AFRL-RQ-WP-TR-2019-0214

COMPUTATIONAL AIRCRAFT PROTOTYPE SYNTHESIS (CAPS)

Robert Haimes and Marshall C. Galbraith

Massachusetts Institute of Technology

John F. Dannenhoffer, III

Syracuse University

David L. Marcum

Mississippi State University

Steve Karman

Pointwise, Inc.

NOVEMBER 2019
Final Report

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
AEROSPACE SYSTEMS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the USAF 88th Air Base Wing (88 ABW) Public Affairs Office (PAO) and is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RQ-WP-TR-2019-0214 has been reviewed and is approved for publication in accordance with assigned distribution statement.

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE *(DD-MM-YY)* November 2019	**2. REPORT TYPE** Final	**3. DATES COVERED *(From - To)*** 8 October 2014 – 7 November 2019

4. TITLE AND SUBTITLE COMPUTATIONAL AIRCRAFT PROTOTYPE SYNTHESIS (CAPS)	**5a. CONTRACT NUMBER** FA8650-14-C-2472
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62201F
6. AUTHOR(S) Robert Haimes and Marshall C. Galbraith (Massachusetts Institute of Technology) John F. Dannenhoffer, III (Syracuse University) David L. Marcum (Mississippi State University) Steve Karman (Pointwise, Inc.)	**5d. PROJECT NUMBER** 2401
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER Q18C
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139 Mississippi State U, 218 Carpenter, Mississippi State, MS 39762 Syracuse University, 263 Link Hall, Syracuse, NY 13244 Pointwise, Inc., 213 Jennings, Fort Worth, TX 76104	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Aerospace Systems Directorate Wright-Patterson Air Force Base, OH 45433-7542 Air Force Materiel Command United States Air Force	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/RQVC
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-RQ-WP-TR-2019-0214

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES
PA Clearance Number: 88ABW-2020-0833; Clearance Date: 1 Mar 2020.

14. ABSTRACT
The objective of this effort is to establish a computational geometry, meshing and analysis model generation tool that can be used across AFRL/RQV. This common tool will enable collaboration between conceptual design, multidisciplinary optimization and high-fidelity simulation efforts.

15. SUBJECT TERMS
multidisciplinary analysis, multi-fidelity geometry, aircraft design, software infrastructure

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dean E. Bryson
Unclassified	Unclassified	Unclassified	SAR	143	**19b. TELEPHONE NUMBER** (Include Area Code) (312) 713-7137

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

TABLE OF CONTENTS

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

LIST OF FIGURES

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

LIST OF TABLES

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AIM	Analysis Interface Module – a CAPS plugin that supports a particular solver and/or mesh generation subsystem
AFLR	Advancing-Front/Local-Reconnection – a series of unstructured mesh generation software subsystems
API	Application Programming Interface – a software library
BEM	Built-up Element Model – a structural analysis view of geometry as non-manifold *sheets*
BRep	Boundary Representation – object hierarchy that holds on to both geometry and topology, see Appendix A
BSpline	B-Spline, or Basis Spline, is a spline function that has minimal support with respect to a given degree, smoothness, and domain partition – used to represent complex shapes in CAD systems
CAD	Computer Assisted Design – a software system that builds geometry
CAPS	Computational Aircraft Prototype Syntheses
Edge	BRep term for a bounded curve (note the capitalization) – see Appendix A
EGADS	The Electronic Geometry Aircraft Design System
ESP	The Engineering SketchPad – a CAD-like system that underpins CAPS
GLOVES	Graphical Layout Of VEhicle Systems
Face	BRep term for a trimmed surface (note the capitalization) – see Appendix A
flend	Fillet-like blend operation
FPC	Full Potential Code
GUI	Graphical User Interface
HSM	Hypergeometric Shell Model
IBL	Integral Boundary Layer
IGES	Initial Graphics Exchange Specification – an archaic file standard for the transmission of geometric data
Jenkins	Software testing harness used for ESP/CAPS
LSM	Lumped Shell Model
MDO	Multidisciplinary Optimization
Node	BRep term for a point in space (note the capitalization) – see Appendix A
OML	Outer Mold Line
OpenCSM	Open-source Constructive Solid Modeler
PLUGS	Parametric Legacy Unstructured Geometry System
pyCAPS	The Python interface to Computational Aircraft Prototype Syntheses software
SBO	Solid Boolean Operator – intersection, subtraction or union
SLUGS	Static Legacy Unstructured Geometry System
STEP	Standard for the Exchange of Product data – a file format for the transmission of BRep data
UDC	An ESP User Defined Component – a script that acts like a macro or subroutine
UDF	An ESP User Defined Function – a complex plugin that provides a parametric sub-build of a BRep based on input BReps
UDP	An ESP User Defined Primitive – a plugin that provides a parametric sub-build of a BRep

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

1 SUMMARY

In the design of real configurations, such as aerospace vehicles, geometry should play a central role. For a given shape, the number of unique geometry models is almost as great as the number of disciplinary simulation models. Each simulation model will usually read certain geometry and mesh formats or have other requirements peculiar to it. However, no matter the geometry/mesh format or requirements, it must be based on realizable and consistent geometric object(s). This fact allows for all geometry and mesh requirements to originate from a single common parametric description.

Beyond the differences caused by disciplinary analyses, there are also the differences created between analysis and manufacturing. When analyzing (or designing/optimizing) some physical object that will ultimately be manufactured, it is common practice to create an additional model beyond those generated for the simulation design tools. This is a fully realizable 3D representation in a CAD or CAD-like system. Generally great care must be taken to ensure that the design and manufacturing representations are close enough to each other so that what is built is the same as what was designed. This care requires a large amount of time (and human intervention), making automation of the process extremely difficult, if not impossible, especially within a Multi-Disciplinary Analysis and Optimization (MDAO) environment.

The most common method for transferring geometry amongst the various analyses is via file standards. The first commonly used standard was the IGES file format which contains data that is defined as disjoint and unconnected surfaces and curves; that is, it only contains geometry with no notion of topology. Topology, in this context, is the hierarchy and connectivity of the various geometric elements. Since 3D meshing software ultimately requires a closed watertight model, much effort is therefore needed to take the geometric data, trim the curves and surfaces, and then deduce the topology. STEP, a more complete file standard, supports the transmittal of topology as well as geometry so that a Boundary Representation (BRep) can be built. This is the preferable file type to hold geometric data. Surprisingly, this format is seldom used, probably due to the fact that constructing a STEP reader is complex and it requires a complete solid-modeling geometry kernel to deal with the data.

A larger problem with both IGES and STEP formats is the fact that they are static (non-parametric) geometry models. The implication of this is that one can only perform physics-based analyses on that particular geometry, with no ability to modify it or perform trade studies. Also, without being driven by Design Parameters, it is impossible to determine the sensitivities of the results of a physics-based simulation with respect to the Design Parameters; the latter is key to generating optimal designs.

This final report for the Computational Aircraft Prototype Syntheses (**CAPS**) project discusses the tasks performed and in some cases the status (if not completed). But unlike many research projects, this report and the associated papers (see the Bibliography) are not the output of this entire effort. The most tangible and significant result of this work is open source software that robustly performs the overall underpinning for a holistic and integrated system that can perform *Design through Analysis*. **CAPS** along with the rest of the **ESP** software can be found and downloaded at: http://acdl.mit.edu/ESP.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

2 INTRODUCTION

2.1 The Engineering SketchPad (ESP) – OpenCSM and EGADS

Currently, most organizations have found it difficult to bridge the gap between conceptual design, where the geometry may be of low fidelity, and fully realizable 3D representations. To alleviate this problem and those associated with transmitting geometry via file standards, geometry kernel APIs that couple directly with the source of the geometry can be utilized. One clear advantage to this approach is that the geometry never needs to be translated and hence remains simpler and closed to within the modeler's tolerance. Also, a geometry system that can be used both at the conceptual level and throughout preliminary/detailed design has obvious advantages.

File standards and kernel APIs are for dealing with a static configuration once it has been defined; such a view is sufficient for analysis. But for design, the ability to deal with the process by which the configuration is defined and built is paramount. In parametric CAD systems, the configuration definition is done through a master-model that consists of both a build recipe (called the *feature tree*) and a set of Design Parameters. This recipe (where the *design intent* is realized) must be made available to the MDAO process since it defines the design space and informs how to build and optimize the configuration. Most CAD systems hold this information in proprietary file formats that cannot easily be read or modified by outside programs.

Foundational work has been accomplished in developing an integrated software suite that solves the issues discussed above. The resulting capability provides the tools to generate various representations of a design (either multi-fidelity or multi-disciplinary, or both) from a single master model. A user accesses this software through a web browser, and this complete suite is referred to as the Engineering Sketch Pad (**ESP**), which is a fully parametric, attributed, feature-based solid-modeling system [1]. The output of ESP is geometry in the form of one or more BReps (see Appendix A).

ESP is built both upon the WebViewer (which is a WebGL-based visualizer for three-dimensional configurations and data) and upon **OpenCSM** [2], which is a constructive solid modeler that is itself built upon **EGADS** [3] and OpenCASCADE. There is no absolute requirement for **ESP**'s dependency on OpenCASCADE; rather this CAD kernel is chosen because it is open-source and can be distributed freely with **ESP**. In fact, all of this software is open-source and available without any licensing restrictions.

2.2 CAPS

It is not an easy task to build a tightly-integrated software system that contains many access points, needs to be able to be user-driven, and fundamentally improves upon the multi-fidelity and multi-disciplinary design process. This is accomplished in **CAPS** (which is **ESP**'s formal connection to various Computational Engineering Analysis suites) by attacking the process-related bottlenecks head-on [4]. For example, when performing a vortex lattice aerodynamic analysis of a wing with the geometric description of an Outer Mold Line (OML), the wing needs to be deflated to a single surface. Typically, this requires difficult, possibly error-prone, user-intensive reverse engineering and may provide situations that are, at best, ambiguous. The strategy taken here is to forward engineer the process, where, in this case, the mid-surface aerodynamic shape is generated directly.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Component or sub-component models can be generated as either compiled-language plug-ins or as scripts that build geometry. **CAPS** allows user and programmatic access (through a high-level API or Python interface) to:

- change a Design Parameter value (or values) and regenerate the geometry;
- annotate the geometry through attribution;
- get geometric sensitivities with respect to the Design Parameters;
- generate geometry at a fidelity commensurate with the analysis to be used;
- mesh (or setup the input for meshing) the geometry, specifically for the analysis at-hand; and
- setup for the execution of the specific analysis code.

A software block diagram for **CAPS** in the **ESP** environment can be seen in Figure 1. An important part of **CAPS**' flexibility in dealing with various analysis codes are the AIMs (Analysis Interface Modules). This plug-in technology leaves the overall framework alone and allows for run-time connections. The geometry passed to the plug-in is specified on the BRep (of appropriate fidelity) by the use of attribution at build. Any inputs (not associated with the BRep) as well as other BRep attributes may also be required depending of the analysis at-hand. The following functions are a part of any AIM plug-in:

- Attribute/Input Checking: this AIM function is invoked before any mesh/input file generation to ensure that all of the required data can be found.
- Meshing: the input BRep and/or tessellation are used to either perform the meshing directly (if possible or the mesh system has an API) or to provide input to a grid generator. Note that the mesh vertices that sit on geometry (as described in the input BRep) need to be associated back to the geometry. This is important for generating parametric sensitivities [5] and performing data fitting [6] (straight forward interpolation or conservative data transfer). Most stand-alone grid generation systems maintain this data internally but do not make it available as output. Any attempt to re-associate this data by inverse evaluations is slow and not robust.
- Analysis Input Generation (Pre-Execution): the input values and attributes found on the geometry are used to construct and output the input file(s) required to run the analysis. If the analysis suite has its own API, then the API can be used directly to avoid the writing and subsequent reading of files. This means that the pre-execution portion of the AIM also performs the analysis execution function, otherwise the **CAPS** user/programmer/MDO framework is responsible for running the solver.
- Post Execution: **CAPS** is informed that the analysis code has successfully run.
- Output Parsing: this is required to get performance data, displacements, pressures or other information required to be used as input to another analysis module or to inform the optimizer of the objective functional value(s). Again, this would involve file reading unless the analysis system has an API that can be used to retrieve the output data directly.
- Data Transfer Functions [6]: a function that computes interpolation within a surface element is required in order to perform the interdisciplinary coupling in an interpolation setting. If the option for conservation is chosen then the interpolation function must be augmented with one that performs integration of quantities over an element (also their backward or dual variants are required for efficient computation of the transfer).

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

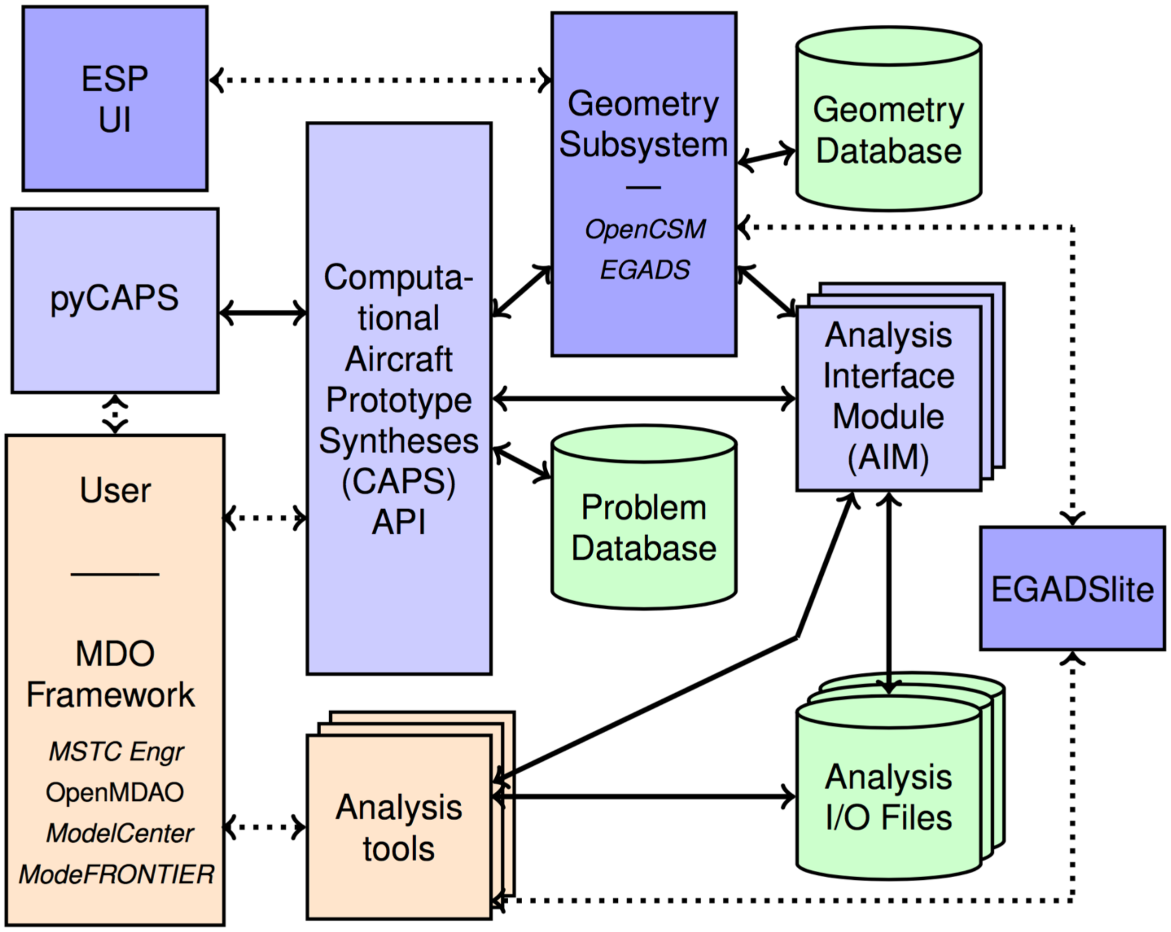


Figure 1. CAPS (in the ESP environment) block diagram.

The block diagram seen in Figure 1 has not changed since the inception of **CAPS** except for these 2 items:

- EGADSlite – This is a subset of the **EGADS** API and entirely free of the OpenCASCADE dependency. The portion of the API supported are those functions useful for grid generation and mesh adaptation and this subset is designed specifically to be placed in High Performance Compute (HPC) environments [7]. EGADSlite contains the functions that allow for parsing the Topology of a geometric model, performing geometric evaluations and inverse evaluations and the in/out predicates – basically everything in **EGADS** except for geometry construction. This work has been funded by NASA.
- pyCAPS – The original **CAPS** proposal assumed that the software would be accessed via compiled applications or would be plugged into an MDO framework (where someone would use the **CAPS** API in C/C++ to make the programming connections). It quickly became apparent that this would limit **CAPS**' access and usefulness. An easier access approach was needed. The decision was made by AFRL personnel that there needed to be a Python connection. pyCAPS [8] was initially written by Ryan Durscher AFRL/RQVC to mirror the C/C++ API but also be appropriate for Python (**CAPS** was *Pythonized*). This has become an integral part of the **ESP**/**CAPS** software suite and has allowed for improved and simplified

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

testing; plus, it gives us the ability to train people in the use of **CAPS** without requiring programming or being dependent on an MDO framework.

2.3 AIMs

The open source suite of **CAPS** AIMs is fairly complete reflecting various fidelities of both Computational Fluid Dynamics (CFD) and Structural Analysis. Also, because of the non-viral open source license and the plug-in nature of the AIMs, AFRL personnel has developed a number of internal AIMs not distributed outside of AFRL and which are not included in the external software releases.

The following is a list of AIMs currently in the **ESP**/**CAPS** distribution:

AFLR2 – A 2D triangle mesher for 2D CFD applications written by Prof. Dave Marcum of Mississippi State University. Used for running SU^2 in 2D mode.

AFLR3 – This is the 3D tetrahedral volume mesher written by Prof. Dave Marcum of Mississippi State University. AFLR3 is an unstructured grid generator that is used by many in the CFD community and by many in the DoD and can generate meshes suitable for Reynolds-Averaged Navier-Stokes (RANS) simulations.

AFLR4 – This 3D surface triangulator (also from Prof. Marcum) can be used instead of the **EGADS** tessellator to prepare surfaces for volume meshing (AFLR3 or TetGen). Its output is an **EGADS** Tessellation Object, which is a container for the triangulation (elements and vertices). The geometric *ownership* of each vertex is also maintained with the associated geometric parameters.

ASTROS – The Automated STRructural Optimization System (ASTROS) is a comprehensive software suite for the multidisciplinary design and analysis of aerospace structures. ASTROS combines optimization algorithms with structural finite element analysis (FEA) disciplines such as statics, dynamics, and aeroelasticity to perform automated design of structures. ASTROS supports both the preliminary design stages of new aircraft/spacecraft structures and design modifications that occur later in the product life cycle. ASTROS, based on the standard NASTRAN data formats, combines finite element modeling and analysis techniques with efficient optimization solutions to deliver significant reductions in the time required to develop superior designs of aerospace structures. ASTROS integrates all of the engineering disciplines that impact the preliminary structural design phase and can simultaneously design to strength, flutter, displacement, and other requirements. It considers a wide scope of conditions in a design task and treats multiple boundary conditions, each permitting a range of analysis such as statics, modes, and flutter. The current AIM supports a subset of the current ASTROS functionality. More can be added when needed.

AVL – Describing geometry appropriate for AVL (the Athena Vortex Lattice) code [9] is different than higher fidelity codes that require a single *Body* representing the OML. AVL requires multiple *Bodies* each referring to an airfoil section. The geometric model needs to be consistent with a build description that is hierarchical and multi-fidelity. That is, the build description that generates the geometric data at this level can be further enhanced to produce the complete OML of the aircraft design under consideration.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

As for the geometric description, AVL requires airfoil section data specified at the appropriate locations that describe the *skeleton* of the aircraft. These sections, when *lofted* as groups and finally *unioned* together, build the OML. Intercepting the state of the geometry before these higher-level operations are applied provides the data appropriate for AVL. This naturally constructs a hierarchal geometric view where a design can progress into higher fidelities and feedback can be achieved where we can go back to this level of description when need be.

AWAVE – AWAVE provides an estimation for wave drag at supersonic Mach numbers at various angles of attack. Inside AWAVE all configurations are assumed to be symmetric with respect to the X-Z plane. Only the +Y axis portion of a given model is used to generate the AWAVE input. This AIM automatically finds the proper portions of the model to create the input. However, it assumes that the model is oriented with the X-axis as the flow direction and the Y-axis out the right-side wing from the pilot's perspective.

Cart3D – Cart3D (written by Michael Aftosmis and team at NASA Ames Research Center) is the *best-in-class* CFD Euler (inviscid) solver. The geometric input to Cart3D is a surface tessellation of the *Body* of interest (in the form of an **EGADS** tessellation Object). Cart3D constructs an AMR (hierarchical Automatic Mesh Refinement) mesh, where the input body triangulation cuts through the Cartesian elements.

Another module in this suite is not a connection to **CAPS**, but is an **ESP** interface into the Cart3D Design Framework. This interface code ("ESPxddm") uses the XML Cart3D design description language XDDM to adjust **OpenCSM** design parameters, rebuild the geometry, tessellate and compute sensitivities for the Design Framework.

Chimera Grid Tools – Chimera Grid Tools (CGT) is a software package containing a variety of tools for the Chimera overset grid approach for solving complex configuration problems. The typical starting point is a description of the surface geometry in the form of triangulations or regular surface patches. This data is converted from the **ESP** description and is the prototype for the current **ESP** integration into the CGT tool "OverGrid".

EGADS Tessellation – The **EGADS** surface meshing AIM provides **CAPS** with the native EGADS triangulation (or quadrilaterals) in the form of a Tessellation Object.

FRICTION – FRICTION is a skin friction and form drag estimation program written by W. Mason (Virginia Tech), which provides an estimate of laminar and turbulent skin friction and form drag suitable for use for aircraft preliminary design.

Fun3D – Fun3D is an unstructured 3D RANS CFD solver from NASA Langley Research Center.

HSM – The Hypergeometric Shell Model (HSM) is formulated in the global 3D cartesian coordinate system, parameterized using local (element) coordinates, which also define a local basis for forming tangential derivatives and material strains. This closely follows the analysis of Simo et al [10,11,12], except that here the equations are obtained directly from stress equilibrium rather than an energy functional. As in Simo's formulation, the present method also defines a material quasi-normal vector (or director) as a primary unknown, whose deviation from the surface normal defines the transverse shear strains. The transverse shear stresses, however, are represented separately by their scalar potential, so that the normal-force equilibrium relation

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

becomes a well-conditioned Poisson equation, and also precludes any shear-locking problems [13].

<u>MASSTRAN</u> – MassTran is a simple solver that approximates the mass properties and is used primarily for training in the use of **CAPS**. It computes the total mass, center of gravity, and moments of inertia of a geometric configuration using structural shell meshes attributed for finite element structural solvers such as ASTROS and NASTRAN.

<u>MYSTRAN</u> – MYSTRAN is an open-source general purpose finite element analysis computer program for structures that can be modeled as linear (i.e. displacements, forces and stresses proportional to applied load). MYSTRAN is an acronym for "My Structural Analysis", to indicate its usefulness in solving a wide variety of finite element analysis problems on a personal computer (although there is no reason that it could not be used on larger computers as well). For anyone familiar with the popular NASTRAN computer program developed by NASA in the 1970's and popularized in several commercial versions since, the input to MYSTRAN will look quite familiar. Indeed, many structural analyses modeled for execution in NASTRAN will execute in MYSTRAN with little, or no, modification. MYSTRAN, however, is not NASTRAN. All of the finite element processing to obtain the global stiffness matrix (including the finite element matrix generation routines themselves), the reduction of the stiffness matrix to the solution set, as well as all of the input/output routines are written in independent, modern, Fortran 90/95 code. The major solution algorithms (e.g., triangular decomposition of matrices and forward/backward substitution to obtain solutions of linear equations and Lanczos eigenvalues extraction code), however, were obtained from the popular LAPACK and ARPACK codes.

<u>NASTRAN</u> – NASTRAN is a finite element analysis (FEA) program that was originally developed for NASA in the late 1960s by Stephen Burns of the University of Rochester under United States government funding for the Aerospace industry. The MacNeal-Schwendler Corporation (MSC) was one of the principal and original developers of the public domain NASTRAN code. NASTRAN source code is integrated in a number of different software packages, which are distributed by a range of companies. This AIM is specifically targeted for the MSC NASTRAN variant.

<u>Pointwise</u> – Pointwise is the premier commercial CFD Mesher. Pointwise has its own internal Geometry Kernel which is referred to as Geometry Engine (GE). A prerequisite for AIM construction is that **EGADS** data be translated into GE, which includes geometry, topology and attribution. This has been done in a separate stand-alone application ("egads2nmb"). This code ("egads2nmb") has been taken over and incorporated into Pointwise Ver 18.2R2 (and higher) so that Pointwise can now fully import **EGADS** models. The AIM drives Pointwise *Glyph* scripts to provide full automation after import.

<u>Skeleton</u> – This AIM is a coding example that individuals interested in writing AIMs can use as a pattern. It is being constructed for the **CAPS** trainings, where (at times) the last session is on AIM writing.

<u>SU^2</u> – SU^2 is an open-source unstructured 3D RANS CFD solver from a team of researchers, students and others) from Stanford University overseen by Prof. Juan Alonso.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

TetGen – TetGen is an open-source tetrahedral mesh generator written by Hang Si [14], which can be used with Fun3D or SU^2 for Euler simulations.

TSFOIL – This code solves the two-dimensional, transonic, small-disturbance equations for flow past lifting airfoils in both free air and various wind-tunnel environments by using a variant of the finite-difference method.

XFOIL – XFOIL [15] is an interactive program for the design and analysis of subsonic isolated airfoils from Mark Drela at MIT.

2.4 Software Engineering and Testing

In that the tangible output for this contract is the open source software, the approach to software generation, testing, release and training is critical in order that the result be robust and usable. To ensure that all of the **ESP** software components function a great deal of testing is required. This has been automated by the use of Jenkins which *watches* the MIT hosted **ESP** repositories for commits.

This testing and the use of Jenkins as the test harness was initiated during this contract (though there was no specific associated task). We have found this invaluable in that it allows for the generation of software that is of an exceptionally high quality (rivaling the best commercial engineering software available). Most of the time the testing suite finds the problems so that the users do not. It allows for us to freely change foundational portions of the software without the fear of inserting bugs or not maintaining backward compatibility. The success of this software is partially attributable to the vast amount of testing and the use of Jenkins for organizing the tests.

The **ESP** tab on the ACDL/MIT Jenkins page can be seen in Figure 2. Currently all testing is done against 2 Releases of OpenCASCADE, 6.8.1 and 7.3.1 (both of which have been hardened and corrected for known bugs). A more complete description of each of these Jenkins Projects follow:

BasicOcsm – **EGADS** and **OpenCSM** are built for the use of the Jenkins project and the basic regression tests are run from the **OpenCSM** data/basic directory.

Beta – This project is run after a new **ESP** Beta release has been put on the **ESP** website. It compiles and builds all of the **ESP** (including **CAPS**) applications to test out the distribution layout. Minimal testing on both **ESP** and **CAPS** is performed.

Bob –Similar to BasicOcsm but is initiated manually to test low-level changes. This allows for finding problems before the regular testing

Commit – This gets run after an *svn* commit from either the **CAPS**, **OpenCSM** or **EGADS** repositories. It compiles, and generally builds all libraries and applications. It also executes a small suite of test examples to ensure that the base-level functionality is intact.
Coverage – The *gcc* suite of compilers supports the ability to map the level of coverage the suite of tests actually touch in the source code-base. This is useful information and provides a "water mark" where testing coverage should only improve over time. This is available only under Linux. Note that this is currently not being monitored.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

MemcheckCaps – Both *clang* and *gcc* have the ability to check some memory usage (for out of bounds memory usage) by compiling the source with certain flags. This is done for **CAPS** and a fairly exhaustive suite of tests are run in order to check for memory problems. This provides some of the same data as has been available through *Valgrind* but runs faster (note that this does not check for the use of *uninitialized values*, so *Valgrind* is still an important tool in the arsenal).

MemcheckOcsm – Same as MemcheckCaps but for **EGADS** and **OpenCSM**.

MemcheckOcsm_7.4 – Same as MemcheckOcsm but testing against OpenCASCADE 7.4.1 (our hardened version of the 7.4 release). This is being done as we deprecate support of OpenCASCADE 6.8 and consider including OpenCASCADE 7.4.

RegCaps – A complete **CAPS** test suite of unit and higher-level tests are built and run for all supported architectures and compilers. Because of the standardization, all sections should provide the same results. If not, this highlights cases to examine closely. This was what was done, by hand, just before a full release was made official. The use of Jenkins in this project significantly reduces the time required to cut a new release.

RegOcsm – **EGADS** and **OpenCSM** are built. The complete **OpenCSM** test suite of unit and higher-level tests are run for all supported architectures and compilers. Because of the standardization, all sections should provide the same results. If not, this highlights cases to examine closely.

UndefinedCaps – Both *clang* and *gcc* have the ability to catch the use of undefined variables. This project builds and executes the **CAPS** applications with this flag and reports any findings.

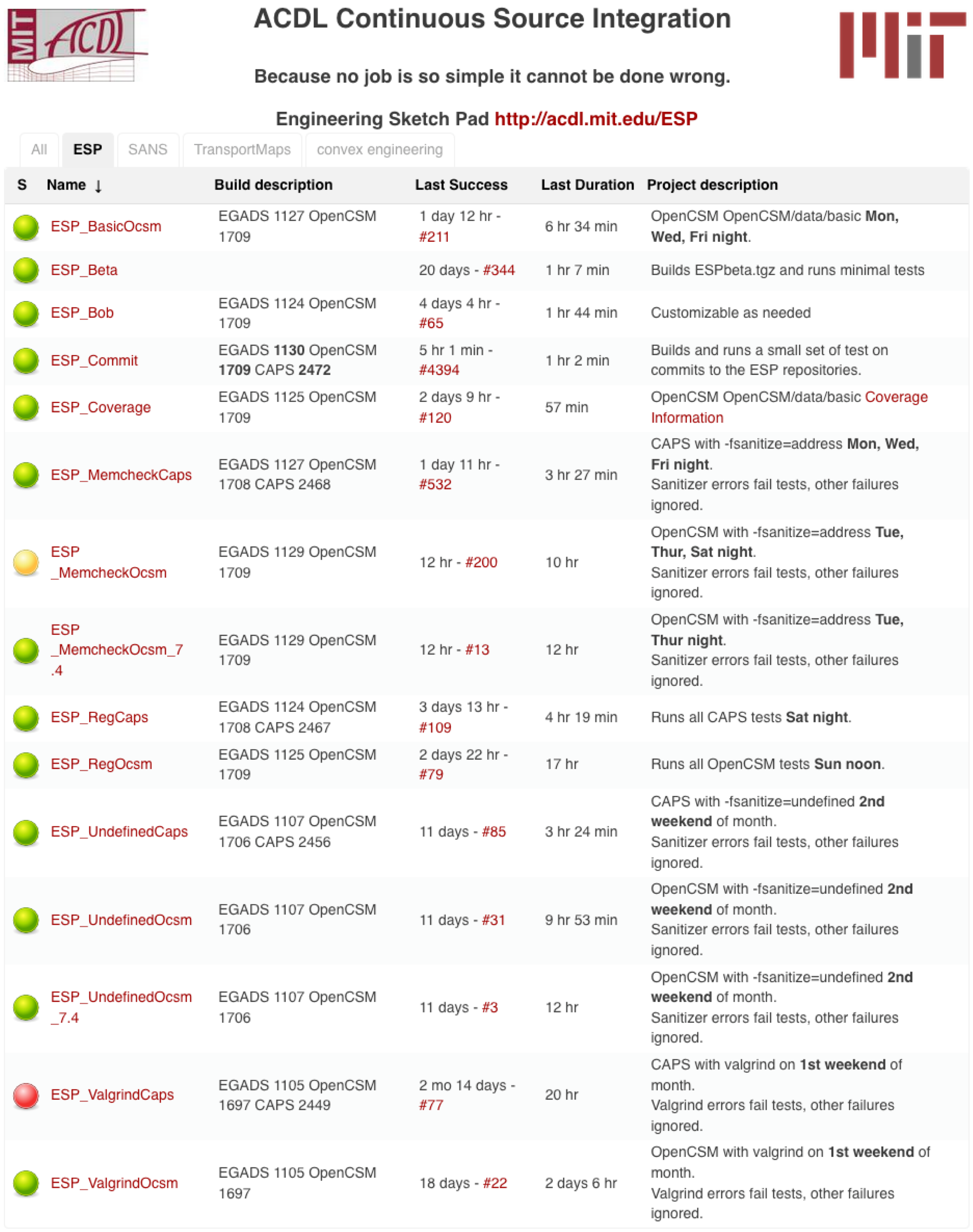
UndefinedOcsm – Both *clang* and *gcc* have the ability to catch the use of undefined variables. This project builds and executes **EGADS** and **OpenCSM** applications with this flag and reports any findings.

UndefinedOcsm_7.4 – Same as UndefinedOcsm but testing against OpenCASCADE 7.4.1

ValgrindCaps – This project runs the same cases as in the RegCaps Project, but does so using the dynamic analyzer *Valgrind*. This is run only once a month because it can take about a day, due to the size of the test suite and the speed penalty (as much as 10 times slower) encountered using this tool.

ValgrindOcsm – This project runs the same cases as in the RegOcsm Project, but does so using the dynamic analyzer *Valgrind*. This is run only once a month because it can take more than two days, due to the size of the test suite and the speed penalty (as much as 10 times slower) encountered using this tool.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



ACDL Continuous Source Integration

Because no job is so simple it cannot be done wrong.

Engineering Sketch Pad http://acdl.mit.edu/ESP

All | **ESP** | SANS | TransportMaps | convex engineering

S	Name ↓	Build description	Last Success	Last Duration	Project description
🟢	ESP_BasicOcsm	EGADS 1127 OpenCSM 1709	1 day 12 hr - #211	6 hr 34 min	OpenCSM OpenCSM/data/basic **Mon, Wed, Fri night**.
🟢	ESP_Beta		20 days - #344	1 hr 7 min	Builds ESPbeta.tgz and runs minimal tests
🟢	ESP_Bob	EGADS 1124 OpenCSM 1709	4 days 4 hr - #65	1 hr 44 min	Customizable as needed
🟢	ESP_Commit	EGADS **1130** OpenCSM **1709** CAPS **2472**	5 hr 1 min - #4394	1 hr 2 min	Builds and runs a small set of test on commits to the ESP repositories.
🟢	ESP_Coverage	EGADS 1125 OpenCSM 1709	2 days 9 hr - #120	57 min	OpenCSM OpenCSM/data/basic Coverage Information
🟢	ESP_MemcheckCaps	EGADS 1127 OpenCSM 1708 CAPS 2468	1 day 11 hr - #532	3 hr 27 min	CAPS with -fsanitize=address **Mon, Wed, Fri night**. Sanitizer errors fail tests, other failures ignored.
🟡	ESP _MemcheckOcsm	EGADS 1129 OpenCSM 1709	12 hr - #200	10 hr	OpenCSM with -fsanitize=address **Tue, Thur, Sat night**. Sanitizer errors fail tests, other failures ignored.
🟢	ESP _MemcheckOcsm_7 .4	EGADS 1129 OpenCSM 1709	12 hr - #13	12 hr	OpenCSM with -fsanitize=address **Tue, Thur night**. Sanitizer errors fail tests, other failures ignored.
🟢	ESP_RegCaps	EGADS 1124 OpenCSM 1708 CAPS 2467	3 days 13 hr - #109	4 hr 19 min	Runs all CAPS tests **Sat night**.
🟢	ESP_RegOcsm	EGADS 1125 OpenCSM 1709	2 days 22 hr - #79	17 hr	Runs all OpenCSM tests **Sun noon**.
🟢	ESP_UndefinedCaps	EGADS 1107 OpenCSM 1706 CAPS 2456	11 days - #85	3 hr 24 min	CAPS with -fsanitize=undefined **2nd weekend** of month. Sanitizer errors fail tests, other failures ignored.
🟢	ESP_UndefinedOcsm	EGADS 1107 OpenCSM 1706	11 days - #31	9 hr 53 min	OpenCSM with -fsanitize=undefined **2nd weekend** of month. Sanitizer errors fail tests, other failures ignored.
🟢	ESP_UndefinedOcsm _7.4	EGADS 1107 OpenCSM 1706	11 days - #3	12 hr	OpenCSM with -fsanitize=undefined **2nd weekend** of month. Sanitizer errors fail tests, other failures ignored.
🔴	ESP_ValgrindCaps	EGADS 1105 OpenCSM 1697 CAPS 2449	2 mo 14 days - #77	20 hr	CAPS with valgrind on **1st weekend** of month. Valgrind errors fail tests, other failures ignored.
🟢	ESP_ValgrindOcsm	EGADS 1105 OpenCSM 1697	18 days - #22	2 days 6 hr	OpenCSM with valgrind on **1st weekend** of month. Valgrind errors fail tests, other failures ignored.

Figure 2. The ESP tab displaying all of the Jenkins testing projects.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3 TASK STATUS

There is no intention of generating a detailed report of each of these tasks. For that information please refer to the associated papers (see the References) or the appropriate monthly technical reports (where there have been 61). The monthly reports are complete and detailed. This final report discusses and summarizes the status of each task/subtask.

3.1 CAPS

This first set of 10 tasks were the ones outlined in the original **CAPS** proposal. This ran from August 2014 to May 2017.

3.1.1 CAPS Infrastructure

These subtasks are associated with the design and initial development of the **CAPS** software system.

3.1.1.1 Overall Architecture

A detailed design for the CAPS infrastructure, including identification of the form and function of all major components and the associated application programming interfaces (APIs) was performed. A design review with customer was had early on in the contract (See Section 3.1.10.1) to ensure that most all of the needs are met. The result can be seen in the current **ESP** distribution in the file $ESP_ROOT/doc/CAPSapi.pdf (also see Appendix B), which has only changed in minor ways since the original design.

3.1.1.2 AIM Plugin Design

The API for the Analysis Interface & Meshing (AIM) subsystem was defined taking the API for the User-defined Primitives/Functions (within the Geometry subsystem) as an example and template. The current AIM development and API document can be found in the **ESP** distribution in the file $ESP_ROOT/doc/AIMdevel.pdf (also see Appendix C).

This design has changed over the course of this effort, in particular for the support of pyCAPS and providing a hierarchal view for the AIMs. For the most part these changes were accomplished by adding AIM access points while avoiding changing the signatures of the functions (so that the functionality of existing AIMs could be maintained – backward compatibility).

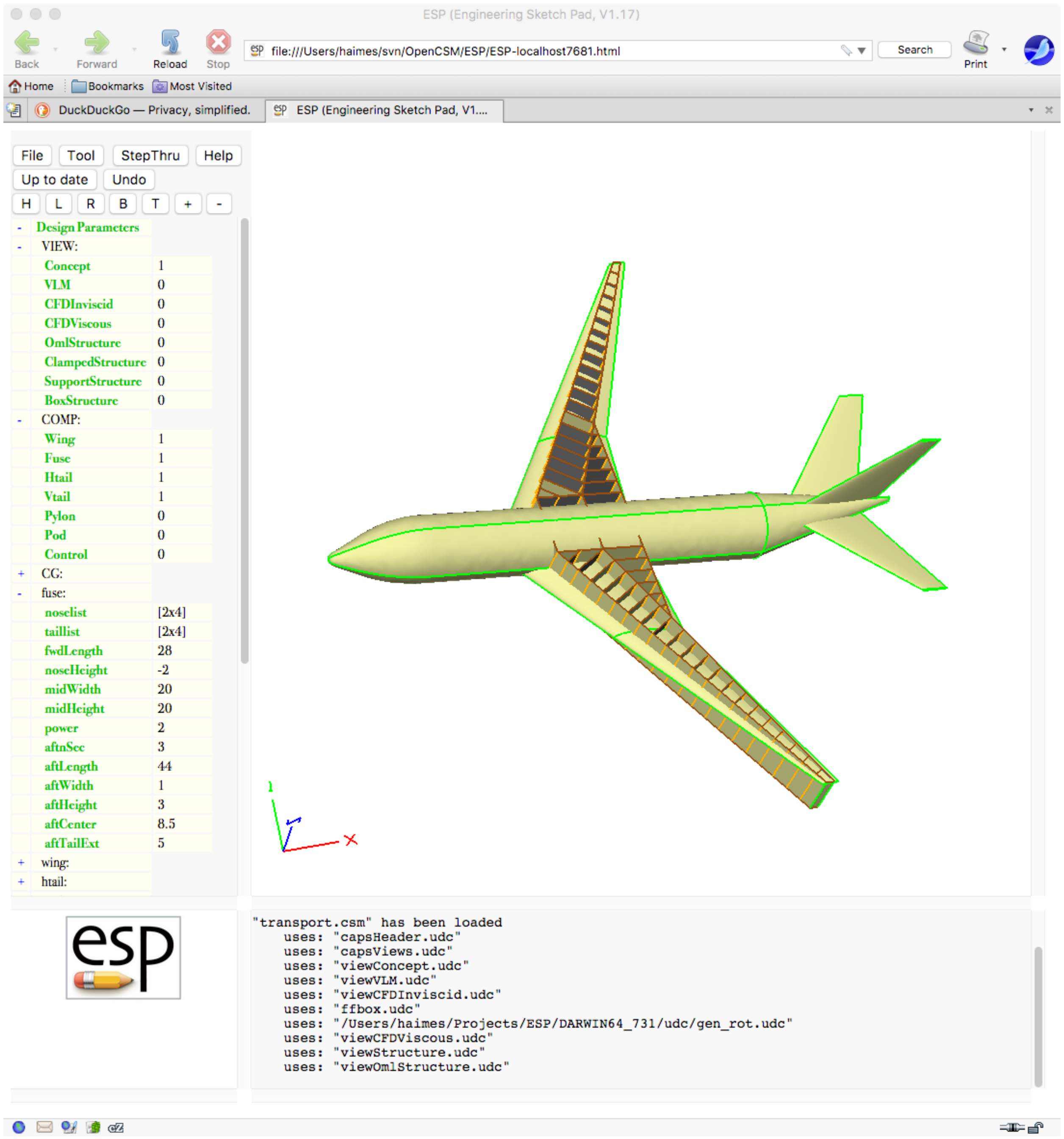
The current design appears quite flexible and adequate as can be seen by the number and breadth of AIMs distributed with **ESP**/**CAPS** (see Section 2.3) and the proprietary AIMs found at AFRL and other sites.

3.1.2 Engineering SketchPad

The subtasks listed below have to do with the Graphical User Interface (GUI) of the geometry subsystem found in **ESP**. These are maintenance, improvement and demonstration efforts.

3.1.2.1 ESP Parameter Manager

A browser-based parameter manager was designed and implemented that allows users to inspect and change all the parameters associated with a model. These parameters include geometric parameters (such as aspect ratio), material property parameters, and control parameters. The parameters will be organized hierarchically such that conceptual, preliminary, and detailed design parameters are grouped together.


ESP (Engineering Sketch Pad, V1.17)
Back
Forward
Reload
Stop
file:///Users/haimes/svn/OpenCSM/ESP/ESP-localhost7681.html
Search
Print
Home
Bookmarks
Most Visited
DuckDuckGo — Privacy, simplified.
ESP (Engineering Sketch Pad, V1....
File
Tool
StepThru
Help
Up to date
Undo
H
L
R
B
T
+
-
esp

- Design Parameters	
- VIEW:	
Concept	1
VLM	0
CFDInviscid	0
CFDViscous	0
OmlStructure	0
ClampedStructure	0
SupportStructure	0
BoxStructure	0
- COMP:	
Wing	1
Fuse	1
Htail	1
Vtail	1
Pylon	0
Pod	0
Control	0
+ CG:	
- fuse:	
noselist	[2x4]
taillist	[2x4]
fwdLength	28
noseHeight	-2
midWidth	20
midHeight	20
power	2
aftnSec	3
aftLength	44
aftWidth	1
aftHeight	3
aftCenter	8.5
aftTailExt	5
+ wing:	
+ htail:	

```
"transport.csm" has been loaded
    uses: "capsHeader.udc"
    uses: "capsViews.udc"
    uses: "viewConcept.udc"
    uses: "viewVLM.udc"
    uses: "viewCFDInviscid.udc"
    uses: "ffbox.udc"
    uses: "/Users/haimes/Projects/ESP/DARWIN64_731/udc/gen_rot.udc"
    uses: "viewCFDViscous.udc"
    uses: "viewStructure.udc"
    uses: "viewOmlStructure.udc"
```

Figure 3. ESP Parameter Management.

The hierarchical **ESP** Parameter Manager can be seen in the left-hand frame of Figure 3. The transport model shown here is from the most recent training and can be found in the current **ESP** distribution at $ESP_ROOT/training/ESP/transport.csm. What is seen is the “Conceptual” *view*

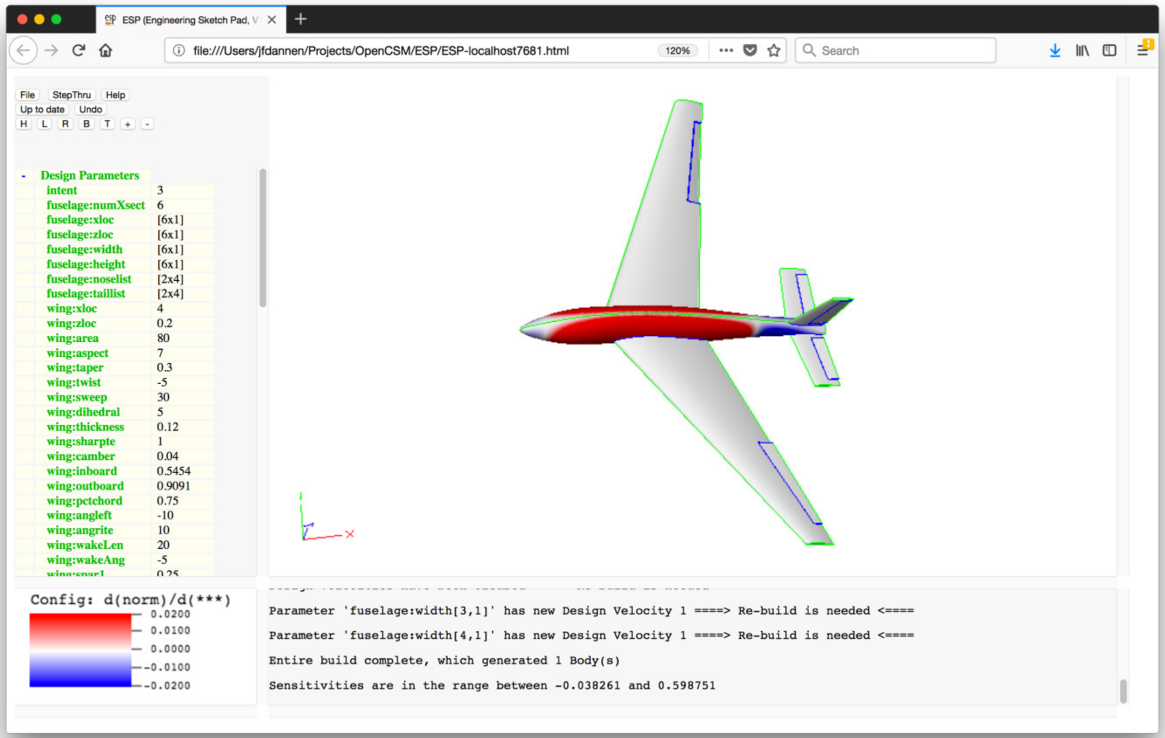
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

(i.e., the one appropriate to examine the model, and not for any specific analysis). The other views can be simply activated by setting the appropriate parameters to 1.

The components *in play* can be selected in a similar manner. In Figure 3 we see that the wing, fuselage, horizontal and vertical tails are expressed, where the pylon, pod and control surfaces have been suppressed. Also, the fuselage parameters have been opened up displaying the fuselage controls. Similar access exists for all of the other components.

