CAD system.

If one is considering CFD or a multi-disciplinary analysis that includes CFD or another physical system (not supported through the CAD system) then some other mode of execution is required. This, historically, has been prone to problems in automation.

2 CAPRI

Instead of the serial approach to analysis as described above, CAPRI uses a geometry centric approach. This makes the actual geometry (not just a discretized version) accessible to all phases of the analysis. The connection to the geometry is made through an Application Programming Interface (API) and NOT a file system. In fact, CAPRI is defined from two components; (1) A definition of the solid part (geometry and topology) and (2) the API that allows access to the part and other data pertinent to the analysis suite.

The CAPRI API isolates the top level applications (grid generators, solvers and visualization components) from the geometry kernel. This allows the replacement of one geometry kernel with another, without effecting the top level applications. For example, if Pro/ENGINEER is used as the CAD package then Pro/TOOLKIT can be used for all geometric queries so that no solid geometry information is lost in a translation. See Figure 2.

This figure depicts a simple analysis suite. A much more complex system can be put together by adding modules as would be required for multi-disciplinary applications.

2.1 Geometry and Topology

To insure that the resulting interface is not overly complex, it is crucial that the geometry description be uncomplicated (but not too simple as to impair functionality). Most systems that deal with CAD data make the distinction between geometry (points, curves and surfaces) and topology (the hierarchical connections between geometric entities). CAPRI mixes these in a simple data definition. The geometry and topology are defined in CAPRI in the following manner:

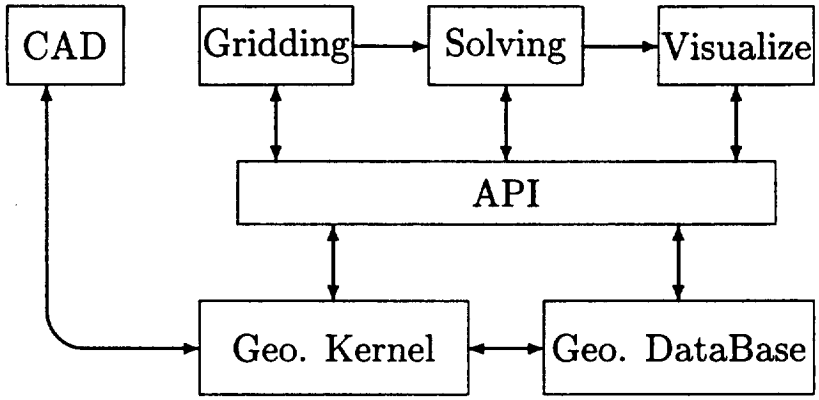

Figure 2: The CAPRI based Computational Analysis Suite

- Nodes

These are the simplest entities and are just points in 3 space.

- Edges

Edges are curves. Each Edge is bounded by two unique Nodes. The Edge is parameterized with t, where the first Node has a value at t_{min} and the second bounding Node has a value of t_{max}. The value of t_{min} is always less than t_{max}.

- Faces

Faces are parameterized (u, v) surfaces. The parameter range for u is u_{min} to u_{max} and v ranges from v_{min} to v_{max}, but the relationship between (u, v) and the bounding Edges is not as simple as the Edge-Node definitions. This is because Faces may be bounded by more than 4 Edges.

The bounds of the Face are defined by closed set(s) of Edges. There may be one or more of these loops for each Face. Stored with each defining Edge is an orientation so that it is known whether to look at the Edge as specified or in the opposite sense. The loop is an ordered suite that defines the orientation of the Face. The outer loop(s), specify the boundary of the surface, and traverse the Face in a right-handed manner – defining the outward pointing normal (out of the volume). Any holes are specified by a left-handed traversal of Edges. See Figure 3.

Each Edge can be found bounding two Faces, one in the forward and one in the opposite sense.

Again, to aid in plotting and to have a complete representation of this (possibly complex) Face, there is an attached discretization (tesselation). This is defined as suite of disjoint triangles of a specified length. Each triangle is right-handed with the normal pointing out of the volume.

- Boundaries

Boundaries are simply collections of one or more Faces. These entities are the connection between the geometry and the rest of the analysis suite,

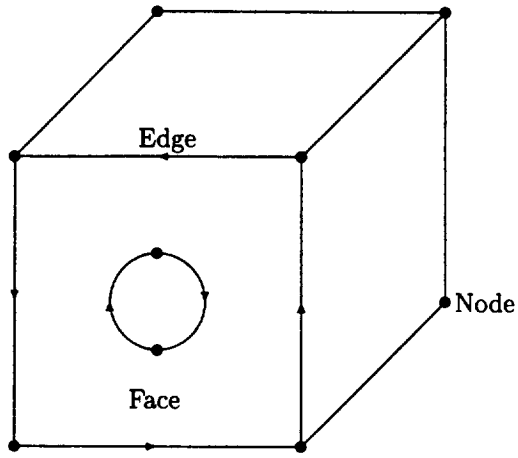

Figure 3: Simple Volume with cylinder cutout

as described above. The Faces need not couple together (i.e., a periodic boundary upstream and downstream from a turbine or compressor blade) but are used to insure that the grid generation knows that these surfaces could be treated in special ways. And, the solver knows which boundary condition to apply to what section of the resultant mesh.