3.1.2.2 WebViewer Updates

The browser-based geometry viewer (WebViewer) was extended to allow for visualization of surface parameters, such as design sensitivities, aerodynamic loads, structural displacements, and temperatures. This was accomplished in the **ESP** GUI and a pyCAPS initiated variant. Much effort as expended to ensure that the WebViewer performs well on supported browsers. Note that Microsoft's Internet Explorer and Edge could not be included due to long standing WebSocket bugs.



Design Parameters	
intent	3
fuselage:numXsect	6
fuselage:xloc	[6x1]
fuselage:zloc	[6x1]
fuselage:width	[6x1]
fuselage:height	[6x1]
fuselage:noselist	[2x4]
fuselage:taillist	[2x4]
wing:xloc	4
wing:zloc	0.2
wing:area	80
wing:aspect	7
wing:taper	0.3
wing:twist	-5
wing:sweep	30
wing:dihedral	5
wing:thickness	0.12
wing:sharpte	1
wing:camber	0.04
wing:inboard	0.5454
wing:outboard	0.9091
wing:pctchord	0.75
wing:angleft	-10
wing:angrite	10
wing:wakeLen	20
wing:wakeAng	-5
wing:spar1	0.25

```
Parameter 'fuselage:width[3,1]' has new Design Velocity 1 ====> Re-build is needed <====

Parameter 'fuselage:width[4,1]' has new Design Velocity 1 ====> Re-build is needed <====

Entire build complete, which generated 1 Body(s)

Sensitivities are in the range between -0.038261 and 0.598751
```

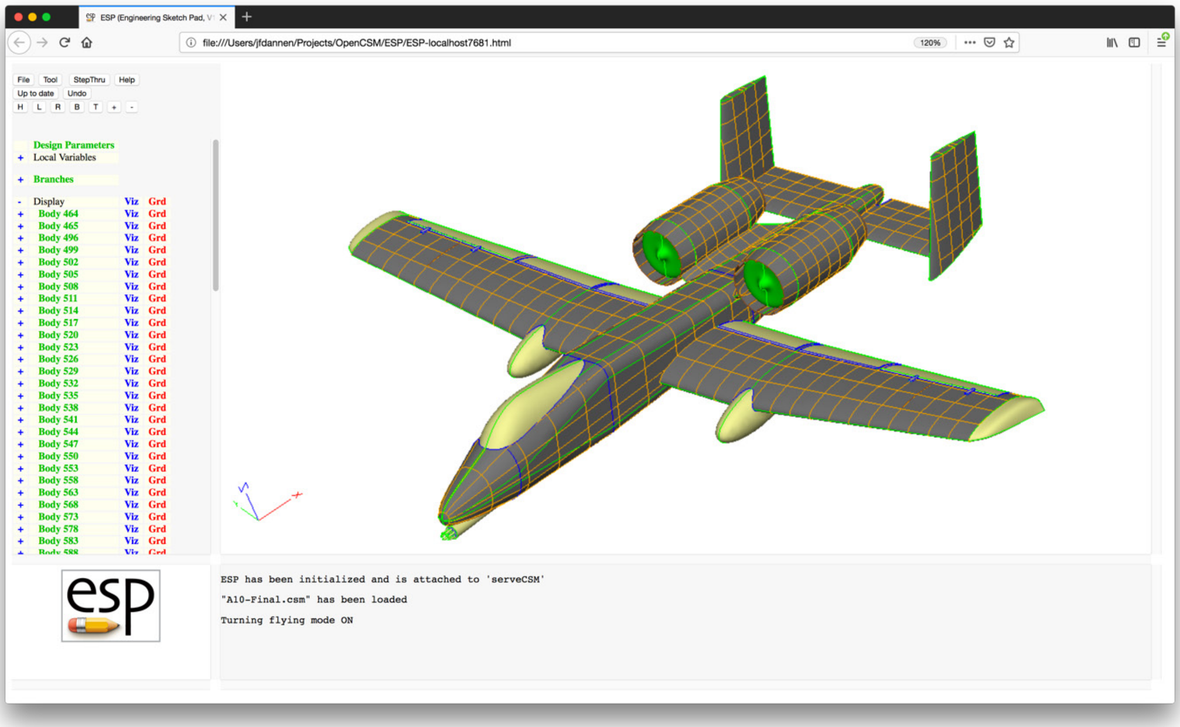
Figure 4. Sensitivity of the fuselage width at a section mid-wing.

Figure 4 shows the design sensitivity colored on the geometry. Because the fuselage was generated by the *blend* operation (which produces a fit cubic BSpline surface), one notes that as you increase the width at the mid-section the sensitivity towards the tail indicates shrinkage.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3.1.2.3 Multidiscipline multi-fidelity fighter

A number of fluid/structure fighter models have been generated, which are driven from a consistent suite of parameters. Most models, at a minimum, include a built-up element model (BEM) and an outer-mode line (OML). Example CSM files can be seen in the distribution $ESP_ROOT/data/fighter*.csm.

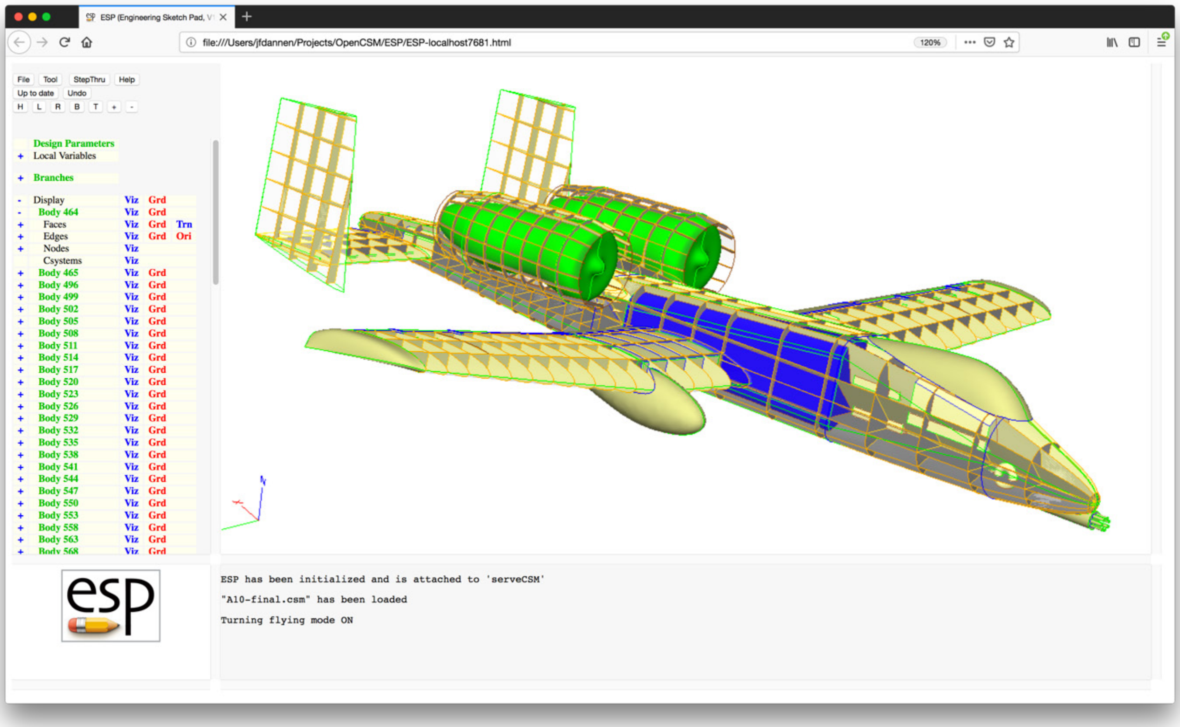


```
ESP has been initialized and is attached to 'serveCSM'
"A10-Final.csm" has been loaded
Turning flying mode ON
```

Figure 5. A10-like fighter – Outer mold-line and the scribed structural supports.

Figure 5 shows a fairly complete and rather detailed **ESP** model of a fighter inspired by the A10. This was put together by Kip Risch-Andrews, a Syracuse University undergraduate. It contains the OML, structural components (ribs, spars, bulkheads, and etc.) as well as subsystems like the engines, weapons and cutouts for the cockpit. Figure 6 displays the same model but with some of the OML skins not rendered, which gives a more complete view of the internal layout of the structures and components.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



```
ESP has been initialized and is attached to 'serveCSM'
"A10-final.csm" has been loaded
Turning flying mode ON
```

Figure 6. A10-like fighter – Showing the internal structures and components.

3.1.3 Geometry Subsystem

The following subtasks involve the continued improvement to **EGADS** and **OpenCSM** and required changes to fully support **CAPS**.

3.1.3.1 EGADS Updates

This was a continuing subtask throughout the original contract and the supplement. Its intention is to enhance and upgrade **EGADS** as required. **EGADS** is the geometry engine used by all of **ESP** (including the **CAPS** layer). The UDP/UDF as well as the AIM plugins are all **EGADS** applets and have full access to the model (geometry, topology, attribution and can modify and/or construct new models). **EGADS** was originally a *thin* veneer over OpenCASCADE. This situation has been changing over time and **EGADS**' dependency on OpenCASCADE has been diminished by its enhancement. This is due to three factors:

1. most of the robustness issues experienced in the **ESP** suite are due to SegFaults and unexpected aborts deep within OpenCASCADE,
2. some of OpenCASCADE's operators don't work as one would expect, and
3. OpenCASCADE does not provide the parametric sensitivities required by **ESP** without resorting to perturbing the geometry.

This subtask included:

- General **EGADS** maintenance. This included bug fixes and the effort in supporting new releases of OpenCASCADE. Much of this subtask is made easier by the vast testing, in particular, of the **OpenCSM** scripts.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

- Work continues to improve the speed and robustness of the Solid Boolean Operators (SBOs). There has been much success in this endeavor. Between the **EGADS** internal changes and the improvements in OpenCASCADE, we have seen as much as a factor of 10 speedup for some SBOs. Also, the success rate of completing all of the tests in the suite continues to improve indicating better robustness.
- Replaced the evaluations, inverse evaluations and in/out predicates for better speed and to be consistent with the **EGADS** parallel variant (**EGADSlite** – funded under a NASA NRA cooperative agreement). This has allowed for scalability to be realized when multithreading **EGADS** applications.
- Many of the low-level geometry functions now provide parametric sensitivities to operators in **EGADS.** This minimizes what currently requires finite-differencing.

3.1.3.2 OpenCSM Updates

Again, this was a continuing subtask throughout the original contract and the supplement to enhance and upgrade the **OpenCSM** as required. **OpenCSM** is the parametric build portion of **ESP**. This foundational software parses the build scripts (*feature tree*) in order to instruct **EGADS** as to how the geometry should be built, based on the Design Parameters. **OpenCSM** also provides parametric sensitivities (when queried) on the resultant geometry by traversing the script and applying the chain-rule.

This maintenance task covers the entire period and is a "catch all" for bug fixes, CSM scripting enhancements, continued development and testing.

3.1.3.3 CSM Components

A library of sub-system components that can be included within any model has been developed. Most of these are in the form of UDCs (User Defined Components), which appear to be CSM subroutines or macros. A complete collection of these scripts can be found in the **ESP** distribution in $ESP_ROOT/udc.

3.1.3.4 UDP Plugins

A large number of User Defined Primitive (UDP) and User Defined Function (UDF) plugins have been generated that can be included within any model. The UDP/UDFs include: bezier, biconvex, box, createPoly, csm, editAttr, ellipse, fitcurve, freeform, hex, import, kulfan, naca, naca456, nurbbody, parsec, pod, radwaf, sew, stiffener, supell, and waffle (see the **OpenCSM** help for more details).

3.1.4 Legacy Geometry Tools

Even though **ESP** is designed to generate *clean* geometry and models commensurate (in fidelity) with the analysis at-hand, there are situations where the import of legacy models is important. These subtasks deal with both static and parametric legacy geometry.

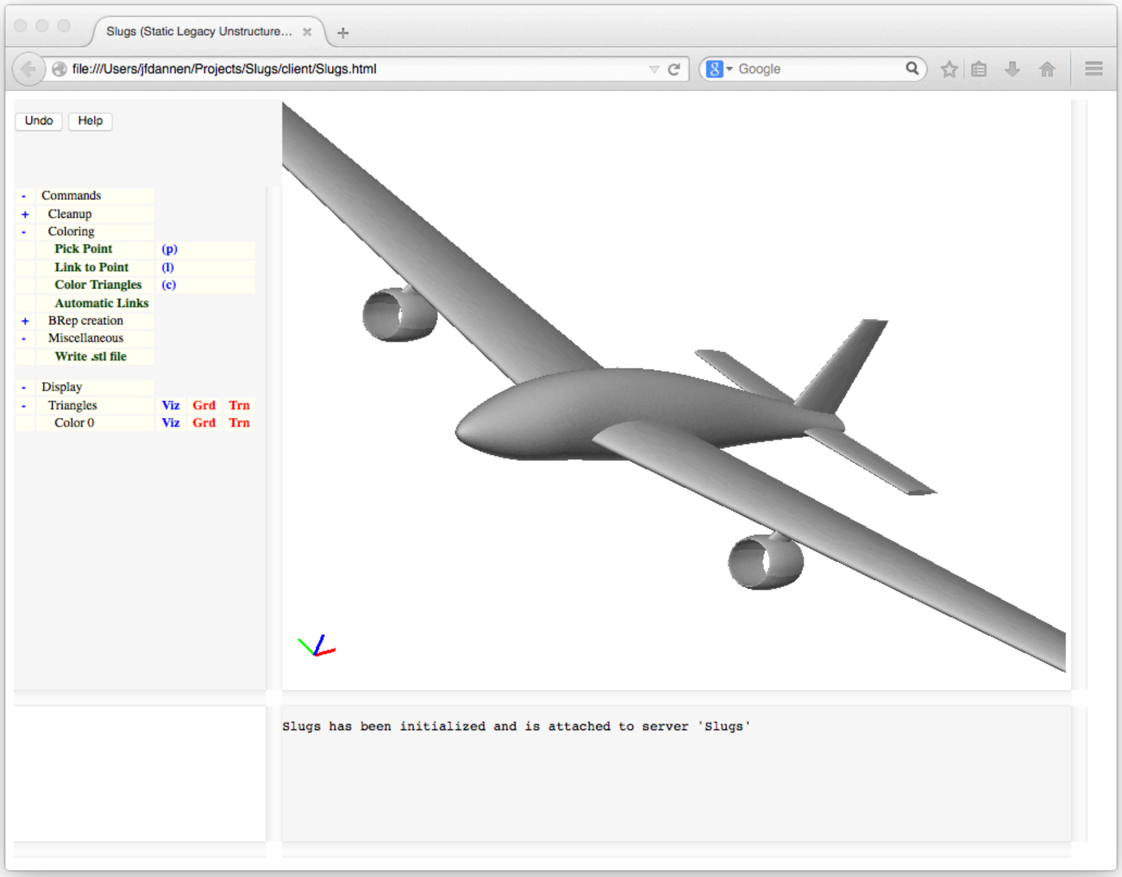
3.1.4.1 Import Legacy Geometry

There are 3 different file formats that can be used to import legacy geometry: IGES, STEP, and STL. **EGADS** through EG_loadModel (and the import command in **OpenCSM**) can load either IGES and/or STEP. In the case of STEP, the import is usually fairly good and may require no intervention. This may not be the case for IGES where the import can be good if the geometry is

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

properly trimmed and written with topology, but this is not the default. More often unconnected individual surfaces are written – these are import to **EGADS** as a series of *Face-Bodies*. The *Face-Bodies* can be put back together by using the **EGADS** function EG_sewFaces or in **OpenCSM** by the 'udprim sew' command. If this fails the imported geometry can be tessellated and the triangles that make up the tessellation can be treated like STL input.

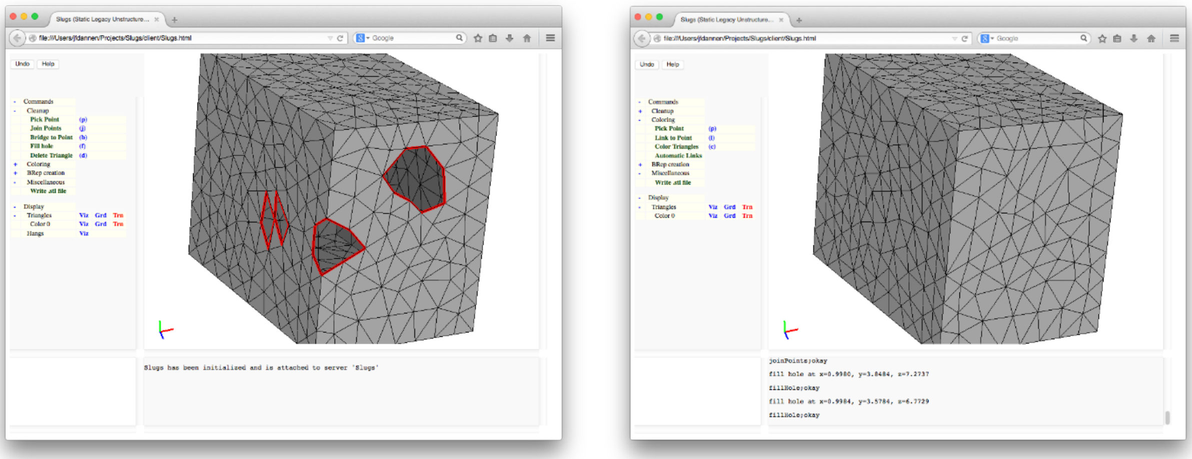
For **CAPS** to properly function (or the geometry to be used during the build of more complex configurations), the imported geometry cannot be discrete but must be in the form of a BRep. This is a problem for the import of triangulations (e.g., from STL files). The **CAPS** application SLUGS (the Static Legacy Geometry System) requires interaction with the user to accomplish the conversion to BRep. The user is presented with a graphical view of the input tessellation in a browser (using the WebViewer) with a very similar layout to **ESP**. Figure 7 shows an example configuration that consists of a transport-type configuration with two under-wing engines.



```
Slugs has been initialized and is attached to server 'Slugs'
```

Figure 7. Initial view of a tessellation that is given to SLUGS.

The first step in the use of SLUGS is to repair the tessellation to fill-in gaps and join nearly-identical points. An example of this is shown in Figure 8 where the three holes were filled in with very few mouse clicks.

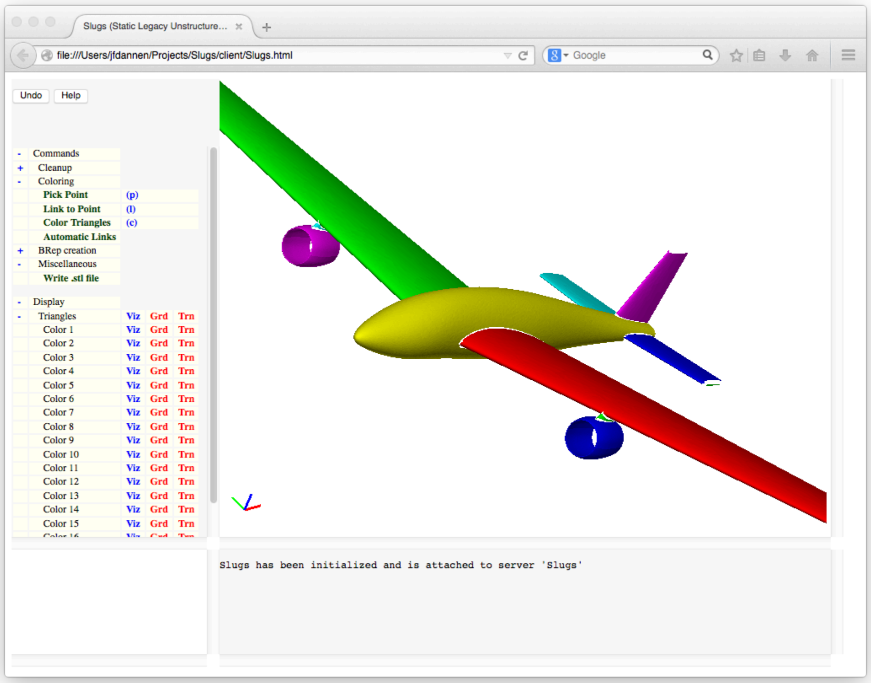


```
Slugs has been initialized and is attached to server 'Slugs'
```

```
joinPoints;okay
fill hole at x=0.9980, y=3.8484, z=7.2737
fillHole;okay
fill hole at x=0.9984, y=3.5784, z=6.7729
fillHole;okay
```

Figure 8. Example of initial and “fixed” tessellation in SLUGS.

The second step in the use of SLUGS is to separate the triangles in the configuration into “colors”, each of which will ultimately become a Face in the static geometry. This is done by marking triangle sides that enclose a group of triangles and then telling SLUGS to color them. The marking of the triangle sides is greatly facilitated by the use of a SLUGS tool that automatically marks the sides that lie on the shortest path between two specified points.



```
Slugs has been initialized and is attached to server 'Slugs'
```

Figure 9. Example of configuration colored by a user in less than 30 minutes.

For the configuration shown in Figure 9 the entire coloring process was performed in less than a half hour. See $ESP_ROOT/SLUGS/Slugs-help.html for a description of the interactive commands.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

The third step, which converts the colored triangulation into the BRep Edges and Faces requires colored patches that are either 3 or 4 sided. Each colored patch becomes a Face when fit with a BSpline control net of *7 x 7* points using the Levenberg-Marquardt algorithm.

This non-parametric geometry is usable in a static manner (i.e., the geometry cannot be parametrically rebuilt) throughout **CAPS**.

3.1.4.2 Legacy Parameterization

The goal of PLUGS (the Parametric Legacy Unstructured Geometry System) is to find the design parameter values (associated with a given CSM model) that most closely match a cloud of unassigned with regards to geometry (and unconnected) points. This is done through the use of a least-squares (Levenberg-Marquardt) optimizer that simultaneously changes the values of the design parameters as well as the *[u,v]* parametric coordinate associated with each point in the cloud. The initial work on PLUGS was done in 2016 by a graduate student (Pengcheng Jia) at Syracuse University [16].

The basic strategy is a nested process. In the outer layer of the process, each cloud point is associated with the most likely face in the configuration. Once these correspondences are made, the inner layer uses the Levenberg-Marquardt optimizer to modify the design parameters and geometric *[u,v]* parametric coordinates. At the end of this process, the “guessed” correspondences might not be very good, so the outer layer is again processed to re-establish the correspondences.

Though Jia showed very promising results, his research code was not integrated into the **ESP** software suite at the time. See Section 3.2.4.3 for the current status and **CAPS** integration.

3.1.5 Structural Design – LSM/HSM

A “Lumped” Structural Model (LSM) that extends traditional beam models into a plate-based model that is fully compatible with the simple aeroelastic analyses, such a vortex lattice method has been developed. Classical shell elasticity theory with complex geometry has traditionally been formulated in curvilinear coordinate systems on the shell. The resulting elasticity equations then involve coordinate Christoffel symbols which account for the curvatures of the coordinate lines. This formalism is not only complex, but creates unwarranted demands on geometry smoothness in computational implementations.

Discretizations which treat the shell elements as degenerate 3D solids circumvent the problems with curvilinear coordinates by formulating the problem in 3D Cartesian space, with the node position vector and transverse material vector (or director) as the primary unknowns. However, they have their own complications in their need for C^1 or even C^2 continuity of assumed element solution modes, and also have other problems such as shear locking. They also do not capture rigid-body rotations exactly without special treatment. Exact representation of rigid-body element rotation is highly desirable for applications such as high aspect ratio High Altitude Long Endurance (HALE) aircraft which can feature large deformations.

The Hypergeometric Shell Model (HSM) [13] is formulated in the global 3D Cartesian coordinate system and parameterized using local (element) coordinates, which also define a local basis for forming tangential and normal derivatives as well as material strains. HSM extends

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

basic bilinear shell models by constructing a conformal higher-order surface (*C^1* continuous in the fine-mesh limit) using the director field which is already present, so that no additional unknowns are introduced. The MITC method [17] is used for transverse shear strain interpolations. Overall, for a given numerical problem size, a large improvement in accuracy is obtained for highly-curved elements and bending-dominated problems, and particularly in problems with strong membrane/bending coupling, e.g., buckling. Both quadrilateral and triangle elements are treated.

The present method also defines a complete local basis for the undeformed geometry, in the form of a normal vector and two in-surface vectors in which general anisotropic materials can be specified independently of the discretization.

Figure 10 demonstrates a tube beam with incipient shell buckling computed using the HSM formulation. The tube is anchored at one end with a simple pinned support boundary condition, and a vertical force is applied at the free end. The tube exhibits local buckling for a sufficiently large tip load. At incipient buckling, a tube-ovalization buckling mode is evident, and features an inward "dent" on the upper surface roughly 1.5 diameters out from the anchored end. Also visible is another conventional column-type buckling mode characterized by an outward bend immediately adjacent to the anchor end.

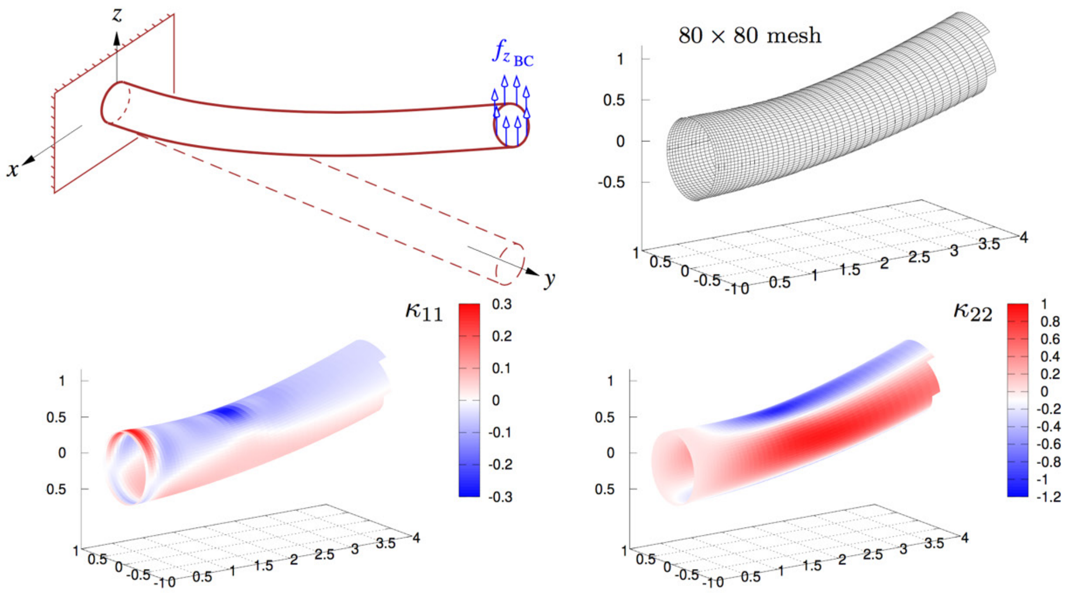


Figure 10. Tube beam is subjected to a vertical load on its free end.

3.1.6 Analysis Subsystem

3.1.6.1 Data Manager

It was originally envisioned that there be a **CAPS** data manager. The object-based data design changed the notion of where (meta)data for an object resides. In the current **CAPS** design information associated with an object is stored as part of the object itself. This includes functions like those associated with a software control system such as versioning, metadata (attribute) properties, and who made the changes. This could be thought of as be beginning of a design system with *Digital Thread*. There are also data objects in **CAPS** that are the receptacle for data associated with analysis input and output as well as those that reflect data distributed on the discrete form of the geometry targeted for a particular analysis (Vertex Sets). The Vertex Sets can have scalar, vector (and state vector) fields attached (Data Sets) to provide flexible methods to store information for the user to view or for multi-physics (multidisciplinary) **CAPS** problems. See in the **ESP** distribution the file $ESP_ROOT/doc/CAPSapi.pdf which describes in detail the construction of Vertex and Data Set objects and their use.

3.1.6.2 Conservative Fitting

The **CAPS** software design performs lazy computations, that is there is a concept of *dirty* associated with an object. This is handled by having a *serial number* associated with an object and if the prerequisite objects have a later *serial number* than the object itself, it is *dirty*. If an object is *dirty* and then requested, **CAPS** initiates the calculation required to update the object and make *clean* by resetting the serial number to the current value.

A **CAPS** Bound object can contain multiple Vertex Sets, each can have Data Sets with the same name, but only one can be the source of the Data Set (see Appendix B for a more complete descriptions of these terms). The source is basically the owning Analysis Object (attached to an AIM), where the other Data Sets (with the same name) are derived from this source Object. For a standard Fluid/Structure interaction the owning Data Set for *pressure* would be the CFD analysis, where the displacement (a vector of 3) source is the structural analysis.

The dependent Data Sets are computed via one of two methods: interpolation or conservative data transfer. If conservative, in a sense the interpolation weights are adjusted so that the area-weighted integrated values of the source and the derived information match [6]. Conservative data transfers are important for weakly coupled inner iterations so that the process can be convergent. See the end of the file $ESP_ROOT/doc/CAPSapi.pdf (Appendix B), which describes setting up these coupled multidisciplinary simulations.

Note that this has been demonstrated and used at AFRL in a number of situations and has become part of the **CAPS** training. See $ESP_ROOT/training/session5.3.pdf entitled "Data Transfer: Loosely-Coupled Aeroelasticity" for a complete description of the current state of **CAPS** coupling using pyCAPS.

3.1.7 Meshing

Meshing tends to be the bottleneck in many simulations. These subtasks are associated with the ability to automatically generate meshes for some of the analysis suites.

3.1.7.1 OverSet Meshing

The original use (and intent) for **OpenCSM** was as an overset mesh generation application. The notion was that you could build up the mesh topology as the geometry was constructed. The prototype for this kind of overset meshing was a tool called OvrCad. For a number of reasons, the priority of this task was always shadowed by more pressing needs. At the same time William Chan (NASA Ames Research Center), the author of the premier overset meshing tool *OverGrid* began using **EGADS** as the base-level geometry kernel. His intension was the same as OvrCad – automation of the overset mesh generation process. William's effort (funded internally by NASA) based the connection to geometry on **EGADS** and the additional information it can provide.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

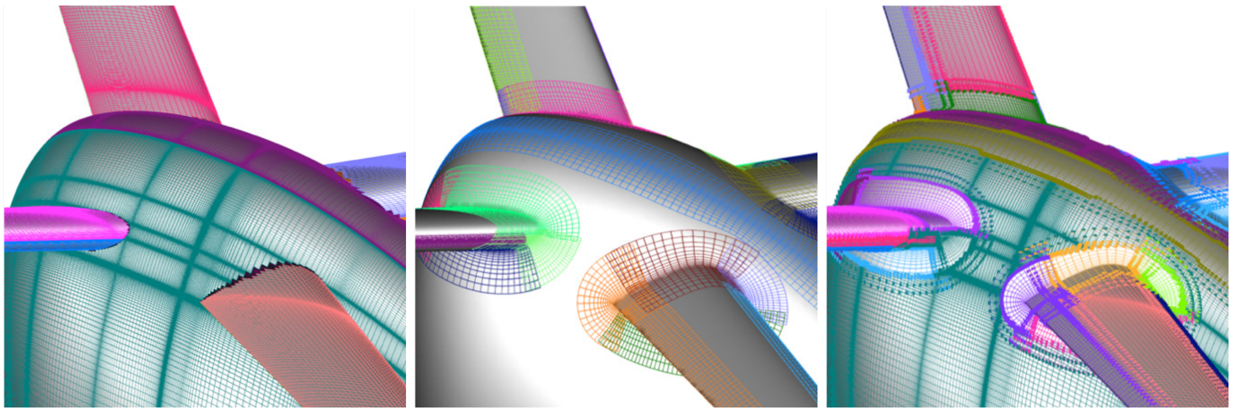


Figure 11. The use of EGADS in automating NASA's *OverGrid*.

The automated meshing results of using **EGADS**/OverGrid on a simple quadcopter can be seen in Figure 11. The left-hand image shows a graded *[u,v]* structured surface mesh *iblanked* where the surface is trimmed. Mesh clustering is seen where Nodes appear in the BRep Topology. The middle image of Figure 11 shows the automatic construction of *collar grids* from the BRep Loops, where the right-hand image shows the final mesh. The entire procedure is documented in [18].

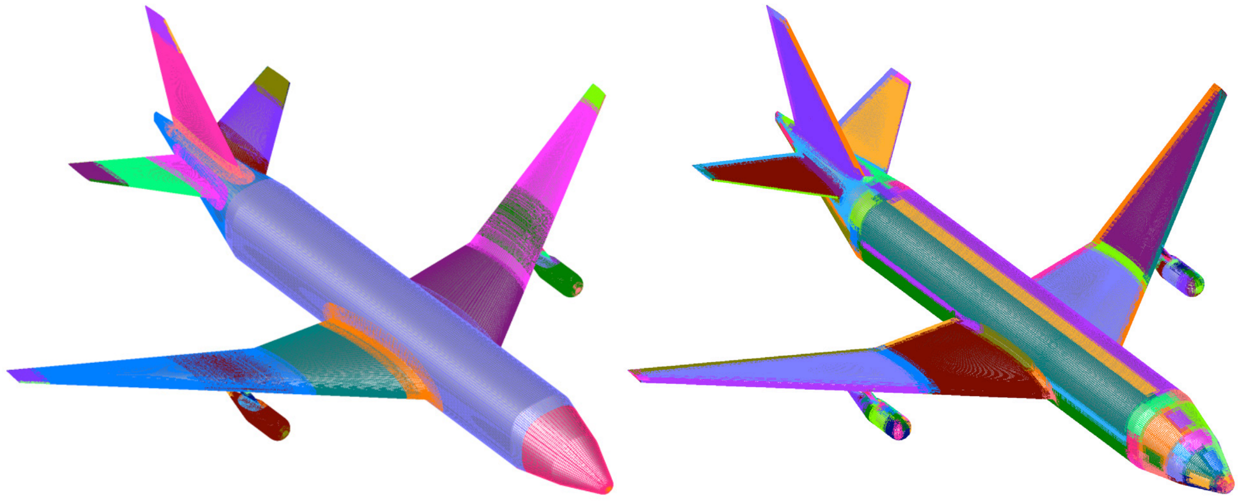


Figure 12. Manual (left) and automated (right) overset meshing of GMGW2 Case 3.

Figure 12 shows the comparison of manual vs. automated overset meshing of the geometry used for the second AIAA Geometry and Mesh Generation Workshop, where Table 1 shows a mesh count and timing comparison.

Table 1. Overset mesh generation timings.

	No. Grids	No. Points	Wall Time
Manual	50	490000	50 hours
Automatic	248	954000	6 minutes

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

The Chimera Grid Tool (CGT) AIM was the starting point for the **EGADS**/OverGrid connection, but OverGrid itself is not distributed with **ESP**/**CAPS** – OverGrid being part of NASA's Chimera Grid Tools is Export Controlled. Users need to make a formal request to NASA in order to get next release of CGT that will have this level of automaton. There currently is no **CAPS** AIM for CGT directly (but the prototype is part of the distribution).

3.1.7.2 BRep-based Meshing

The **EGADS** tessellation AIM has been developed to generate either triangle-based, or under some circumstances, quadrilateral-based meshes. The output is an **EGADS** Tessellation Object, which gets used by **CAPS** to easily generate Vertex Sets that are suitable for holding on to sensitivities or other Data Sets suitable for viewing or to be used for mesh->geometry->mesh transfers of information (see Section 3.1.6.2). The Tessellation Object can also be used as input to 3D meshers, in particular TetGen and AFLR3 where the Object is used to specify the bounds of the domain in order to generate tetrahedral grids.

3.1.8 Analysis Interface and Meshing (AIM) Plugins

These tasks were originally envisioned to ensure that the AIM design was sufficient to support the following analysis tools. As can be seen in Section 2.3, the suite of supported tools (in the open source **ESP** distribution) is far more complete that the subtasks listed below. Also, since the complete **ESP** distribution (including **CAPS**) continues to be upgraded and made available to AFRL personnel the original demonstration tasks became redundant because the *real* users were testing the code out on their actual problems.

3.1.8.1 Cart3D

An AIM plugin was built Cart3D analysis. This included both the preparation of the inputs files needed by Cart3D as well as transfer of information that is contained in Cart3D's output files back into the **CAPS** system. The geometry is represented as a triangulation and can be generated by the **EGADS** Tessellation or the AFLR4 AIMs.

3.1.8.2 ASTROS

A **CAPS** AIM plugin for many modes of ASTROS analysis has been implemented. This AIM can support both the full featured version or mASTROS that is shipped with the **ESP** distribution. The AIM includes both the preparation of the inputs files needed by ASTROS as well as transfer of information that is contained in ASTROS's output files back into the **CAPS** system.

3.1.8.3 SU^2

An AIM plugin for SU^2 analysis codes was designed and built. It supports versions 4.1.1 (Cardinal), 5.0.0 (Raven), 6.1.0, and 6.2.0 (Falcon). The AIM includes both the preparation of the inputs files needed by SU^2 as well as transfer of information that is contained in SU^2's output files back into the **CAPS** system.

3.1.8.4 OverFlow

Of the listed subtasks this is the only one not completed. Because OverCad was never resurrected (see Section 3.1.7.1) and the **EGADS** connection to OverGrid has just been completed there were no automated meshing tools to generate grids for OverFlow. In the near future (but not as part of this contract) an AIM plugin for an OVERFLOW analysis could easily be generated (it would not be too different from Cart3D, SU^2 or Fun3D).

3.1.8.5 LSM/HSM

Drela's FORTRAN API of the HSM software is directly linked into an AIM. The AIM is equipped with a reverse Cuthill-Mckee algorithm to improve the perform of the linear solves. At this point only a limited set of boundary and loading conditions are currently exposed via the AIM, and additional effort is required to fully expose the complete functionality of the HSM software. However, the HSM AIM has been exercised with simple cantilever shapes (including shapes with multiple faces), and the results exhibit significantly lower errors compared to traditional shell model discretizations. For a full description of HSM see Section 3.1.5 and [13].

3.1.9 Demonstrations

This demonstration task was placed in the original proposal as capstone examples of the use of the software. Because of the close interaction between the **CAPS** team and AFRL personnel as well as the fairly continuous delivery of software (see Section 3.1.10.3) these demonstration subtasks became less critical. Restated, *real* demonstrations were on-going during the contract by individuals (at AFRL and elsewhere) using the **CAPS** software in their workflow and to solve their problems of interest.

3.1.9.1 Fighter (Cart3D only)

Though this subtask calls for a fighter configuration, we used a simpler model where the geometry is known and published along with complete wind tunnel results. This is one of the test configurations for the Full Potential Code (see Section 3.2.4.4). The data can be found in the AGARD report AR-138 and the case is known as "Wing A Body B2". The chapter is from D.A. Treadgold, A.F. Jones, and K.H. Wilson entitled "Pressure Distribution Measured in the RA 8ft x 6ft Transonic Wind Tunnel on RAE Wing 'A' in Combination with an Axi-Symmetric Body at Mach Numbers of 0.4, 0.8 and 0.9".

The "Wing A Body B2" geometry was constructed via **ESP** and the case run through pyCAPS exercising both **CAPS** and the Cart3D AIM. Figure 13 shows Mach number from a converged (and mesh adapted) Cart3D run against a finely discretized resultant geometry (performed in the AIM), where surface pressure for the same case can be seen in Figure 14.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

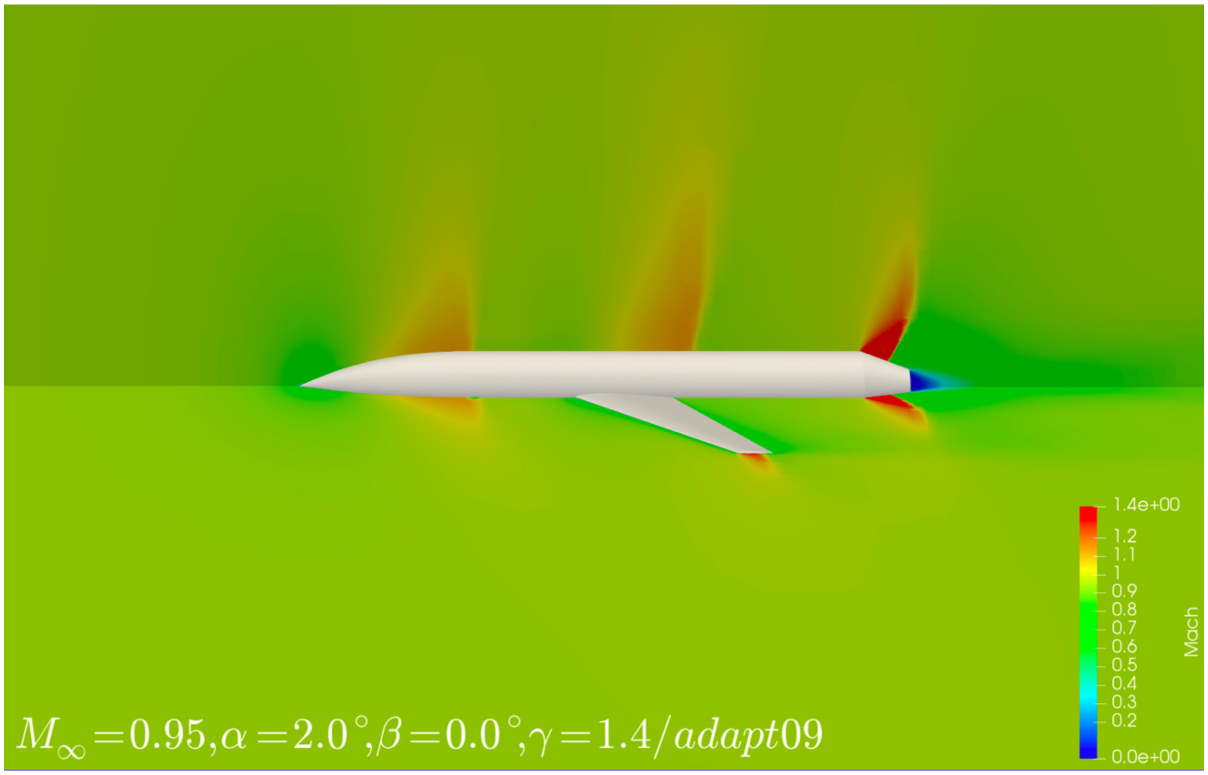


Figure 13. Mach number results from Cart3D shown on 2 planar cuts.

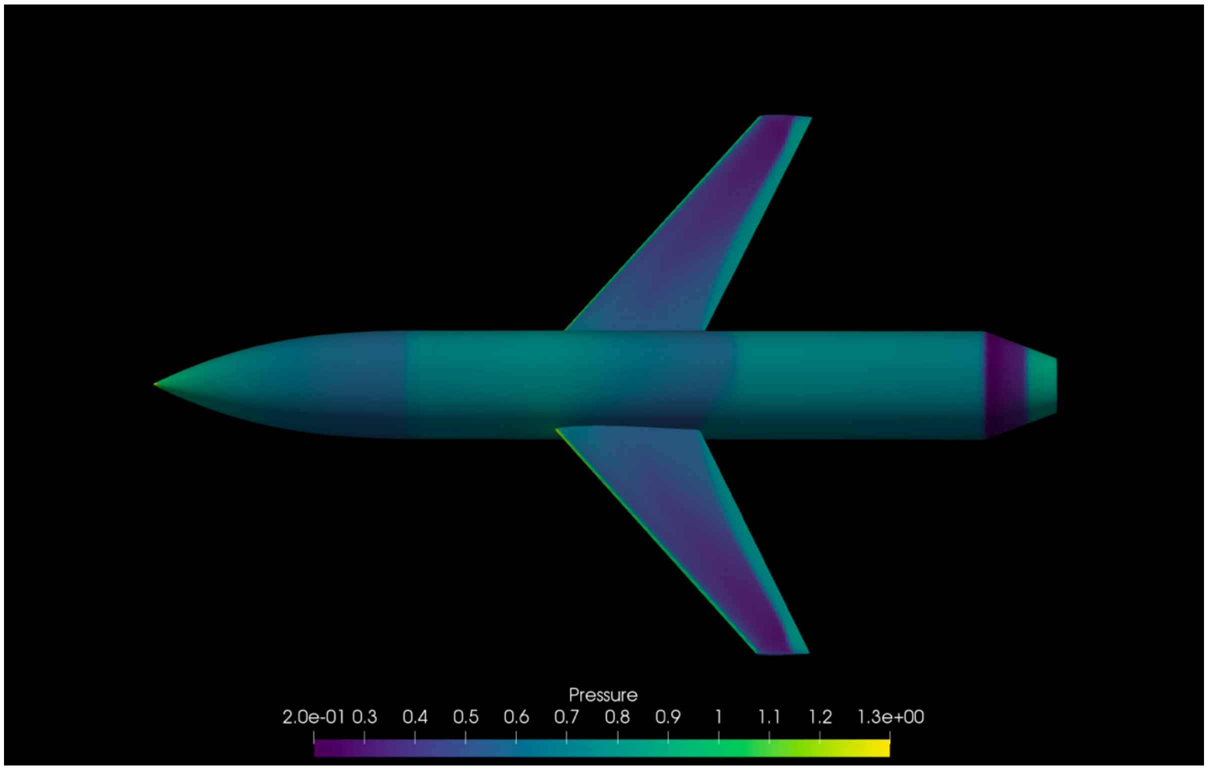


Figure 14. Surface pressures from the same case as seen in Figure 13.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3.1.9.2 Transport

A transport example has become part of the **CAPS** training. The model shown in Figure 3 can be found in the **ESP** distribution at $ESP_ROOT/training/ESP/transport.csm. It can be configured to generate geometry for various forms of analysis (*views*) and run through the appropriate solvers.

3.1.10 Support

This task includes general support for **ESP**/**CAPS** within RQVC (and other AFRL branches). The task includes subtasks for a software design review, training, software delivery and reporting. But it was found that another form of support was required during the original **CAPS** contract: maintenance. As the software was being deployed and used, individual would stumble through its learning curve. This would be found out by meeting with users to see how the software was being utilized. The end result was, at times, changes to the trainings to properly reflect *best practices*, but would also, at times, require additional functionality. This was particularly true when trying to determine the best way to attribute the geometry in preparation for analysis. And, of course, fixing bugs found by the user-base also required much effort unattributable to a specific subtask.

3.1.10.1 Design Review

Early in the contract (at the kick-off meeting) a **CAPS** software design review was executed. This presentation fully described the **CAPS** software layout, API and plugin functionality to the entire team and to selected AFRL personnel. Feedback was critical to ensure that the design provided a proper foundation and satisfies AFRL's perceived needs. This Design Review was a prerequisite for the Section 3.1.1 subtasks.

3.1.10.2 Training

All of the trainings given were either at AFRL or at a location off-base but in the local vicinity of Wright-Patterson Air Force Base. A list of the trainings given can be found in Table 2.

Table 2. Training Dates

Training	Days	Participants
ESP	July 2015 – 3 ½	41
ESP	August 2016 – 3	28
ESP	June 2018 – 2 ½	46
CAPS	August 2018 – 3	23 (limited to invitees)
ESP/CAPS	June 2019 – 5	31

The trainings have developed into a successful mix of informational material and hands-on exercises. Participants show up with their own laptops. They have previously downloaded the most recent **ESP** distribution (or show up early to do so before the training officially begins). This means that all exercises are performed on their equipment and the students walk away with the software functional, and the knowledge of how to use it in a familiar operating environment.

In general, there are sessions that discuss a topic (or suite of topics) lasting for an hour or two and then roughly an hour of hands on assignments. These assignments are carefully crafted not only to augment the class/session's material, but also are challenging enough to keep the better students from getting bored. At the end of each session there is the request for the students to fill in "muddy cards". These allow the students to ask questions anonymously (what was not clear), which can inform us when we are not getting the points across. Also, this mechanism is useful in correcting the class material and reporting bugs and making feature requests. At times, we reissue another release of the software "shortly" after a training where the problems that were discovered are fixed.

3.1.10.3 Code Delivery

A process for software delivery has been developed during the course of this contract. The delivered files include a directory structure to place all components and modules, source and installation directions / *makefiles* for all software written for a variety of platforms (Windows 7 & 10, LINUX and MAC OSX). The source is written in a variety of languages, such as ANSI C, C++, JavaScript (and some testing and example code is in Python). API bindings have been made available for C/C++ and, in some cases, FORTRAN.

There are 3 modes for code capture and updates:

1. For a select few individuals at AFRL access has been granted to the MIT software repositories. These individuals always have access to the most current state of the software and in some cases act as developers (that is, they can commit code to the MIT repositories).
2. Beta source releases are made available when the code-base is functional and stable (mostly "green balls" reported by Jenkins – see Section 2.4). This packaged *tar* image can be found on the MIT **ESP** website, so it can be easily downloaded by anyone who has access to the web. Note that compilation and building is required to use this form of software distribution.
3. Official releases (see Table 3) are made periodically. These are numbered and fully supported. They come in two forms: a source release (which is the same as the Beta, except that it reflects the official release) and fully built software. The PreBuilt distributions (one file for each supported OS) requires no software building, can be installed (under most circumstances) without any system privileges, and sets up a desktop icon that can be double-clicked (for all OSs) that can initiate the appropriate environment and allow full access to the **ESP** modules and components. This is clearly well suited to either novice users or those that are not software savvy.

There is a tension between the desire to continue to improve a large and complex software suite and the ability to get the results of these efforts into the user's hands. Development must stop at some point and the code *frozen* except for bug fixes. The state of the code base needs rigorous testing (beyond the testing done during continuous software integration), which is rather time consuming. This is the nature of the **ESP** test matrix, where the tests are executed on LINUX, MAC and Windows, against various versions of OpenCASCADE and differing compilers.

Building the distribution also requires care and time. It must be tested against all target combinations and needs to work without intervention. And there are now two variants: (1) build from source and (2) pre-built distributions (which are especially useful for Windows without Visual Studio – the compiler).

To ensure that this is done periodically, **ESP** releases have been *cut* before any formal training, so that users are trained on the most recent software available. The trainings are also useful in

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

finding issues in the most up-to-date release and has initiated another release soon after the training (if significant problems were found).

The software release schedule during the course of this contract can be seen in Table 3.

Table 3. Software Releases

ESP Revision	DATE
1.07	July 2015
1.08	October 2015
1.09	August 2016
1.10	September 2016
1.11	June 2017
1.12	December 2017
1.13	May 2018
1.14	December 2018
1.15	May 2019
1.16	August 2019
1.17	January 2020

3.1.10.4 Reporting

Monthly technical and financial reports were delivered on-time throughout the duration of the contract. The technical reports were detailed and contained the information on the on-going research as well as the implementation delivered as functioning source code (see Section 3.1.10.3).

3.2 CAPS Supplement

The tasks listed below reflect the **CAPS** supplement (referred to as *In-Scope Work Modification numbered P00011*) where the technical portion of the contract ran from May 2017 (the end of the original **CAPS** work) to November 2019. This included 4 major thrusts: continued work on geometry and geometry construction for aircraft design, meshing to facilitate an automated workflow, an effort to incorporate *packaging* into **ESP**/**CAPS** and tasks to continue on with the efforts started during the initial phase (Section 3.1).

3.2.1 Geometry

Inside the **CAPS** environment, parametric geometry is generated based on combining a selected number of solids through Boolean operations. This process makes the generation of fully blended aircraft configurations, such as the D-8 or YF-23 very difficult. The objective of this task is to generate parametric methods to construct fully blended aircraft bodies that can maintain a user-defined continuity level. Additionally, methods should be capable of representing aircraft defined previously in a parametric way.

3.2.1.1 BSpline Morphing

Often it is easy to build a parametric model that is close to the desired shape, but which must be adjusted locally to satisfy local shape requirements. The objective of this task is to create tools

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

that allow one to morph a given boundary representation by changing the BSpline control points associated with either a Face, an Edge (and its supporting Faces), or a Node (and its supporting Edges and Faces). This was accomplished at the **EGADS** level by allowing any general surface to be replaced by a BSpline surface and then giving the programmer that ability to move individual BSpline Control Points. This was never elevated to the **OpenCSM** level because it is not clear the best way to have the user control the movement of one or more ganged Control Points.

There are 2 phases that are required for performing this "free form" modification of individual Faces. The first of which is to prepare an **EGADS** Body for the operations, the second is to actually do the shape changes. The assumption is that we will be moving/adjusting control points of a BSpline/NURBS surface and that the surface is used at full extent (or at least trimmed by 4 Edges where the underlying curves are isoclines). This will ensure that we do not open up models that are closed, as long as we don't move the control points at the bounds of the surface. Obviously, there may need to be some scribing done to the Body to prepare for this.

The following documented functions already existed in **EGADS** that let you preform much of the first phase:

```
stat = EG_convertToBSpline(face, &newSurface);
```

which takes an existing Face and generates the BSpline/NURBS equivalent surface trying to preserve the *[u,v]* parameterization. The new surface can be enhanced (see EG_addKnots below) and then made into a new Face by EG_makeTopology. Note that the Loops from the source Face will need to be remade (EG_makeTopology) specifying the new surface, but the Edges themselves do not need to be modified.

```
stat = EG_replaceFaces(body, n, replacements, &newBody);
```

which takes a list of n Face pairs (the original and the replacement), does the Face swapping and generates a new Body with the updated Faces.

The following undocumented function can be used to add "degrees of freedom" to the operation by adding to the knot sequence and therefore providing more control points to adjust:

```
stat = EG_addKnots(surface, nU, Us, nV, Vs, &newSurface);
```

which takes as input the BSpline/NURBS surface, the number of additional knots in the U direction and a vector of new U knot values, the number of additional knots in the V direction and a vector of new V knot values and outputs a new surface that has the same shape of the original but with additional knots/control points.

The following **EGADS** function can be used during the second phase (in the design setting):

```
stat = EG_adjustCPs(body, face, CPs, &newBody, &newFace);
```

```
where: body   the input Body ego
   face   the Face ego to adjust (ref surface must be BSPLINE)
       and must have a single Loop with 4 Edges at IsoClines
   CPs    the control points (the same setup as the data for
```

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

BSPLINEs without the knot information)

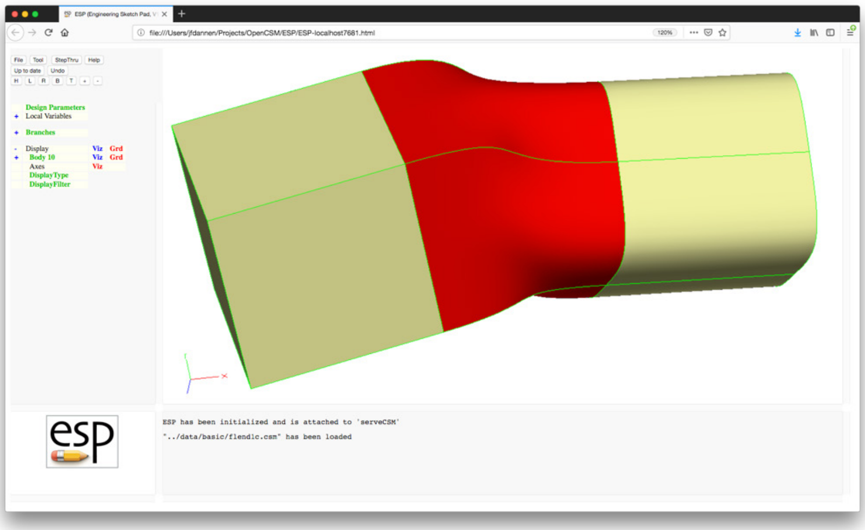
newBody the returned new Body ego

newFace the returned Face in newBody that corresponds to face

Note that this function could be used to simplify some of the first phase setup. Also, it should be noted that to fully control shapes (allow for changes across Edges) the scheme outlined above would need to be modified to include the Face's bounding Edges.

3.2.1.2 Sculpting

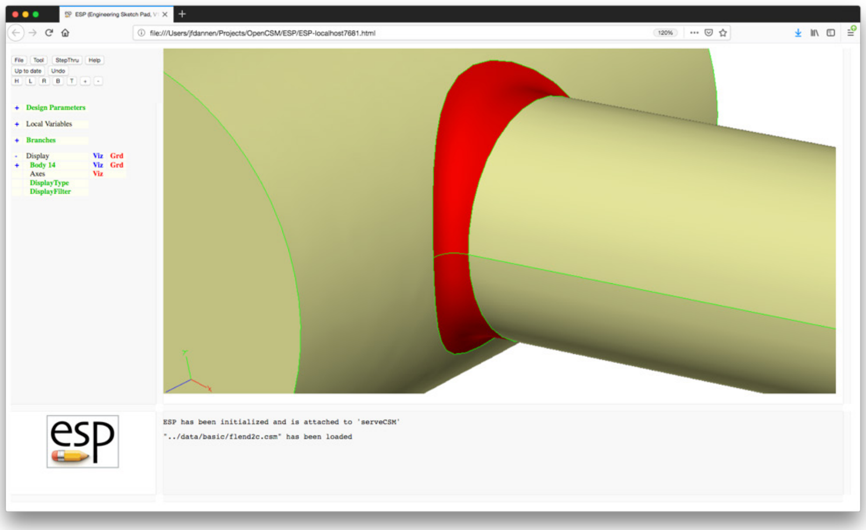
The purpose of sculpting is to produce smooth transitions between various parts of a configuration. The current implementation combines two bodies. This smooth transition is produced with a *Flend* (fillet-like blend) [19], which generates B-spline surfaces that are at least slope-continuous (C^1) at their Edges, and generally almost curvature-continuous (C^2). From an aerodynamics perspective, this "almost C^2" condition is advantageous. Figure 15 through Figure 17 shows examples of *Flends*, where in each figure the *Flend* surfaces are depicted in red. Figure 15 shows a *Flend* between two adjacent bodies, Figure 16 shows a *Flend* at the junction of two bodies, and Figure 17 shows a *Flend* the root of a turbomachinery blade. In each case, all that was required of the user is a set of scribing curves.



```
ESP has been initialized and is attached to 'serveCSM'
"../data/basic/flend1c.csm" has been loaded
```

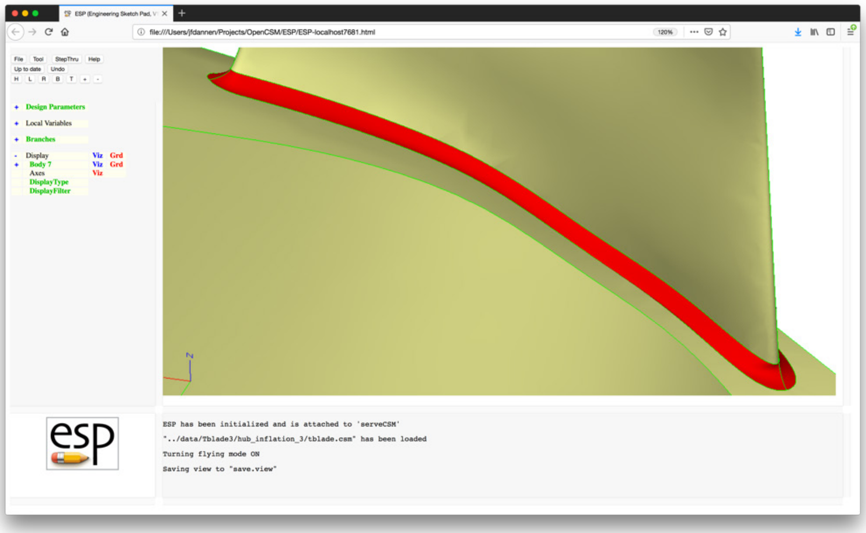
Figure 15. *Flend* as a continuation between 2 Bodies.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



```
ESP has been initialized and is attached to 'serveCSM'
"../data/basic/flend2c.csm" has been loaded
```

Figure 16. *Flend* attaching 2 cylindrical Bodies.



```
ESP has been initialized and is attached to 'serveCSM'
"../data/Tblade3/hub_inflation_3/tblade.csm" has been loaded
Turning flying mode ON
Saving view to "save.view"
```

Figure 17. *Flend* as a fillet replacement.

3.2.1.3 EGADS

See Section 3.1.3.1.

3.2.2 Meshing

The initial **CAPS** capability can generate meshes for structural finite-element and RANS analyses, but with lesser quality than is desired. The first objective of this task is to generate unstructured fully-quadrilateral meshes for structural analysis. The second is to do research into the mesh mechanics required to perform solver-based adaptation. And, the third objective is to create links to industry-standard RANS meshing software so that these packages can be accessed seamlessly through the **CAPS** environment.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3.2.2.1 Surface Quadrilaterals

Most structural solvers provide much more accurate results when given a pure quadrilateral (as opposed to triangle or mixed) mesh as input. At the start of this contract the **EGADS** tessellation subsystem could provide quadrilateral meshes if it can determine the four sides of the Face being handled. The goal of this subtask is to provide a general unstructured quadrilateral meshing scheme that will be watertight and will be consistent with the rest of **EGADS**. This has required the following steps:

- perform a coarse initial triangulation of the Body of interest;
- subdivide all Edge discretizations, which provides an extra vertex along each Edge segment;
- for each Face, subdivide the internal triangle sides and insert a vertex at each triangle centroid, generating 3 quadrilaterals per triangle;
- regularize the mesh by local operations in order to achieve as many valence (the number of quad sides touching a vertex) 4 vertices as possible;
- adjust and/or smooth the resulting vertices supporting the quads (per Face) (in *[u,v]*) in order to drive the angles in each quadrilateral toward 90^o;
- place the final resulting quadrilateral body tessellation into an **EGADS** tessellation object; and
- attribute the tessellation object so that functions that use the object can determine that the Faces have been discretized with unstructured quadrilaterals.

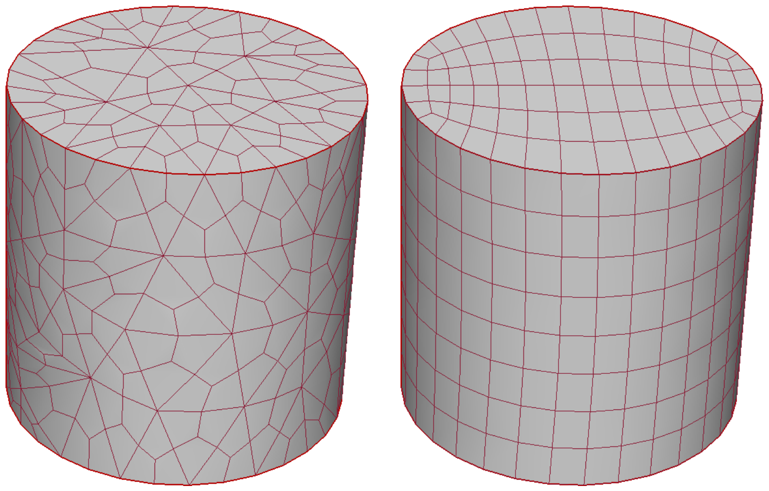


Figure 18. Cylinder at initial quadding and after regularization.

The complete algorithm used is fully described in [20, 21]. Examples on simple shapes can be seen in Figure 18 and Figure 19.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

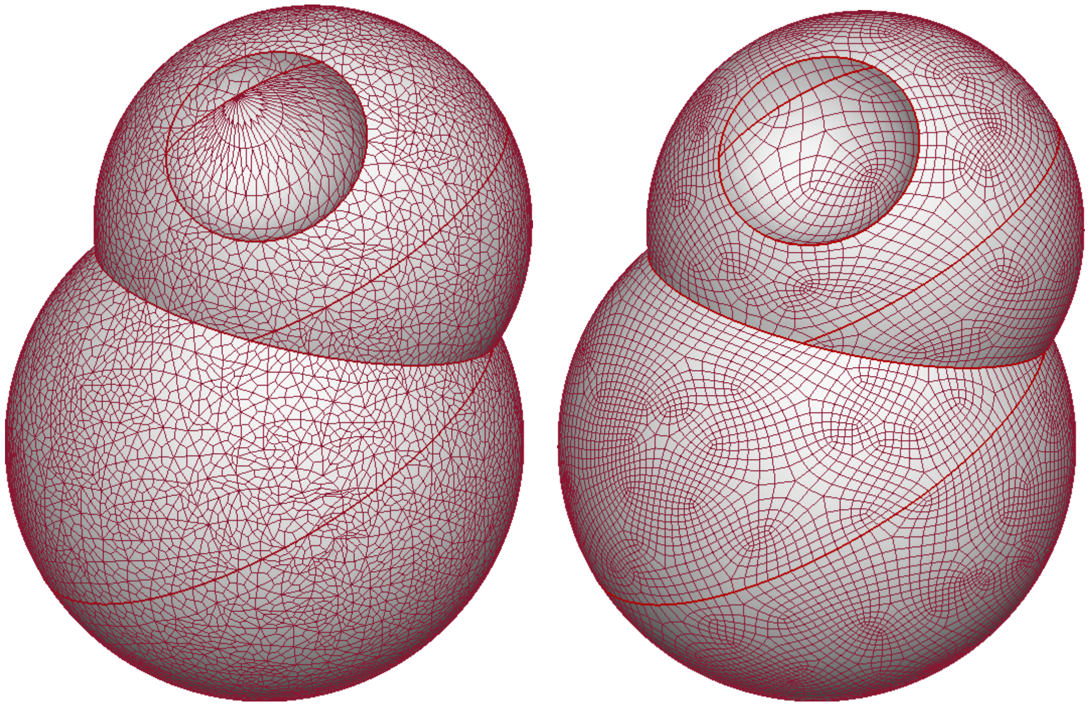


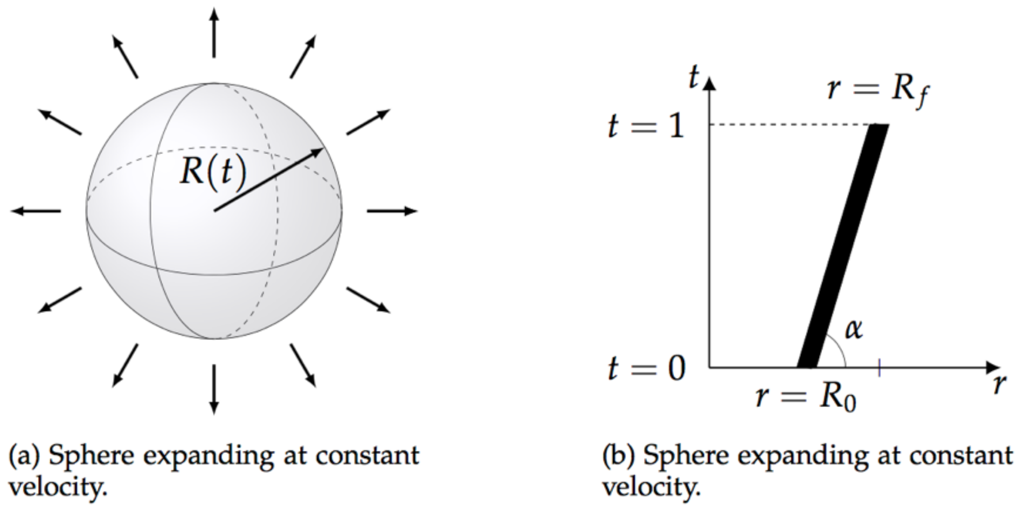
Figure 19. Multi-sphere case at initial quadding and after regularization.

3.2.2.2 Adaptation

RANS meshing is more prone to discretization error when the mesh does not conform to the *features* found in the solution. This has been shown repeatedly in the Drag Prediction Workshops, and is a *chicken-and-egg* problem – how do you generate a good mesh when you don't know the solution, and why would you run the simulation if you knew the answer! This is obviously a problem when you wish to accurately predict a result (such as in a design setting). And because each new mesh (from a new design iteration) may have differing and unknown errors, it is questionable whether a convergent design process exists. In some settings this problem is ignored by morphing an existing mesh and assuming that the errors are related.

In any case, the current situation does not provide a robust ability to do design. To mitigate this problem research into methods for 3D mesh adaptation have been undertaken (in fact the work is in a 4D setting). The mesh can either be driven from the features found in a resultant solution or, more importantly, from an Adjoint solver in which error estimation can be used to deal with the solution-based error directly. The latter case is a more robust way to actually provide error bounds on the solution, which can then be used in a process that can guarantee convergence.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



(a) Sphere expanding at constant velocity. (b) Sphere expanding at constant velocity.

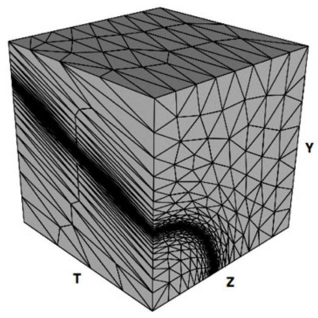
Figure 20. Illustration of a 4D case and expected refinement.

A 4D metric field is modeled after an expanding spherical wave in 3D (see Figure 20a). Consider a spherical wave of radius $R_0 = 0.4$ centered about the origin in 3-space at time $t = 0$. If the wave expands at a constant velocity Vp to a radius $R_f = 0.8$ at time $t = 1$, then the expanding sphere traces the geometry of a hyper-cone in 4D.

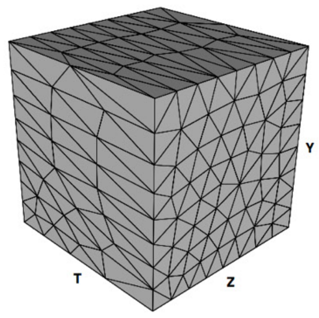
Figure 20b exhibits the behavior of the expanding $(d - 1)$-sphere in a spherical-temporal coordinate system. Note that a slice of the $(d + 1)$-dimensional cone with a hyperplane with non-constant temporal component yields a d-cone. Here, this appears as a line but rotational symmetry implies the hyper-cone sliced by a hyperplane with non-constant temporal component yields a three- dimensional cone. Hence, when extracting the eight cubes bounding the unit tesseract (4D cube), we expect to see three-dimensional cones along hyperplanes with a varying temporal component.

For clarity, all eight 3D meshes bounding the tesseract for this wave case are shown in Figure 21. Note that the expected refinement of the cones are observed along hyperplanes with non-constant temporal component. At $t = 0$ and $t = 1$, the sphere at the initial and final radii are respectively observed. See [22, 23] for a more complete explanation of the algorithms used.

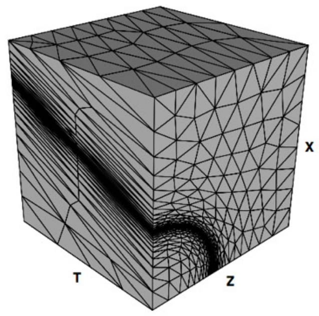
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



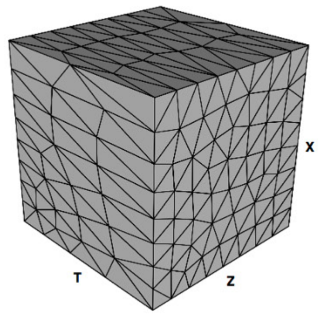
(a) $x = 0$



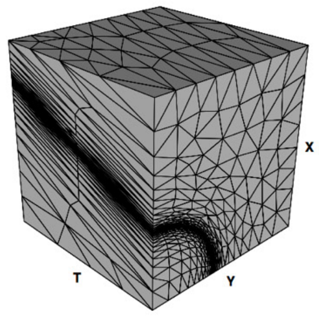
(b) $x = 1$



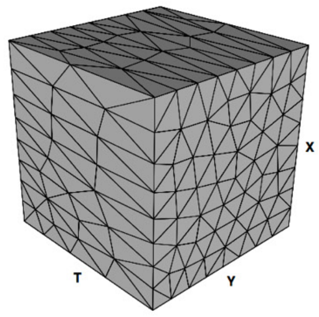
(c) $y = 0$



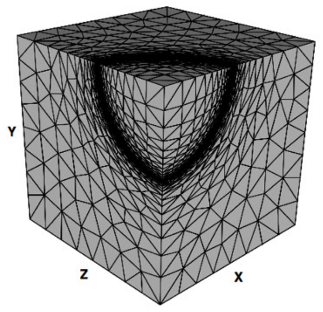
(d) $y = 1$



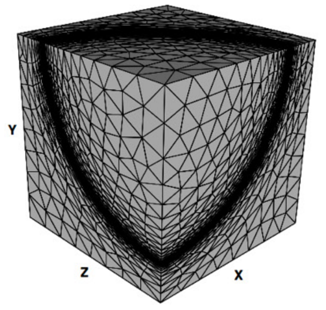
(e) $z = 0$



(f) $z = 1$



(g) $t = 0$



(h) $t = 1$

Figure 21. Meshes of the eight bounding cubes for the 4D adaptation case.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3.2.2.3 Robust Integration of AFLR4

The objective for this subtask was to modify the Advancing-Front/Local-Reconnection (AFLR) surface meshing software, AFLR4, to meet the needs of the **CAPS** system. AFLR meshing software (AFLR3-volume, AFLR4-surface and AFLR2-planar) is widely used, readily available to DoD users, and has been very successful with relevant problems. Furthermore, AFLR4 generated surface meshes are optimal for AFLR3 (volume meshing). This subtask, however, involved only modifications to AFLR4 surface meshing and its integration within an AIM. The intent was to provide a capability within **CAPS** to automatically generate a high-quality surface mesh that is optimal for generation of a mesh using AFLR3 volume meshing, both with and without specified boundary layers.

AFLR4 surface meshing uses the overall AFLR strategy of advancing-front-type point placement, combined with local-reconnection-based connectivity optimization. For surface meshing, a valid mesh (i.e., no folds in the triangulation) is maintained in both the *[u, v]* mapped space and in the physical space. A novel physical space approximation (PSA) is used to eliminate the need for expensive underlying geometry evaluations during mesh generation. The process is as follows:

1. generate a 2D mesh in mapped space;
2. evaluate physical space coordinates using the true geometry at the generated *[u,v]* mapped space coordinates;
3. use the mesh in physical space as a linear approximation (PSA) of the true geometry and regenerate a new mesh in both mapped space and the PSA. All projections and geometric operations in physical space are done with the PSA. Point placement and connectivity optimization is done in PSA;
4. evaluate physical space coordinates using the true geometry at the regenerated *[u,v]* mapped space coordinates. Due to the linear approximation of the PSA, there is a slight perturbation in the coordinate locations. For a reasonable mapping this is never an issue; and
5. for highly distorted mappings, generate a revised *[u, v]* mapping layer. Then return to step 3. One or two iterations of steps 3 through 4 eliminates the impact of distorted mappings.

While the overall framework for AFLR4 is ideal for system integration, substantial additions were needed to develop a robust, fully-integrated and fully-automated version that can produce optimal meshes with no user intervention within the **CAPS** system. Several phases were proposed and implemented to achieve this result.

- Develop and implement a means for automatically specifying the point spacing/length-scale from the given geometry definition of each surface patch. AFLR4 allows for three different methods to specify length scale variation; a boundary driven approach (defined by point spacing on surface patch edges), a background mesh, or a call-back function defining point spacing throughout physical space. The automated process developed herein accounts for surface curvature and proximity of components and is implemented with a multi-pass procedure outlined below:

    1. Generate an initial surface mesh that uses point spacing derived from curvature on edges.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

2. Generate a curvature driven surface mesh that uses a background mesh with length-scales derived from surface curvature. The surface mesh from pass 1 is used to create the background mesh and specify length-scale variation. The resulting mesh appropriately captures the geometric curvature and features of the given surface definitions. For a single body and single component this mesh is the final mesh.
3. If there are multiple bodies or components, then proximity between them is considered. In this discussion a body is a closed set of surfaces and a component is a set of surfaces, e.g. an aircraft with multiple stores may be a single body with multiple components – fuselage, tail, wing, store, struts, etc. An overall volume background mesh is generated from all of the discretized surface meshes (generated on the previous pass). Distance between surfaces is then evaluated by the volume background mesh edges. Length scale is locally reduced if the distance is not sufficient to produce a set number of volume layers and/or support generation of a boundary-layer (BL) region. The surface mesh is then regenerated with the modified background mesh. Multiple sub-passes of this pass are then taken to provide a smooth length scale transition. At completion, the resulting surface mesh is considered the final mesh.

Note that if an existing background mesh derived from a previous solution is available then only steps 2 and 3 are needed. Also, surfaces that are considered far-field surfaces (see next point) are simply discretized with a single bounding-box derived length-scale. The multi-pass procedure described above for other surfaces is driven primarily by a single length scale that should be based on physical information available for the intended application, e.g. wing chord. In addition, BL thickness, if applicable, can be estimated from the physical information. Further control of this procedure is available to the user via **ESP** attribution parameters. However, in general additional meshing control parameters are not required for a suitable mesh.

- Develop an appropriate **ESP** attribution scheme for parameters that control AFLR surface and volume meshing within the **CAPS** system. For surface and volume meshing various user parameters can be set to control the overall meshing process. The following describes the case dependent parameters available:

1. Mesh generation boundary conditions (BC) by Face. Each Face should have an appropriate BC specified via attribute (similar to what is required for the solution process). BC's available include far-field surface, BL generating surface, symmetry plane, curved surface that intersects the BL region (similar to symmetry), embedded/transparent surface, etc.
2. Specification of "components" by Face if desired for proximity-based refinement.
3. Global surface mesh generation parameters include configuration reference length, BL thickness (if applicable), along with numerous optional parameters that are available to adjust the surface meshing. However, these optional parameters are not required or expected to be used by most users and are available primarily for expert power-users that want very specific and unique mesh characteristics. The description of these optional inputs can be found in the AFLR4 documentation.
4. Local surface mesh generation parameters that can be applied by Face include various control parameters for unique mesh characteristics. These include, a local Face scaling parameter to increase/decrease length-scale, Edge mesh refinement for

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

resolution of sharp edges, along with other parameters that are available to adjust the surface meshing locally. Again, these optional parameters are not required or expected to be used by most users and are available primarily for expert power-users that want very specific and unique mesh characteristics.

5. For volume meshing the BCs are passed directly and additional AFLR3 volume meshing parameters can be specified globally or locally.

- Develop documentation and tutorials on AFLR4 usage. Complete web-based documentation on all available parameters along with a tutorial with multiple cases is provided with all AFLR software.
- Develop a process to automatically derive, from the geometry definition, a directional anisotropic metric that specifies the point spacing directionally along with curvature orientation. The automated curvature driven process previously described generates both isotropic and anisotropic metric parameters. However, modification of AFLR4 surface meshing to enable this capability was determined to be outside the scope of the present effort and is saved for future work.

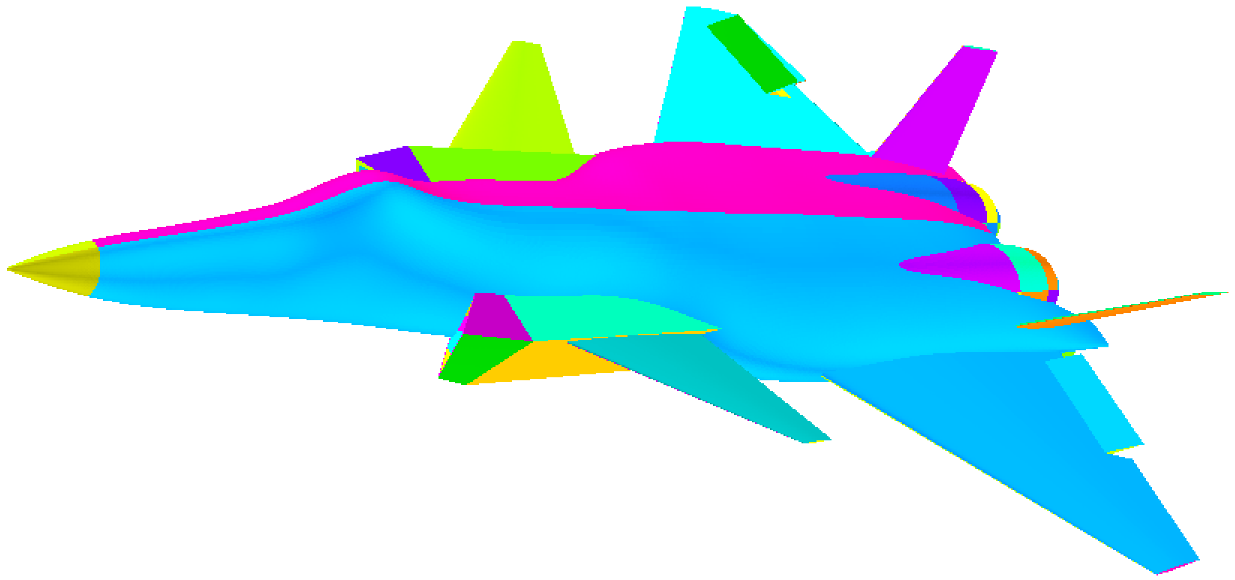


Figure 22. Fighter body configuration for AFLR example.

Some example cases are presented in the following discussion to illustrate the current capability. All cases are automatically generated by AFLR4 with length scale and BL thickness specified. The first case is that of a fighter aircraft. In this case the impact of Edge discontinuity refinement is compared. The overall configuration is shown in Figure 22. One view of the resulting surface mesh with and without Edge discontinuity refinement is shown in Figure 23. All of the sharp edges are fully refined in the case of Edge refinement. An additional close-up view is shown in Figure 24, and note the refinement along the flap (green surface). Typical CFD simulations of such configurations require refinement of sharp geometric features in addition to curvature.

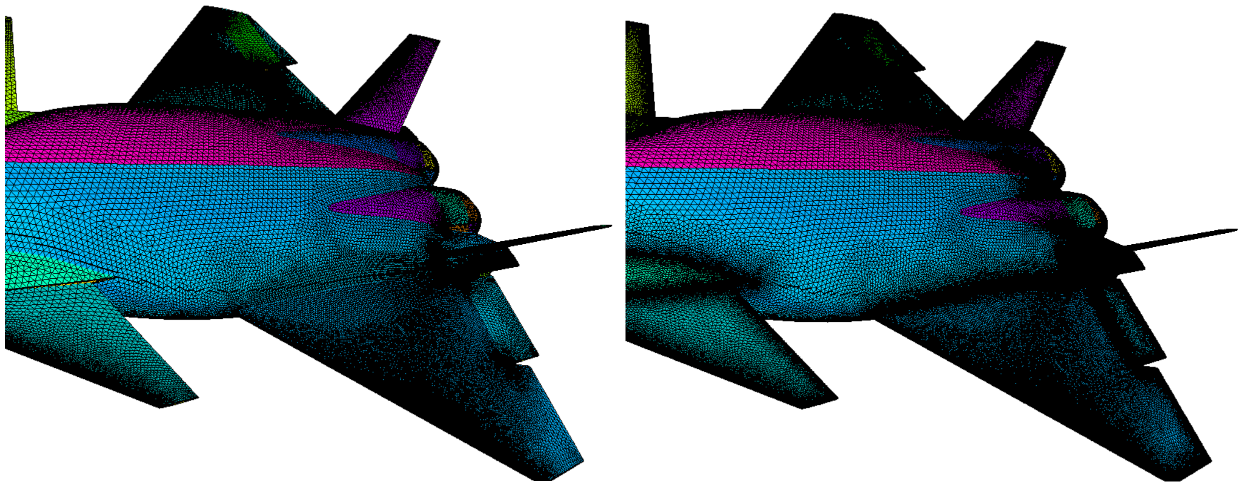


Figure 23. Fighter surface mesh without (left) and with (right) discontinuous Edge refinement.

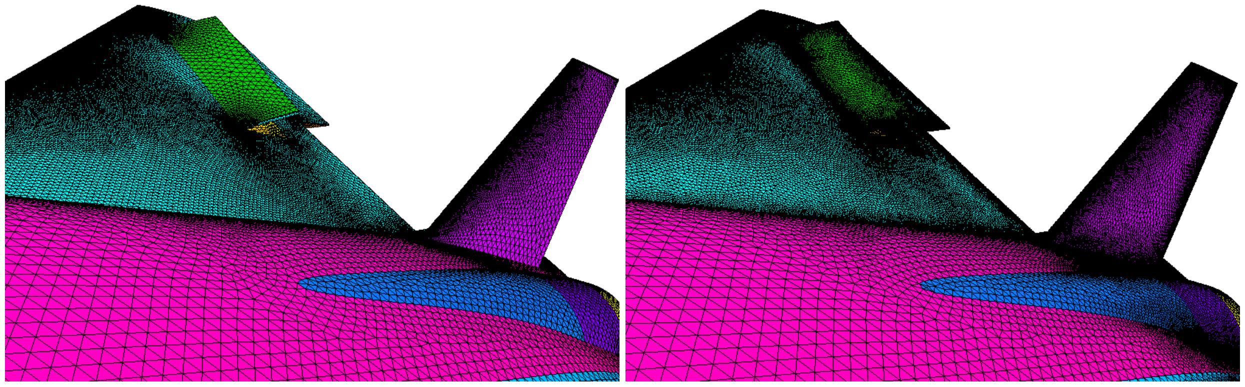


Figure 24. Close-up of fighter surface mesh without (left) and with (right) discontinuous Edge refinement.

The second case is that of a launch vehicle with strap-on boosters shown in Figure 25. Two views of the resulting surface mesh are shown in Figure 26. As shown both surface curvature and proximity regions are refined. An AFLR3 volume mesh with BL region was generated using this surface mesh. Figure 27 shows two views of the volume mesh field cut. Refinement in the region between the main and strap-on boosters allows full resolution of the BL region.

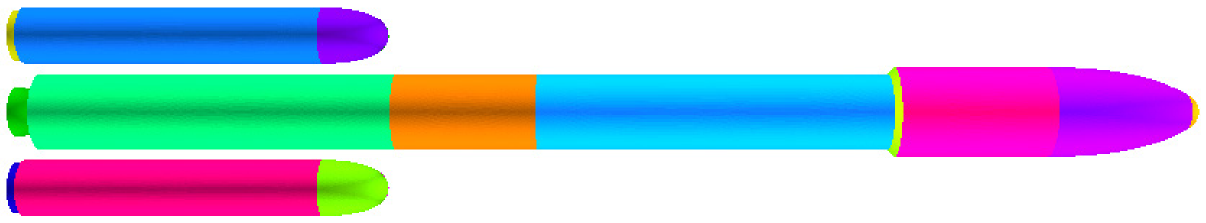


Figure 25. AFLR launch vehicle test configuration.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

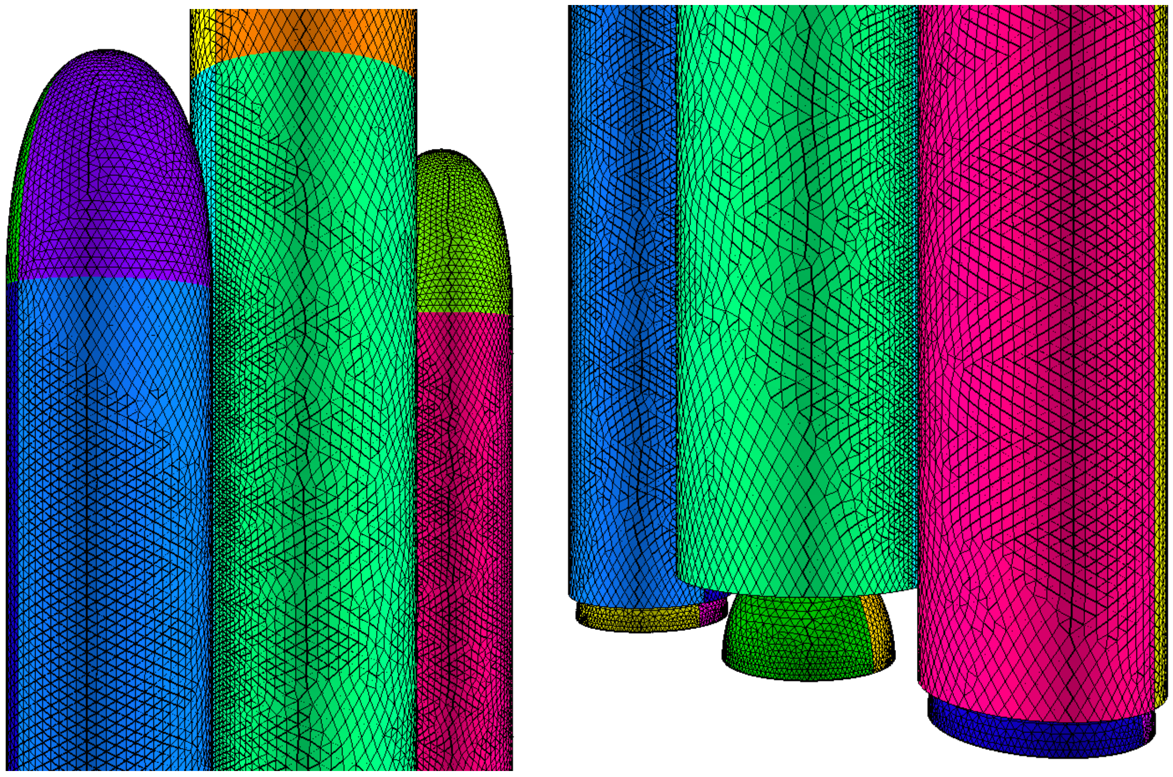


Figure 26. Launch vehicle surface mesh views showing proximity refinement between main and strap-on boosters.

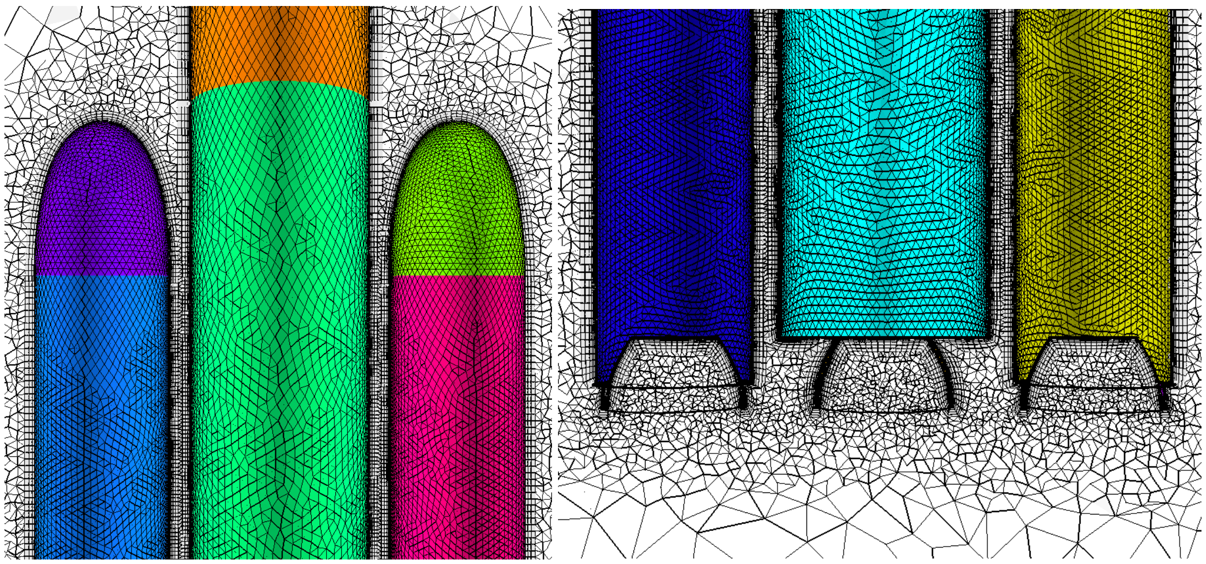


Figure 27. Launch vehicle volume mesh cut views showing refinement and BL region between main and strap-on boosters.

The third case is that of a jet engine nacelle. This case has multiple bodies/components that have several regions in close proximity. Two views of the overall configuration are shown in Figure 28.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

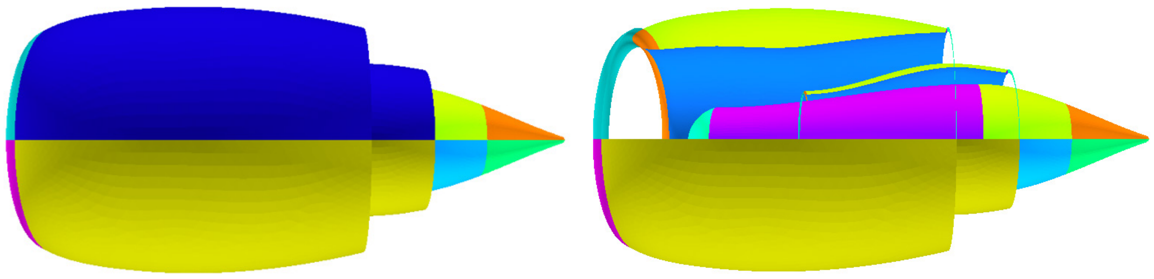


Figure 28. AFLR jet engine nacelle test configuration.

The resulting surface mesh with both curvature and proximity refinement is shown for two views in Figure 29.

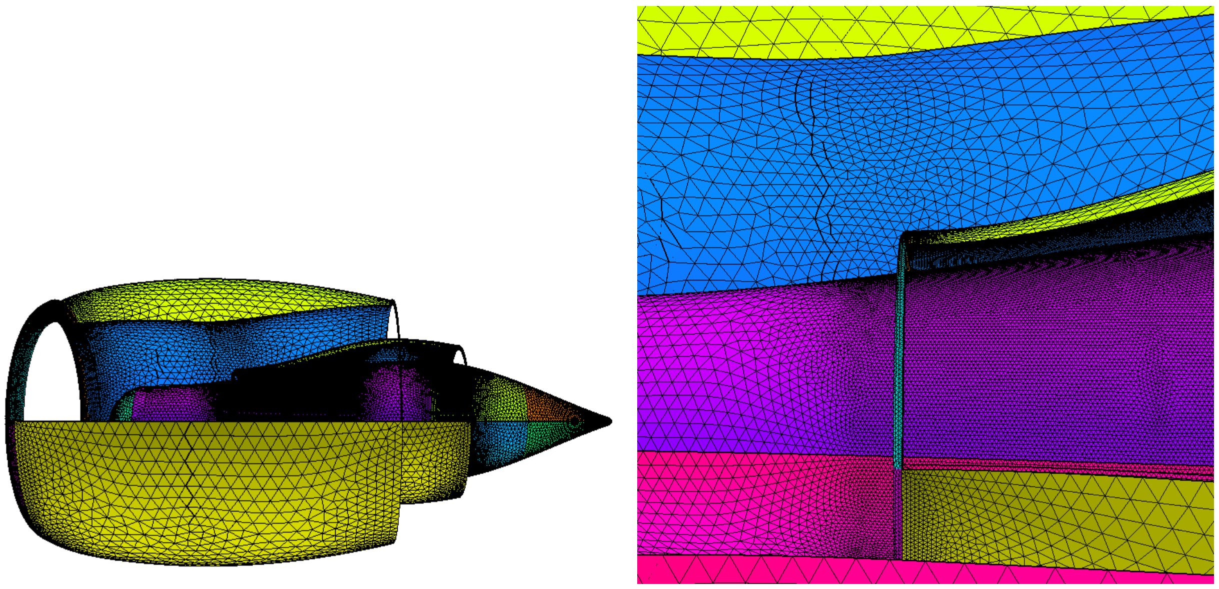


Figure 29. Nacelle surface mesh views showing proximity refinement between components.