Boundaries have an associated name (i.e., far-field, body, wing and etc).

- Volumes

Volumes are completely closed regions of 3 space. Volumes are bounded by the sum of all of the Faces found in the Boundaries. These Faces match up at the shared Edges, that terminate at the Nodes. CAPRI can handle one or more Volumes at a time. Each Volume can be named with strings like; 'Fluids passage', 'Blade', and etc.

2.2 Volumes

All coordinates for a Volume reported back through CAPRI are in the CADs native coordinate system, scale and units for that individual component part.

- Replication

There is a Replication matrix and count associated with each Volume. The count refers to the number of times to apply the matrix to produce the complete representation of the object. This matrix is used to multiply all Volume coordinates in order to produce additional instances. This is only referenced within CAPRI for the interpolation functions.

Mirroring can be considered a simple form of this type of replication, where the count is one and the matrix is all zero except for the diagonal. The diagonal will contain 1.0 for 2 of the 3 components and -1.0 for the other.

- Displacement

There is a Displacement matrix associated with each Volume. The matrix is used to multiply all Volume coordinates in order to specify movement of the Volume. This allows the support of any combination of translation, scaling and rotation. This may be used for transient problems where Volumes move by other Volumes. For the interpolation routines, Replication is applied before Displacement.

- Volume Tags

Tags are character strings associated with the Volume. Each Tag has an attached value string. These Tag entities are useful for specifying conditions or material information for the entire Volume. For example; the Volume may have a Tag 'gamma' with the associated value '1.4'.

2.3 Boundaries

Boundaries are the pivotal data objects used within CAPRI. Boundaries are the entities that the grid generators should build the exposed parts of the mesh about. Different solver functions (boundary conditions) are then applied across these facets of the volume. When Multi-disciplinary analysis are run, boundaries are where these different physical models share information to drive the coupled solution.

The data that comes from CAD systems does not always provide a proper separation of surfaces (Edges, as specified above) that coincide with what is required by the analysis suite. This is for two reasons; (1) the CAD operator, by the order of construction, may produce artifacts (such as sliver surfaces) or detail at a level more complex than the analysis suite requires. (2) Curved surfaces such as fillets have breaks, on where these surfaces mate with other surfaces, usually not at the center of curvature where the analysis suite would require the edge of the boundary.

The first of these problems is resolved in CAPRI by allowing the collection of CAD surfaces. The analysis suite can query this collection and get to the detailed CAD surfaces if required. This has the advantage over what is done in automated techniques used for grid generation in that the CAD artifacts can be meshed through as opposed to becoming features in the grid. For example, a sliver surface would end up completely resolved, in an automated surface gridding procedure, requiring potentially large numbers of small cells in those regions.

Scribing and splitting CAD surfaces so that the analysis boundaries can be defined is a function of CAPRI. Initially this is done interactively or through program control (if the analysis suite can determine where to break the surfaces). In the future, work will be done to attempt to automate this procedure.

- Boundary Discretization

Each Boundary can have an attached discretization. This discretization can be from different mesh topologies that touch the Boundary. There are 3 types of cell faces that build this structure (supported via Boundary nodes

– *bnodes*): (1) Disjoint Triangles – 3 *bnodes* per entity, (2) Disjoint Quadrangles – 4 *bnodes* per entity and (3) Quad-Meshes – these are produced from grid 'planes' of structured blocks.

- Special Groupings

Special groupings are simply lists of *bnodes* that may be required by the solver's boundary condition routines. This is to flag "special" nodes. For example, if *IBlanking* is used, there could be a list that contains the *IBlanked* nodes.

- Boundary Attachments

Boundary attachments are collections of data that are associated with the *bnodes* of the Boundary discretization. The attachments are identified by a name and can have an additional string that can indicate information on how and/or when the attachment was created. These attachments can be used to communicate boundary level data between modules (i.e., heat transfer to the visualization module), perform *zooming* or otherwise couple like simulations at boundaries and perform multi-disciplinary coupling between Volumes.

- Boundary Tags

Tags are character strings associated with the Boundary. Each string has an attached value string. These Tag entities are useful for specifying conditions or material information for the application of boundary conditions by the solver. For example; the Boundary named 'Wall' may have a Tag 'temperature' with the associated value '300K'.

2.4 Interpolation Handles

An integer *hook* is used in CAPRI to simplify the specification of Boundary to Boundary interpolation required by single and multi-disciplinary coupling. This Handle is an index to internal storage that contains information such as the indices and weights used in the source *bnode* space to create Attachments to the destination Boundary discretization.