An AFLR3 volume mesh with BL region was generated using this surface mesh. Figure 30 shows the volume mesh field cut. Refinement in the region between the components allows full resolution of the BL region. Two additional views of the volume mesh cut are shown in Figure 31.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

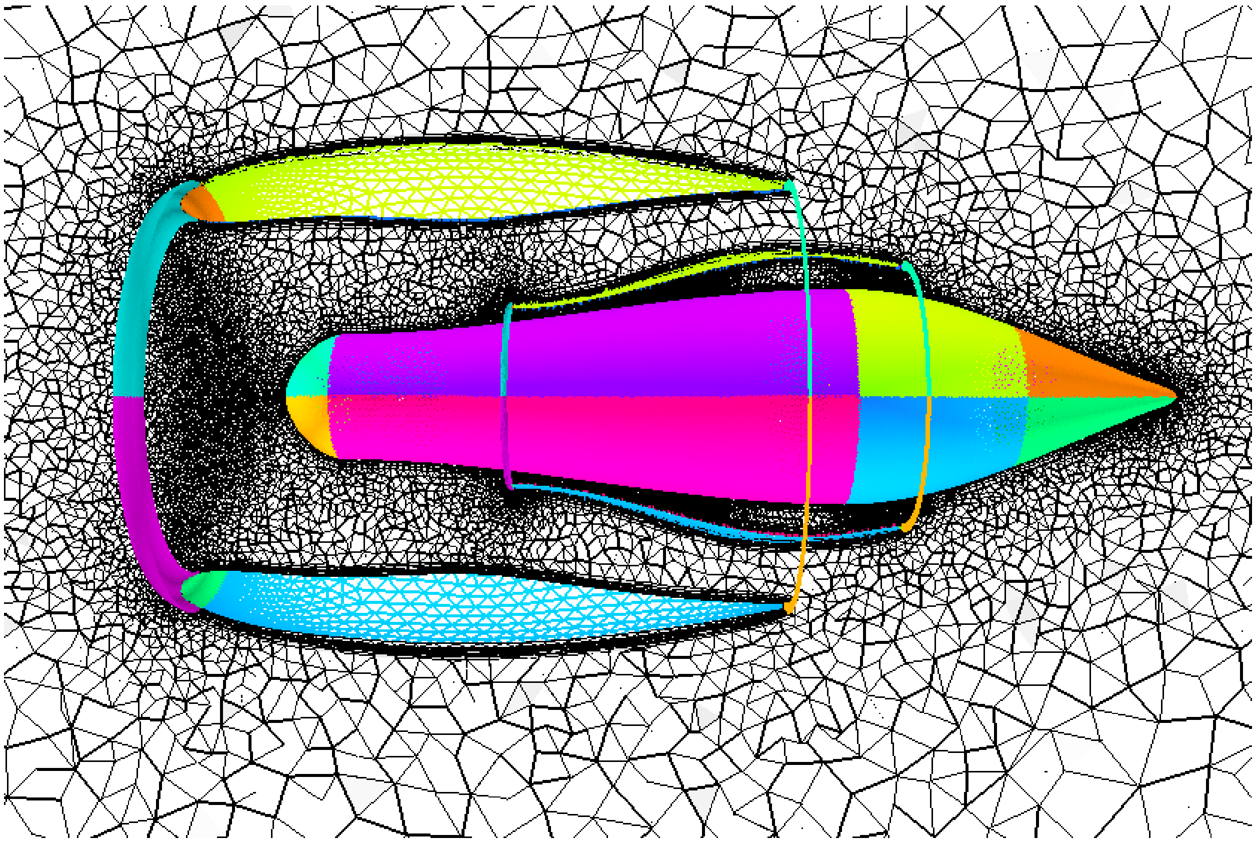


Figure 30. Nacelle mesh cut showing refinement and BL region between components.

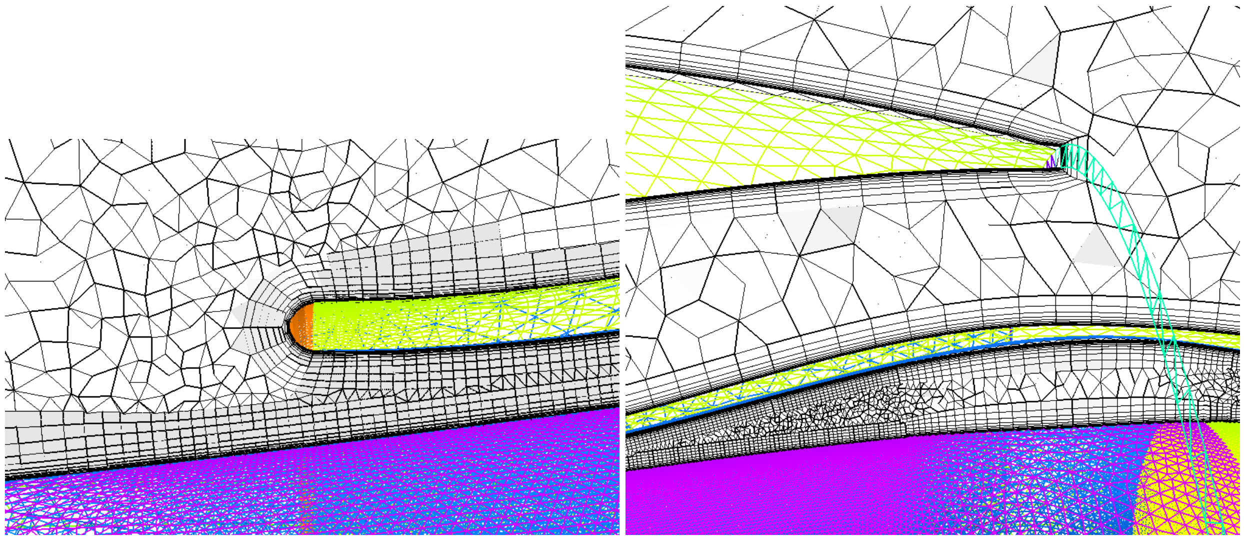


Figure 31. Closeup of nacelle volume mesh cut showing refinement and BL region between components.

3.2.2.4 Pointwise Automation

The mesh generation software **Pointwise** and its scripting language **Glyph** were used to create a system for automatically generating unstructured meshes given a closed, watertight geometry.

The system, called GeomToMesh, was developed to work with attributed geometry created in **ESP**. Three subtasks were identified to enable this automated capability.

The first subtask required modification of the Pointwise software to allow import of **ESP** geometry. This was initially performed with an external program that converted **ESP** geometry file in **EGADS** format into the native Pointwise NMB file format. The newly created NMB file was then read by Pointwise for processing. Eventually this external program was incorporated into the Pointwise software enabling import of **EGADS** files directly. Additional modifications to Pointwise were made to store the attributed information for each geometry entity internally. New Glyph script function calls were also created permitting the scripts to access this attributed data on the geometry surfaces, curves and points.

The second subtask developed the language or schema that communicated the meshing instructions from the attributed geometry to the Glyph scripts. The attributed data on the geometry contained key-value pairs that were recognized by the Glyph scripts. Figure 32 shows a sample of many of the key-value pairs read by the GeomToMesh script during the meshing process. Many of these key names mirror the meshing parameters exposed in the GUI to an interactive user of Pointwise. If no attribution information is provided the scripts will attempt to generate an isotropic unstructured tetrahedral mesh given a closed geometry input file. With attribution the system will produce a mesh more closely aligned with the user's intent. The more commonly used attributes are the naming functions and specifying normal wall spacing values on surfaces associated with viscous boundary conditions.

The third subtask was to develop and evolve the system of Glyph scripts, known as GeomToMesh, to import the attributed **ESP** geometry and generate a completed unstructured volume mesh ready for flowfield analysis. These scripts are completely general in the sense that no assumptions are made with respect to the configuration shape or purpose. An additional Glyph file containing other meshing parameters can be provided by the user to further control the meshing process. These parameters have default values that are loaded at startup. Any parameters provided by this user file override the default values.

Using the GeomToMesh system simply involves providing the **ESP** geometry file and the optional user parameter file to the scripts run in Pointwise. The system can be executed through the GUI or can be run in batch mode on the command line. A detailed description of the processing and capabilities can be found in [24].

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Key	Value	Geometry Location	Description
	Preceding $ means it is a character string		
PW:Name		Face	Boundary name for domain or collection of domains.
PW:QuiltName		Face	Name to give one or more quilts that are assembled into a single quilt. No angle test is performed.
PW:Baffle	$Baffle or $Intersect	Face	Either a true baffle surface or a surface intersected by a baffle.
PW:DomainAlgorithm	$Delaunay, $AdvancingFront, $AdvancingFrontOrtho	Face	Surface meshing algorithm.
PW:DomainIsoType	$Triangle, $TriangleQuad	Face	Surface cell type. Global default is Triangle.
PW:DomainMinEdge	$Boundary or > 0.0	Face	Cell Minimum Equilateral Edge Length in domain.
PW:DomainMaxEdge	$Boundary or > 0.0	Face	Cell Maximum Equilateral Edge Length in domain.
PW:DomainMaxAngle	[0, 180)	Face	Cell Maximum Angle in domain (0.0 = NOT APPLIED)
PW:DomainMaxDeviation	[0, infinity)	Face	Cell Maximum Deviation in domain (0.0 = NOT APPLIED)
PW:DomainSwapCells	true or false	Face	Swap cells with no interior points.
PW:DomainQuadMaxAngle	(90, 180)	Face	Quad Maximum Included Angle in domain.
PW:DomainQuadMaxWarp	(0, 90)	Face	Cell Maximum Warp Angle in domain.
PW:DomainDecay	[0, 1]	Face	Boundary decay applied on domain.
PW:DomainMaxLayers	[0, infinity)	Face	Maximum T-Rex layers in domain.
PW:DomainFullLayers	[0, infinity)	Face	Number of full T-Rex layers in domain. (0 allows multi-normals)
PW:DomainTRexGrowthRate	[1, infinity)	Face	T-Rex growth rate in domain.
PW:DomainTRexType	$Triangle, $TriangleQuad	Face	Cell types in T-Rex layers in domain.
PW:DomainTRexIsoHeight	> 0.0	Face	Isotropic height for T-Rex cells in domain. Default is 1.0.
PW:WallSpacing	> 0.0	Face	Viscous normal spacing for T-Rex extrusion.
PW:TRexIsoHeight	> 0.0	Model	Isotropic height for volume T-Rex cells. Default is 1.0.
PW:TRexCollisionBuffer	> 0.0	Model	T-Rex collision buffer. Default is 0.5.
PW:TRexMaxSkewAngle	[0, 180]	Model	T-Rex maximum skew angle. Default 180 (Off)
PW:TRexGrowthRate	[1, infinity)	Model	T-Rex growth rate.
PW:TRexType	$TetPyramid, $TetPyramidPrismHex, or $AllAndConvertWallDoms	Model	T-Rex cell type
PW:BoundaryDecay	[0, 1]	Model	Volumetric boundary decay. Default is 0.5.
PW:EdgeMaxGrowthRate	[1, infinity)	Model	Volumetric edge maximum growth rate. Default is 1.8.
PW:MinEdge	$Boundary or > 0.0	Model	Tetrahedral Minimum Equilateral Edge Length in block.
PW:MaxEdge	$Boundary or > 0.0	Model	Tetrahedral Maximum Equilateral Edge Length in block.
PW:ConnectorMaxEdge	> 0.0	Edge	Maximum Edge Length in connector.
PW:ConnectorEndSpacing	> 0.0	Edge	Specified connector endpoint spacing.
PW:ConnectorDimension	> 0	Edge	Specify connector dimension.
PW:ConnectorAverageDS	> 0.0	Edge	Specified average delta spacing for connector dimension.
PW:ConnectorMaxAngle	[0, 180)	Edge	Connector Maximum Angle. (0.0 = NOT APPLIED)
PW:ConnectorMaxDeviation	[0, infinity)	Edge	Connector Maximum Deviation. (0.0 = NOT APPLIED)
PW:NodeSpacing	> 0.0	Node	Specified connector endpoint spacing for a node.

Figure 32. Sample key-value pairs recognized by the GeomToMesh scripts.

Additional capabilities were incorporated into the scripts since the paper was presented. These include exporting geometry-to-mesh associativity data used by **ESP**, enabling the use of Point Cloud Datasets for performing mesh adaptation and enhanced geometry feature detection techniques that can recognize high curvature regions and convex/concave Edges. The scripts are used routinely to generate meshes for many different configurations. The automation afforded by the scripts allows a user to create a complete mesh sequences for a grid convergence study, such as the two meshes shown in Figure 33 and Figure 34 for an upcoming AIAA workshop on high-order CFD methods.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

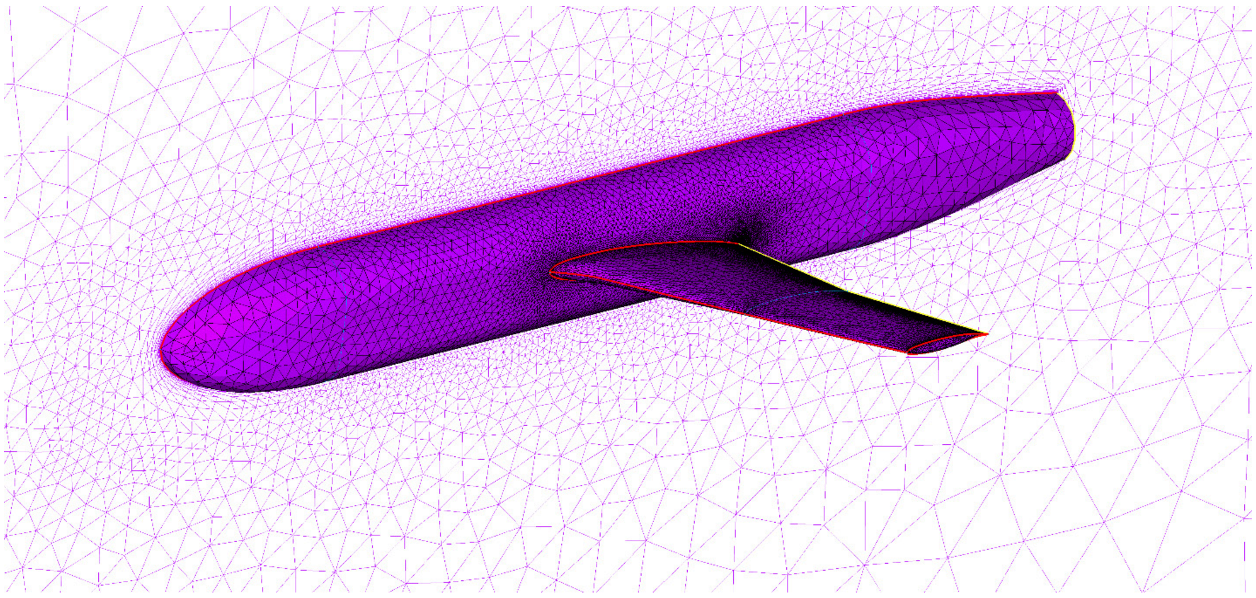


Figure 33. First mesh in the Juncture Flow Model mesh series.

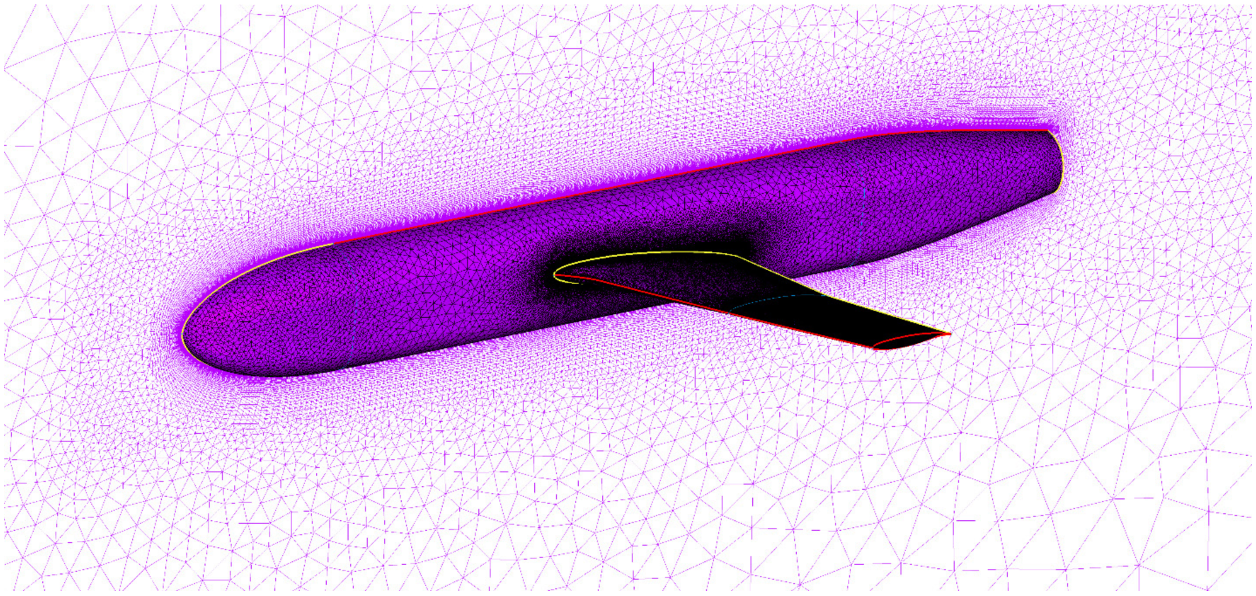


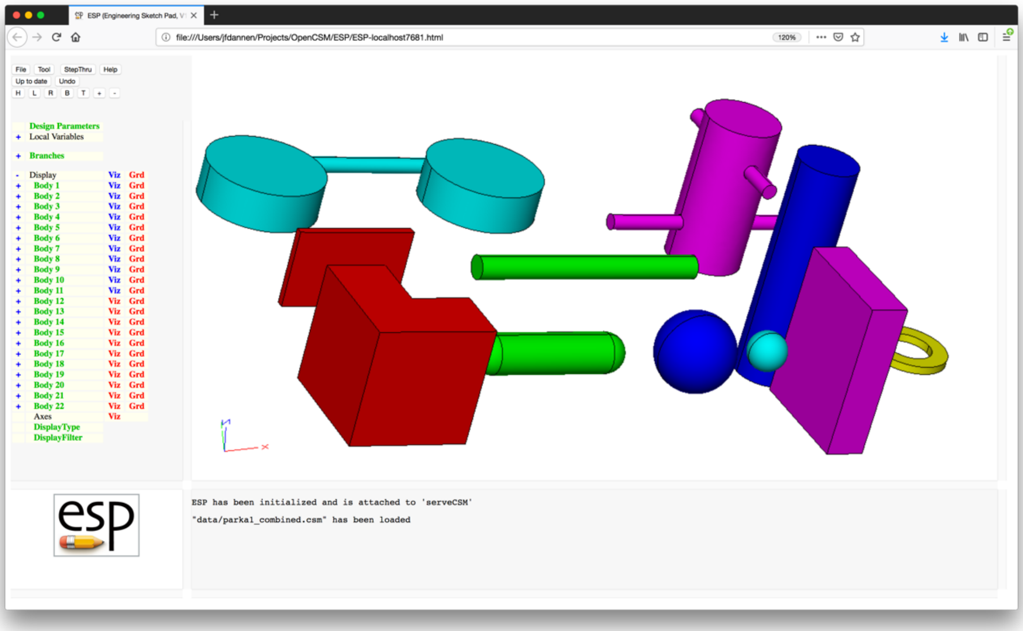
Figure 34. Eleventh mesh in the Juncture Flow Model mesh series.

The GeomToMesh system has been posted on GitHub for any Pointwise user to download. It is also included in the **ESP** distribution.

3.2.3 Packaging

The goal of the packaging application is to determine the location and orientation of a set of given components (here called “packages”) so as to use up the least volume in a prescribed outer container. Figure 35 shows an example set of 11 packages that are to be placed in a rectangular box with the minimum volume.

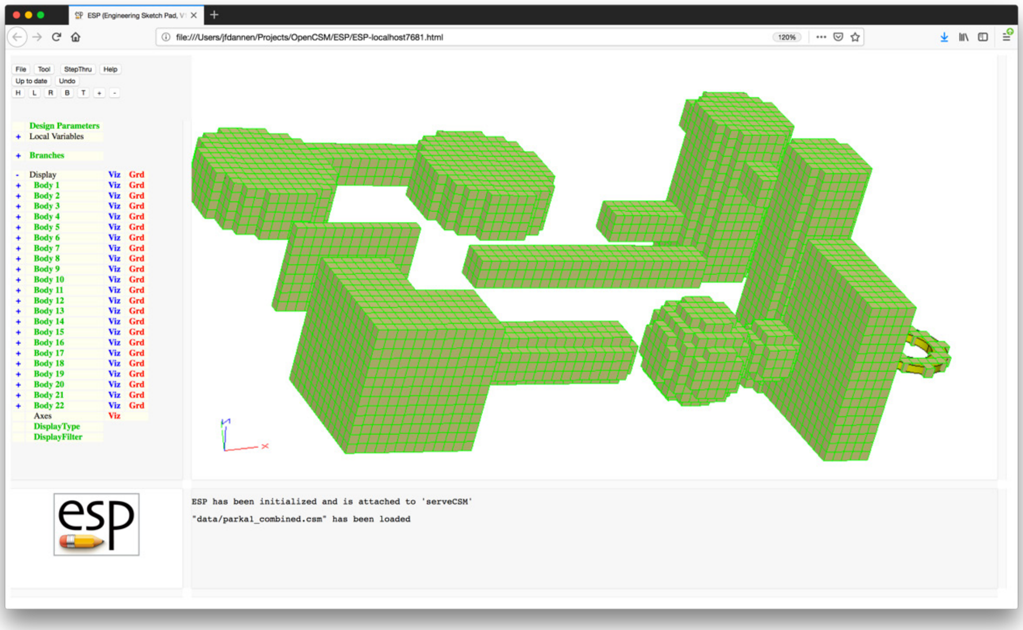
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



```
ESP has been initialized and is attached to 'serveCSM'
"data/parkal_combined.csm" has been loaded
```

Figure 35. Initial "Packages" to place in a minimal box.

The packaging application performs this optimization using a genetic algorithm. To do this, a voxelated version of each of the packages is created, as shown in Figure 36. This voxelated representation forms a "skin" around each package.



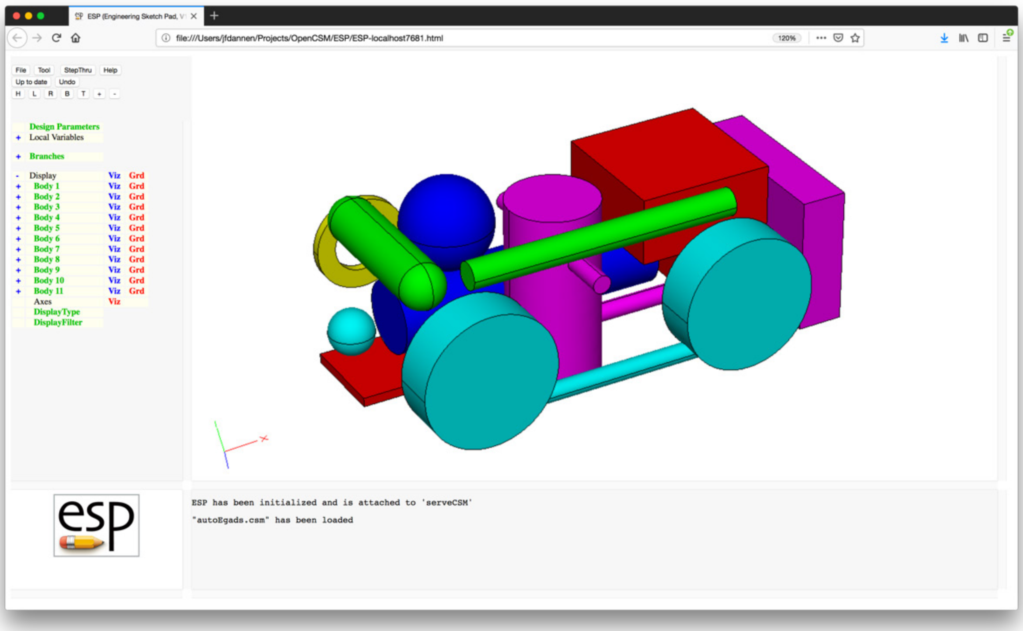
```
ESP has been initialized and is attached to 'serveCSM'
"data/parkal_combined.csm" has been loaded
```

Figure 36. Voxelated representation of "Packages"

The genetic algorithm picks a packing order that the packages should be placed onto a background grid, using a *greedy* (iterative, locally optimal) algorithm for each placement. For each placement, up to 24 package orientations are considered. The "fitness" associated with each packing order is the necessary outer volume size.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

When complete, the genetic algorithm produces several configurations that could be used as the starting points for another optimizer. Figure 37 shows one such packaging for this problem. Note that there is a small buffer around each package, due to the granularity in the voxelation process. The user must choose this granularity as a balance between packing tightness and computational speed.



```
ESP has been initialized and is attached to 'serveCSM'
"autoEgads.csm" has been loaded
```

Figure 37. A minimal packing configuration for the Packages seen in Figure 35.

The "packages" are then tessellated and an interference computation is performed on the discrete geometry via pairs of packages. The distance and derivatives of the translation and Euler angles (6 values per pair) are computed. This will be used to do a gradient-based optimization from the number of seed points made available via the genetic algorithm. The appropriate gradient-base optimization scheme has yet to be found. This requires a global optimizer that can take as input the interference data from the pairs (as well as the derivatives for movement) and move the packages, constrained by avoiding interference, to satisfy some objective function such as minimal volume and/or the center of gravity at some point in space.

3.2.4 Support and Continuation Tasks

The final task involves user support to the Air Force in the use of **CAPS** and those subtasks from **CAPS** Phase #1 that include long-term maintenance, require integration, user testing, and/or have been deferred.

As the number of **ESP**/**CAPS** users grows, there is a continual set of requests for expanded capabilities (new commands), training, documentation, and general user support to ensure that AFRL personnel use the tools in the most productive manner. The objective of this task is to provide the continued user support in order to maximize the effectiveness of AFRL personnel in the use of the **ESP** environment and its various tools.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3.2.4.1 OpenCSM

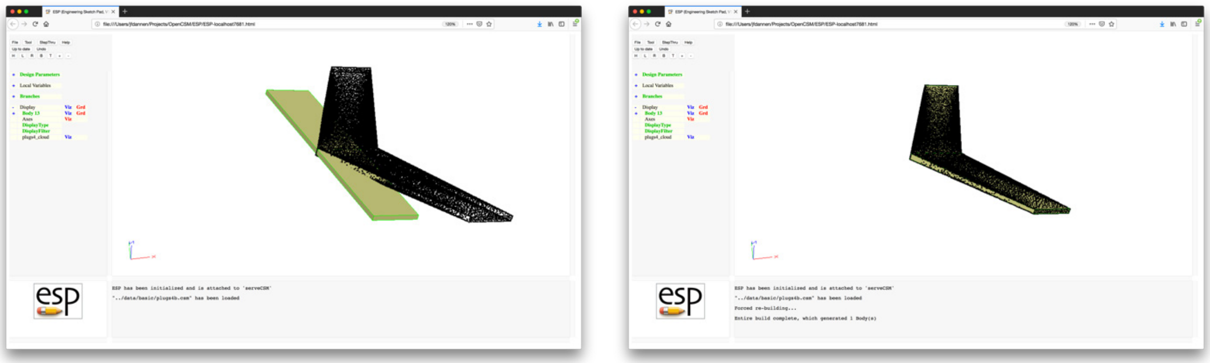
See Section 3.1.3.2.

3.2.4.2 OverSet Meshing

See Section 3.1.7.1.

3.2.4.3 PLUGS

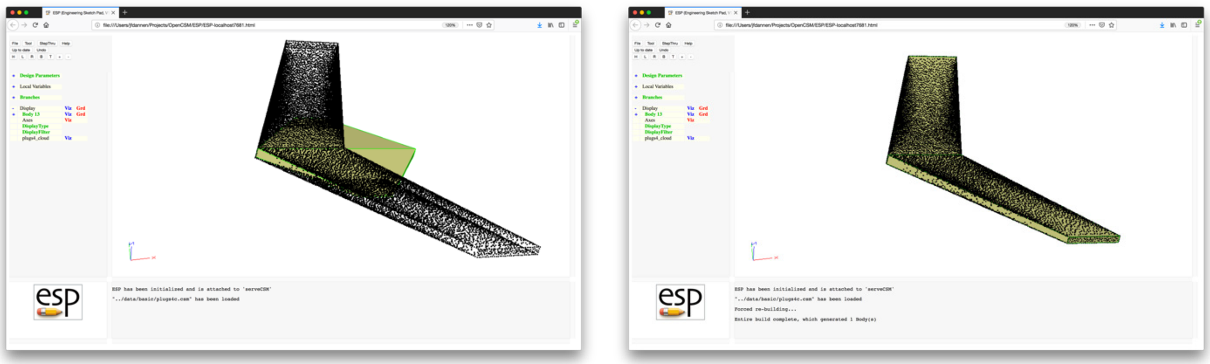
The process described in section 3.1.4.2 for PLUGS was reprogrammed and included within **CAPS**. The major activity here was sanitizing the code base so as to be easy to maintain and is thoroughly tested. The results of this integration are shown in Figure 38 and Figure 39. The configuration is a wing, with the initial "guess" parameter values (the yellow expressed geometry) and point clouds (black points in space) shown on the left-hand side of the figures and the final optimized fits shown on the right-hand side. Note that for this case, PLUGS can start from a very poor initial guess and produce good results.



```
ESP has been initialized and is attached to 'serveCSM'
"../data/basic/plugs4b.csm" has been loaded
```

```
ESP has been initialized and is attached to 'serveCSM'
"../data/basic/plugs4b.csm" has been loaded
Forced re-building...
Entire build complete, which generated 1 Body(s)
```

Figure 38. PLUGS start and best fit for a wing case.



```
ESP has been initialized and is attached to 'serveCSM'
"../data/basic/plugs4c.csm" has been loaded
```

```
ESP has been initialized and is attached to 'serveCSM'
"../data/basic/plugs4c.csm" has been loaded
Forced re-building...
Entire build complete, which generated 1 Body(s)
```

Figure 39. PLUGS wing case from another start parameterization.

3.2.4.4 FPC/HSM/IBL

Simulation efficiency for design can be improved through the use of medium-fidelity (instead of high-fidelity) analysis techniques. A Full-Potential (FP) method formulated on the full outer mold line (OML) geometry definition is arguably a medium fidelity aerodynamic model in-between Vortex-Lattice and Euler/RANS methods. Simulations with the Full-Potential Code (FPC) can be orders of magnitude faster than RANS, and will thus allow a corresponding increase in the number of design iterations which can be performed on any given project. The FP

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

formulation includes the effects of frame motion and supports stability and control derivatives, thus allowing rapid flight-dynamics analyses and control law development for rigid aircraft. The overall objective is to obtain a framework for relatively rapid development of air vehicle configurations, similar to the existing AVL [9] and ASWING [25] methods, but with full-OML geometric fidelity and the ability to handle transonic and supersonic flows.

A preliminary implementation of the 3D Full-Potential solver with the addition of static and dynamic stability derivatives already exists in the MIT Solution Adaptive Numerical Simulator (SANS) [26, 27] framework. The solver has been developed in a modern C++ framework leveraging templates and template-based automatic differentiation to provide development flexibility and while maintaining performance comparable to highly tuned equivalent software written in FORTRAN. The solver framework can utilize both shared and distributed memory parallelism, although the typical size of FPC cases tends not to warrant distributed parallelism.

The Full-Potential formulation also allows an opportunity to flexibly expand the simulation capabilities to include viscous modeling, structural modeling, and geometric deformations associated with design modes and/or control surfaces. Initial implementations for a 3D integral boundary layer (IBL) and the HSM structural shell model (see Section 3.1.5) exist within the same code base as the current Full-Potential solver. The FPC, IBL and HSM modules could be strongly coupled in a Newton-based nonlinear solver formulation. Coupling between the FPC, IBL and HSM will use wall transpiration on the fixed baseline geometry to model both the surface's displacement resulting from aeroelastic effects, as well as viscous displacement effects. Virtual geometric displacements resulting from control-surface deflections and geometry design modes can be modeled by the same transpiration formalism. Simulation results indicate that the transpiration method can accurately model surprisingly large geometry deformations, and can also be used to capture aeroelastic deformations and viscous-displacement effects. This formalism makes FPC even more economical in a design setting compared to RANS, since with the latter approach transpiration cannot be used and remeshing is always required to capture any geometry changes.

Full-Potential. Full-Potential solvers have been successfully developed by a number of researchers using finite-volume schemes and finite-element methods. Similar to Vortex-Lattice methods, FP formulations solve irrotational inviscid flows, but better account for compressibility effects and are suitable for capturing transonic flows with relatively weak shocks. While more costly than Vortex-Lattice methods, FP solutions can be computed on the order of seconds to minutes. The FP formulation also does not rely on small-disturbance assumptions, and as a result requires the definition of an OML. However, since numerical errors in FP solutions tend to be localized, details such as the rounding of wing tips and wing-body fairings do not significantly impact the overall solution. Thus, OML designs that lack complete geometric detail can still be analyzed with some confidence. Finally, the complete Jacobian which is used as part of a Newton method for solving the FP equations also provides the means to compute stability and control derivatives directly via only back-substitutions, as can be done with Vortex-Lattice methods. Hence, a complete set of derivatives can be evaluated for any point solution in the flight envelope without resorting to multiple or unsteady simulations, or constructing reduced order models

Two versions of the potential formulation are currently implemented in the SANS code base: an incompressible potential formulation, and a two-field compressible Full-Potential

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

formulation. The incompressible potential and compressible FP formulations are solved using a continuous Galerkin finite-element method, following well established methods. New formulations (yet to be published – see Appendix D) of the Kutta and wake boundary conditions have recently been developed that allow for adjoint consistency and higher order methods. In contrast, all potential formulations in the literature fail to produce well defined adjoint Kutta and wake boundary conditions. As a result, FP solvers based on these conventional formulations are restricted to linear potential approximations within elements; they also do not lend themselves easily to output-based adaptation methods.

Compressible Full-Potential formulations applied to transonic flows must account for the presence of shocks. Conventional approaches use some variant of density or mass-flux up-winding in supersonic regions to eliminate expansion shocks. Finite volume or finite difference implementations of up-winding either result in first order discretization in supersonic regions or use extended stencils. To provide for density up-winding while preserving a nearest-neighbor stencil, we adopt a two-field approach. The equivalent FEM formulation solves for density and potential separately using two weighted residuals: one for mass conservation and another for the density-velocity relation.

Potential flow formulations require wake sheets to model vortical flow. Consistent with classical FP implementations, the current formulation requires airfoil shapes with sharp trailing edges where wakes originate. Wake sheets are currently generated as part of the geometric build process with limited user input.

Currently mesh generation is performed internally in the **CAPS** framework using the attributed BRep geometric definition and global meshing parameters as inputs. The BRep surface mesh is generated with AFLR4, and volume mesh generation uses either AFLR3 or TetGen.

While AFLR and TetGen provide means of generating meshes in an automated way, both mesh generators only generate isotropic grids. However, a significant portion of the potential solution surrounding wakes is anisotropic. That is, there is large variation in the potential field in the spanwise direction of the wake which requires fine resolution to capture. However, there is little variation in the potential in the streamwise direction. Unfortunately, using isotropic grids results in excessive resolution in the streamwise direction, producing unnecessarily long runtimes for the analysis. This problem is mitigated by the use of the adaptive meshing work described in Section 3.2.2.2 – see Figure 40, which shows a simple wing and wake sheet before and after adaptation.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

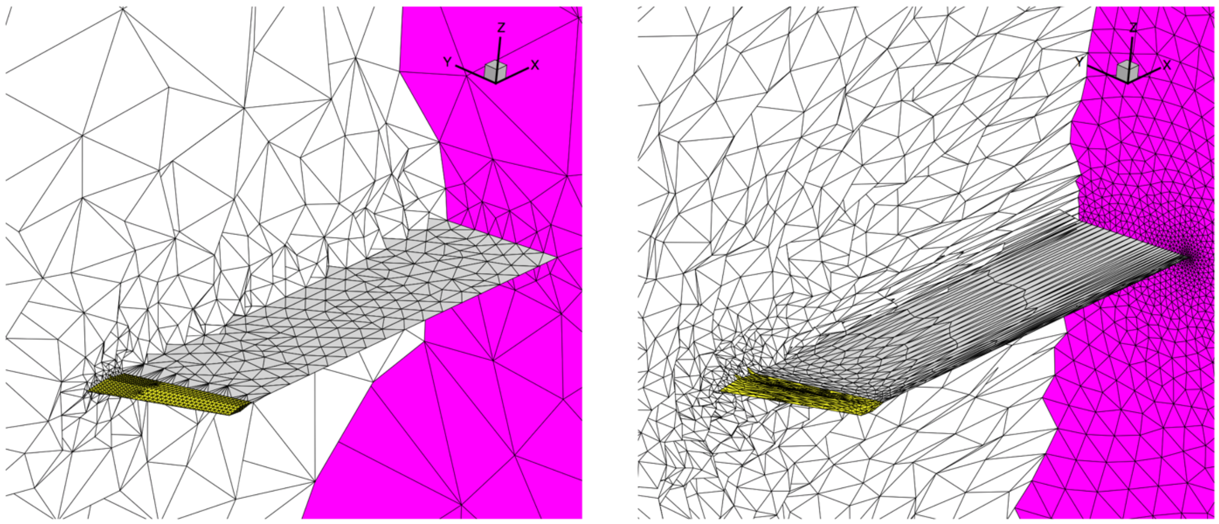


Figure 40. Crinkle cut of linearized incompressible potential initial and adapted grids.

<u>Integral Boundary Layer</u>. This NASA funded project explores the viscous/inviscid zonal formulations together with strongly-coupled solution methods that have proven to be extremely effective in rapid viscous analyses of 2D aerodynamic flows. 2D example applications are MSES [28] and XFOIL [15] both from Prof. Mark Drela of MIT. The relative robustness of the strong-coupling method, and also its ability to handle limited flow separation, both stem from its simultaneous solution of the viscous and inviscid equations as a fully-coupled system via a global Newton method.

Numerous 3D integral boundary layer formulations have been developed in the past. All these methods were formulated in curvilinear coordinates covering the body surface. A practical difficulty with such coordinates is their relative intolerance of surface slope discontinuities, which appear as singularities in the surface curvatures and in the corresponding metrics of the equations. Also, if non-orthogonal curvilinear coordinates are employed, as required for complete coverage of a general body shape, the resulting transformed equations become extremely complex. These traditional difficulties have been sidestepped in the IBL 3D approach [29] again of Mark Drela.

The major development was to formulate the integral boundary layer equations in finite-element form using local Cartesian coordinates defined for each residual. This eliminates the need to construct curvilinear body surface coordinates, and thus largely sidesteps most of the geometry smoothness requirements. It also allows solving the equations on arbitrary triangular or quadrilateral surface meshes, and does not require the identification of stagnation points or attachment lines for the application of initial conditions. All these features greatly simplify the application of the 3D integral boundary layer equations to relatively complex surface shapes.

Another issue which has received scant attention is the incorporation of a suitable transition prediction method into the integral boundary layer methods. The approach has been to either specify the transition line explicitly, or to use 2D correlations or e^N type methods along strips or streamlines to set the transition location in an *ad-hoc* loosely coupled manner. This approach is unreliable if transition is triggered by laminar separation, as frequently occurs in low

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Reynolds number flows. In the "envelope e^N" formulation used in the 2D MSES [28] and XFOIL [15] codes, the amplification equation which governs the transition location is solved simultaneously with the inviscid and boundary layer equations, giving a robust overall method for transitional flows. In IBL this strongly-coupled transition prediction formulation was extended and applied to the 3D boundary layer case, so that the transition location is in effect a fundamental unknown of the solution.

The IBL 3D formulation supports a standard inviscid/viscous interaction model that imposes a transpiration boundary condition on the inviscid formulation at the body surface. The specified inviscid transpiration mass flux is equal to the surface-divergence of the mass defect of the viscous layer, which is equivalent to the physical requirement that the normal mass fluxes in the viscous and inviscid zones are equal immediately outside of the boundary layer. This normal-flow imposition is the only mechanism by which the viscous layer can influence the overall outer inviscid flow, and thus is fully consistent with the physics of high Reynolds number flows.

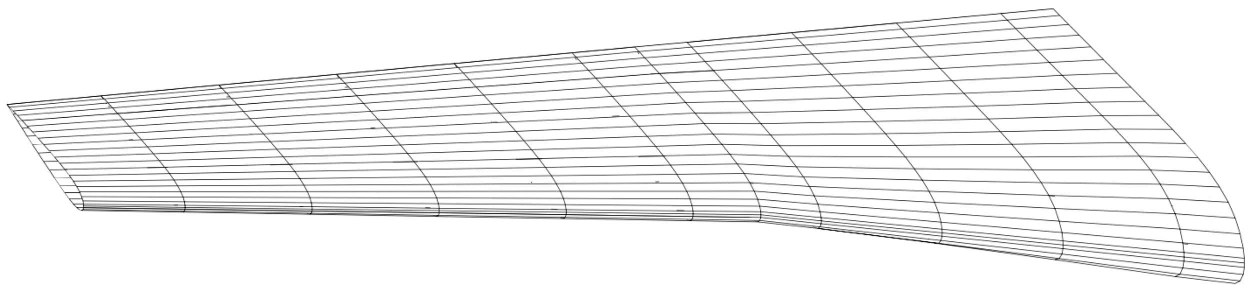


Figure 41. Double-taper wing with 41 by 12 paneling. Wake 9 by 12 mesh not shown.

The basic feasibility of the IBL 3D formulation for application to non-trivial geometry has been demonstrated, where IBL was strongly coupled to a low order panel method using constant doublet strengths, together with constant source-panel strengths to impose the wall transpiration. The overall coupled formulation was solved using a global Newton method, and was used to predict separated flow over a double-taper wing. Figure 41 shows the wing paneling, and Figure 42 shows the computed wall streamlines with a zoom-in on the right. The attachment and separation lines are captured in the solution by the strongly coupled formulation. This is in contrast to the classical 3D boundary layer solvers, which typically require the identification of an attachment line where the space-marching procedure is started.

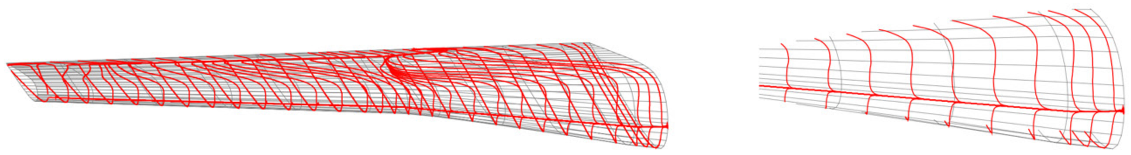


Figure 42. Computed wall streamlines on double-taper wing at 4° angle of attack.

It should be noted that this subtask is, overall, still work in progress. HSM has yet to be fully integrated into the SANS framework (but is available in **CAPS** from the original FORTRAN implementation – see Section 3.1.8.5). And, because the Full Potential Code is grid sensitive it requires adaptation and the proper technique to adjust the mesh is through error estimation via

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

the adjoint, which is not yet complete. Only recently has an analytic adjoint been developed, which is required for error estimation, see Appendix D.

The full potential code is currently incorporated directly into an unpublished AIM. The AIM automatically generates a mesh using either TetGen or AFLR3, where the surface mesh is generated directly by the AIM due to the non-manifold nature of the wake sheet. A number of aerodynamic quantities of interest such as lift, drag, and pitching moment are available. The ability to deflect control surface is also incorporated into the AIM. In addition, both dynamic and static stability derivatives are available. A set of unpublished UDPs have also been created in order to simply the geometry generation (including wake sheets off of lifting surfaces) and application of attributions for full potential computations. This was used during the first **CAPS** training (see Section 3.1.10.2) but was provided only in the PreBuilt distributions. The UDPs and AIM have since been removed waiting for mesh adaptation to ensure better accuracy.

3.2.4.5 CAPS API and AIMs

This task is similar to the maintenance subtask described in Section 3.1.3.2, but now that the **CAPS** software infrastructure is maturing (and is in constant use at AFRL and elsewhere) this maintenance includes all of the **CAPS** software. As deficiencies were found, the API has been enhanced to provide a better interface to the *attached* analysis suites. As pyCAPS (the Python connection to **CAPS**) improves, it drives some minor changes to the **CAPS** API.

The suite of AIMs developed during the first phase of the **CAPS** contract included many more analysis connections than originally proposed (see Sections 2.3 and 3.1.8).

3.2.4.6 SLUGS

SLUGS is the “Static Legacy Unstructured Geometry System” (see Section 3.1.4.1) is the part of **CAPS** that allows a user to generate a watertight BSpline-based BRep from a cloud of points.

Although SLUGS has been exercised on several test cases, the state of the code is such that new cases often reveal minor extensions (and bug fixes) that would improve its utility.

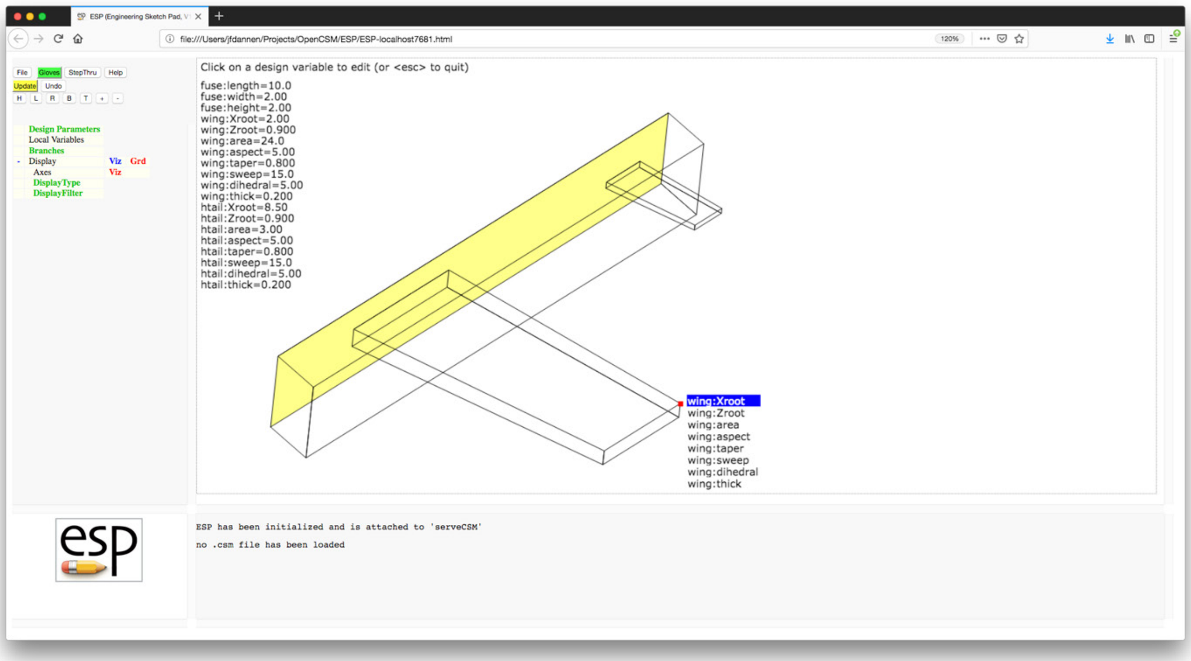
3.2.4.7 Vehicle Configurator (GLOVES)

When modeling a new aircraft, one of the first steps is to create the baseline parametric model. Fortunately, there is a fair amount of commonality between aircraft of different types (such as tube and wing). The Vehicle Sketch Pad from NASA was developed to assemble aircraft components (wing and fuselage) quickly to generate a visual representation that is useful in very early design phases. Unfortunately, VSP’s output is not a watertight BRep; that is, VSP’s resultant geometry is not all that useful for medium to high fidelity analyses, such as aeroelastic analysis. Also, VSP’s user interface does not allow one to naturally interact with a design, but instead has the use adjusting sliders, etc.

GLOVES (the Graphical Layout Of VEhicle Systems) is a tool for creating a vehicle model using a set of standard primitives, such as a wing-like lifting surface or a fuselage-like blended body. Each body type has a set of standard design parameters. For example, the wing-like lifting surface is defined in terms of its root (origin), area, aspect ratio, taper ratio, sweep, dihedral, twist, thickness (distribution) and camber (distribution).

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Once the vehicle components are assembled, GLOVES presents the user with a wire-frame representation, such as shown in Figure 43. This example is a transport-like configuration, consisting of a fuselage, a wing, and a horizontal tail.



```
Click on a design variable to edit (or <esc> to quit)
fuse:length=10.0
fuse:width=2.00
fuse:height=2.00
wing:Xroot=2.00
wing:Zroot=0.900
wing:area=24.0
wing:aspect=5.00
wing:taper=0.800
wing:sweep=15.0
wing:dihedral=5.00
wing:thick=0.200
htail:Xroot=8.50
htail:Zroot=0.900
htail:area=3.00
htail:aspect=5.00
htail:taper=0.800
htail:sweep=15.0
htail:dihedral=5.00
htail:thick=0.200
```

```
ESP has been initialized and is attached to 'serveCSM'
no .csm file has been loaded
```

Figure 43. GLOVES Graphical User Interface.

Once this configuration is shown on the screen, the user interacts with it by hovering over one of the corner points for the wireframe. When over the point, a pop-up menu is displayed that informs the user as to which of the design parameters effect the location of this point. In the figure, hovering over the point at the upper-surface trailing-edge wingtip tells the user that the location of this point is determined by the wing's Xroot, Zroot, area, aspect (ratio), taper (ratio), sweep, dihedral, and thickness. When the user click on one of these items, motion of the mouse will cause the wireframe to track the mouse (as best as possible) by only changing the selected item.

GLOVES is particularly useful in creating a "first guess" for PLUGS. By having a configuration closer to the cloud, many less iterations would be required to find the closest parameter fit.

3.2.4.8 Documentation

One of the biggest challenges in generating useful software of a complex nature is the writing of a clear, concise, and understandable suite of documentation. This is even more difficult when the software is under heavy development. Without the documentation, even the best implemented software is of dubious value (if no one can figure out how to use it, how useful can it be?).

A great deal of effort is spent before a software release in order to fill-in the missing parts of the documentation and to ensure that the current state of the documentation set is consistent with the software to be released. Writing the documentation is a team effort because the software generated under the **ESP** umbrella is a team endeavor. The location of the documents with the **ESP** distribution is listed in Table 4.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Table 4. Documentation and location.

Document	Location in the distribution
EGADS API	doc/EGADS/egads.pdf
OpenCSM API	include/OpenCSM.h
CAPS API	doc/CAPSapi.pdf
AIM Development	doc/AIMdevel.pdf
CAPS Discretization	doc/capsDiscr.pdf
ESP	ESP/ESP-help.html
AIM References	doc/CAPSdoc/*
pyCAPS	doc/pyCAPS/*
Training	training/*
WebViewer API	doc/Viewer.pdf

The training material has also become an important part of the documentation suite. Much effort is expended before a training in reviewing the contents, updating the material, including new features and adding updated "best practices" (also possibly deemphasizing aspects of the material when timing associated with overall content becomes an issue).

3.2.4.9 Software Releases

See Section 3.1.10.3

3.2.4.10 Software Installation on Air Force Computational Facilities

As stated in Section 3.1.10.3 there are 3 basic ways that **ESP**/**CAPS** is installed on individual workstations at AFRL: direct access to the MIT source code repositories, the use of source Beta releases, or the use of official releases (either source or PreBuilt distributions). This has caused some problems at AFRL in regards to the dissemination of plugins that are not a part of the official release (3 above). These UDP/UDFs and/or AIMs either have not been cleared (to Distribution A) or are proprietary and/or sensitive. The problem is that these plugins do not go through the same rigorous testing and need to be distributed separately by some other internal procedure. A better distribution solution is required.

3.2.4.11 ESP and CAPS Training

See Section 3.1.10.2.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

4 CONCLUSIONS

A recent overview of the **CAPS** project [30] concluded with:

> “We believe that a shift in the role and representation of geometry plays a central role in enabling an environment capable of rapid, multifidelity, multidisciplinary design. Most significantly, we advocate for constructing design models that encode the design intent and conceptual elements that comprise a vehicle, rather than a producing a single – albeit typically parametric – view of geometry, which is traditionally approached from a manufacturing rather than analysis mindset. Such a design model enables the construction of multiple, analysis-specific views from a single specification, eliminating the necessity and ambiguity of reinterpreting geometry for different purposes. From this representation, it naturally extends that geometry serves as a conduit for transferring data between coupled analyses. Hence, the geometry should play an active role throughout the analysis process extending beyond analysis preprocessing.
>
> We have also found persistent attribution of geometry to be a critical element of the design environment. When coupled with trimmed, watertight geometry, attributes drive the automated generation of analysis meshes and inputs, removing a bottle neck that precludes using many high-fidelity analyses in the early design process. Ultimately, the attributes provide a linkage between geometry and non-geometric information required for analysis. However, in our experience, the application of analysis attributes in the midst of the design model specification can present more of a conceptual burden than advantage, particularly for structural models requiring identification of BRep Edges and Nodes. A more practicable approach is to attribute geometry with its conceptual purpose as it is built, and to apply analysis-specific attributes to the analysis-specific views after their construction. This dichotomy has the advantage of supporting the typical separation of the modeler, or configurator, from the analyst.
>
> Underpinned by the attributed design model, we have produced and demonstrated the **CAPS** infrastructure to manage the flow of information between the geometry subsystem, various analysis interface modules (AIMs), and the environment driven by an external design process. Beyond producing the design model, the user’s primary interaction with **CAPS** is in the configuration and coordination of AIMs within an executive process. The AIMs themselves perform pre- and post-processing, having a one-to-one mapping with a particular analysis package. The actual execution of an analysis within the computational environment is managed external to the AIM by design, as to permit interoperability across a wide range of environments.
>
> Aside from the shift of cultural mindset required, perhaps the greatest barrier to adoption of **CAPS** technology is learning to script the design model. While the Engineering Sketch Pad (**ESP**) provides a native viewer with built-in script editor, the graphical process entails selecting model operations from a menu and auto-generating the corresponding script. In our experience, users typically construct the design model within **ESP** by making small script modifications and viewing the results. To users accustomed to graphical geometry layout and perhaps uninitiated in computer programming, producing a design model script can be a daunting process. One step we have recently taken to lower this barrier is the introduction pre-coded analysis view generators. When loaded,

> these scripts take geometry attributed under a convention and produce the representations required by certain analyses. Similarly, a library of typical components with a predefined design intent could be envisioned, allowing users to construct vehicle models with a building block approach.
>
> Looking more outwardly, widespread adoption will require the incorporation of these modeling philosophies into industry-standard modeling packages and design frameworks. Our hope is that by demonstrating continued success, the ideas espoused by **CAPS** will attain broad acceptance to advance the state of design by multifidelity, multidisciplinary analysis."

By any measure, the **CAPS** contract is a success. Throughout the course of the contract there has been a great deal of communication and this has changed the priority of various tasks, to continually accommodate the use of **ESP**/**CAPS** within AFRL. In a real sense this contract has been handled as a *Cooperative Agreement* (due to the close collaboration) to the benefit of both the **CAPS** team and AFRL. Since the useful output of the effort has been the software and changes are (continuously) available, the feedback we have gotten has improved our knowledge of the problems at-hand and has provided a better "product" overall.

Even though there has been some adjustment of priorities, most all of the tasks listed above have been successfully completed. Those that were not, either fit into the category of finding an alternative (OverSet Meshing – Section 3.1.7.1) or required a great deal of research, where significant progress can be seen. Examples of the latter are Sculpting (Section 3.2.1.2), Packaging (Section 3.2.3) and FPC/HSM/IBL (Section 3.2.4.4).

Much has been learned [30] (as mentioned above). A number of both Masters and PhD students at MIT and Syracuse University have been funded through **CAPS** and have graduated. The Bibliography is, yet again, another reflection of the intellectual output of this effort overall. And most importantly the software is available (http://acdl.mit.edu/ESP) and is being continuously used at AFRL and elsewhere.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

5 REFERENCES

1. Robert Haimes and John Dannenhoffer, “The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry”, AIAApaper2013-3073.

2. John Dannenhoffer, “OpenCSM: An Open-Source Constructive Solid Modeler for MDAO”, AIAApaper2013-0701.

3. Robert Haimes and Mark Drela, “On the Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design”, AIAApaper2012-0683.

4. Edward Alyanak, Ryan Durscher, Robert Haimes, John Dannenhoffer, Nitin Bhagat and Darcy Allison, “Multi-fidelity Geometry-centric Multi-disciplinary Analysis for Design”, AIAApaper2016-4007.

5. John Dannenhoffer and Robert Haimes, “Design Sensitivity Calculations Directly on CAD-based Geometry”, AIAApaper2015-1370.

6. John Dannenhoffer and Robert Haimes, “Conservative Fitting for Multi-Disciplinary Analysis”, AIAApaper2014-0294.

7. Robert Haimes and John Dannenhoffer, “EGADSlite: A Lightweight Geometry Kernel for HPC”, AIAApaper2018-1401.

8. Ryan Durscher and Dennis Reedy, “pyCAPS: A Python Interface to the Computational Aircraft Prototype Syntheses”, AIAApaper2019-2226.

9. Mark Drela and Harold Youngren, AVL – http://web.mit.edu/drela/Public/web/avl.

10. J.C. Simo and D.D. Fox, “On a Stress Resultant Geometrically Exact Shell Model, Part 1: Formulation and Optimal Parameterization”, *Computer Meth. in Appl. Mechanics and Eng.*, 72:267-304, 1989.

11. J.C. Simo, D.D. Fox and M.S. Rifai, “On a Stress Resultant Geometrically Exact Shell Model, Part 2: The Linear Theory; Computational Aspects”, *Computer Meth. in Appl. Mechanics and Eng.*, 73:53-92, 1989.

12. J.C. Simo, D.D. Fox and M.S. Rifai, “On a Stress Resultant Geometrically Exact Shell Model, Part 3: Computational Aspects of Linear Theory”, *Computer Meth. in Appl. Mechanics and Eng.*, 79:21-70, 1990.

13. Mark Drela, Marshall Galbraith and Steven Allmaras, “Hybrid Shell Model for Aeroelastic Modeling”, AIAApaper2019-2227.

14. Hang Si, “TetGen, a Delaunay-based Quality Tetrahedral Mesh Generator”, *ACM Trans. On Mathematical Software*, 41(2), 2015.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

15. Mark Drela, “XFOIL: An Analysis and Design System for low Reynolds Number Aitrfoils”, In *Low Reynolds Number Aerodynamics*, Springer-Verlag, 1989. http://web.mit.edu/drela/Public/web/xfoil

16. Pengcheng Jia and John Dannenhoffer, “Generation of Parametric Aircraft Models from a Cloud of Points”, AIAApaper2016-1926.

17. E.N. Dvorkin and K.J. Bathe, “A Continuum Mechanics Based Four-node Shell Element for General Non-linear Analysis”, *Engineering Computations*, 1(1):77-88, 1984.

18. William Chan, Shishir Pandya, and Robert Haimes, “Automation of Overset Structured Mesh Generation on Complex Geometries”, AIAApaper2019-3671.

19. Zachary Eager and John Dannenhoffer, “Flends: Generalized Fillets via B-splines”, AIAApaper2019-1717.

20. Julia Docampo-Sánchez and Robert Haimes, “Towards Fully Regular Quad Mesh Generation”, AIAApaper2019-1988.

21. Julia Docampo-Sánchez and Robert Haimes, “A Regularization Approach for Automatic Quad Mesh Generation”, Presented at the 28th Interntional Meshing Roundtable.

22. Philip Caplan, Robert Haimes, David Darmofal and Marshall Galbraith, “Extension of local cavity operators to 3d + t spacetime mesh adaptation”, AIAApaper2019-1992.

23. Philip Caplan, Robert Haimes and David Darmofal, “Four-dimentional anisotropic mesh adaptation”, Submitted to *Computer-Aided Design*.

24. Steve Karman and Nick Wyman, “Automatic Unstructured Mesh Generation with Geometry Attribution”, AIAApaper2019-1721.

25. Mark Drela, “Integrated Simulation Model for Preliminary Aerodynamic, Structural and Control-law Design of Aircraft”, AIAApaper1999-1394.

26. Marshall Galbraith, Steven Allmaras, and Robert Haimes, “Full Potential Revisited: A Medium Fdelity Aerodynamic Analysis Tool”, AIAApaper2017-0290.

27. Marshall Galbraith, Steven Allmaras and David Darmofal, “A Verification Driven Process for Rapid Development of CFD Software”, AIAApaper2015-1530.

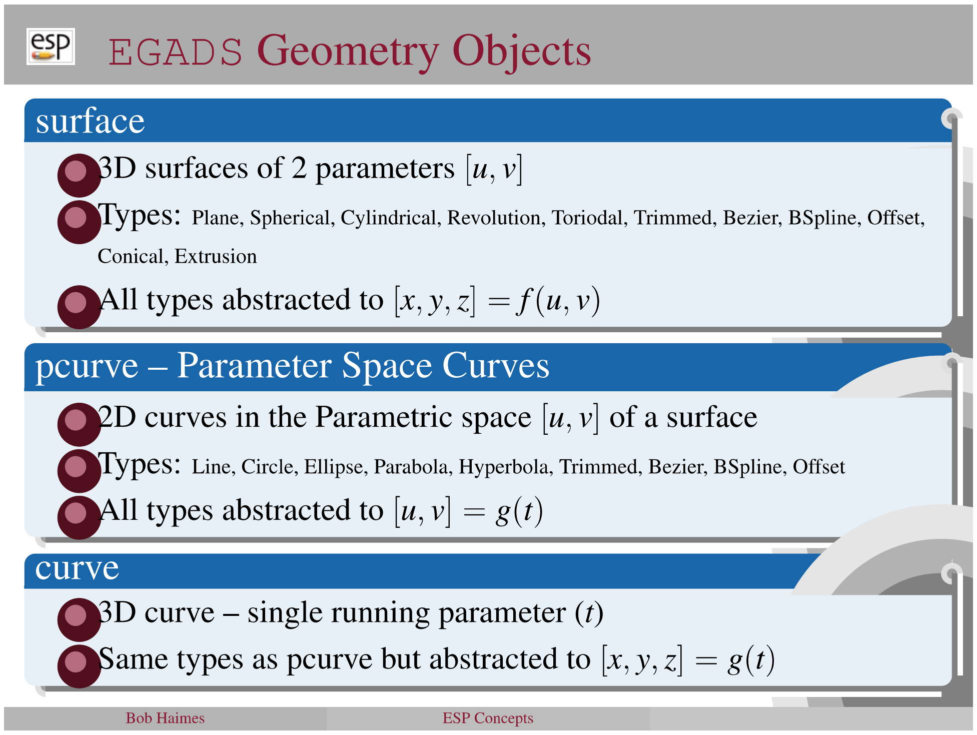
28. Mark Drela and Michael Giles, “Viscous-invicid Analysis of Transonic and Low Reynolds Number Airfoils”, AIAA Journal, 25(10):1347-1355, 1987.

29. Mark Drela, “Three-dimensional Integral Boundaty Layer Formulations for Gereral Configurations”, AIAApaper2013-2437.

30. Dean Bryson, Robert Haimes and John Dannenhoffer, “Toward the Realization of a Highly Integrated, Multidisciplinary, Multifidelity Design Environment”, AIAApaper2019-2225.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

APPENDIX A – Geometry Concepts



EGADS Geometry Objects

surface

- 3D surfaces of 2 parameters $[u, v]$
- Types: Plane, Spherical, Cylindrical, Revolution, Toriodal, Trimmed, Bezier, BSpline, Offset, Conical, Extrusion
- All types abstracted to $[x, y, z] = f(u, v)$

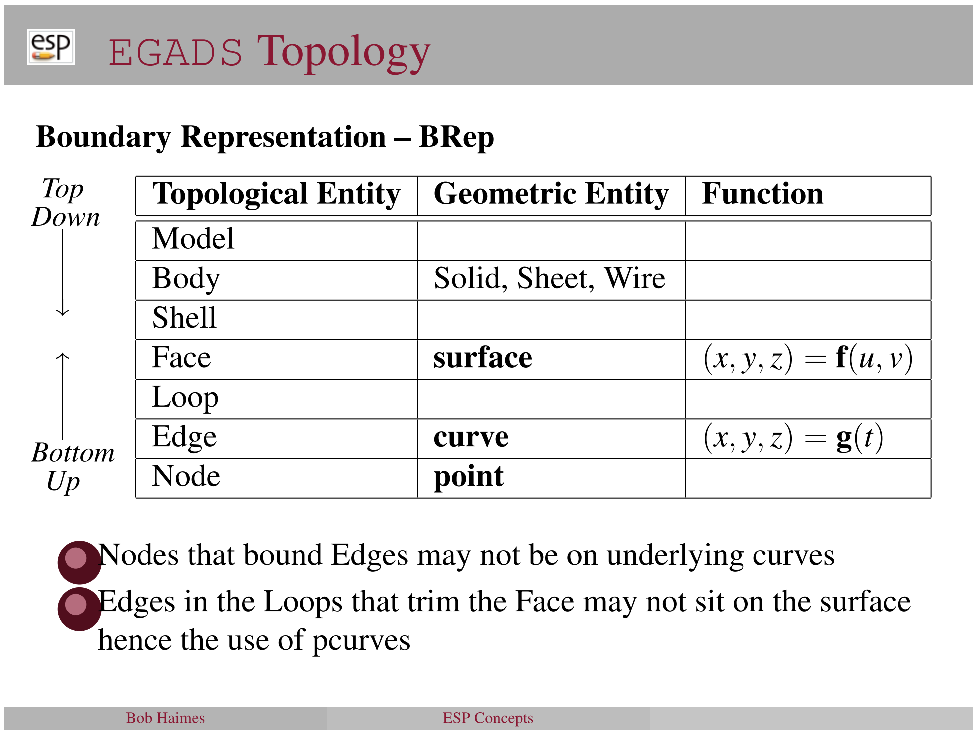
pcurve – Parameter Space Curves

- 2D curves in the Parametric space $[u, v]$ of a surface
- Types: Line, Circle, Ellipse, Parabola, Hyperbola, Trimmed, Bezier, BSpline, Offset
- All types abstracted to $[u, v] = g(t)$

curve

- 3D curve – single running parameter (t)
- Same types as pcurve but abstracted to $[x, y, z] = g(t)$

Bob Haimes ESP Concepts



EGADS Topology

Boundary Representation – BRep

Top Down $\downarrow$

Topological Entity	Geometric Entity	Function
Model		
Body	Solid, Sheet, Wire	
Shell		
Face	**surface**	$(x, y, z) = \mathbf{f}(u, v)$
Loop		
Edge	**curve**	$(x, y, z) = \mathbf{g}(t)$
Node	**point**	

$\uparrow$ *Bottom Up*

- Nodes that bound Edges may not be on underlying curves
- Edges in the Loops that trim the Face may not sit on the surface hence the use of pcurves

Bob Haimes ESP Concepts

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

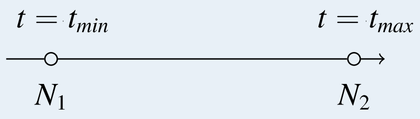
EGADS Topology Objects

Node

- Contains a point – $[x, y, z]$
- Types: **none**

Edge

- Has a 3D curve (if not Degenerate)
- Has a t range (t_{min} to t_{max}, where $t_{min} < t_{max}$)
  Note: The positive orientation is going from t_{min} to t_{max}
- Has a Node for t_{min} and for t_{max} – can be the same Node
- Types: **ONENODE** – periodic, **TWONODE** – normal, **DEGENERATE** – single Node, t range used for the pcurve



EGADS Topology Objects – Loops

Loop – without a reference surface

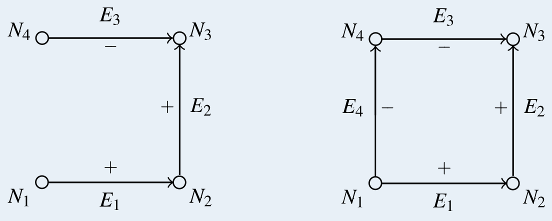
1. Free standing connected Edges that can be used in a non-manifold setting (for example in WireBodies)
2. A list of connected Edges associated with a Plane (which does not require pcurves)

- An ordered collection of Edge objects with associated senses that define the connected *Wire/Contour/Loop*
- Segregates space by maintaining material to the left of the running Loop (or traversed right-handed pointing out of the intended volume)
- No Edges should be Degenerate
- Types: **OPEN** or **CLOSED** (comes back on itself)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

EGADS Topology Objects – Loops

Loop – without a reference surface



Open: $+E_1$ $+E_2$ $-E_3$ Closed: $+E_1$ $+E_2$ $-E_3$ $-E_4$

EGADS Topology Objects – Loops

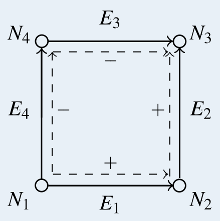
Loop – with a reference surface

- Collections of Edges (like without a surface) followed by a corresponding collection of pcurves that define the $[u, v]$ trimming on the surface
- Degenerate Edges are required when the $[u, v]$ mapping collapses like at the apex of a cone (note that the pcurve is needed to be fully defined using the Edge's t range)
- An Edge may be found in a Loop twice (with opposite senses) and with different pcurves. For example a closed cylindrical surface at the seam – one pcurve would represent the beginning of the period where the other is the end of the periodic range.
- Types: **OPEN** or **CLOSED** (comes back on itself)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

EGADS Topology Objects – Loops

Loop – with a reference surface (**CLOSED**)



dotted lines indicate associated pcurves

EGADS Topology Objects – Face

Face

- A surface bounded by one or more Loops with associated senses
- Only one outer Loop (sense = 1) and any number of inner Loops (sense = -1). Note that under very rare conditions a Loop may be found in more than 1 Face – in this case the one marked with sense = +/- 2 must be used in a reverse manner.
- All Loops must be **CLOSED**
- Loop(s) must not contain reference geometry for Planar surfaces
- If the surface is not a Plane then the Loop's reference Object must match that of the Face
- Type is the orientation of the Face based on surface's $U \otimes V$:
  - **SFORWARD** or **SREVERSE** when the orientations are opposed

Note that this is coupled with the Loop's orientation (i.e. an outer Loop traverses the Face in a right-handed manner defining the outward direction)

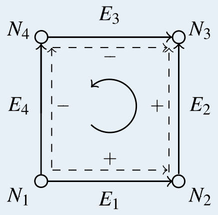
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

EGADS Topology Objects – Face

Face

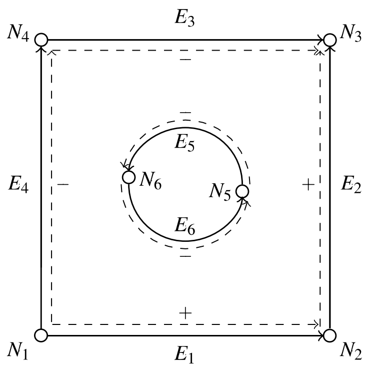
- An outer Loop traverses the Face in a right-handed manner
- Inner Loops trim the Face in a left-handed manner
- *Material* is to the left of the Edges going around the Loops

surface normal
is out of the page



Single Outer Loop – right handed/counterclockwise: $+E_1$ $+E_2$ $-E_3$ $-E_4$

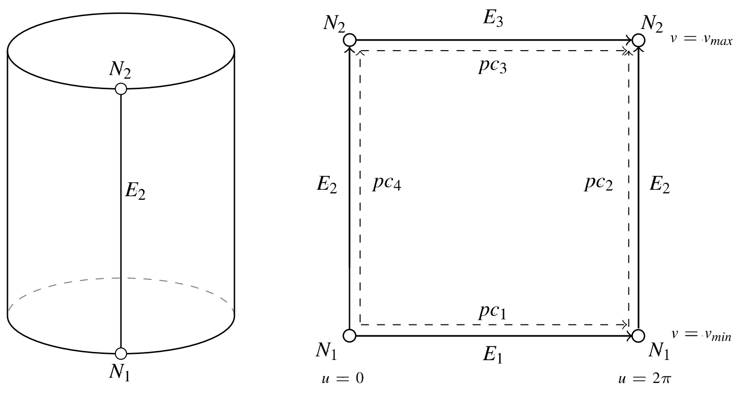
EGADS Topology Objects – Face



- Outer Loop – right handed/counterclockwise: $+E_1$ $+E_2$ $-E_3$ $-E_4$
- Inner Loop – left handed/clockwise: $-E_5$ $-E_6$

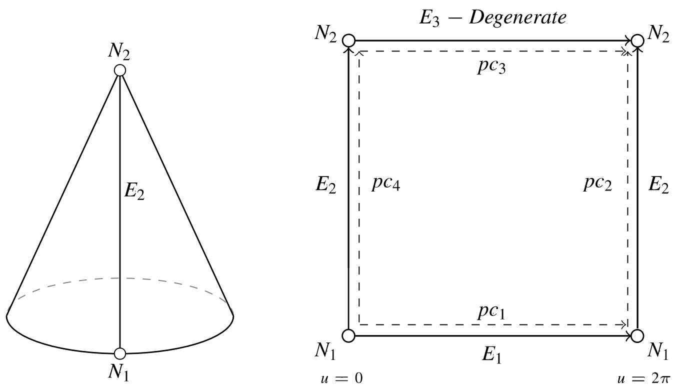
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

EGADS Topology Objects – Face



Unrolled periodic cylinder Face
Single Outer Loop – right handed/counterclockwise:
$+E_1$ $+E_2$ $-E_3$ $-E_2$

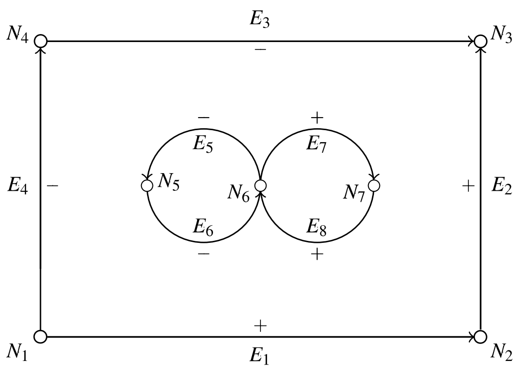
EGADS Topology Objects – Face



Unrolled Cone

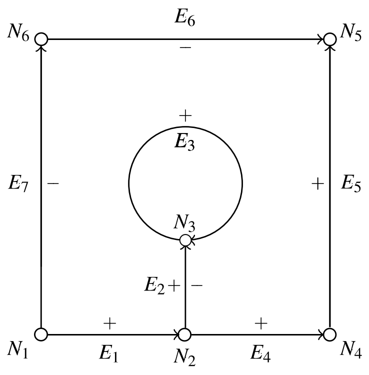
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

EGADS Topology Objects – Face



- Outer Loop – right handed/counterclockwise: $+E_1$ $+E_2$ $-E_3$ $-E_4$
- Inner Loop #1 – left handed/clockwise: $-E_5$ $-E_6$
- Inner Loop #2 – left handed/clockwise: $+E_7$ $+E_8$

EGADS Topology Objects – Face



Single Outer Loop – right handed/counterclockwise:
$+E_1$ $+E_2$ $+E_3$ $-E_2$ $+E_4$ $+E_5$ $-E_6$ $-E_7$

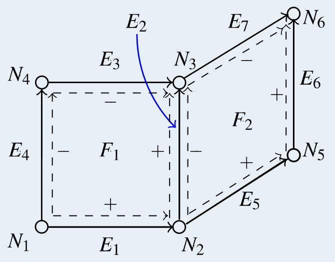
Note: pcurve is the same for both sides of E_2

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

EGADS Topology Objects

Shell

- A collection of one or more connected Faces that if **CLOSED** segregates regions of 3-Space
- All Faces must be properly oriented
- Non-manifold Shells can have more than 2 Faces sharing an Edge
- Types: **OPEN** (including non-manifold) or **CLOSED**



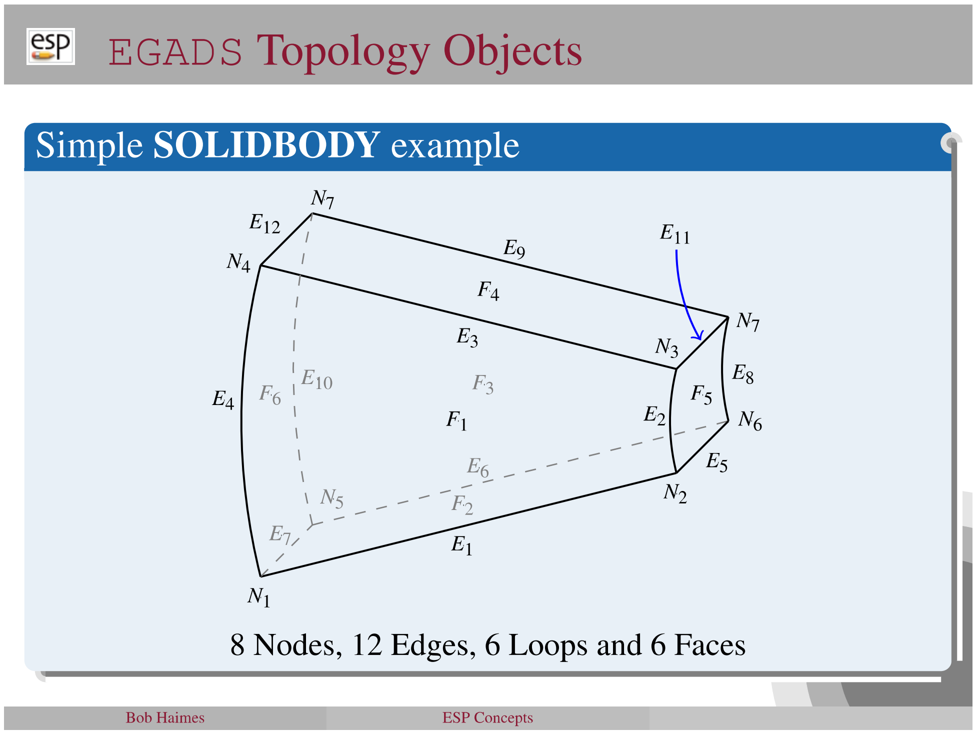
Face #1 Loop: $+E_1$ $+E_2$ $-E_3$ $-E_4$
Face #2 Loop: $+E_5$ $+E_6$ $-E_7$ $-E_2$

EGADS Topology Objects

Body

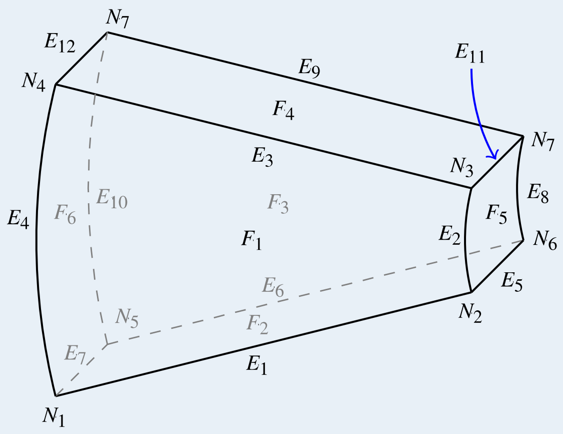
- Container used to aggregate Topology
- Connected to support non-manifold collections at the Model level
- *Owns* all the Objects contained within
  - A **WIREBODY** type contains a single Loop
  - A **FACEBODY** contains a single Face – IGES import
  - A **SHEETBODY** contains one or more Shell(s) which can be either non-manifold or manifold (though usually a manifold Body of this type is promoted to a **SOLIDBODY**)
  - **SOLIDBODY**:
    - A manifold collection of one or more **CLOSED** Shells with associated senses
    - There may be only one outer Shell (sense = 1) and any number of inner Shells (sense = -1)
    - Edges (except **DEGENERATE**) found exactly twice (sense = ± 1)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

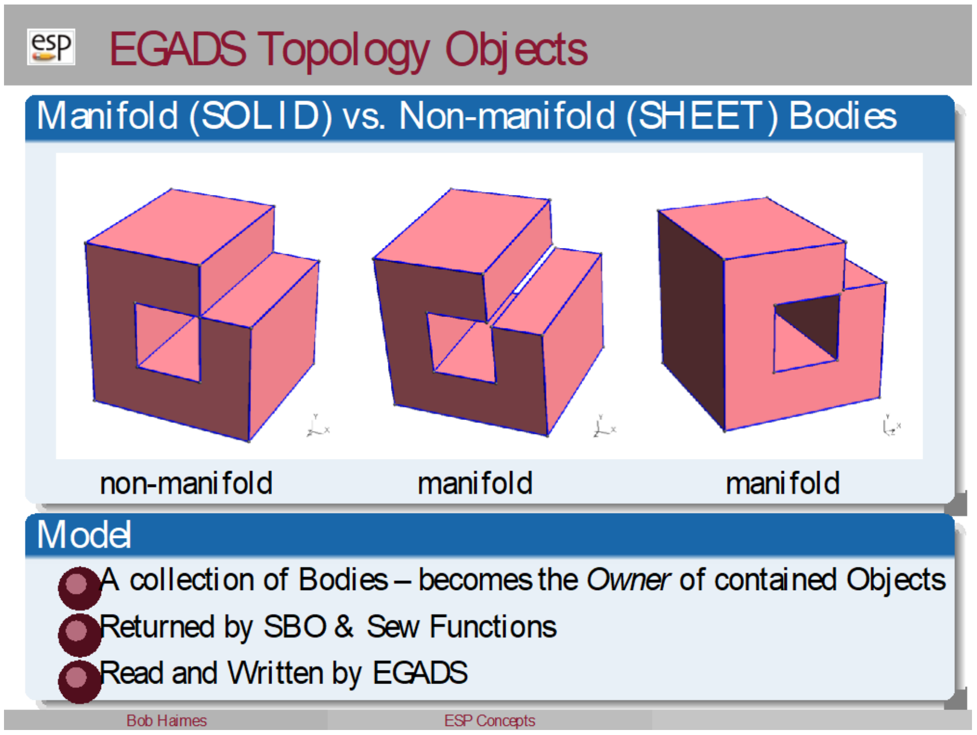


EGADS Topology Objects

Simple **SOLIDBODY** example

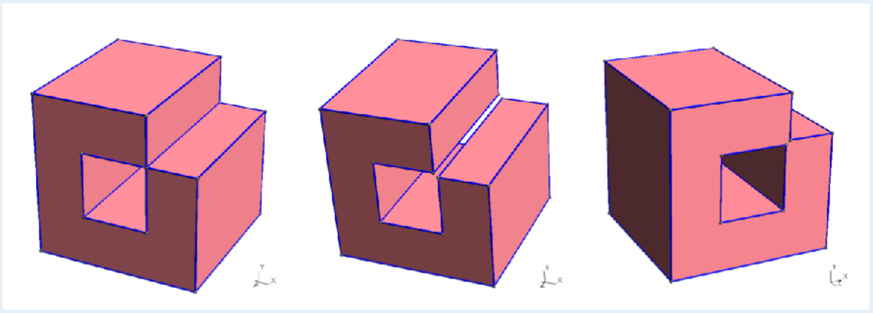


8 Nodes, 12 Edges, 6 Loops and 6 Faces



EGADS Topology Objects

Manifold (SOLID) vs. Non-manifold (SHEET) Bodies



non-manifold manifold manifold

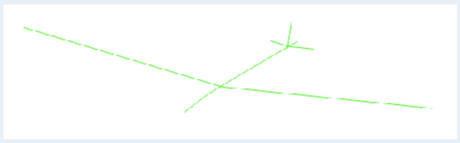
Model

- A collection of Bodies – becomes the *Owner* of contained Objects
- Returned by SBO & Sew Functions
- Read and Written by EGADS

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

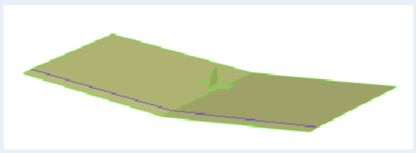
EGADS Topology Objects

Body Examples

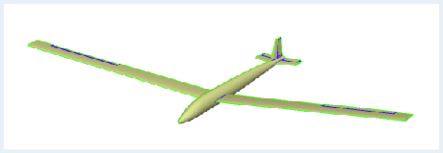


Wire Bodies

Face (Sheet) Bodies *



Sheet Bodies



Solid Body

* OpenCSM treats all FACEBODYs as SHEETBODYs

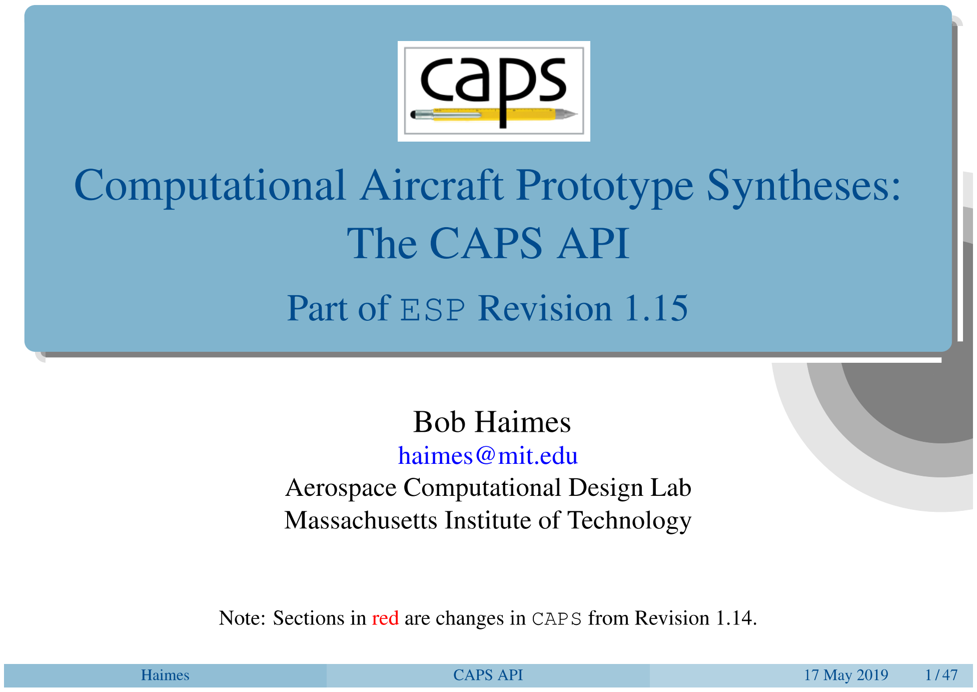
EGADS Objects – Attribution

- Attributes – metadata consisting of name/value pairs
  - Unique name – no spaces
  - A single type: Integer *, Real, String, CSys (Coordinate Systems)
  - A length (not for strings)
- Objects
  - Any EGADS Object can have multiple Attributes (each with a unique name)
  - Only Attributes on Topological Objects are copied and are persistent (saved)
- SBO & Intersection Functions
  - Unmodified Topological Objects maintain their Attributes
  - Face Attributes are carried through to the resultant fragments
  - All other Attributes are lost
- CSys Attributes are modified through Transformations

* OpenCSM supports only Real numeric attributes (integer values are converted)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

APPENDIX B – CAPS API



Computational Aircraft Prototype Syntheses: The CAPS API

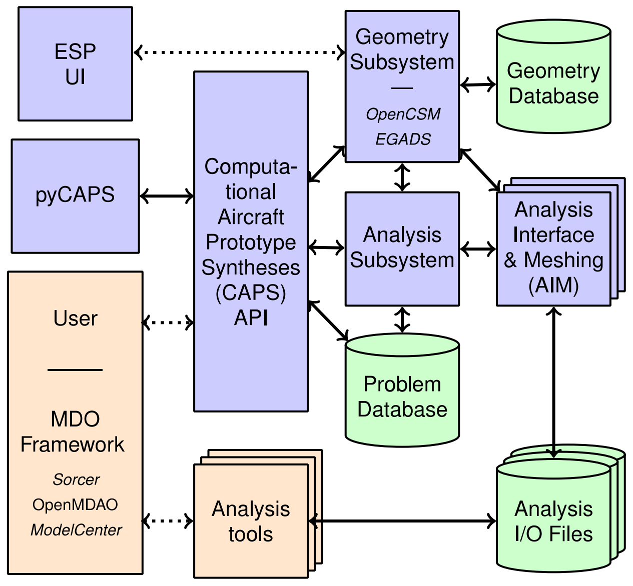
Part of ESP Revision 1.15

Bob Haimes
haimes@mit.edu
Aerospace Computational Design Lab
Massachusetts Institute of Technology

Note: Sections in red are changes in CAPS from Revision 1.14.

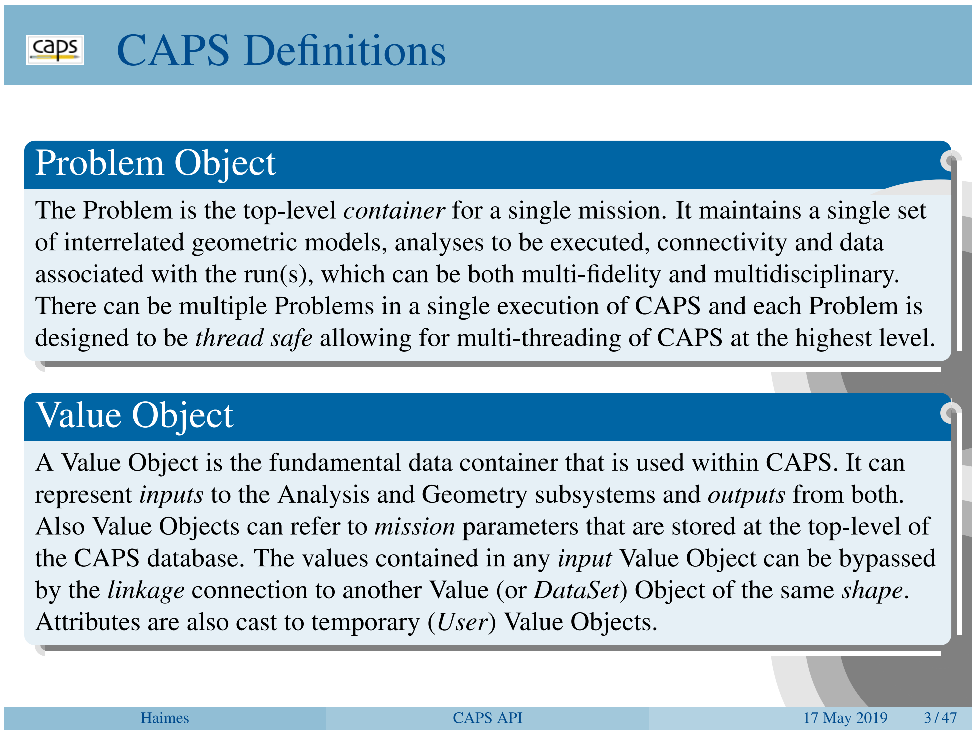
Haimes | CAPS API | 17 May 2019 | 1 / 47

CAPS Infrastructure in ESP



Haimes | CAPS API | 17 May 2019 | 2 / 47

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



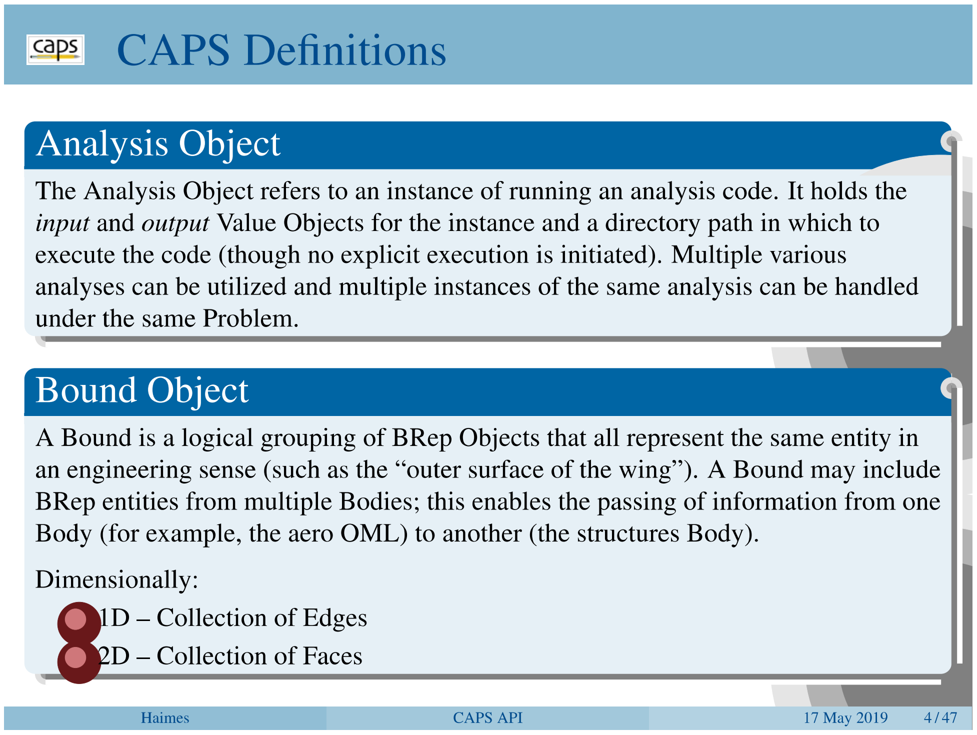
CAPS Definitions

Problem Object

The Problem is the top-level *container* for a single mission. It maintains a single set of interrelated geometric models, analyses to be executed, connectivity and data associated with the run(s), which can be both multi-fidelity and multidisciplinary. There can be multiple Problems in a single execution of CAPS and each Problem is designed to be *thread safe* allowing for multi-threading of CAPS at the highest level.

Value Object

A Value Object is the fundamental data container that is used within CAPS. It can represent *inputs* to the Analysis and Geometry subsystems and *outputs* from both. Also Value Objects can refer to *mission* parameters that are stored at the top-level of the CAPS database. The values contained in any *input* Value Object can be bypassed by the *linkage* connection to another Value (or *DataSet*) Object of the same *shape*. Attributes are also cast to temporary (*User*) Value Objects.



CAPS Definitions

Analysis Object

The Analysis Object refers to an instance of running an analysis code. It holds the *input* and *output* Value Objects for the instance and a directory path in which to execute the code (though no explicit execution is initiated). Multiple various analyses can be utilized and multiple instances of the same analysis can be handled under the same Problem.

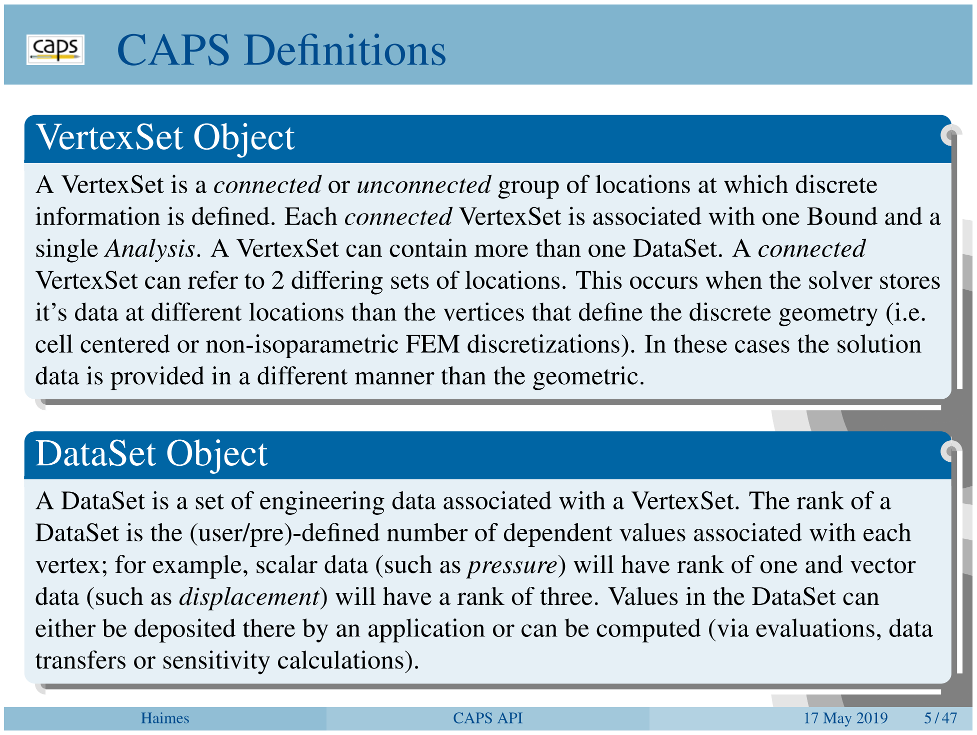
Bound Object

A Bound is a logical grouping of BRep Objects that all represent the same entity in an engineering sense (such as the "outer surface of the wing"). A Bound may include BRep entities from multiple Bodies; this enables the passing of information from one Body (for example, the aero OML) to another (the structures Body).