When a new Handle is created the current state of the Displacement and Replication data for the destination Volume is used as well as the Displacement matrix for the source when calculating the interpolation. Therefore, for transient simulations, the Handles associated with Volumes moving (with respect to the coupled volume) must be deleted and recreated every iteration.

2.5 The CAPRI API

The CAPRI API is sub-divided into the following components:

1. Utility routines.
2. Geometry data-base queries.
3. Point queries.
4. Calculated or geometrically derived queries.
5. Boundary data routines.
6. Geometry based interpolation routines.
7. Tag based routines.
8. Geometric modification.

3 The CAPRI Top-level Modules

CAPRI does not specify either the use or control of the suite. The design system or analysis software can be different programs controlled at some higher level (such as command line language scripts or another code). The software could also be built as a single integrated application. For example, an automated optimizer could be used to drive the entire suite, and in this case, it would obviously be in control. Software control could also be specified by some visually based GUI that allows plugging the modules together as envisioned by NPSS.

For this to work, and for general plug and play, there needs to be agreement on how input parameters get passed through the system(s). Within CAPRI, the Tag and Attachment concepts are used for specifying this data. Tags place simple (or integrated quantities) associated with the Volume or Boundary of interest into CAPRI so that other modules have direct access. Complex specification of input can be achieved by mapping quantities to Boundaries via Attachments.

It is the responsibility of the controlling software to fill the required Tags and/or Attachments for the target module (or cause them to be filled by executing some other module in the system) before initiating execution.

3.1 Grid Generator

The Grid Generator task is the most straight forward. It uses CAPRI to query the geometry and topology of the part of interest and then performs the meshing. Once complete, it specifies the Boundary discretization for the Volume so that the other modules can communicate their data on the Boundary.

A CAPRI interface was added to the grid generation code (APG) used in average passage turbomachinery flow analysis. This is a difficult test because this analysis requires that spanwise grid surfaces lie on surfaces of revolution, leading to many surface-surface intersections. Previously, the APG code was limited to using a single parameratized surface to define the blade, which is often not easily obtained from a CAD model. CAPRI provides an alternative approach, using two basic features. First, CAPRI provides a discretized version of the surfaces in the form of a tessellation. Second, CAPRI allows *snapping* points on to a surface from a given location. A new routine was written for APG which calculates the intersection of a surface of revolution with the blade tessellation. Points calculated on this intersection are then brought to the surface using the closest point calculation.

Figure 4 shows a generic compressor grid produced from a Parasolid (UniGraphics) model using APG and CAPRI.

3.2 Solver

The simulation software gets the input data from CAPRI via Tags and Attachments, but will need to read the cloud of data (both mesh point

4 A Coupling Example – Zooming

The task of coupling a 2D axisymmetric fluids code to a full 3D CFD simulation for turbomachinery would, in the past, require special programming. This would produce a *one-off* application and the coding would have to be done again with a different component (if different modules were required). CAPRI simplifies this problem by providing the interpolation infrastructure. A proper representation of both Volumes is required (with the shared Boundary) because CAPRI, and all geometric representations within CAPRI, is assumed to be 3D.

The first step for this example would be to define the Replication information for each Volume. This allows the use of a single passage to represent the entire wheel and informs the CAPRI interpolation routines how to perform the mating when the interfaces do not match exactly.

If the information is to be passed from the 2D axisymmetric simulation to the 3D, then the axisymmetric code needs to produce its 3D discretization on to the Boundary. This is simply done by revolving the data about the arch producing a radial structured mesh. The data on the mesh points (in the circumferencial direction) is duplicated – axisymmetricly. The 3D code can then get data on the Boundary by a single call that provides an interpolated Attachment. Scalar, vector or state-vector quantities are supported.

If the information is to be passed from the 3D solution to the axisymmetric simulation, the 2D code gets the Attachment desired by interpolation on to the constructed Boundary. Averaging then needs to be applied so that the coupled values can be used within the axisymmetric context.

5 Conclusion

CAPRI offers a solution to this real world dilemma which is comprised of multi-vendor, differing need companies that require access to geometry which preserves all the inherent features of that part while allowing each discipline the ability to register its effect.

Acknowledgments

This work was partially sponsored by NASA Lewis Research Center under grant NAG3-2019. Additional support for this research was obtained from the IBM SUR Project and the IBM UUP Project.

References

[1] A. L. Evans, J. Lytle, G. Follen and I. Lopez. An Integrated Computing and Interdisciplinary Systems Approach to Aeropropulsion Simulation. AMSE IGTI, Orlando, FL, 1997.

[2] R. Haimes and M. Giles. Visual3: Interactive Unsteady Unstructured 3D Visualization. AIAA Paper 91-0794, 1991.

[3] R. Haimes and D. Edwards. Visualization in a Parallel Processing Environment. AIAA Paper 97-0348, 1997.