Dimensionally:

- 1D – Collection of Edges
- 2D – Collection of Faces

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



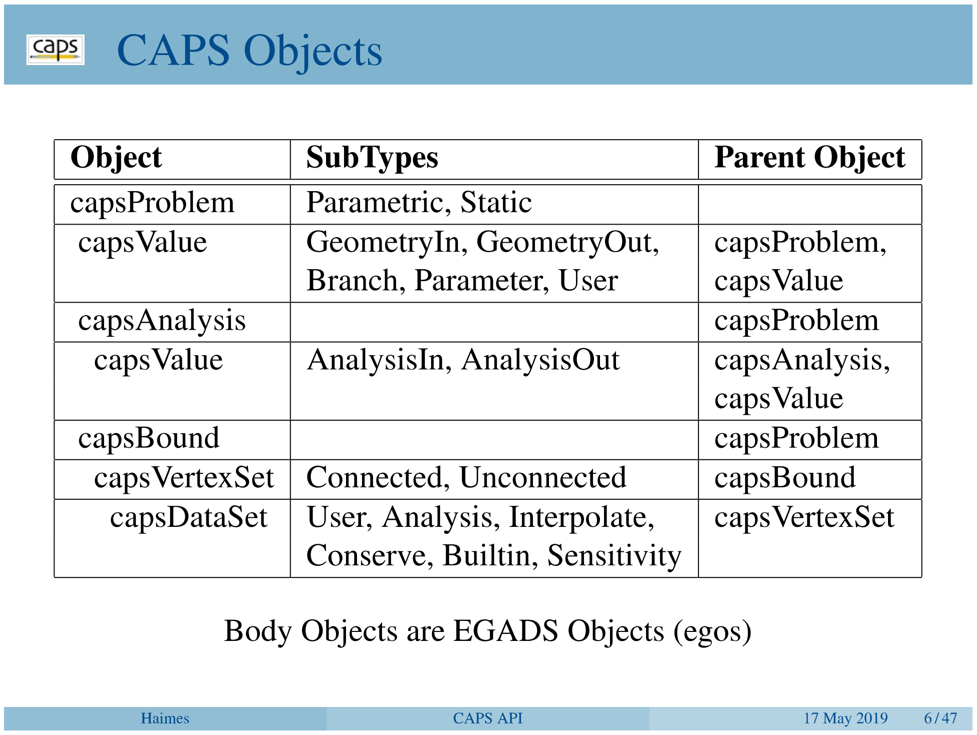
CAPS Definitions

VertexSet Object

A VertexSet is a *connected* or *unconnected* group of locations at which discrete information is defined. Each *connected* VertexSet is associated with one Bound and a single *Analysis*. A VertexSet can contain more than one DataSet. A *connected* VertexSet can refer to 2 differing sets of locations. This occurs when the solver stores it's data at different locations than the vertices that define the discrete geometry (i.e. cell centered or non-isoparametric FEM discretizations). In these cases the solution data is provided in a different manner than the geometric.

DataSet Object

A DataSet is a set of engineering data associated with a VertexSet. The rank of a DataSet is the (user/pre)-defined number of dependent values associated with each vertex; for example, scalar data (such as *pressure*) will have rank of one and vector data (such as *displacement*) will have a rank of three. Values in the DataSet can either be deposited there by an application or can be computed (via evaluations, data transfers or sensitivity calculations).



CAPS Objects

Object	**SubTypes**	**Parent Object**
capsProblem	Parametric, Static	
capsValue	GeometryIn, GeometryOut, Branch, Parameter, User	capsProblem, capsValue
capsAnalysis		capsProblem
capsValue	AnalysisIn, AnalysisOut	capsAnalysis, capsValue
capsBound		capsProblem
capsVertexSet	Connected, Unconnected	capsBound
capsDataSet	User, Analysis, Interpolate, Conserve, Builtin, Sensitivity	capsVertexSet

Body Objects are EGADS Objects (egos)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS Body Filtering

Filtering the active CSM Bodies occurs at two different stages, once in the CAPS framework, and once in the AIMs. The filtering in the CAPS framework creates sub-groups of Bodies from the CSM stack that are passed to the specified AIM. Each AIM instance is then responsible for selecting the appropriate Bodies from the list it has received.

The filtering is performed by using two Body attributes: "capsAIM" and "capsIntent".

Filtering within AIM Code

Each AIM can adopt it's own filtering scheme for down-selecting how to use each Body it receives. The "capsIntent" string is accessible to the AIM, but it is for information only.

CAPS Body Filtering

CSM AIM targeting: "capsAIM"

The CSM script generates Bodies which are designed to be used by specific AIMs. The AIMs that the Body is designed for is communicated to the CAPS framework via the "capsAIM" string attribute. This is a semicolon-separated string with the list of AIM names. Thus, the CSM author can give a clear indication to which AIMs should use the Body. For example, a body designed for a CFD calculation could have:

ATTRIBUTE capsAIM $su2AIM;fun3dAIM;cart3dAIM

CAPS AIM Instantiation: "capsIntent"

The "capsIntent" Body attribute is used to disambiguate which AIM instance should receive a given Body targeted for the AIM. An argument to `caps_load` accepts a semicolon-separated list of keywords when an AIM is instantiated in CAPS/pyCAPS. Bodies from the "capsAIM" selection with a matching string attribute "capsIntent" are passed to the AIM instance. The attribute "capsIntent" is a semicolon-separated list of keywords. If the string to `caps_load` is NULL, all Bodies with a "capsAIM" attribute that matches the AIM name are given to the AIM instance.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Other Reserved CAPS Attribute names

capsLength

This string Attribute must be applied to an EGADS Body to indicate the length units used in the geometric construction.

capsBound

This string Attribute must be applied to EGADS BRep Objects to indicate which CAPS Bound(s) are associated with the geometry. A entity can be assigned to multiple Bounds by having the Bound names separated by a semicolon. Face examples could be “Wing”, “Wing;Flap”, “Fuselage”, and etc.

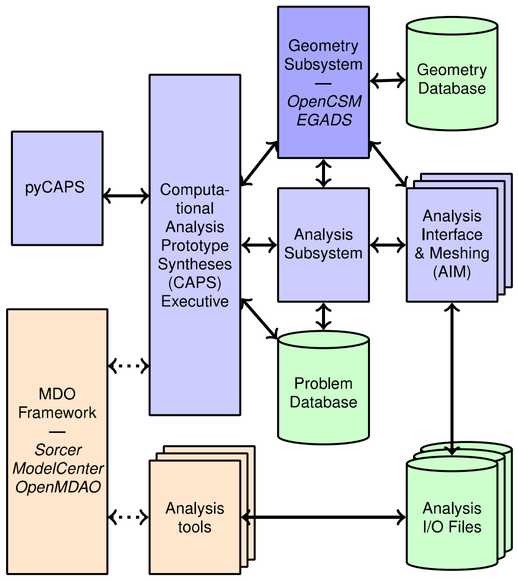
Note: Bound names should not cross dimensional lines.

capsGroup

This string Attribute can be applied to EGADS BRep Objects to assist in grouping geometry into logical sets. A geometric entity can be assigned to multiple groups in the same manner as the capsBound attribute.

Note: CAPS does not internally use this, but is suggested of classifying geometry.

CAPS Execution



Setup (or read) the Problem:

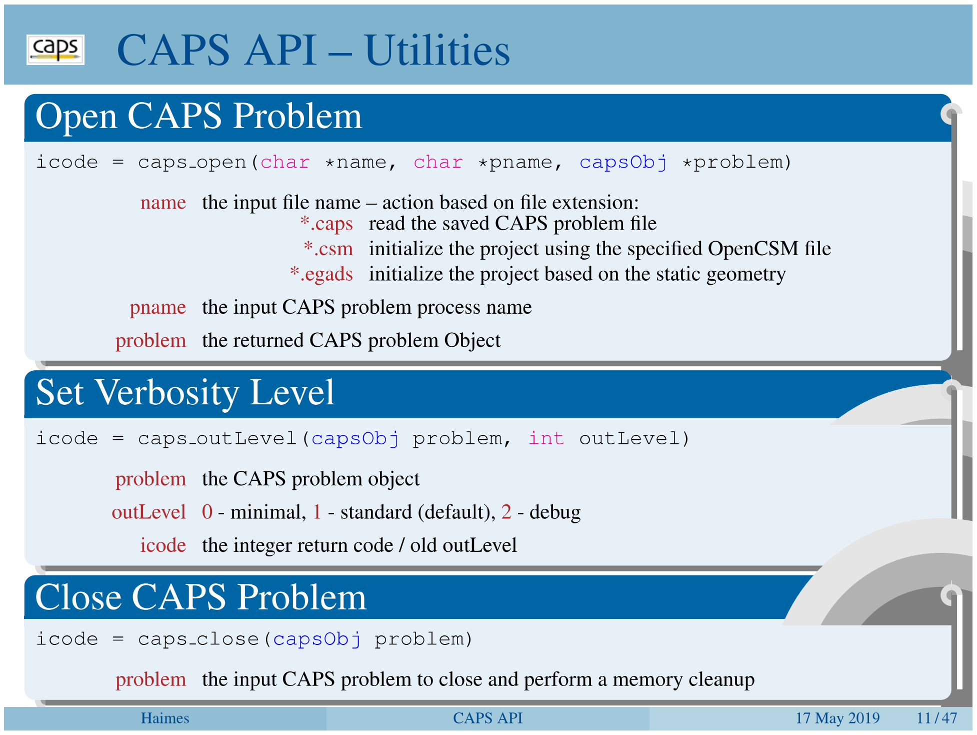
- Initialize Problem with *csm* (or *static*) file
  GeomIn and GeomOut parameters
- Specify *mission* parameters
- Make Analysis instances
  AnalysisIn and AnalysisOut params
- Create *Bounds*, *VetrexSets* & *DataSets*
- Establish linkages between parameters

Run the Problem:

- Adjust the appropriate parameters
- Regenerate Geometry (if *dirty*)
- Call for Analysis Input file generation
- Framework/user runs each *solver*
- Inform CAPS that an Analysis has run
  fills AnalysisOut params & *DataSets* (lazy)
- Generate *Objective Function*

Save the Problem DB (*checkpointing*)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



CAPS API – Utilities

Open CAPS Problem

```
icode = caps_open(char *name, char *pname, capsObj *problem)
```

name	the input file name – action based on file extension:
	*.caps read the saved CAPS problem file
	*.csm initialize the project using the specified OpenCSM file
	*.egads initialize the project based on the static geometry
pname	the input CAPS problem process name
problem	the returned CAPS problem Object

Set Verbosity Level

```
icode = caps_outLevel(capsObj problem, int outLevel)
```

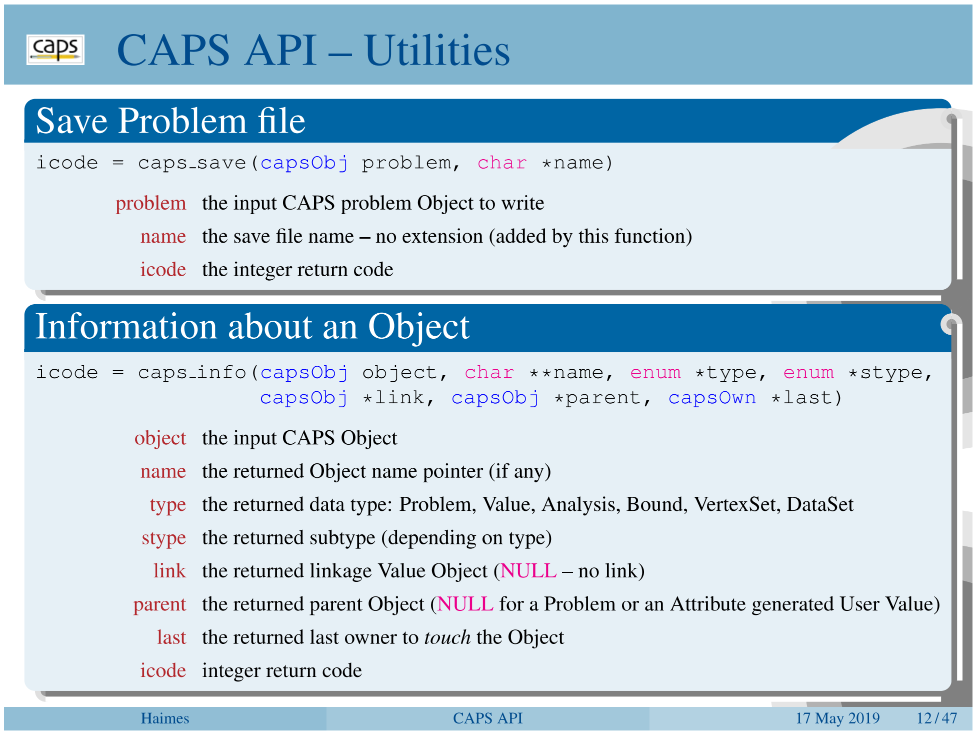
problem	the CAPS problem object
outLevel	0 - minimal, 1 - standard (default), 2 - debug
icode	the integer return code / old outLevel

Close CAPS Problem

```
icode = caps_close(capsObj problem)
```

problem	the input CAPS problem to close and perform a memory cleanup

Haimes CAPS API 17 May 2019 11 / 47



CAPS API – Utilities

Save Problem file

```
icode = caps_save(capsObj problem, char *name)
```

problem	the input CAPS problem Object to write
name	the save file name – no extension (added by this function)
icode	the integer return code

Information about an Object

```
icode = caps_info(capsObj object, char **name, enum *type, enum *stype,
                  capsObj *link, capsObj *parent, capsOwn *last)
```

object	the input CAPS Object
name	the returned Object name pointer (if any)
type	the returned data type: Problem, Value, Analysis, Bound, VertexSet, DataSet
stype	the returned subtype (depending on type)
link	the returned linkage Value Object (NULL – no link)
parent	the returned parent Object (NULL for a Problem or an Attribute generated User Value)
last	the returned last owner to *touch* the Object
icode	integer return code

Haimes CAPS API 17 May 2019 12 / 47

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Utilities

Children Sizing info from a Parent Object

```
icode = caps_size(capsObj object, enum type, enum stype, int *size)
```

object	the input CAPS Object
type	the data type to size: Bodies, Attributes, Value, Analysis, Bound, VertexSet, DataSet
stype	the subtype to size (depending on type)
size	the returned size
icode	integer return code

Get Child by Index

```
icode = caps_childByIndex(capsObj object, enum type, enum stype,
                          int index, capsObj *child)
```

object	the input parent Object
type	the Object type to return: Value, Analysis, Bound, VertexSet, DataSet
stype	the subtype to find (depending on type)
index	the index [1-`size`]
child	the returned CAPS Object
icode	integer return code

CAPS API – Utilities

Get Child by Name

```
icode = caps_childByName(capsObj object, enum type, enum stype,
                         char *name, capsObj *child)
```

object	the input parent Object
type	the Object type to return: Value, Analysis, Bound, VertexSet, DataSet
stype	the subtype to find (depending on type)
name	a pointer to the index character string
child	the returned CAPS Object
icode	integer return code

Delete an Object

```
icode = caps_delete(capsObj object)
```

object	the Object to be deleted Note: only Value Objects of subtype User and Bound Objects may be deleted!
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Utilities

Get Body by index

```
icode = caps_bodyByIndex(capsObj obj, int ind, ego *body, char **unit)
```

obj	the input CAPS Problem or Analysis Object
ind	the index [1-`size`]
body	the returned EGADS Body Object
units	pointer to the string declaring the length units – NULL for unitless values
icode	integer return code

Set Owner Data

```
icode = caps_setOwner(capsObj prob, char *pname, capsOwn *owner)
```

prob	the input CAPS Problem Object
pname	a pointer to the process name character string
owner	a pointer to the CAPS Owner structure to fill
icode	integer return code

Notes: (1) This increases the Problem's sequence number
(2) This does not return the owner pointer, but uses the address to fill
(3) The internal strings can be freed up with `caps_freeOwner`

CAPS API – Utilities

Free Owner Information

```
caps_freeOwner(capsOwn *owner)
```

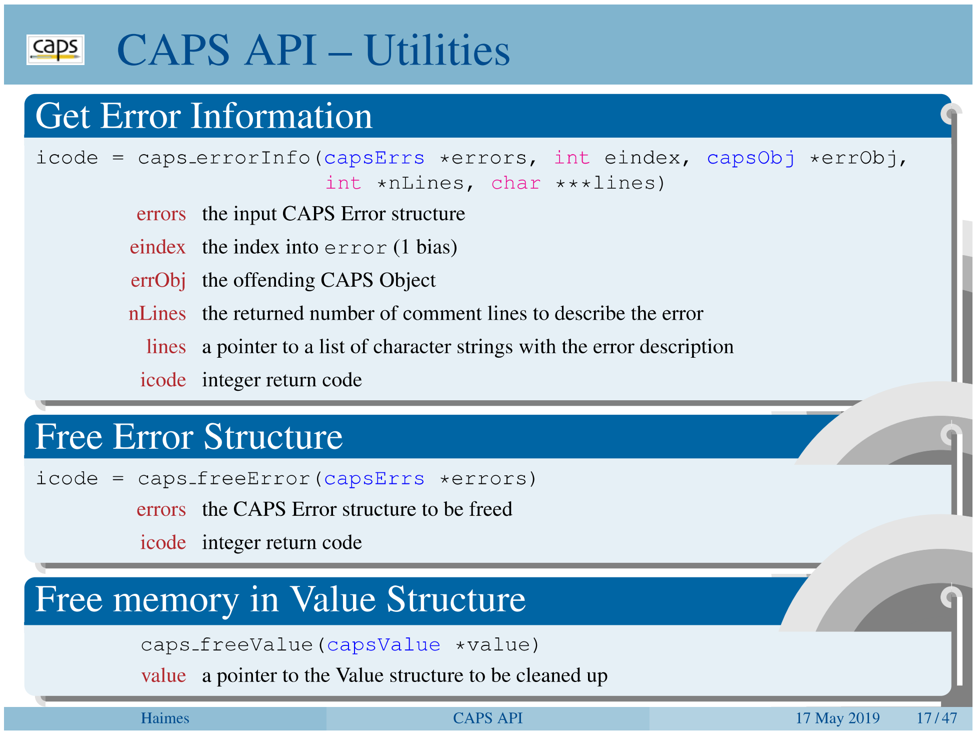
owner	a pointer to the CAPS Owner structure to free up the members `pname`, `pID` and `user`

Get Owner Information

```
icode = caps_ownerInfo(capsOwn owner, char **pname, char **pID,
                       char **userID, short datetime[6], long *sNum)
```

owner	the input CAPS Owner structure
pname	the returned pointer to the process name
pID	the returned pointer to the process ID
userID	the returned pointer to the user ID
datetime	the filled date/time stamp info [year, month, day, hour, minute, second]
sNum	the sequence number (always increasing)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



CAPS API – Utilities

Get Error Information

```
icode = caps_errorInfo(capsErrs *errors, int eindex, capsObj *errObj,
                       int *nLines, char ***lines)
```

errors	the input CAPS Error structure
eindex	the index into `error` (1 bias)
errObj	the offending CAPS Object
nLines	the returned number of comment lines to describe the error
lines	a pointer to a list of character strings with the error description
icode	integer return code

Free Error Structure

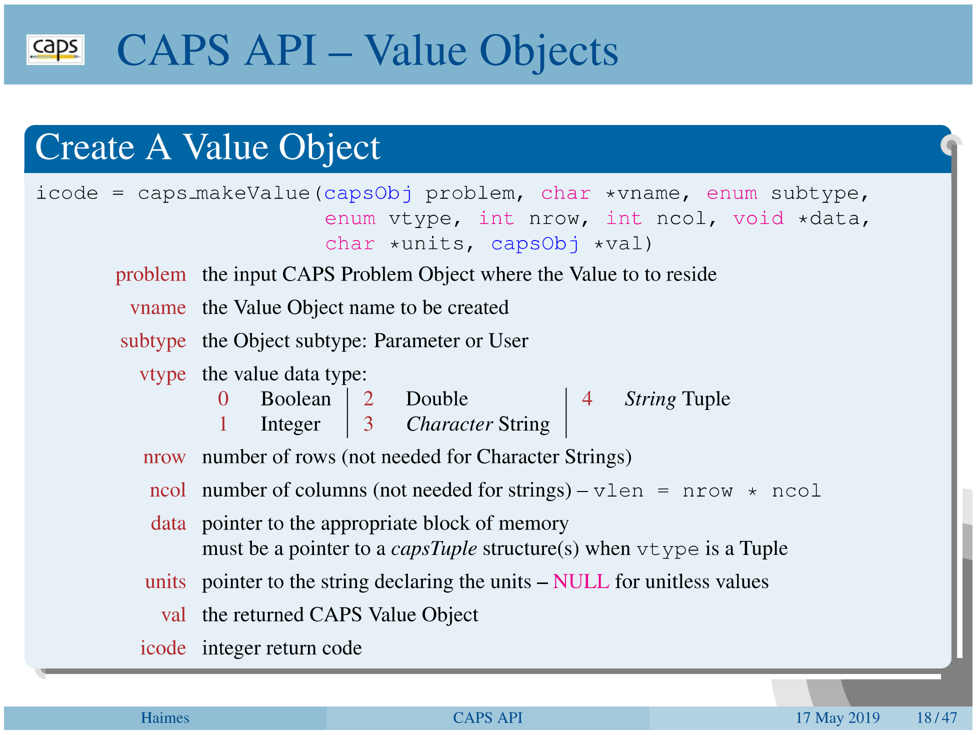
```
icode = caps_freeError(capsErrs *errors)
```

errors	the CAPS Error structure to be freed
icode	integer return code

Free memory in Value Structure

```
caps_freeValue(capsValue *value)
```

value	a pointer to the Value structure to be cleaned up



CAPS API – Value Objects

Create A Value Object

```
icode = caps_makeValue(capsObj problem, char *vname, enum subtype,
                       enum vtype, int nrow, int ncol, void *data,
                       char *units, capsObj *val)
```

problem	the input CAPS Problem Object where the Value to to reside
vname	the Value Object name to be created
subtype	the Object subtype: Parameter or User
vtype	the value data type:

0	Boolean	2	Double	4	*String* Tuple
1	Integer	3	*Character* String		

nrow	number of rows (not needed for Character Strings)
ncol	number of columns (not needed for strings) – `vlen = nrow * ncol`
data	pointer to the appropriate block of memory must be a pointer to a *capsTuple* structure(s) when `vtype` is a Tuple
units	pointer to the string declaring the units – NULL for unitless values
val	the returned CAPS Value Object
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Value Objects

Retrieve Values

```
icode = caps_getValue(capsObj val, enum *vtype, int *vlen, void **data,
                      char **units, int *nErr, capsErrs **errs)
```

val	the input Value Object
vtype	the returned data type:
vlen	the returned value length
data	a filled pointer to the appropriate block of memory (NULL – don't fill) Can use `childByIndex` to get Value Objects
units	the returned pointer to the string declaring the units
nErr	the returned number of errors generated – 0 means no errors
errs	the returned CAPS error structure – NULL with no errors
icode	integer return code

0	Boolean	2	Double	4	*String* Tuple
1	Integer	3	*Character* String	5	Value *Object*

Use the structure *capsTuple* when casting `data` if a Tuple (4)

CAPS API – Value Objects

Reset A Value Object

```
icode = caps_setValue(capsObj val, int nrow, int ncol, void *data)
```

val	the input CAPS Value Object (not for GeometryOut or AnalysisOut)
nrow	number of rows (not needed for Character Strings)
ncol	number of columns (not needed for strings) – `vlen = nrow * ncol`
data	pointer to the appropriate block of memory used to reset the values

Get Valid Value Range

```
icode = caps_getLimits(capsObj val, void **limits)
```

val	the input Value Object
limits	an returned pointer to a block of memory containing the valid range [2*sizeof(`vtype`) in length] – or – NULL if not yet filled

Set Valid Value Range

```
icode = caps_setLimits(capsObj val, void *limits)
```

val	the input Value Object (only for the User & Parameter subtypes)
limits	a pointer to the appropriate block of memory which contains the minimum and maximum range allowed (2 in length)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Value Object

Get Value Shape/Dimension

```
icode = caps_getValueShape(capsObj val, int *dim, enum *lfixed,
                           enum *sfixed, enum *ntype,
                           int *nrow, int *ncol)
```

val	the input Value Object
dim	the returned dimensionality: 0 scalar only 1 vector or scalar 2 scalar, vector or 2D array
lfixed	0 – the length(s) can change, 1 – the length is fixed
sfixed	0 – the Shape can change, 1 – Shape is fixed
ntype	0 – NULL invalid, 1 – not NULL, 2 – is NULL
nrow	number of rows – parent index for `Value` vtypes
ncol	number of columns Note: `vlen = nrow * ncol`
icode	integer return code

CAPS API – Value Object

Set Value Shape/Dimension

```
icode = caps_setValueShape(capsObj val, int dim, enum lfixed,
                           enum sfixed, enum ntype)
```

val	the input Value Object (only for the User & Parameter subtypes)
dim	the dimensionality: 0 scalar only 1 vector or scalar 2 scalar, vector or 2D array
lfixed	0 – the length(s) can change, 1 – the length is fixed
sfixed	0 – the Shape can change, 1 – Shape is fixed
ntype	0 – NULL invalid, 1 – not NULL, 2 – is NULL

Units conversion

```
icode = caps_convert(capsObj val, char *units, double in, double *out)
```

val	the reference Value Object
units	the pointer to the string declaring the source units
in	the source value to be converted
out	the returned converted value in the Value Object's units

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Value Object

Transfer Values

```
icode = caps_transferValues(capsObj src, enum tmethod, capsObj dst,
                            int *nErr, capsErrs **errs)
```

src	the source input Value Object (not for `Value` or `Tuple` vtypes) – or – DataSet Object
tmethod	0 – copy, 1 – integrate, 2 – weighted average – (1 & 2 only for DataSet `src`)
dst	the destination Value Object to receive the data Notes: ● Must not be GeometryOut or AnalysisOut ● Shapes must be compatible ● Overwrites any Linkage
nErr	the returned number of errors generated – 0 means no errors
errs	the returned CAPS error structure – NULL with no errors
icode	integer return code

CAPS API – Value Object

Establish Linkage

```
icode = caps_makeLinkage(capsObj link, enum tmethod, capsObj trgt)
```

link	linking Value Object (not for `Value` or `Tuple` vtypes or Value subtype User) – or – DataSet Object
tmethod	0 – copy, 1 – integrate, 2 – weighted average – (1 & 2 only for DataSet `link`)
trgt	the target Value Object which will get its data from `link` Notes: ● Must not be GeometryOut or AnalysisOut ● Shapes must be compatible ● `link` = NULL removes any Linkage
icode	integer return code

Note: circular linkages are not allowed!

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Attributes

Get Attribute by name

```
icode = caps_attrByName(capsObj object, char *name, capsObj *attr)
```

object	any CAPS Object
name	a string referring to the Attribute name
attr	the returned User Value Object (must be deleted when no longer needed)
icode	integer return code

Get Attribute by index

```
icode = caps_attrByIndex(capsObj object, int in, capsObj *attr)
```

object	any CAPS Object
in	the index (bias 1) to the list of Attributes
attr	the returned User Value Object (must be deleted when no longer needed) Attribute name is the Value Object name
icode	integer return code

Note: The *shape* of the original Value Object is not maintained, but the length is correct.

CAPS API – Attributes

Set an Attribute

```
icode = caps_setAttr(capsObj object, char *name, capsObj attr)
```

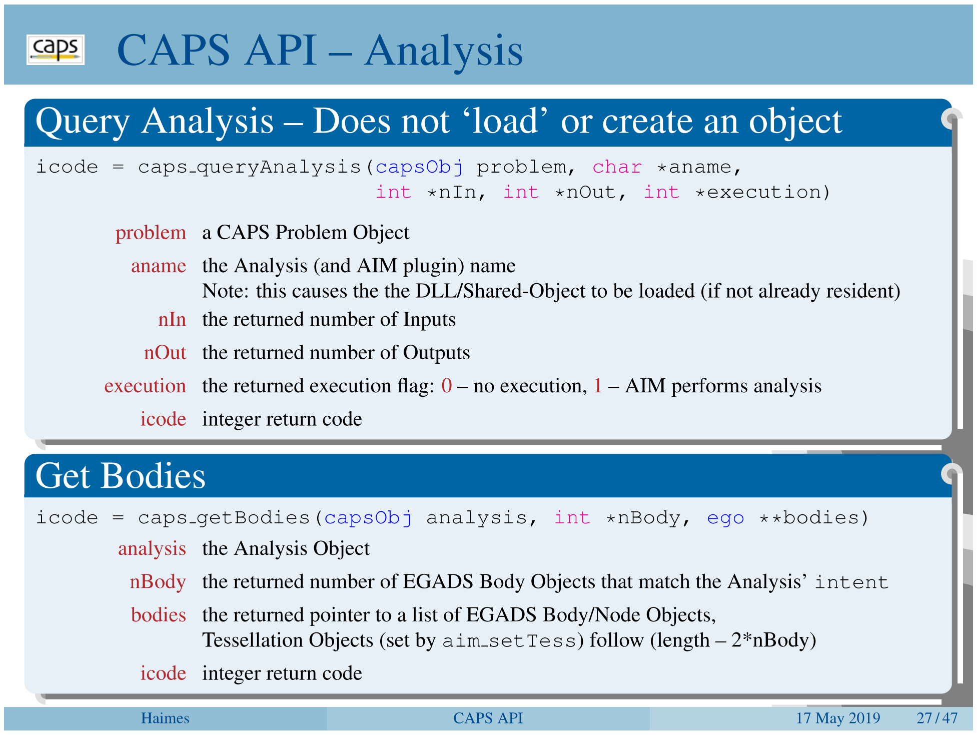
object	any CAPS Object
name	a string referring to the Attribute name – NULL: use name in `attr` Note: an existing Attribute of this name is overwritten with the new value
attr	the Value Object containing the attribute The attribute will not maintain the Value Object's *shape*
icode	integer return code

Delete an Attribute

```
icode = caps_deleteAttr(capsObj object, char *name)
```

object	any CAPS Object
name	a string referring to the Attribute to delete NULL deletes all attributes attached to the Object
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



CAPS API – Analysis

Query Analysis – Does not ‘load’ or create an object

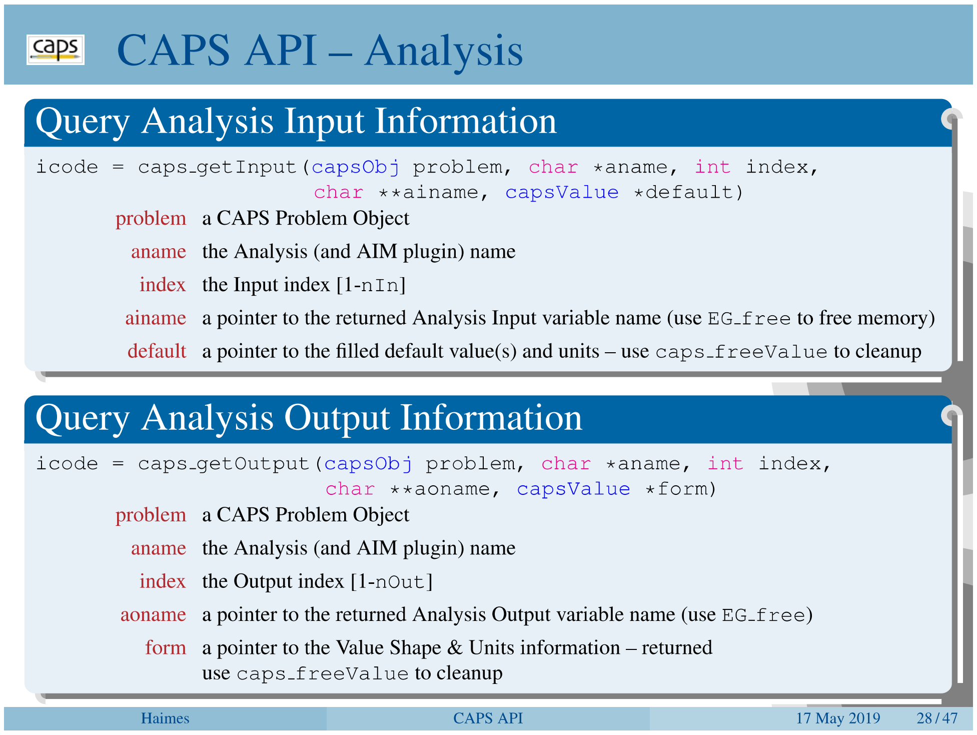
```
icode = caps_queryAnalysis(capsObj problem, char *aname,
                           int *nIn, int *nOut, int *execution)
```

problem	a CAPS Problem Object
aname	the Analysis (and AIM plugin) name Note: this causes the the DLL/Shared-Object to be loaded (if not already resident)
nIn	the returned number of Inputs
nOut	the returned number of Outputs
execution	the returned execution flag: 0 – no execution, 1 – AIM performs analysis
icode	integer return code

Get Bodies

```
icode = caps_getBodies(capsObj analysis, int *nBody, ego **bodies)
```

analysis	the Analysis Object
nBody	the returned number of EGADS Body Objects that match the Analysis’ `intent`
bodies	the returned pointer to a list of EGADS Body/Node Objects, Tessellation Objects (set by `aim_setTess`) follow (length – 2*nBody)
icode	integer return code



CAPS API – Analysis

Query Analysis Input Information

```
icode = caps_getInput(capsObj problem, char *aname, int index,
                      char **ainame, capsValue *default)
```

problem	a CAPS Problem Object
aname	the Analysis (and AIM plugin) name
index	the Input index [1-`nIn`]
ainame	a pointer to the returned Analysis Input variable name (use `EG_free` to free memory)
default	a pointer to the filled default value(s) and units – use `caps_freeValue` to cleanup

Query Analysis Output Information

```
icode = caps_getOutput(capsObj problem, char *aname, int index,
                       char **aoname, capsValue *form)
```

problem	a CAPS Problem Object
aname	the Analysis (and AIM plugin) name
index	the Output index [1-`nOut`]
aoname	a pointer to the returned Analysis Output variable name (use `EG_free`)
form	a pointer to the Value Shape & Units information – returned use `caps_freeValue` to cleanup

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Analysis

Load Analysis into a Problem

```
icode = caps_load(capsObj problem, char *aname, char *apath,
                  char *unitSys, char *intent, int naobj,
                  capsObj *aobjs, capsObj *analysis)
```

problem	a CAPS Problem Object
aname	the Analysis (and AIM plugin) name Note: this causes the the DLL/Shared-Object to be loaded (if not already resident)
apath	the absolute filesystem path to both read and write files this is required even if the AIM does not use the the filesystem, so that the combination of `aname` and `apath` is unique
unitSys	pointer to string describing the unit system to be used by the AIM (can be NULL) see specific AIM documentation for a list of strings for which the AIM will respond
intent	the *intent* character string used to pass Bodies to the AIM, NULL – no filtering
naobj	the number of *parent* Analysis Object(s)
aobjs	a list of the *parent* Analysis Object(s) – may be NULL if `naobj` == 0
analysis	the resultant Analysis Object
icode	integer return code

CAPS API – Analysis

Initialize Analysis from another Analysis Object

```
icode = caps_dupAnalysis(capsObj from, char *apath, int naobj,
                         capsObj *aobjs, capsObj *analysis)
```

from	an existing CAPS Analysis Object
apath	the absolute filesystem path to both read and write files required so that the combination of `aname` and `apath` is unique
naobj	the number of *parent* Analysis Object(s)
aobjs	a list of the *parent* Analysis Object(s) – may be NULL if `naobj` == 0
analysis	the resultant Analysis Object
icode	integer return code

Get Dirty Analysis Object(s)

```
icode = caps_dirtyAnalysis(capsObj object, int *nAobj, capsObj **aobjs)
```

problem	a CAPS Problem, Bound or Analysis Object
nAobjs	the returned number of *dirty* Analysis Objects
aobjs	a returned pointer to the list of *dirty* Analysis Objects (*freeable*)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Analysis

Initialize Analysis from another Analysis Object

```
icode = caps_dupAnalysis(capsObj from, char *apath, int naobj,
                         capsObj *aobjs, capsObj *analysis)
```

from	an existing CAPS Analysis Object
apath	the absolute filesystem path to both read and write files required so that the combination of `aname` and `apath` is unique
naobj	the number of *parent* Analysis Object(s)
aobjs	a list of the *parent* Analysis Object(s) – may be NULL if `naobj` == 0
analysis	the resultant Analysis Object
icode	integer return code

Get Dirty Analysis Object(s)

```
icode = caps_dirtyAnalysis(capsObj object, int *nAobj, capsObj **aobjs)
```

problem	a CAPS Problem, Bound or Analysis Object
nAobjs	the returned number of *dirty* Analysis Objects
aobjs	a returned pointer to the list of *dirty* Analysis Objects (*freeable*)
icode	integer return code

CAPS API – Analysis

Generate Analysis Inputs

```
icode = caps_preAnalysis(capsObj analysis, int *nErr, capsErrs **errs)
```

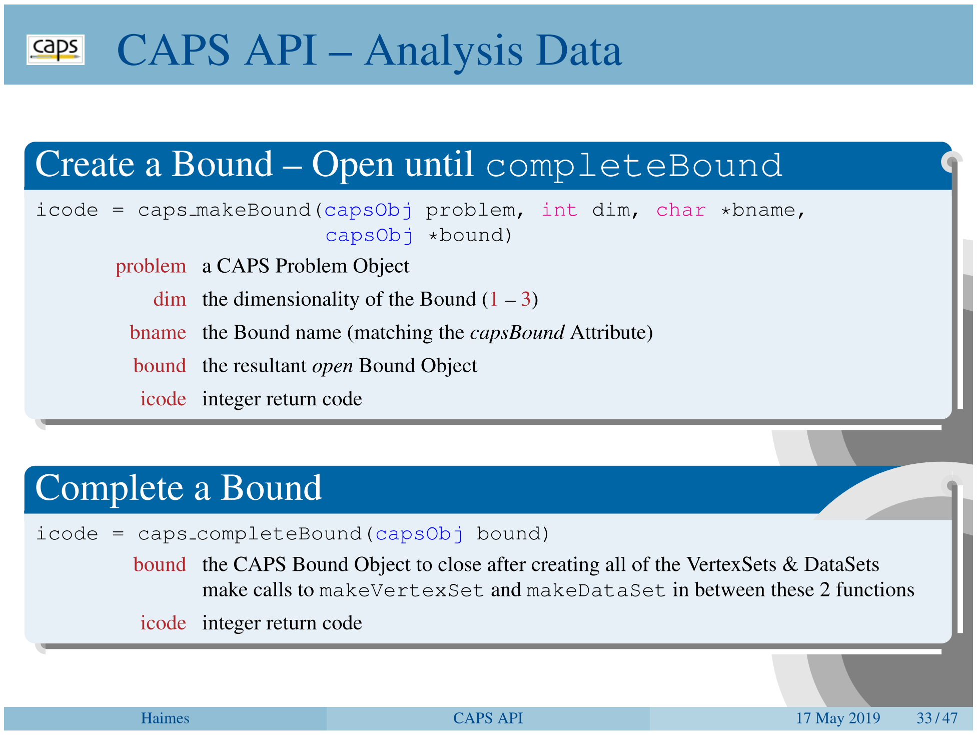
analysis	the Analysis (or Problem) Object a *Geometry*-only regen is forced when this is a Problem Object
nErr	the returned number of errors generated – 0 means no errors
errs	the returned CAPS error structure – NULL with no errors
icode	integer return code

Mark Analysis as Run

```
icode = caps_postAnalysis(capsObj analysis, capsOwn current, int *nErr,
                          capsErrs **errors)
```

analysis	the Analysis Object Note: this clears all Analysis Output Objects to force reloads/recomputes
current	the CAPS owner structure information for the run
nErr	the returned number of errors generated – 0 means no errors
errors	the returned CAPS error structure – NULL with no errors
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



CAPS API – Analysis Data

Create a Bound – Open until `completeBound`

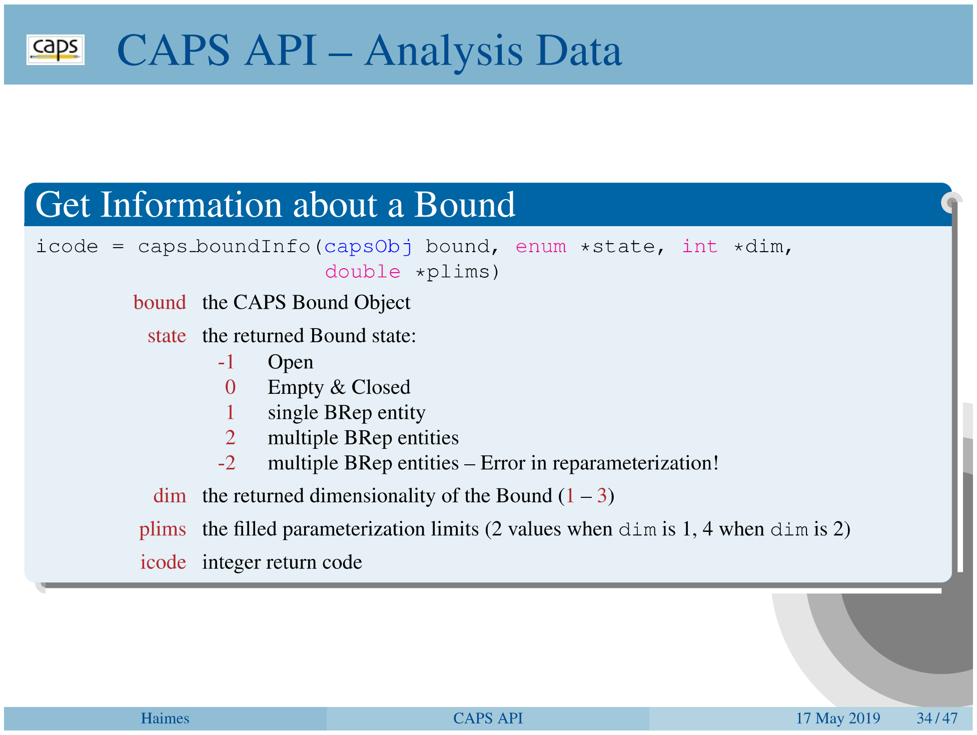
```
icode = caps_makeBound(capsObj problem, int dim, char *bname,
                       capsObj *bound)
```

problem	a CAPS Problem Object
dim	the dimensionality of the Bound (1 – 3)
bname	the Bound name (matching the *capsBound* Attribute)
bound	the resultant *open* Bound Object
icode	integer return code

Complete a Bound

```
icode = caps_completeBound(capsObj bound)
```

bound	the CAPS Bound Object to close after creating all of the VertexSets & DataSets make calls to `makeVertexSet` and `makeDataSet` in between these 2 functions
icode	integer return code



CAPS API – Analysis Data

Get Information about a Bound

```
icode = caps_boundInfo(capsObj bound, enum *state, int *dim,
                       double *plims)
```

bound	the CAPS Bound Object
state	the returned Bound state:

-1	Open
0	Empty & Closed
1	single BRep entity
2	multiple BRep entities
-2	multiple BRep entities – Error in reparameterization!

dim	the returned dimensionality of the Bound (1 – 3)
plims	the filled parameterization limits (2 values when `dim` is 1, 4 when `dim` is 2)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Analysis Data

Make a VertexSet

```
icode = caps_makeVertexSet(capsObj bound, capsObj analysis,
                           char *vname, capsObj *vset)
```

bound	an input *open* CAPS Bound Object
analysis	the Analysis Object (NULL – Unconnected)
vname	a character string naming the VertexSet (can be NULL for a Connected VertexSet)
vset	the returned VertexSet Object
icode	integer return code

Get Info about a VertexSet

```
icode = caps_vertexSetInfo(capsObj vset, int *nGpts, int *nDpts,
                           capsObj *bound, capsObj *analysis)
```

vset	the VertexSet Object
nGpts	the returned number of *Geometry* points in the VertexSet
nDpts	the returned number of point *Data* positions in the VertexSet
bound	the returned associated Bound Object
analysis	the returned associated Analysis Object (NULL – Unconnected)
icode	integer return code

CAPS API – Analysis Data

Fill VertexSets for cyclic/incremental transfers

```
icode = caps_fillVertexSets(capsObj bound, int *nErr, capsErrs **errs)
```

bound	an input *closed* CAPS Bound Object
nErr	the returned number of errors generated – 0 means no errors
errs	the returned CAPS error structure – NULL with no errors
icode	integer return code

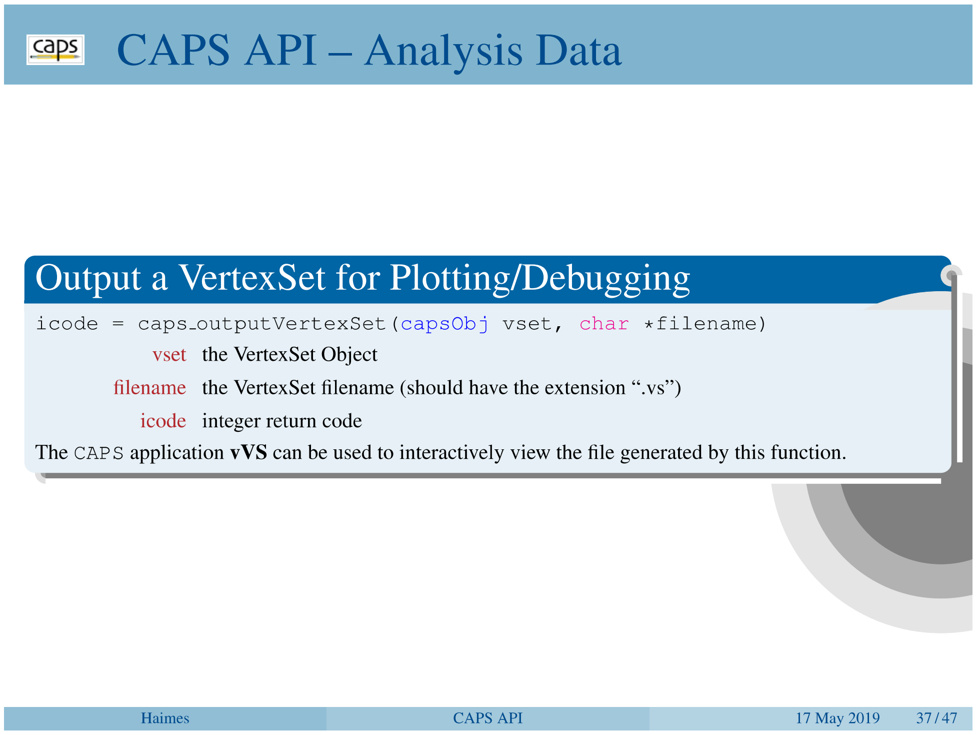
Note: Causes the filling of the VertexSets owned by the Bound by forcing the invocation of the appropriate `aimDiscr` functions in the AIM. Under normal circumstances this is deferred to the last postAnalysis call of the collected VertexSets.

Fill an Unconnected VertexSet

```
icode = caps_fillUnVertexSet(capsObj vset, int npts, double *xyzs)
```

vset	the input Unconnected VertexSet Object
npts	the number of points in the VertexSet
xyzs	the point positions (3*`npts` in length)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



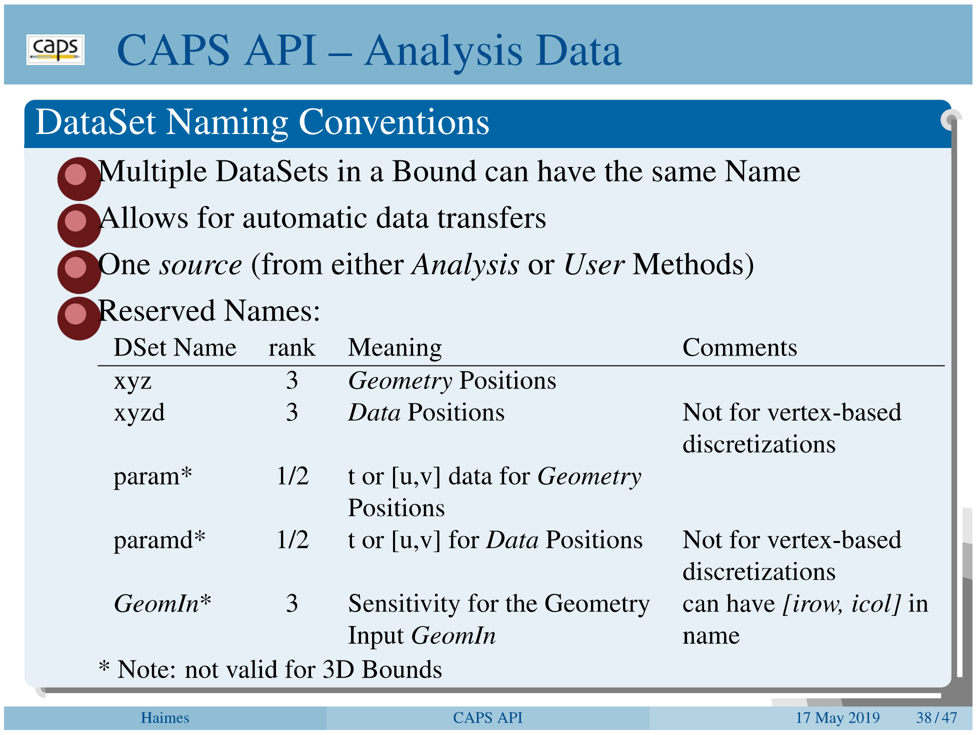
CAPS API – Analysis Data

Output a VertexSet for Plotting/Debugging

```
icode = caps_outputVertexSet(capsObj vset, char *filename)
```

vset	the VertexSet Object
filename	the VertexSet filename (should have the extension ".vs")
icode	integer return code

The `CAPS` application **vVS** can be used to interactively view the file generated by this function.



CAPS API – Analysis Data

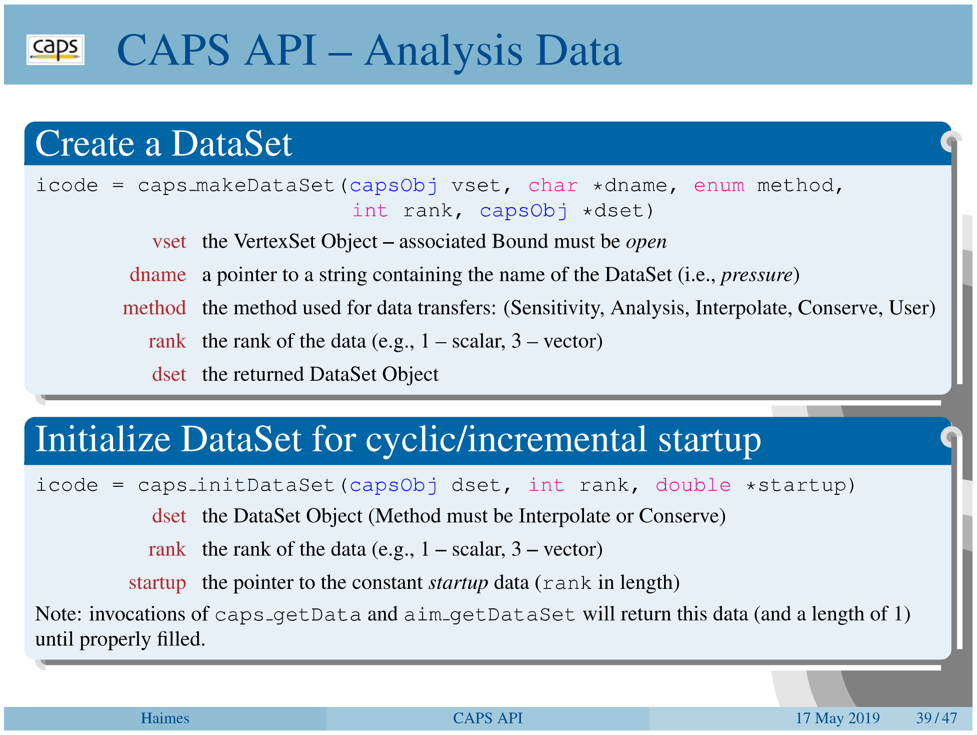
DataSet Naming Conventions

- Multiple DataSets in a Bound can have the same Name
- Allows for automatic data transfers
- One *source* (from either *Analysis* or *User* Methods)
- Reserved Names:

DSet Name	rank	Meaning	Comments
xyz	3	*Geometry* Positions	
xyzd	3	*Data* Positions	Not for vertex-based discretizations
param*	1/2	t or [u,v] data for *Geometry* Positions	
paramd*	1/2	t or [u,v] for *Data* Positions	Not for vertex-based discretizations
*GeomIn**	3	Sensitivity for the Geometry Input *GeomIn*	can have *[irow, icol]* in name

* Note: not valid for 3D Bounds

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



CAPS API – Analysis Data

Create a DataSet

```
icode = caps_makeDataSet(capsObj vset, char *dname, enum method,
                         int rank, capsObj *dset)
```

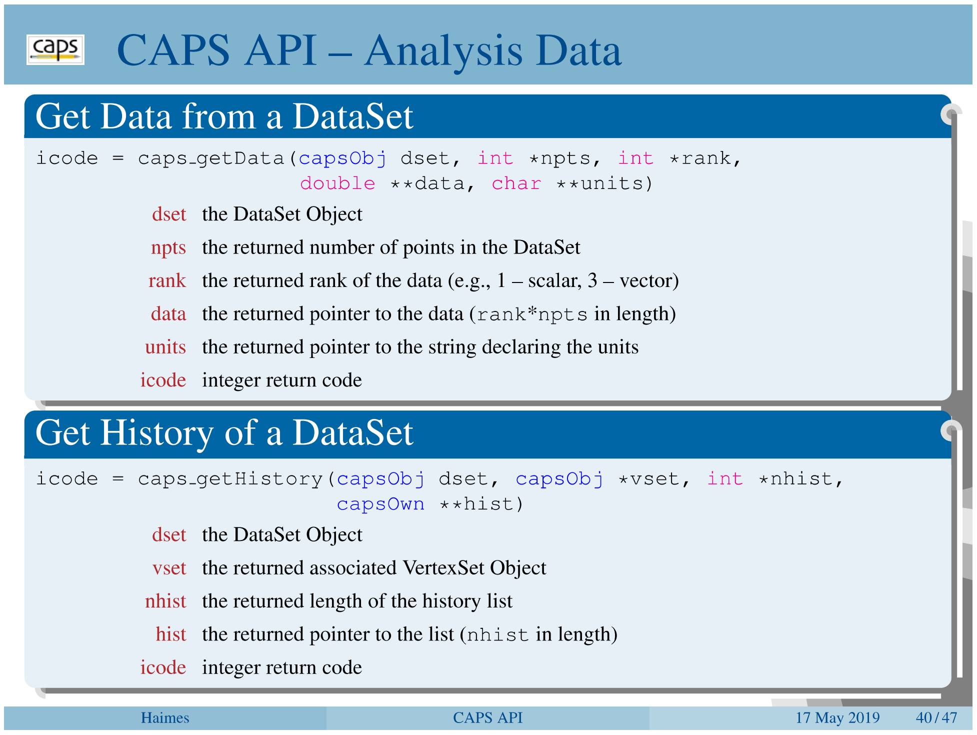
vset	the VertexSet Object – associated Bound must be *open*
dname	a pointer to a string containing the name of the DataSet (i.e., *pressure*)
method	the method used for data transfers: (Sensitivity, Analysis, Interpolate, Conserve, User)
rank	the rank of the data (e.g., 1 – scalar, 3 – vector)
dset	the returned DataSet Object

Initialize DataSet for cyclic/incremental startup

```
icode = caps_initDataSet(capsObj dset, int rank, double *startup)
```

dset	the DataSet Object (Method must be Interpolate or Conserve)
rank	the rank of the data (e.g., 1 – scalar, 3 – vector)
startup	the pointer to the constant *startup* data (`rank` in length)

Note: invocations of `caps_getData` and `aim_getDataSet` will return this data (and a length of 1) until properly filled.



CAPS API – Analysis Data

Get Data from a DataSet

```
icode = caps_getData(capsObj dset, int *npts, int *rank,
                     double **data, char **units)
```

dset	the DataSet Object
npts	the returned number of points in the DataSet
rank	the returned rank of the data (e.g., 1 – scalar, 3 – vector)
data	the returned pointer to the data (`rank*npts` in length)
units	the returned pointer to the string declaring the units
icode	integer return code

Get History of a DataSet

```
icode = caps_getHistory(capsObj dset, capsObj *vset, int *nhist,
                        capsOwn **hist)
```

dset	the DataSet Object
vset	the returned associated VertexSet Object
nhist	the returned length of the history list
hist	the returned pointer to the list (`nhist` in length)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Analysis Data

Put *User* Data into a DataSet

```
icode = caps_setData(capsObj dset, int nverts, int rank, double *data,
                     char *units)
```

dset	the DataSet Object
nverts	the number of points in data – must match declared `npts`
rank	the rank of the data – must match declared `rank` (e.g., 1 – scalar, 3 – vector)
data	a pointer to the data (`rank*nverts` in length)
units	the pointer to the string declaring the units
icode	integer return code

Get DataSet Objects by Name

```
icode = caps_getDataSets(capsObj bound, char *dname, int *nobj,
                         capsObj **dsets)
```

bound	an input CAPS Bound Object
dname	a pointer to a string containing the name of the DataSet
nobj	the returned number of Objects with the name
dsets	a returned pointer to the list of DataSet Objects (*freeable*)
icode	integer return code

CAPS API – Analysis Data

Get Triangulations for a 2D VertexSet

```
icode = caps_triangulate(capsObj vset, int *nGtris, int **Gtris,
                         int *nDtris, int **Dtris)
```

vset	the input CAPS Connected VertexSet Object
nGtris	the returned number of *Geometry*-based Triangles
Gtris	the returned pointer to a list of indices (bias 1) referencing *Geometry*-based points (3*`nGtris` in length) – *freeable*
nDtris	the returned number of *Data*-based Triangles (0 if discretization is vertex based)
Dtris	the returned pointer to a list of indices (bias 1) referencing *Data*-based points (3*`nDtris` in length) – *freeable*
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS API – Analysis (AIM) Debug

Backdoor AIM Specific Communication

```
icode = caps_AIMbackdoor(capsObj analysis, char *JSONin,
                         char **JSONout)
```

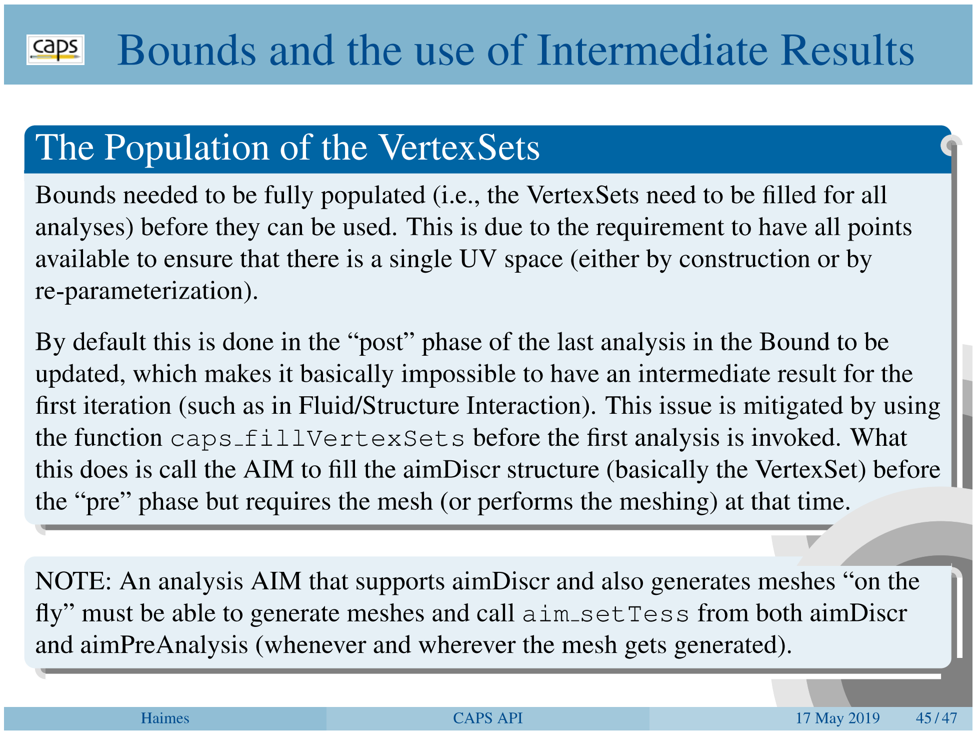
analysis	the Analysis Object
JSONin	a pointer to a character string that AIM function `aimBackdoor` will respond to.
JSONout	a returned pointer to a character string that AIM function `aimBackdoor` creates and passes back as the result to the request (may be *freeable* – depending on the AIM).
icode	integer return code

Note: Look at the specific AIM documentation to determine if it will respond and to what JSONin commands.

CAPS Return Codes

CAPS_SUCCESS	0	CAPS_CIRCULARLINK	-319
CAPS_BADRANK	-301	CAPS_UNITERR	-320
CAPS_BADDSETNAME	-302	CAPS_NULLBLIND	-321
CAPS_NOTFOUND	-303	CAPS_SHAPEERR	-322
CAPS_BADINDEX	-304	CAPS_LINKERR	-323
CAPS_NOTCHANGED	-305	CAPS_MISMATCH	-324
CAPS_BADTYPE	-306	CAPS_NOTPROBLEM	-325
CAPS_NULLVALUE	-307	CAPS_RANGEERR	-326
CAPS_NULLNAME	-308	CAPS_DIRTY	-327
CAPS_NULLOBJ	-309	CAPS_HIERARCHERR	-328
CAPS_BADOBJECT	-310	CAPS_STATEERR	-329
CAPS_BADVALUE	-311	CAPS_SOURCEERR	-330
CAPS_PARAMBNDERR	-312	CAPS_EXISTS	-331
CAPS_NOTCONNECT	-313	CAPS_IOERR	-332
CAPS_NOTPARMTRIC	-314	CAPS_DIRERR	-333
CAPS_READONLYERR	-315	CAPS_NOTIMPLEMENT	-334
CAPS_FIXEDLEN	-316	CAPS_EXECERR	-335
CAPS_BADNAME	-317	CAPS_CLEAN	-336
CAPS_BADMETHOD	-318	CAPS_BADINTENT	-337

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



Bounds and the use of Intermediate Results

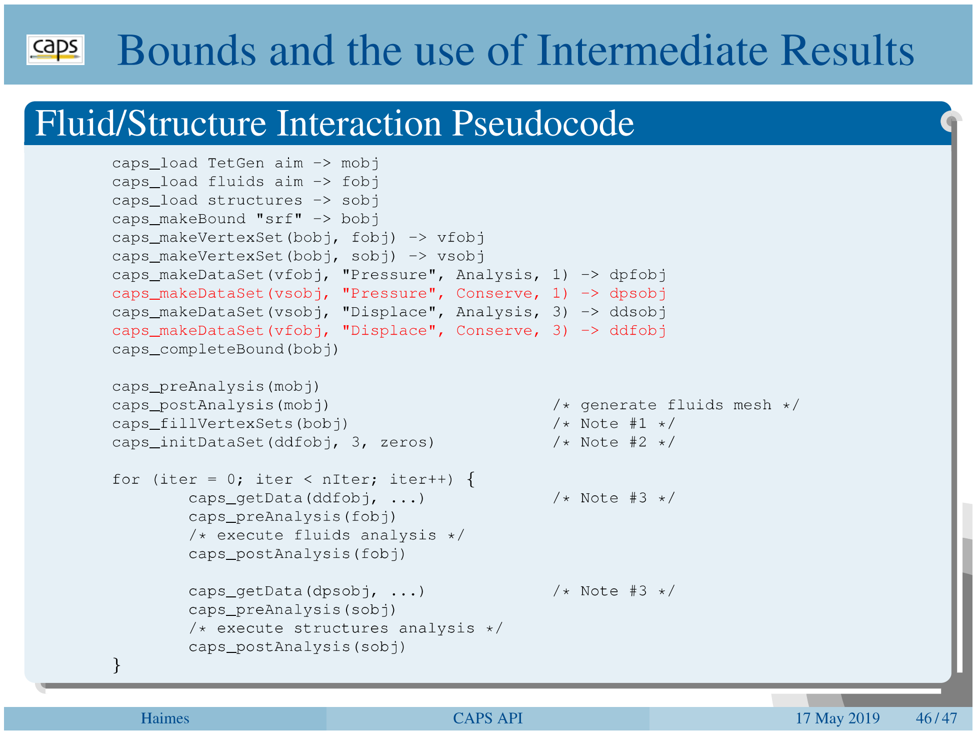
The Population of the VertexSets

Bounds needed to be fully populated (i.e., the VertexSets need to be filled for all analyses) before they can be used. This is due to the requirement to have all points available to ensure that there is a single UV space (either by construction or by re-parameterization).

By default this is done in the "post" phase of the last analysis in the Bound to be updated, which makes it basically impossible to have an intermediate result for the first iteration (such as in Fluid/Structure Interaction). This issue is mitigated by using the function `caps_fillVertexSets` before the first analysis is invoked. What this does is call the AIM to fill the aimDiscr structure (basically the VertexSet) before the "pre" phase but requires the mesh (or performs the meshing) at that time.

NOTE: An analysis AIM that supports aimDiscr and also generates meshes "on the fly" must be able to generate meshes and call `aim_setTess` from both aimDiscr and aimPreAnalysis (whenever and wherever the mesh gets generated).

Haimes   CAPS API   17 May 2019   45 / 47



Bounds and the use of Intermediate Results

Fluid/Structure Interaction Pseudocode

```
caps_load TetGen aim -> mobj
caps_load fluids aim -> fobj
caps_load structures -> sobj
caps_makeBound "srf" -> bobj
caps_makeVertexSet(bobj, fobj) -> vfobj
caps_makeVertexSet(bobj, sobj) -> vsobj
caps_makeDataSet(vfobj, "Pressure", Analysis, 1) -> dpfobj
caps_makeDataSet(vsobj, "Pressure", Conserve, 1) -> dpsobj
caps_makeDataSet(vsobj, "Displace", Analysis, 3) -> ddsobj
caps_makeDataSet(vfobj, "Displace", Conserve, 3) -> ddfobj
caps_completeBound(bobj)

caps_preAnalysis(mobj)
caps_postAnalysis(mobj)                    /* generate fluids mesh */
caps_fillVertexSets(bobj)                  /* Note #1 */
caps_initDataSet(ddfobj, 3, zeros)         /* Note #2 */

for (iter = 0; iter < nIter; iter++) {
        caps_getData(ddfobj, ...)          /* Note #3 */
        caps_preAnalysis(fobj)
        /* execute fluids analysis */
        caps_postAnalysis(fobj)

        caps_getData(dpsobj, ...)          /* Note #3 */
        caps_preAnalysis(sobj)
        /* execute structures analysis */
        caps_postAnalysis(sobj)
}
```

Haimes   CAPS API   17 May 2019   46 / 47

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Bounds and the use of Intermediate Results

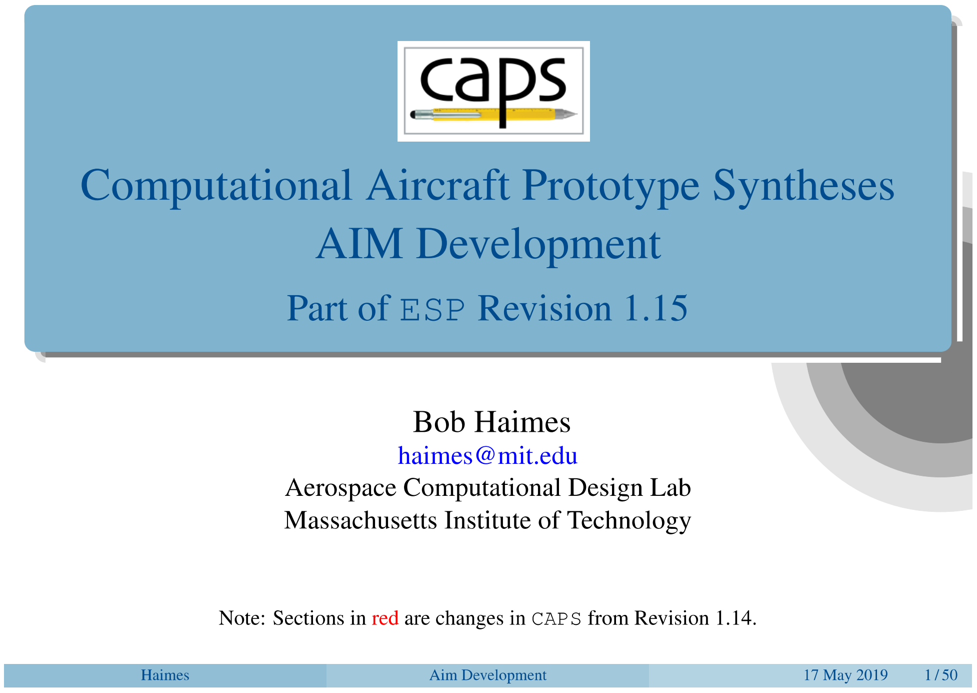
Pseudocode Notes

The fluids AIM requires the "Displace" values during its "pre" phase, just as the structural analysis AIM requires "Pressure" (i.e., loads) during its "pre" phase to fill in all the inputs.

1. `caps_fillVertexSets` calls aimDiscr in the fluids AIM, so that AIM must transfer the data from the TetGen AIM to populate the aimDiscr structure. The structures AIM can still do the tessellation in its aimDiscr function, but it will be invoked before any "pre" phase. Care must be taken so that any tessellation input data can be taken from the AIM inputs.
2. `caps_initDataSet` gets called to set the first displacement data to zeros, in that no structural analysis will have been run at start, but is needed by the fluids.
3. `caps_getData` is currently required to actually do the interpolation/conservative data transfer (i.e., it cannot be done in the AIM by the invocation of `aim_getDataSet`). This will be changed in the future, so these calls will not be required, but current scripts and code will still function.
4. The lines in red now cause `aimUsesDataSet` to be invoked to determine if the DataSet is required by the Analysis (and will make it *dirty*).

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

APPENDIX C – AIM DEVELOPMENT



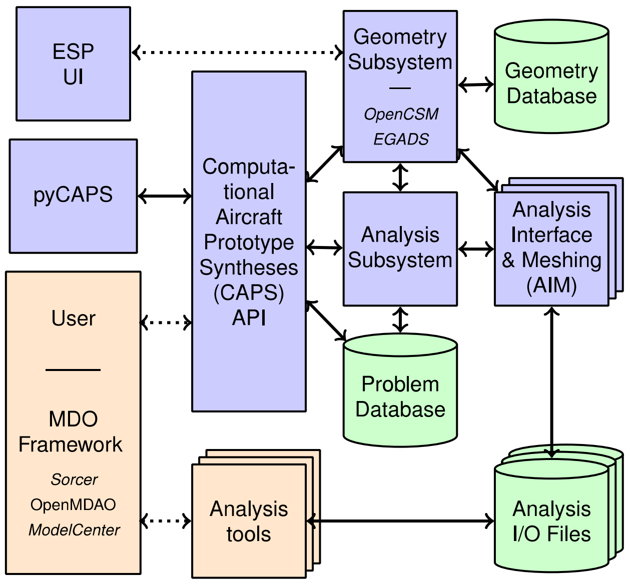
Computational Aircraft Prototype Syntheses
AIM Development
Part of ESP Revision 1.15

Bob Haimes
haimes@mit.edu
Aerospace Computational Design Lab
Massachusetts Institute of Technology

Note: Sections in red are changes in CAPS from Revision 1.14.

Haimes | Aim Development | 17 May 2019 | 1 / 50

CAPS Infrastructure in ESP



Haimes | Aim Development | 17 May 2019 | 2 / 50

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS Objects

Object-based Not *Object Orientated*

- Like *egos* in `EGADS`
- Pointer to a C structure – allows for an function-based API
- Treated as *blind pointers* (i.e., not meant to be dereferenced)
  Header info used to determine how to dereference the *pointer*
- API Functions
  - Returns an `int` error code or `CAPS_SUCCESS`
  - Usually have one (or more) input Objects
  - Can have an output Object (usually at the end of the argument list)
- Can interface with multiple compiled languages

See $ESP_ROOT/doc/CAPSapi.pdf

CAPS Definitions

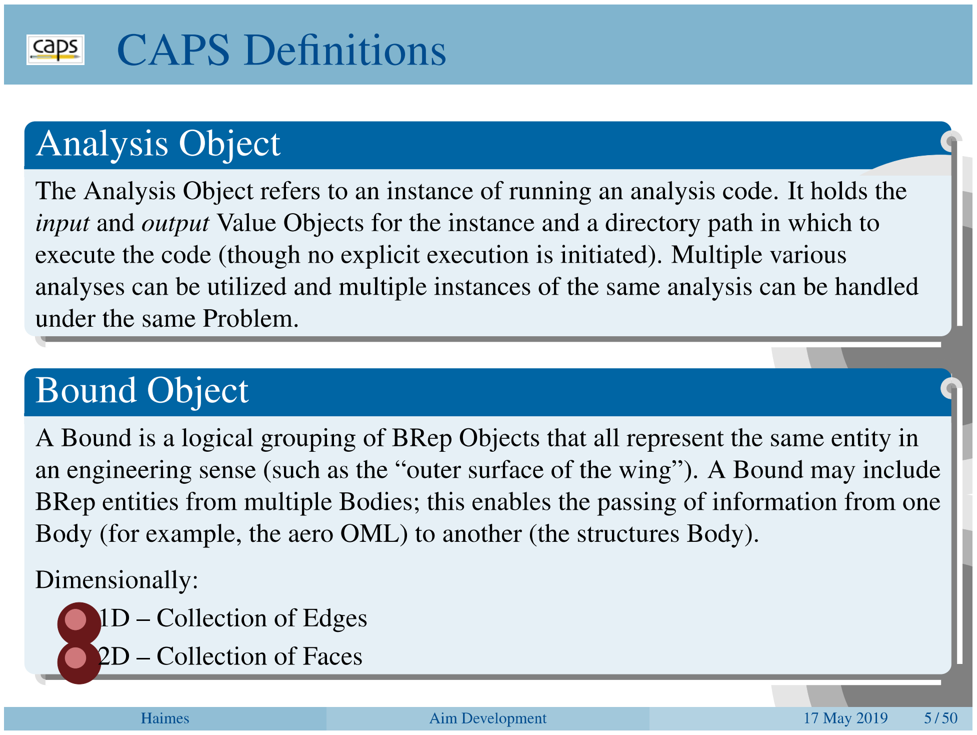
Problem Object

The Problem is the top-level *container* for a single mission. It maintains a single set of interrelated geometric models, analyses to be executed, connectivity and data associated with the run(s), which can be both multi-fidelity and multidisciplinary. There can be multiple Problems in a single execution of CAPS and each Problem is designed to be *thread safe* allowing for multi-threading of CAPS at the highest level.

Value Object

A Value Object is the fundamental data container that is used within CAPS. It can represent *inputs* to the Analysis and Geometry subsystems and *outputs* from both. Also Value Objects can refer to *mission* parameters that are stored at the top-level of the CAPS database. The values contained in any *input* Value Object can be bypassed by the *linkage* connection to another Value (or *DataSet*) Object of the same *shape*. Attributes are also cast to temporary (*User*) Value Objects.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.



CAPS Definitions

Analysis Object

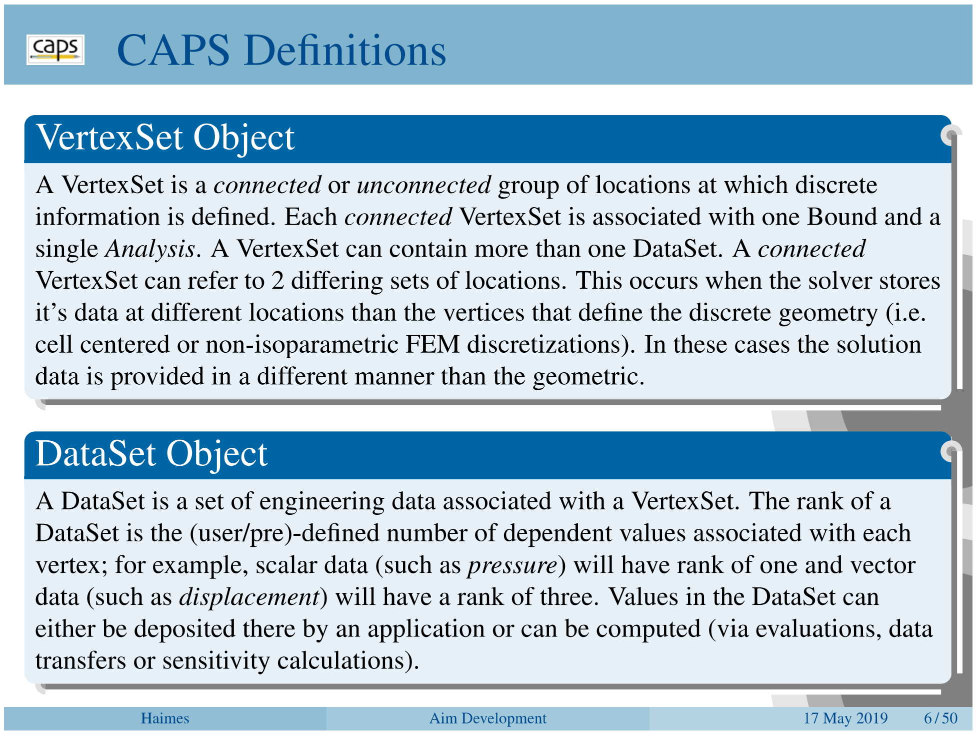
The Analysis Object refers to an instance of running an analysis code. It holds the *input* and *output* Value Objects for the instance and a directory path in which to execute the code (though no explicit execution is initiated). Multiple various analyses can be utilized and multiple instances of the same analysis can be handled under the same Problem.

Bound Object

A Bound is a logical grouping of BRep Objects that all represent the same entity in an engineering sense (such as the "outer surface of the wing"). A Bound may include BRep entities from multiple Bodies; this enables the passing of information from one Body (for example, the aero OML) to another (the structures Body).

Dimensionally:

- 1D – Collection of Edges
- 2D – Collection of Faces



CAPS Definitions

VertexSet Object

A VertexSet is a *connected* or *unconnected* group of locations at which discrete information is defined. Each *connected* VertexSet is associated with one Bound and a single *Analysis*. A VertexSet can contain more than one DataSet. A *connected* VertexSet can refer to 2 differing sets of locations. This occurs when the solver stores it's data at different locations than the vertices that define the discrete geometry (i.e. cell centered or non-isoparametric FEM discretizations). In these cases the solution data is provided in a different manner than the geometric.

DataSet Object

A DataSet is a set of engineering data associated with a VertexSet. The rank of a DataSet is the (user/pre)-defined number of dependent values associated with each vertex; for example, scalar data (such as *pressure*) will have rank of one and vector data (such as *displacement*) will have a rank of three. Values in the DataSet can either be deposited there by an application or can be computed (via evaluations, data transfers or sensitivity calculations).

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS Objects

Object	SubTypes	Parent Object
capsProblem	Parametric, Static	
capsValue	GeometryIn, GeometryOut, Branch, Parameter, User	capsProblem, capsValue
capsAnalysis		capsProblem
capsValue	AnalysisIn, AnalysisOut	capsAnalysis, capsValue
capsBound		capsProblem
capsVertexSet	Connected, Unconnected	capsBound
capsDataSet	User, Analysis, Interpolate, Conserve, Builtin, Sensitivity	capsVertexSet

Body Objects are EGADS Objects (egos)

CAPS Body Filtering

Filtering the active CSM Bodies occurs at two different stages, once in the CAPS framework, and once in the AIMs. The filtering in the CAPS framework creates sub-groups of Bodies from the CSM stack that are passed to the specified AIM. Each AIM instance is then responsible for selecting the appropriate Bodies from the list it has received.

The filtering is performed by using two Body attributes: "capsAIM" and "capsIntent".

Filtering within AIM Code

Each AIM can adopt it's own filtering scheme for down-selecting how to use each Body it receives. The "capsIntent" string is accessible to the AIM, but it is for information only.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

CAPS Body Filtering

CSM AIM targeting: "capsAIM"

The CSM script generates Bodies which are designed to be used by specific AIMs. The AIMs that the Body is designed for is communicated to the CAPS framework via the "capsAIM" string attribute. This is a semicolon-separated string with the list of AIM names. Thus, the CSM author can give a clear indication to which AIMs should use the Body. For example, a body designed for a CFD calculation could have:

ATTRIBUTE capsAIM $su2AIM;fun3dAIM;cart3dAIM

CAPS AIM Instantiation: "capsIntent"

The "capsIntent" Body attribute is used to disambiguate which AIM instance should receive a given Body targeted for the AIM. An argument to `caps_load` accepts a semicolon-separated list of keywords when an AIM is instantiated in CAPS/pyCAPS. Bodies from the "capsAIM" selection with a matching string attribute "capsIntent" are passed to the AIM instance. The attribute "capsIntent" is a semicolon-separated list of keywords. If the string to `caps_load` is NULL, all Bodies with a "capsAIM" attribute that matches the AIM name are given to the AIM instance.

Analysis Interface & Meshing – Intro 1/2

- Hides all of the individual Analysis details (and peculiarities)
  - Individual plugin functions *translate* from the Analysis' perspective back and forth to CAPS
  - Provides a direct connection to BRep geometry and attribution through EGADS
- Outside the CAPS Object infrastructure
  - Use of C structures
  - AIM Utility library (with the *context* enbedded in `aimInfo`)
- An AIM plugin is required for each Analysis code at:
  - a specific *intent*
  - a specific *mode* (i.e., where the inputs may be different)

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Analysis Interface & Meshing – Intro 2/2

- AIMs can be hierarchical
  - Parent Analysis Objects specified at CAPS Analysis load
  - Parent and child AIMs can directly communicate
- Dynamically loaded at runtime – extendibility and extensibility

Windows	Dynamically Loaded Libraries (`name.dll`)
LINUX	Shared Objects (`name.so`)
MAC	*Bundles*, CAPS will use the `so` file extension

- Plugin names must be unique – loaded by the name

- † indicates memory handled by CAPS in the following functions i.e., CAPS will free these memory blocks when necessary

capsValue Structure 1/5

The capsValue Structure is simply the data found within a CAPS Value Object. `aimInputs` and `aimOutputs` must fill the structure with the *type*, *form* and optionally *units* of the data. `aimInputs` also sets the default value(s) in the *vals* member. The structure's members listed below must be filled (most have defaults).

Value Type – no default

The value *type* can be one of:

```
  enum capsvType {Boolean, Integer, Double, String, Tuple, Value};
```

Note:
The Value type in a capsValue is only supported at the CAPS level and not in AIMs

The tuple structure

```
typedef struct {
  char *name;                  /* the name */
  char *value;                 /* the value for the pair */
} capsTuple;
```

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

capsValue Structure 2/5

Shape of the Value – 0 is the default

dim can be one of:

- 0 scalar only
- 1 vector or scalar
- 2 scalar, vector or 2D array

Value Dimensions – 1 is the default

nrow and *ncol* set the dimension of the Value. If both are 1 this has a `scalar` shape. If either *nrow* or *ncol* are one then the shape is `vector`. If both are greater than 1 then this represents a 2D array of values.

Other enumerated constants

```
enum capsFixed   {Change, Fixed};
enum capsNull    {NotAllowed, NotNull, IsNull};
enum capstMethod {Copy, Integrate, Average};
```

capsValue Structure 3/5

Varying Length – the default is “Fixed”

The member *lfixed* indicates whether the length of the Value is allowed to change.

Varying Shape – the default is “Fixed”

The member *sfixed* indicates whether the *shape* of the Value is allowed to change.

Can Value be NULL? – the default is “NotAllowe

The member *nullVal* indicates whether the Value is or can be NULL
Options are found in `enum capsNULL`

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

capsValue Structure 4/5

capsValue Member Usage Notes

- *sfixed & dim*
  If the shape is “Fixed” then *nrow* and *ncol* must fit that shape (or a lesser dimension). [Note that the length can change if *lfixed* is “Change”.] If *sfixed* is “Change” then you change *dim* before changing *nrow* and *ncol* to a higher dimension than the current setting.
- *lfixed & nrow/ncol*
  If the length is “Fixed” then all updates of the Value(s) must match in both *nrow* and *ncol* (which presumes a “Fixed” shape).
- *nullVal & nrow/ncol*
  nrow and *ncol* should remain at their values even if the Value is NULL to maintain the dimension (and possibly length) when “Fixed”. To indicate a NULL all that is necessary is to set *nullVal* to “IsNull”. The actual allocated storage can remain in the *vals* member or set to NULL.
- Use `EG_alloc` to allocate any memory required for the *vals* member.

capsValue Structure 5/5

```
/*
 * structure for CAPS object -- VALUE
 */
typedef struct {
  int          type;              /* value type -- capsvType */
  int          length;            /* number of values */
  int          dim;               /* the dimension */
  int          nrow;              /* number of rows */
  int          ncol;              /* the number of columns */
  int          lfixed;            /* length is fixed -- capsFixed */
  int          sfixed;            /* shape is fixed -- capsFixed */
  int          nullVal;           /* NULL handling -- capsNull */
  int          pIndex;            /* parent index for vType = Value */
  union {
    int        integer;           /* single int -- length == 1 */
    int        *integers;         /* multiple ints */
    double     real;              /* single double -- length == 1 */
    double     *reals;            /* mutiple doubles */
    char       *string;           /* character string (no single char) */
    capsTuple  *tuple;            /* tuple (no single tuple) */
    capsObject *object;           /* single object -- not used in AIMs*/
    capsObject **objects;         /* multiple objects -- not used in AIMs */
  } vals;
  union {
    int        ilims[2];          /* integer limits */
    double     dlims[2];          /* double limits */
  } limits;
  char         *units;            /* the units for the values */
  capsObject   *link;             /* the linked object (or NULL) */
  int          linkMethod;        /* the link method -- capstMethod */
} capsValue;
```

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Plugin Functions

- Registration & Declaring Inputs / Outputs
- Pre-Analysis & Retrieving Output
  Write and read files – or – use Analyses API if available
- Discrete Support – Interpolation & Integration

AIM – Registration/Initialization

```
icode = aimInitialize(int ngIn, capsValue *gIn, int *qeFlg,
                      const char *unitSys, int *nIn, int *nOut,
                      int *nFields, char ***fnames, int **ranks)
```

ngIn	the number of *Geometry* Input value structures
gIn	a pointer to the list of *Geometry* Input value structures
qeFlg	on Input: 1 indicates a query and not an analysis instance; on Output: 1 specifies that the AIM executes the analysis
unitSys	a pointer to a character string declaring the unit system – can be NULL
nIn	the returned number of Inputs (minimum of 1)*
nOut	the returned number of possible Outputs*
nFields	the returned number of fields to responds to for DataSet filling
fnames	a returned pointer to a list of character strings with the field/DataSet names †
ranks	a returned pointer to a list of ranks associated with each field †
icode	integer return code (-) or AIM *instance* counter

*nIn & nOut should not depend on the intent

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Initialization

```
icode = aimInputs(int inst, void *aimInfo, int index, char **ainame,
                  capsValue *defval)
```

inst	the AIM *instance* index
aimInfo	the AIM context – NULL if called from `caps_getInput`
index	the Input index [1-`nIn`]
ainame	a returned pointer to the returned Analysis Input variable name
defval	a pointer to the filled default value(s) and units – CAPS will free any allocated memory
icode	integer return code

```
icode = aimOutputs(int inst, void *aimInfo, int index, char **aonam,
                   capsValue *form)
```

inst	the AIM *instance* index
aimInfo	the AIM context (used by the Utility Functions)
index	the Output index [1-`nOut`]
aonam	a returned pointer to the returned Analysis Output variable name
form	a pointer to the Value Shape & Units information – to be filled any actual values stored are ignored/freed
icode	integer return code

AIM – Dependent DataSet

Is the DataSet required by aimPreAnalysis – Optional

```
icode = aimUsesDataSet(int inst, void *aimInfo, const char *bname,
                       const char *dname, enum capsdMethod dMethod)
```

inst	the AIM *instance* index
aimInfo	the AIM context (used by the Utility Functions)
bname	the Bound name
dname	the DataSet name
dMethod	the data method used (either *Interpolate* or *Conserve*)
icode	integer return code – use `CAPS_NOTNEEDED` if not required

Called at `caps_makeDataSet`, when the data method used is either *Interpolate* or *Conserve*, for possible dependent VertexSets with `dname`. If it is dependent then the Analysis Object is made *dirty* when the DataSet needs updating.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – PreAnalysis

Parse Input data & Optionally Generate Input File(s)

```
icode = aimPreAnalysis(int inst, void *aimInfo, const char *apath,
                       capsValue *inputs, capsErrs **errs)
```

inst	the AIM *instance* index
aimInfo	the AIM context (used by the Utility Functions)
apath	the filesystem path where the input file(s) are to be written
inputs	the complete suite of Analysis inputs (`nIn` in length)
errs	a pointer to the returned structure where input error(s) occurred – NULL no errors
icode	integer return code

Called to prepare the input to an Analysis or prepare the input and execute the Analysis (based on qeFlg).

AIM – PostAnalysis & Termination

Perform any processing after the Analysis is run – Optional

```
icode = aimPostAnalysis(int inst, void *aimInfo, const char *apath,
                        capsErrs **errs)
```

inst	the AIM *instance* index
aimInfo	the AIM context (used by the Utility Functions)
apath	the filesystem path where the file(s) have been written
errs	a pointer to the returned structure where error(s) may have occurred – NULL no errors
icode	integer return code

Free up any memory the AIM has stored

```
void aimCleanup()
```

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Output Parsing

Calculate/Retrieve Output Information

```
icode = aimCalcOutput(int inst, void *aimInfo, const char *apath,
                      int index, capsValue *val, capsErrs **errors)
```

inst	the AIM *instance* index
aimInfo	the AIM context (used by the Utility Functions)
apath	the filesystem path where the Analysis output file(s) should be read
index	the Output index [1-`nOut`] for this single result
val	a pointer to the capsValue data to fill – CAPS will free any allocated memory
errors	a pointer to the returned error structure where output parsing error(s) occurred NULL with no errors
icode	integer return code

Called in a *lazy* manner and only when the output is needed (and after the Analysis is run).

AIM – Discrete Structure 1/5

Discrete Structure – Used to define a VertexSet

The CAPS *Discrete* data structure holds the spatial discretization information for a Bound. It defines reference positions for the location of the vertices that support the geometry and optionally the positions for the data locations (if these differ). This structure can contain a homogeneous or heterogeneous collection of element types and optionally specifies match positions for conservative data transfers.

EGADS Tessellation Object

- Not a requirement – but useful in dealing with sensitivities
- Requires triangles
- Can be constructed from an external mesh generator
  - Look at `EG_initTessBody`, `EG_setTessEdge`, `EG_setTessFace` & `EG_statusTessBody`
  - Make it part of CSM & CAPS by `aim_setTess`

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Discrete Structure 2/5

```
/* defines the element discretization type by the number of reference positions
 * (for geometry and optionally data) within the element.
 * simple tri:  nref = 3; ndata = 0; st = {0.0,0.0, 1.0,0.0, 0.0,1.0}
 * simple quad: nref = 4; ndata = 0; st = {0.0,0.0, 1.0,0.0, 1.0,1.0, 0.0,1.0}
 * internal triangles are used for the in/out predicates and represent linear
 *   triangles in [u,v] space.
 * ndata is the number of data referece positions, which can be zero for simple
 *   nodal or isoparametric discretizations.
 * match points are used for conservative transfers. Must be set when data
 *   and geometry positions differ, specifically for discontinuous mappings.
 * For example:
 *                        neighbors                          neighbors
 *       2          tri-side   vertices          4          side    vertices
 *      / \             0        1 2            / \          0        1 2
 *     /   \            1        2 0           5   3         1        2 3
 *    /     \           2        0 1          /  6  \        2        3 4
 *   0-------1                               /       \       3        4 5
 *                                          0----1----2      4        5 0
 *                        neighbors                          5        0 1
 *   3-------2      quad-side  vertices             nref = 7
 *   |       |          0        1 2
 *   |       |          1        2 3            6              neighbors
 *   |       |          2        3 0        3---.---2    quad-side  vertices
 *   0-------1          3        0 1        |       |        0        1 2
 *                                         7.   8   .5       1        2 3
 *                        neighbors         |       |        2        3 0
 *   4-------3        side     vertices     0---.---1        3        0 1
 *   |       |          0        1 2            4
 *   |       2          1        2 3
 *   |       |          2        3 4                nref = 9
 *   0-------1          3        4 0
 *                      4        0 1
 *       nref = 5
```

AIM – Discrete Structure 3/5

```
 */

typedef struct {
  int    nref;                 /* number of geometry reference points */
  int    ndata;                /* number of data ref points -- 0 data at ref */
  int    nmat;                 /* number of match points (0 -- match at
                                  geometry reference points) */
  int    ntri;                 /* number of triangles to represent the elem */
  double *gst;                 /* [s,t] geom reference coordinates in the
                                  element -- 2*nref in length */
  double *dst;                 /* [s,t] data reference coordinates in the
                                  element -- 2*ndata in length */
  double *matst;               /* [s,t] positions for match points - NULL
                                  when using reference points (2*nmat long) */
  int    *tris;                /* the triangles defined by geom reference indices
                                  (bias 1) -- 3*ntri in length */
} capsEleType;
```

You will usually have only a small number of element types.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Discrete Structure 4/5

```
/*
 * defines the element discretization for geometric and optionally data
 * positions.
 */
typedef struct {
  int   bIndex;                /* the Body index (bias 1) */
  int   tIndex;                /* the element type index (bias 1) */
  int   eIndex;                /* element owning index -- dim 1 Edge, 2 Face */
  int   *gIndices;             /* local indices (bias 1) geom ref positions,
                                  tess index -- 2*nref in length */
  int   *dIndices;             /* the vertex indices (bias 1) for data ref
                                  positions -- ndata in length or NULL */
  union {
    int tq[2];                 /* tri or quad (bias 1) for ntri <= 2 */
    int *poly;                 /* the multiple indices (bias 1) for ntri > 2 */
  } eTris;                     /* triangle indices  that make up the element */
} capsElement;
```

See AIAA paper 2014-0294 in the distribution for a more complete description ($ESP_ROOT/doc/Papers/AIAApaper2014-0294.pdf).

AIM – Discrete Structure 5/5

```
/* defines a discretized collection of Elements
 *
 * specifies the connectivity based on a collection of Element Types and the
 * elements referencing the types.
 */
typedef struct {
  int         dim;             /* dimensionality [1-3] */
  int         instance;        /* analysis instance */
  void        *aInfo;          /* AIM info */

                               /* below handled by the AIMs: */
  int         nPoints;         /* number of entries in the point definition */
  int         *mapping;        /* tessellation indices to the discrete space
                                  2*nPoints in len (body, global tess index) */
  int         nVerts;          /* number of data ref positions or unconnected */
  double      *verts;          /* data ref (3*nVerts) -- NULL if same as geom */
  int         *celem;          /* element containing vert (nVerts in len) or NULL */
  int         nTypes;          /* number of Element Types */
  capsEleType *types;          /* the Element Types (nTypes in length) */
  int         nElems;          /* number of Elements */
  capsElement *elems;          /* the Elements (nElems in length) */
  int         nDtris;          /* number of triangles to plot data */
  int         *dtris;          /* NULL for NULL verts -- indices into verts */
  void        *ptrm;           /* pointer for optional AIM use */
} capsDiscr;
```

See $ESP_ROOT/doc/capsDiscr.pdf for a more complete description.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Discrete Support

Fill-in the Discrete data for a Bound Object – Optional

```
icode = aimDiscr(char *tname, capsDiscr *discr)
```

tname	the Bound name Note: all of the BRep entities are examined for the attribute **capsBound**. Any that match `tname` must be included when filling this `capsDiscr`.
discr	the Discrete structure to fill Note: the AIM *instance*, AIM *info* pointer and the dimensionality have been filled in before this function is invoked.
icode	integer return code

Frees up data in a Discrete Structure – Optional

```
icode = aimFreeDiscr(capsDiscr *discr)
```

discr	the Discrete Structure to have its members freed
icode	integer return code

AIM – Discrete Support

Return Element in the *Mesh* – Optional

```
icode = aimLocateElement(capsDiscr *discr, double *params,
                         double *param, int *eIndex, double *bary)
```

discr	the input Discrete Structure
params	the input global *parametric* space (at all of the *geometry* support positions) rank is the dimensionality (t for 1D, $[u, v]$ for 2D and $[x, y, z]$ for 3D)
param	the input requested parametric position in `params` (dimensionality in length)
eIndex	the returned element index in the `discr` where the position was found (1 bias)
bary	the resultant Barycentric/reference position in the element `eIndex`
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Data Transfers

Data Associated with the Discrete Structure – Optional

```
icode = aimTransfer(capsDiscr *discr, const char *fname, int npts,
                    int rank, double *data, char **units)
```

discr	the input Discrete Structure
fname	the field name to that corresponds to the fill
npts	the number of points to be filled
rank	the rank of the data
data	a pointer associated with the data to be filled (`rank*npts` in length)
units	the returned pointer to the string declaring the units † return NULL to indicate unitless values
icode	integer return code

Fills in the DataSet Object

AIM – Data Transfers

Interpolation on the Bound – Optional

```
icode = aimInterpolation(capsDiscr *discr, const char *name,
                         int eIndex, double *bary, int rank,
                         double *data, double *result)
icode = aimInterpolateBar(capsDiscr *discr, const char *name,
                          int eIndex, double *bary, int rank,
                          double *r_bar, double *d_bar)
```

discr	the input Discrete Structure
name	a pointer to the input DataSet name string
eIndex	the input target element index (1 bias) in the Discrete Structure
bary	the input Barycentric/reference position in the element `eIndex`
rank	the input rank of the data
data	values at the data (or geometry) positions
result	the filled in results (`rank` in length)
r_bar	input d(objective)/d(result)
d_bar	returned d(objective)/d(data)
icode	integer return code

Forward and *reverse differentiated* functions

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM– Data Transfers

Element Integration on the Bound – Optional

```
icode = aimIntegration(capsDiscr *discr, const char *name,
                       int eIndex, int rank,
                       double *data, double *result)
icode = aimIntegrateBar(capsDiscr *discr, const char *name,
                        int eIndex, int rank,
                        double *r_bar, double *d_bar)
```

discr	the input Discrete Structure
name	a pointer to the input DataSet name string
eIndex	the input target element index (1 bias) in `discr`
rank	the input rank of the data
data	values at the data (or geometry) positions – NULL length/area/volume of element
result	the filled in results (`rank` in length)
r_bar	input d(objective)/d(result)
d_bar	returned d(objective)/d(data)
icode	integer return code

Forward and *reverse differentiated* functions

AIM – Data Transfers

Data Transfer to Child AIM – Optional

```
icode = aimData(int inst, const char *name, enum *vtype, int *rank,
                int *nrow, int *ncol, void **data, char **units)
```

inst	the AIM *instance* index
name	the agreed-upon data name to transfer
vtype	value data type – returned
rank	the rank of the data – returned (negative – child should free `data`)
nrow	the number of rows – returned
ncol	the number of columns – returned
data	a void pointer associated with the data – returned
units	the pointer to the string declaring the units (will be free'd by child) – returned

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM – Backdoor

AIM specific Communication – Optional

```
icode = aimBackdoor(int inst, void *aimInfo, const char *JSONin,
                    char **JSONout)
```

inst	the AIM *instance* index
aimInfo	the AIM context
JSONin	a pointer to a character string that represents the inputs.
JSONout	a returned pointer to a character string that is the output of the request.

AIM Helper Functions

- provides useful functions for the AIM programmer
- gives access to CAPS Object data
- note that all function names begin with `aim_`
- if any of these functions are used, then the library must be included in the AIM so/DLL build

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library – Body handling

Get Bodies

```
icode = aim_getBodies(void *aimInfo, char **intent, int *nBody,
                      ego **bodies)
```

aimInfo	the AIM context
intent	the returned pointer to the capsIntent string used to filter the Bodies
nBody	the returned number of EGADS Body Objects that match the `intent`
bodies	the returned pointer to a list of EGADS Body/Node Objects, Tessellation Objects (set by `aim_setTess`) follow (length – 2*nBody)
icode	integer return code

Is Node Body

```
icode = aim_isNodeBody(ego body, double *xyz)
```

body	the EGADS Body Objects to query
xyz	the returned XYZ of the Node (if a Node Body)
icode	integer return code

AIM Utility Library – Units

Units conversion

```
icode = aim_convert(void *aimInfo, char *inUnits, double inValue,
                    char *outUnits, double *outValue)
```

aimInfo	the AIM context
inUnits	the pointer to the string declaring the source units
inValue	the value to be converted
outUnits	the pointer to the string declaring the desired units
outValue	the returned converted value
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library – Units

Units multiplication

```
icode = aim_unitMultiply(void *aimInfo, char *inUnits1, char *inUnits2,
                         char **outUnits)
```

aimInfo	the AIM context
inUnits1	the pointer to the string declaring left units
inUnits2	the pointer to the string declaring right units
outUnits	the returned string units = inUnits1*inUnits2 (freeable)
icode	integer return code

Units division

```
icode = aim_unitDivision(void *aimInfo, char *inUnits1, char *inUnits2,
                         char **outUnits)
```

aimInfo	the AIM context
inUnits1	the pointer to the string declaring numerator units
inUnits2	the pointer to the string declaring denominator units
outUnits	the returned string units = inUnits1/inUnits2 (freeable)
icode	integer return code

AIM Utility Library – Units

Units invert

```
icode = aim_unitInvert(void *aimInfo, char *inUnits,
                       char **outUnits)
```

aimInfo	the AIM context
inUnits	the pointer to the string declaring units
outUnits	the returned string units = 1/inUnits (freeable)
icode	integer return code

Units raise to power

```
icode = aim_unitRaise(void *aimInfo, char *inUnits, const int power,
                      char **outUnits)
```

aimInfo	the AIM context
inUnits	the pointer to the string declaring units
outUnits	the returned string units = inUnits ˆ power (freeable)
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library – Conversions

Name to Index lookup

```
icode = aim_getIndex(void *aimInfo, char *name, enum stype)
```

aimInfo	the AIM context
name	the pointer to the string specifying the name to look-up NULL returns the total number of members in the subtype
stype	GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT
icode	index (1 bias) or negative integer return code

Index to Name lookup

```
icode = aim_getName(void *aimInfo, int index, enum stype, char **name)
```

aimInfo	the AIM context
index	the index to use (1 bias)
stype	GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT
name	the returned pointer to the string specifying the name
icode	integer return code

AIM Utility Library

Get Discretization State

```
icode = aim_getDiscrState(void *aimInfo, char *bname)
```

aimInfo	the AIM context
bname	the Bound name
icode	integer return code – CAPS_SUCCESS is clean

Get Value Structure

```
icode = aim_getValue(void *aimInfo, int index, enum stype,
                     capsValue *value)
```

aimInfo	the AIM context
index	the index to use (1 bias)
stype	GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT
value	the returned pointer to the capsValue structure
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library – Conversions

Data Transfer from Parent AIM(s)

```
icode = aim_getData(void *aimInfo, char *name, enum *vtype, int *rank,
                    int *nrow, int *ncol, void **data, char **units)
```

aimInfo	the AIM context
name	the requested agreed-upon name to fill
vtype	the returned value data type
rank	the returned rank of the data (negative – `data` should be free'd when done)
nrow	the returned number of rows
ncol	the returned number of columns
data	a returned void pointer associated with the data
units	the returned pointer to the string declaring the units (should be free'd) NULL indicates unitless values
icode	integer return code

Notes: All parent AIMs are queried. If none properly respond, this function returns `CAPS_NOTFOUND`. If multiple parents respond then this function returns `CAPS_SOURCEERR`. Parents must not be *dirty*.

AIM Utility Library

Establish Linkage from Parent or Geometry

```
icode = aim_link(void *aimInfo, char *name, enum stype,
                 capsValue *default)
```

aimInfo	the AIM context
name	the requested Value Object name to link
stype	Value subtype (GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALSYSOUT)
default	the pointer from `aimInputs`
icode	integer return code

Note: For ANALYSISIN or ANALYSISOUT subtypes all parent Analyses are queried. If none is found in the parent hierarchy, this function returns `CAPS_NOTFOUND`. The query is performed from the *oldest* ancestor down. The first match is used.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library

Get Geometry State WRT the Analysis

```
icode = aim_newGeometry(void *aimInfo)
```

aimInfo	the AIM context
icode	CAPS_SUCCESS for new, CAPS_CLEAN if not regenerated since last here

Set Tessellation for a Body

```
icode = aim_setTess(void *aimInfo, ego object)
```

aimInfo	the AIM context
object	the EGADS Tessellation Object to use for the associated Body –or – the Body Object to remove and delete an existing tessellation Note that *the Body Object is part of the Tessellation Object*
icode	integer return code

An error is raised when trying to set a Tessellation Object when one exists.

If the Problem is STATIC then the AIM (or CAPS application) is responsible for deleting the Tessellation Object. Otherwise removal of the Tessellation Object is controlled internally during Body operations. If a Tessellation Object is removed (no longer associated with the Body) then CAPS deletes the Tessellation Object.

AIM Utility Library

Get Discretization Structure

```
icode = aim_getDiscr(void *aimInfo, char *bname, capsDiscr **discr)
```

aimInfo	the AIM context
bname	the Bound name
discr	pointer to the returned Discrete structure
icode	integer return code

Get Data from Existing DataSet

```
icode = aim_getDataSet(capsDiscr *discr, char *dname, enum *method,
                       int *npts, int *rank, double **data)
```

discr	the input Discrete Structure
dname	the requested DataSet name
method	the returned method used for data transfers
npts	the returned number of points in the DataSet
rank	the returned rank of the DataSet
data	a returned pointer to the data within the DataSet
icode	integer return code

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library

Get Bound Names

```
icode = aim_getBounds(void *aimInfo, int *nBname, char ***bnames)
```

aimInfo	the AIM context
nBname	returned number of Bound names
bnames	returned pointer to list of Bound names (freeable)
icode	integer return code

Get Unit System

```
icode = aim_unitSys(void *aimInfo, char **unitSys)
```

aimInfo	the AIM context
unitSys	a returned pointer to a character string declaring the unit system – can be NULL
icode	integer return code

AIM Utility Library – Sensitivities

Setup for Sensitivities

```
icode = aim_setSensitivity(void *aimInfo, char *GIname, int *irow,
                           int *icol)
```

aimInfo	the AIM context
GIname	the pointer to the string that matches the *Geometry Input* Parameter name
irow	the parameter row to use – 1 bias
icol	the parameter column to use – 1 bias
icode	integer return code

Notes: (1) `aim_setTess` must have been invoked sometime before calling this function to set the tessellations for the Bodies of interest.
(2) Call `aim_setSensitivity` before call(s) to `aim_getSensitivity`.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

AIM Utility Library – Sensitivities

Get Sensitivities based on Tessellation Components

```
icode = aim_getSensitivity(void *aimInfo, ego tess, int ttype,
                           int index, int *npts, double **dxyz)
```

aimInfo	the AIM context
tess	the EGADS Tessellation Object
ttype	topological type – 0 - NODE, 1 - EDGE, 2 - FACE *Configuration Sensitivities* – -1 - EDGE, -2 - FACE
index	the index in the Body (associated with the tessellation) based on the *type*
npts	the returned number of sensitivities (number of tessellation points)
dxyz	a pointer to the returned sensitivities – 3*npts in length (*freeable*)
icode	integer return code

Note: Call `aim_setSensitivity` before call(s) to `aim_getSensitivity`.

AIM Utility Library – Sensitivities

Get Global Tessellation Sensitivities

```
icode = aim_sensitivity(void *aimInfo, char *GIname, int irow,
                        int icol, ego tess, int *npts, double **dxyz)
```

aimInfo	the AIM context
GIname	the pointer to the string that matches the *Geometry Input* Parameter name
irow	the parameter row to use – 1 bias
icol	the parameter column to use – 1 bias
tess	the EGADS Tessellation Object
npts	the returned number of sensitivities (number of global vertices)
dxyz	a pointer to the returned sensitivities – 3*npts in length (*freeable*)
icode	integer return code

Note: Used to get the tessellation sensitivities for the entire Tessellation Object. The number of points is the global number of vertices in the tessellation.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

APPENDIX D – LINEARIZED INCOMPRESSIBLE POTENTIAL: WAKE AND KUTTA CONDITION

1 Preliminaries

Wake Sum/Difference Operators

Wake-sheet summation and difference operators are defined as,

$$\Sigma(c) = c_l + c_u, \qquad \Delta(c) = c_l - c_u \tag{1}$$

where c is a scalar, and subscripts l and u are lower and upper, respectively.

Product formulas:

$$\Sigma(a\,b) = \frac{1}{2}\Big[\Sigma(a)\,\Sigma(b) \,+\, \Delta(a)\,\Delta(b)\Big], \qquad \Delta(a\,b) = \frac{1}{2}\Big[\Sigma(a)\,\Delta(b) \,+\, \Delta(a)\,\Sigma(b)\Big]. \tag{2}$$

Surface Tangentials and Surface Integration-by-Parts

Surface tangential component:

$$\vec{f}_{||} = \vec{f} - \left(\vec{f}\cdot\hat{n}\right)\hat{n} = -\hat{n}\times\left(\hat{n}\times\vec{f}\right) \tag{3}$$

Surface gradient,

$$\widetilde{\nabla}\phi = \nabla\phi - \frac{\partial\phi}{\partial n}\hat{n} \tag{4}$$

From vector triple product,

$$\hat{n}\times(\hat{n}\times\nabla\phi) = (\hat{n}\cdot\nabla\phi)\,\hat{n} - (\hat{n}\cdot\hat{n})\,\nabla\phi = \hat{n}\frac{\partial\phi}{\partial n} - \nabla\phi \qquad \rightarrow \qquad \widetilde{\nabla}\phi = -\hat{n}\times(\hat{n}\times\nabla\phi) \tag{5}$$

Surface divergence (from applying above to each component and summing trace),

$$\widetilde{\nabla}\cdot\vec{f} = \nabla\cdot\vec{f} - \hat{n}\cdot\frac{\partial\vec{f}}{\partial n} \tag{6}$$

Stokes theorem gives

$$\iint_S \hat{n}\cdot(\nabla\times\vec{g}) = \oint_{C_w} \hat{t}\cdot\vec{g}, \tag{7}$$

where $\hat{t}$ is the unit tangential to ∂S (oriented right-hand rule). Substituting $\vec{g} = \hat{n}\times\vec{f}$,

$$\iint_S \hat{n}\cdot\nabla\times\left(\hat{n}\times\vec{f}\right) = \iint_S \widetilde{\nabla}\cdot\vec{f} = \oint_{C_w} \hat{t}\cdot\left(\hat{n}\times\vec{f}\right) = \oint_{C_w} \left(\hat{t}\times\hat{n}\right)\cdot\vec{f} \tag{8}$$

The unit vector $\hat{\mu} = \hat{t}\times\hat{n}$ is called the conormal; it is outward pointing, tangent to the surface and normal to its bounding contour line.

Surface curl,

$$\hat{n}\times\left[\hat{n}\times\left(\nabla\times\vec{f}\right)\right] = \left[\hat{n}\cdot\left(\nabla\times\vec{f}\right)\right]\hat{n} \,-\, \left(\hat{n}\cdot\hat{n}\right)\left(\nabla\times\vec{f}\right) \tag{9}$$

Define surface curl as,

$$\widetilde{\nabla}\times\vec{f} \equiv \nabla\times\vec{f} \,-\, \hat{n}\left[\hat{n}\cdot\left(\nabla\times\vec{f}\right)\right] = -\hat{n}\times\left[\hat{n}\times\left(\nabla\times\vec{f}\right)\right] \tag{10}$$

Note that $\widetilde{\nabla}\times\vec{f}$ is a surface vector,

$$\hat{n}\cdot\left(\widetilde{\nabla}\times\vec{f}\right) = \hat{n}\cdot\left(\nabla\times\vec{f}\right) - \left(\hat{n}\cdot\hat{n}\right)\left[\hat{n}\cdot\left(\nabla\times\vec{f}\right)\right] = 0 \tag{11}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

2 Linearized Incompressible Potential: PDE and BCs

PDE

The full (Φ) and perturbation (ϕ) potentials are related by,

$$\Phi = \vec{U} \cdot \vec{r} + \phi \tag{12}$$

where $\vec{U}$ is freestream velocity, and $\vec{r} = x\hat{\imath} + y\hat{\jmath} + z\hat{k}$ is physical position.

Mass conservation assuming incompressible,

$$\nabla \cdot (\rho \nabla \Phi) = \nabla \cdot \left(\rho \left[\vec{U} + \nabla\phi\right]\right) = \rho \nabla^2 \phi = 0. \tag{13}$$

Wall BC

Flow tangency condition along the body; this is a Neumann condition,

$$\frac{\partial \Phi}{\partial n} = \vec{U} \cdot \hat{n} + \frac{\partial \phi}{\partial n} = 0, \qquad \text{on } \partial\Omega_b. \tag{14}$$

Farfield BC

Dirichlet/Neumann...

Wake BCs

$$\text{mass flux:} \qquad \Sigma\left(\frac{\partial \phi}{\partial n}\right) = 0 \tag{15}$$

$$\text{potential jump:} \qquad \Delta(\phi) = -\Gamma \tag{16}$$

$$\text{pressure jump:} \qquad \Delta\left(\vec{U} \cdot \nabla\phi\right) \tag{17}$$

We can combine these into a surface PDE for circulation Γ,

$$0 = \Delta\left(\vec{U} \cdot \nabla\phi\right) = \Delta\left(U_n \frac{\partial \phi}{\partial n} + \vec{U}_{||} \cdot \widetilde{\nabla}\phi\right) = \frac{1}{2}\Delta(U_n)\,\Sigma\left(\frac{\partial \phi}{\partial n}\right) + \vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi) = \vec{U}_{||} \cdot \widetilde{\nabla}\Gamma, \tag{18}$$

since U_n flips sign (giving $\Sigma(U_n) = 0$), and $\vec{U}_{||}$ and $\widetilde{\nabla}$ are uniquely defined in the wake.

Kutta Condition

Kutta condition

$$\Delta(\phi)_{\mathsf{TE}} = -\Gamma_{\mathsf{TE}}, \qquad \text{on } C_{\mathsf{TE}} \tag{19}$$

where $\Delta(\phi)_{\mathsf{TE}}$ is evaluated on the wing in the limit as the TE is approached. In addition, there probably needs to be a BC at the wingtip

$$\Delta(\phi)_{\mathsf{wingtip}} = \Gamma_{\mathsf{wingtip}} = 0, \qquad \text{at } C_{\mathsf{wingtip}} \tag{20}$$

Tranair (FTJ, 1992) implements $\Delta(\phi) = -\Gamma$ along the TE, effectively treating this as an initial condition for the wake-surface PDE for Γ: $\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma = 0$.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3 BC Formulations without Lagrange Multipliers: *sans-Lagrange*

3.1 Wake BCs: Mass Flux, Potential Jump and Circulation Gradient

$$\text{mass flux:} \qquad \Sigma\left(\frac{\partial\phi}{\partial n}\right) = 0 \tag{21}$$

$$\text{potential jump:} \qquad \Delta(\phi) = -\Gamma \tag{22}$$

$$\text{circulation:} \qquad \vec{U}_{||}\cdot\tilde{\nabla}\Gamma = 0 \tag{23}$$

Primal strong form: find $\phi \in \mathcal{V}$ and $\Gamma \in \mathcal{W}$ such that $\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) = 0$ for all $\psi \in \mathcal{V}$ and $\Upsilon \in \mathcal{W}$, where,

$$\mathcal{R}_s(\phi,\Gamma;\psi,\Upsilon) = -\iiint_\Omega \psi\left(\nabla^2\phi\right) + \iint_{\partial\Omega_w}\left\{w_a(\psi,\Upsilon)\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b(\psi,\Upsilon)\big[\Delta(\phi)+\Gamma\big] + w_c(\psi,\Upsilon)\big[\vec{U}_{||}\cdot\tilde{\nabla}\Gamma\big]\right\}, \tag{24}$$

where w_a, w_b and w_c are weighting functions of ψ and Υ,

$$w_a = a_1\Sigma(\psi) + a_2\Delta(\psi) + a_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + a_4\Delta\left(\frac{\partial\psi}{\partial n}\right) + a_5\Upsilon + a_6\big[\vec{U}_{||}\cdot\tilde{\nabla}\Upsilon\big] + a_7\big[\vec{U}_{||}\cdot\tilde{\nabla}\Sigma(\psi)\big] + a_8\big[\vec{U}_{||}\cdot\tilde{\nabla}\Delta(\psi)\big] \tag{25}$$

and similarly for w_b ($a_i \to b_i$) and w_c ($a_i \to c_i$).

Consider the linear primal output functional,

$$\mathcal{J}(\phi) = \iiint_\Omega g\phi + \oiint_{\partial\Omega}\left[h\phi + h_n\frac{\partial\phi}{\partial n}\right], \tag{26}$$

where g, h and h_n are functions of position but do not depend on the solution. The boundary contributions include possibly different values of h and h_n on upper and lower sides of the wake sheet,

$$\oiint_{\partial\Omega}\left[h\phi + h_n\frac{\partial\phi}{\partial n}\right] = \iint_{\partial\Omega_b} + \iint_{\partial\Omega_\infty} + \iint_{\partial\Omega_w}\left[\Sigma(h\phi) + \Sigma\left(h_n\frac{\partial\phi}{\partial n}\right)\right], \tag{27}$$

and

$$\Sigma(h\phi) = \frac{1}{2}\Big[\Sigma(h)\,\Sigma(\phi) + \Delta(h)\,\Delta(\phi)\Big], \qquad \Sigma\left(h_n\frac{\partial\phi}{\partial n}\right) = \frac{1}{2}\left[\Sigma(h_n)\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + \Delta(h_n)\,\Delta\left(\frac{\partial\phi}{\partial n}\right)\right], \tag{28}$$

Duality:

$$\mathcal{J}(\phi) - \mathcal{R}_s(\phi,\,\psi) = \mathcal{J}^*(\psi) - \mathcal{R}_s^*(\psi,\,\phi) \tag{29}$$

Invoke IBP twice to isolate adjoint PDE,

$$-\iiint_\Omega \psi\left(\nabla^2\phi\right) = \iiint_\Omega \nabla\psi\cdot\nabla\phi - \oiint_{\partial\Omega}\psi\frac{\partial\phi}{\partial n} = -\iiint_\Omega \phi\left(\nabla^2\psi\right) + \oiint_{\partial\Omega}\left[\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right] \tag{30}$$

For the wake the IBP terms become

$$\begin{aligned}\oiint_{\partial\Omega_w}\left[\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right] &= \iint_{\partial\Omega_{wu}} + \int_{\partial\Omega_{wl}}\left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right)\\ &= \iint_{\partial\Omega_{w=wu}}\left[\left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right)_u - \left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right)_l\right]\\ &= \iint_{\partial\Omega_w} -\Delta\left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right)\end{aligned} \tag{31}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

where

$$\Delta\left(ab\right) = \frac{1}{2}\Big[\Sigma\left(a\right)\Delta\left(b\right) + \Delta\left(a\right)\Sigma\left(b\right)\Big] \tag{32}$$

Substitute into duality,

$$\begin{aligned}
&\mathcal{J}(\phi) - \mathcal{R}_s(\phi,\,\psi)\\
&\qquad = \iiint_\Omega g\phi + \iint_{\partial\Omega_w}\Big[\Sigma\left(h\phi\right) + \Sigma\left(h_n\frac{\partial\phi}{\partial n}\right)\Big]\\
&\qquad\quad - \iiint_\Omega \psi\Big(-\nabla^2\phi\Big) - \iint_{\partial\Omega_w}\left\{w_a\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b\big[\Delta\left(\phi\right) + \Gamma\big] + w_c\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big]\right\},\\
&\qquad = -\iiint_\Omega \phi\Big(-\nabla^2\psi - g\Big) - \iint_{\partial\Omega_w} -\Delta\left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right)\\
&\qquad\quad - \iint_{\partial\Omega_w}\left\{w_a\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b\big[\Delta\left(\phi\right) + \Gamma\big] + w_c\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big]\right\}, + \iint_{\partial\Omega_w}\Big[\Sigma\left(h\phi\right) + \Sigma\left(h_n\frac{\partial\phi}{\partial n}\right)\Big]
\end{aligned} \tag{33}$$

Accumulating wake terms in the strong-form adjoint residual and grouping by weights,

$$\begin{aligned}
\mathcal{R}_s^* :& \iint_{\partial\Omega_w}\left\{-\Delta\left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right) + w_a\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b\big[\Delta\left(\phi\right) + \Gamma\big] + w_c\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big] - \Sigma\left(h\phi\right) - \Sigma\left(h_n\frac{\partial\phi}{\partial n}\right)\right\}\\
&= \iint_{\partial\Omega_w}\mathbf{u}^t\mathbf{M}\mathbf{w}
\end{aligned} \tag{34}$$

where

$$\mathbf{u} = \begin{pmatrix} \Sigma\left(\phi\right)\\ \Delta\left(\phi\right)\\ \Sigma\left(\partial\phi/\partial n\right)\\ \Delta\left(\partial\phi/\partial n\right)\\ \Gamma\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\phi\right)\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\phi\right) \end{pmatrix}, \qquad \mathbf{w} = \begin{pmatrix} \Sigma\left(\psi\right)\\ \Delta\left(\psi\right)\\ \Sigma\left(\partial\psi/\partial n\right)\\ \Delta\left(\partial\psi/\partial n\right)\\ \Upsilon\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\psi\right)\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\psi\right) \end{pmatrix}, \tag{35}$$

$$\mathbf{M} = \begin{pmatrix} 0 & 0 & 0 & -1/2 & 0 & 0 & 0 & 0\\ b_1 & b_2 & b_3 - 1/2 & b_4 & b_5 & b_6 & b_7 & b_8\\ a_1 & a_2 + 1/2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8\\ 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8\\ c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{36}$$

We note that Γ and $\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma$ cannot be independent weights for the adjoint residual; similarly for $\Sigma\left(\phi\right)$ and $\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\phi\right)$, etc. Anticipating possible issues related to this, the adjoint residual can be rewritten with integration-by-parts within the wake surface; given some τ,

$$\iint_{\partial\Omega_w}\tau\big(\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big) = -\iint_{\partial\Omega_w}\Gamma\widetilde{\nabla}\cdot\big(\tau\vec{U}_{||}\big) + \oint_{C_w}\big(\tau\Gamma\vec{U}_{||}\big)\cdot\hat{\mu} \tag{37}$$

where C_w is the bounding contour of the wake, and $\hat{\mu}$ is the wake surface co-normal ($\hat{\mu} = \hat{t}\times\hat{n}$). Note that the co-normal $\hat{\mu}$ is the same on lower and upper (since both $\hat{t}$ and $\hat{n}$ flip signs). The first term on the right-hand side can be written

$$\widetilde{\nabla}\cdot\big(\tau\vec{U}_{||}\big) = \vec{U}_{||}\cdot\widetilde{\nabla}\tau + \tau\widetilde{\nabla}\cdot\vec{U}_{||} \tag{38}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Without changing the adjoint residual, we can then add the term,

$$\theta \left\{ \iint_{\partial\Omega_w} \left[\tau\big(\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big) + \Gamma\big(\vec{U}_{||} \cdot \widetilde{\nabla}\tau\big) + \tau\Gamma\widetilde{\nabla} \cdot \vec{U}_{||} \right] - \oint_{C_w} \big(\tau\Gamma\vec{U}_{||}\big) \cdot \hat{\mu} \right\} = 0 \tag{39}$$

with θ a parameter. Expanding this to other possible combinations of terms, the following will have a surface IBP adjustment:

$$\theta_1 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\phi)\big]\Sigma(\psi) \tag{40}$$
$$\theta_2 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\phi)\big]\Delta(\psi) \tag{41}$$
$$\theta_3 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\phi)\big]\Upsilon \tag{42}$$
$$\theta_4 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi)\big]\Sigma(\psi) \tag{43}$$
$$\theta_5 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi)\big]\Delta(\psi) \tag{44}$$
$$\theta_6 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi)\big]\Upsilon \tag{45}$$
$$\theta_7 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big]\Sigma(\psi) \tag{46}$$
$$\theta_8 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big]\Delta(\psi) \tag{47}$$
$$\theta_9 \quad : \quad \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big]\Upsilon \tag{48}$$

The resulting adjusted strong-form adjoint residual is

$$\begin{aligned}
\mathcal{R}_s^* : \iint_{\partial\Omega_w} \Bigg\{ & -\Delta\left(\phi\frac{\partial\psi}{\partial n} - \psi\frac{\partial\phi}{\partial n}\right) + w_a\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b\big[\Delta(\phi) + \Gamma\big] + w_c\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big] - \Sigma(h\phi) - \Sigma\left(h_n\frac{\partial\phi}{\partial n}\right) \\
& + \theta_1\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\phi)\big]\Sigma(\psi) + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\psi)\big]\Sigma(\phi) + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Sigma(\phi)\,\Sigma(\psi)\Big) \\
& + \theta_2\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\phi)\big]\Delta(\psi) + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\psi)\big]\Sigma(\phi) + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Sigma(\phi)\,\Delta(\psi)\Big) \\
& + \theta_3\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\phi)\big]\Upsilon + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Upsilon\big]\Sigma(\phi) + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Sigma(\phi)\,\Upsilon\Big) \\
& + \theta_4\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi)\big]\Sigma(\psi) + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\psi)\big]\Delta(\phi) + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Delta(\phi)\,\Sigma(\psi)\Big) \\
& + \theta_5\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi)\big]\Delta(\psi) + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\psi)\big]\Delta(\phi) + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Delta(\phi)\,\Delta(\psi)\Big) \\
& + \theta_6\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\phi)\big]\Upsilon + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Upsilon\big]\Delta(\phi) + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Delta(\phi)\,\Upsilon\Big) \\
& + \theta_7\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big]\Sigma(\psi) + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Sigma(\psi)\big]\Gamma + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Gamma\Sigma(\psi)\Big) \\
& + \theta_8\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big]\Delta(\psi) + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Delta(\psi)\big]\Gamma + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Gamma\Delta(\psi)\Big) \\
& + \theta_9\Big(\big[\vec{U}_{||} \cdot \widetilde{\nabla}\Gamma\big]\Upsilon + \big[\vec{U}_{||} \cdot \widetilde{\nabla}\Upsilon\big]\Gamma + \big[\widetilde{\nabla} \cdot \vec{U}_{||}\big]\Gamma\Upsilon\Big) \Bigg\} \\
- \oint_{C_w} & \Big(\theta_1\Sigma(\phi)\,\Sigma(\psi) + \theta_2\Sigma(\phi)\,\Delta(\psi) + \theta_3\Sigma(\phi)\,\Upsilon \\
& + \theta_4\Delta(\phi)\,\Sigma(\psi) + \theta_5\Delta(\phi)\,\Delta(\psi) + \theta_6\Delta(\phi)\,\Upsilon \\
& + \theta_7\Gamma\Sigma(\psi) + \theta_8\Gamma\Delta(\psi) + \theta_9\Gamma\Upsilon\Big)\Big(\vec{U}_{||} \cdot \hat{\mu}\Big) \\
= \iint_{\partial\Omega_w} & \mathbf{u}^t\tilde{\mathbf{M}}\mathbf{w} - \oint_{C_w}(\cdots)
\end{aligned} \tag{49}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

with

$$\tilde{\mathbf{M}} = \begin{pmatrix} \theta_1(\tilde{\nabla}\cdot\vec{U}_{||}) & \theta_2(\tilde{\nabla}\cdot\vec{U}_{||}) & 0 & -1/2 & \theta_3(\tilde{\nabla}\cdot\vec{U}_{||}) & \theta_3 & \theta_1 & \theta_2 \\ b_1+\theta_4(\tilde{\nabla}\cdot\vec{U}_{||}) & b_2+\theta_5(\tilde{\nabla}\cdot\vec{U}_{||}) & b_3-1/2 & b_4 & b_5+\theta_6(\tilde{\nabla}\cdot\vec{U}_{||}) & b_6+\theta_6 & b_7+\theta_4 & b_8+\theta_5 \\ a_1 & a_2+1/2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\ 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ b_1+\theta_7(\tilde{\nabla}\cdot\vec{U}_{||}) & b_2+\theta_8 & b_3 & b_4 & b_5+\theta_9 & b_6+\theta_9 & b_7+\theta_7 & b_8+\theta_8 \\ c_1+\theta_7 & c_2+\theta_8 & c_3 & c_4 & c_5+\theta_9 & c_6 & c_7 & c_8 \\ \theta_1 & \theta_2 & 0 & 0 & \theta_3 & 0 & 0 & 0 \\ \theta_4 & \theta_5 & 0 & 0 & \theta_6 & 0 & 0 & 0 \end{pmatrix} \tag{50}$$

We seek 3 adjoint BCs. This means determining parameters such that the matrix $\tilde{\mathbf{M}}$ is rank 3, and its zero eigenvalue has algebraic and geometric multiplicity of 5 (i.e. $8-3=5$). The zero eigenvector system is

$$\tilde{\mathbf{M}}\mathbf{x} = \mathbf{0} \tag{51}$$

with $\mathbf{x} = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}^t$. This zero eigenvector will have 5 free parameters. The solution process is as follows: first determine 3 of the x_i components in terms of the remaining 5; then determine parameters a_i, b_i, c_i and θ_i so that all coefficients of these remaining 5 x_i components in $\tilde{\mathbf{M}}\mathbf{x}$ vanish, satisfying the above zero eigenvector equation. The resulting matrix will then have rank 3.

By inspection, the easiest path forward is to solve equations 1, 4 and 2 minus 5 for x_1, x_3 and x_4, giving

$$x_1 = 0, \qquad x_3 = 2\Big[(\theta_5-\theta_8)\left[(\tilde{\nabla}\cdot\vec{U}_{||})x_2 + x_8\right] + (\theta_6-\theta_9)\left[(\tilde{\nabla}\cdot\vec{U}_{||})x_5 + x_6\right] - (\theta_7-\theta_4)x_7\Big], \tag{52}$$

$$x_4 = 2\Big[\theta_2\left[(\tilde{\nabla}\cdot\vec{U}_{||})x_2 + x_8\right] + \theta_3\left[(\tilde{\nabla}\cdot\vec{U}_{||})x_5 + x_6\right] + \theta_1 x_7\Big] \tag{53}$$

Substituting in and requiring the remaining equations in $\tilde{\mathbf{M}}\mathbf{x} = \mathbf{0}$ to be satisfied, leads to the parameter constraints

$$a_2 = -\frac{1}{2} + 2a_3\theta_8(\tilde{\nabla}\cdot\vec{U}_{||}), \quad a_5 = 2a_3\theta_9(\tilde{\nabla}\cdot\vec{U}_{||}), \quad a_6 = 2a_3\theta_9,$$
$$a_7 = -2\big(a_4\theta_1 - a_3(\theta_7-\theta_4)\big), \quad a_8 = 2a_3\theta_8, \tag{54}$$

$$b_2 = -(1-2b_3)\theta_8(\tilde{\nabla}\cdot\vec{U}_{||}), \quad b_5 = -(1-2b_3)\theta_9(\tilde{\nabla}\cdot\vec{U}_{||}), \quad b_6 = -(1-2b_3)\theta_9,$$
$$b_7 = -\theta_7 + 2b_3(\theta_7-\theta_4) - 2b_4\theta_1, \quad b_8 = -(1-2b_3)\theta_8, \tag{55}$$

$$c_2 = -(1-2c_3(\tilde{\nabla}\cdot\vec{U}_{||}))\theta_8, \quad c_5 = -(1-2c_3(\tilde{\nabla}\cdot\vec{U}_{||}))\theta_9, \quad c_6 = 2c_3\theta_9,$$
$$c_7 = -2(c_4\theta_1 - c_3(\theta_7-\theta_4), \quad c_8 = 2c_3\theta_8, \tag{56}$$

$$\theta_2 = \theta_3 = \theta_5 = \theta_6 = 0 \tag{57}$$

Designate $\tilde{\mathbf{M}}_0$ as the matrix with these parameter value substitutions. The matrix can be re-written as a product of rank-3 left and right matrices (ala SVD),

$$\tilde{\mathbf{M}}_0 = \mathbf{R}_l\mathbf{R}_r^t, \tag{58}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

$$\tilde{\mathbf{M}}_0\mathbf{N}_r = \mathbf{0}_{8\times 5}, \qquad \mathbf{N}_r^t\mathbf{R}_r = \mathbf{0}_{5\times 3}, \tag{59}$$

$$\mathbf{N}_r = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & \theta_7-\theta_4 & \theta_9 & \theta_9 & 1 \\ 0 & 0 & 0 & 0 & -2\theta_8\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) \\ 0 & -2\theta_1\theta_8\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & 0 & 0 & 0 \\ 0 & 0 & 0 & -\theta_8 & 0 \\ 0 & 0 & -\theta_8\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & 0 & 0 \\ 0 & -\theta_8\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & 0 & 0 & 0 \\ -\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad \mathbf{R}_r = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 2\theta_8\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) \\ 0 & 0 & 1 \\ 0 & 1 & (\theta_7-\theta_4)/\theta_1 \\ 0 & 0 & 2\theta_9\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) \\ 0 & 0 & 2\theta_9 \\ 0 & -2\theta_1 & 0 \\ 0 & 0 & 2\theta_8 \end{pmatrix} \tag{60}$$

$$\mathbf{R}_l = \tilde{\mathbf{M}}_0\mathbf{R}_r\left(\mathbf{R}_r^t\mathbf{R}_r\right)^{-1} = \begin{pmatrix} \theta_1\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & -1/2 & 0 \\ b_1+\theta_4\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & b_4-(b_3-1/2)(\theta_7-\theta_4)/\theta_1 & b_3-1/2 \\ a_1 & a_4-a_3(\theta_7-\theta_4)/\theta_1 & a_3 \\ 1/2 & 0 & 0 \\ b_1+\theta_7\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big) & b_4-b_3(\theta_7-\theta_4)/\theta_1 & b_3 \\ c_1+\theta_7 & c_4-c_3(\theta_7-\theta_4)/\theta_1 & c_3 \\ \theta_1 & 0 & 0 \\ \theta_4 & 0 & 0 \end{pmatrix} \tag{61}$$

The adjoint BCs are given by $\mathbf{R}_r^t\mathbf{w}$ and the primal weights are $\mathbf{R}_l^t\mathbf{u}$,

$$\begin{aligned} \mathbf{u}^t\tilde{\mathbf{M}}_0\mathbf{w} &= \left(\mathbf{u}^t\mathbf{R}_l\right)\left(\mathbf{R}_r^t\mathbf{w}\right) \\ &= \omega_1\Big[\Sigma\left(\psi\right)\Big] + \omega_2\left[\Delta\left(\frac{\partial\psi}{\partial n}\right) - 2\theta_1\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\psi\right)\big]\right] \\ &+ \omega_3\left[2\theta_8\left(\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big)\Delta\left(\psi\right) + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\psi\right)\big]\right) + \Sigma\left(\frac{\partial\psi}{\partial n}\right) + \frac{\theta_7-\theta_4}{\theta_1}\Delta\left(\frac{\partial\psi}{\partial n}\right) + 2\theta_9\left(\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big)\Upsilon + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\big]\right)\right] \end{aligned} \tag{62}$$

$$\begin{aligned} \omega_1 &= \theta_1\left(\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big)\Sigma\left(\phi\right) + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\phi\right)\big]\right) + \theta_4\left(\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big)\Delta\left(\phi\right) + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\phi\right)\big]\right) + b_1(\Delta\left(\phi\right)+\Gamma) \\ &\quad + a_1\Sigma\left(\frac{\partial\phi}{\partial n}\right) + \frac{1}{2}\Delta\left(\frac{\partial\phi}{\partial n}\right) + \theta_7\left(\big(\widetilde{\nabla}\cdot\vec{U}_{||}\big)\Gamma + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big]\right) + c_1\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big] \end{aligned} \tag{63}$$

$$\begin{aligned} \omega_2 &= -\frac{1}{2}\Sigma\left(\phi\right) + \frac{\theta_7-\theta_4}{2\theta_1}\Delta\left(\phi\right) + \left(b_4 - b_3\frac{\theta_7-\theta_4}{\theta_1}\right)\left(\Delta\left(\phi\right)+\Gamma\right) + \left(a_4 - a_3\frac{\theta_7-\theta_4}{\theta_1}\right)\Sigma\left(\frac{\partial\phi}{\partial n}\right) \\ &\quad + \left(c_4 - c_3\frac{\theta_7-\theta_4}{\theta_1}\right)\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big] \end{aligned} \tag{64}$$

$$\omega_3 = -\frac{1}{2}\Delta\left(\phi\right) + b_3(\Delta\left(\phi\right)+\Gamma) + a_3\Sigma\left(\frac{\partial\phi}{\partial n}\right) + c_3\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big] \tag{65}$$

The remaining wake bounding contour terms for the strong-form adjoint residual are

$$\mathcal{R}_s^* \quad : \quad -\oint_{C_w}\Big(\theta_1\Sigma\left(\phi\right)\Sigma\left(\psi\right) + \theta_4\Delta\left(\phi\right)\Sigma\left(\psi\right) + \theta_7\Gamma\Sigma\left(\psi\right) + \theta_8\Gamma\Delta\left(\psi\right) + \theta_9\Gamma\Upsilon\Big)\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) \tag{66}$$

These must be combined with the Kutta condition and wake edge BCs.

Note that without the wake surface IBP (i.e. $\theta_i = 0$ and $(\theta_7-\theta_4)/\theta_1 = 0$), the formulation still produces the correct number of adjoint BCs, but these BCs become nonsensical. The quantities involved in the adjoints BCs become

$$\Sigma\left(\psi\right), \qquad \Sigma\left(\frac{\partial\psi}{\partial n}\right), \qquad \Delta\left(\frac{\partial\psi}{\partial n}\right) \tag{67}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

which effectively sets the normal derivatives on both upper and lower, as well as setting the average value, but does not prescribe any conditions on the adjoint circulation.

The final strong-form primal residual is

$$\mathcal{R}_s(\phi,\Gamma;\psi,\Upsilon) \quad : \quad \iint_{\partial\Omega_w} \left\{ w_a \, \Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b\big[\Delta(\phi)+\Gamma\big] + w_c\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big] \right\} \tag{68}$$

where

$$\begin{aligned}
w_a &= a_1\Sigma(\psi) - \frac{1}{2}\Delta(\psi) + a_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + a_4\Delta\left(\frac{\partial\psi}{\partial n}\right) + 2a_3\theta_8\Big((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta(\psi) + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\psi)\big]\Big)\\
&\quad + 2a_3\theta_9\Big((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\big]\Big) - 2(a_4\theta_1 - a_3(\theta_7-\theta_4))\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\big] && (69)\\
w_b &= b_1\Sigma(\psi) + b_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + b_4\Delta\left(\frac{\partial\psi}{\partial n}\right)\\
&\quad - (1-2b_3)\Big(\theta_8\Big((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta(\psi) + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\psi)\big]\Big) + \theta_9\Big((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\big]\Big)\Big)\\
&\quad - (\theta_7 - 2b_3(\theta_7-\theta_4) + 2b_4\theta_1)\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\big] && (70)\\
w_c &= c_1\Sigma(\psi) + c_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + c_4\Delta\left(\frac{\partial\psi}{\partial n}\right) - \theta_8\Delta(\psi) - \theta_9\Upsilon\\
&\quad + 2c_3\Big[\theta_8\Big((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta(\psi) + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\psi)\big]\Big) + \theta_9\Big((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \big[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\big]\Big)\Big]\\
&\quad + 2\left(c_3(\theta_7-\theta_4) - c_4\theta_1\right)\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\big] && (71)
\end{aligned}$$

Units

$\vec{U} \sim \mathsf{V}$; ϕ, ψ, Γ, $\Upsilon \sim \mathsf{VL}$

$$\iiint_\Omega \psi\Big(\nabla^2\phi\Big) \sim \mathsf{V}^2\mathsf{L}^3$$

$$\iint_{\partial\Omega_w} \left\{ w_a(\psi,\Upsilon)\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b(\psi,\Upsilon)\big[\Delta(\phi)+\Gamma\big] + w_c(\psi,\Upsilon)\big[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\big] \right\} \sim \mathsf{V}^2\mathsf{L}^3$$

This gives $w_a \sim \mathsf{VL}$, $w_b \sim \mathsf{V}$, and $w_c \sim \mathsf{L}$. Free parameters are then: $a_1 \sim 1$, $a_3 \sim \mathsf{L}$, $a_4 \sim \mathsf{L}$, $b_1 \sim 1/\mathsf{L}$, $b_3 \sim 1$, $b_4 \sim 1$, $c_1 \sim 1/\mathsf{V}$, $c_3 \sim \mathsf{L}/\mathsf{V}$, $c_4 \sim \mathsf{L}/\mathsf{V}$, $\theta_i \sim 1/\mathsf{V}$.

3.2 Wake Free Edge BCs:

$$\text{free edge potential jump:} \qquad \Delta(\phi) = 0, \qquad \text{on } C_{\text{free}} \tag{72}$$

Primal strong-form residual

$$\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) \quad : \quad \int_{C_{\text{free}}} w_{\text{free}}(\psi,\Upsilon)\,\Delta(\phi) \tag{73}$$

with

$$w_{\text{free}} = a_{\text{free}}\Sigma(\psi) + b_{\text{free}}\Delta(\psi) + c_{\text{free}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\text{free}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + e_{\text{free}}\Upsilon \tag{74}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Strong-form adjoint residual from duality, including terms from wake surface IBP,

$$\begin{aligned}
\mathcal{R}_s^* \quad : \quad & \int_{C_{\text{free}}} \Big\{ -\Big(\theta_1 \Sigma(\phi)\Sigma(\psi) + \theta_4 \Delta(\phi)\Sigma(\psi) + \theta_7 \Gamma \Sigma(\psi) + \theta_8 \Gamma \Delta(\psi) + \theta_9 \Gamma \Upsilon\Big)\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) \\
& \qquad + \Big(a_{\text{free}}\Sigma(\psi) + b_{\text{free}}\Delta(\psi) + c_{\text{free}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\text{free}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + e_{\text{free}}\Upsilon\Big)\Big(\Delta(\phi)\Big)\Big\} \\
& = \int_{C_{\text{free}}} \mathbf{u}^t \mathbf{M} \mathbf{w}
\end{aligned} \tag{75}$$

with

$$\mathbf{u} = \begin{pmatrix} \Sigma(\phi) \\ \Delta(\phi) \\ \Sigma(\partial\phi/\partial n) \\ \Delta(\partial\phi/\partial n) \\ \Gamma \end{pmatrix}, \qquad \mathbf{w} = \begin{pmatrix} \Sigma(\psi) \\ \Delta(\psi) \\ \Sigma(\partial\psi/\partial n) \\ \Delta(\partial\psi/\partial n) \\ \Upsilon \end{pmatrix}, \tag{76}$$

$$\mathbf{M} = \begin{pmatrix} -\theta_1\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) & 0 & 0 & 0 & 0 \\ -\theta_4\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) + a_{\text{free}} & b_{\text{free}} & c_{\text{free}} & d_{\text{free}} & e_{\text{free}} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -\theta_7\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) & -\theta_8\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) & 0 & 0 & -\theta_9\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) \end{pmatrix} \tag{77}$$

Zero eigenvector has multiplicity of 4 ($5 - 1 = 4$), so set one x_i in terms of all others. Obvious choice is $x_1 = 0$. Zero coefficients for all other x_i gives,

$$b_{\text{free}} = c_{\text{free}} = d_{\text{free}} = e_{\text{free}} = \theta_8 = \theta_9 = 0 \tag{78}$$

giving the strong-form adjoint residual

$$\mathcal{R}_s^*(\psi, \Upsilon;\, \phi, \Gamma) \quad : \quad \int_{C_{\text{free}}} \Big[-\theta_1\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Sigma(\phi) + \Big(a_{\text{free}} - \theta_4\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Big)\Delta(\phi) - \theta_7\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Gamma\Big]\Big[\Sigma(\psi)\Big] \tag{79}$$

and strong-form primal residual

$$\mathcal{R}_s(\phi, \Gamma;\, \psi, \Upsilon) \quad : \quad \int_{C_{\text{free}}} a_{\text{free}}\Sigma(\psi)\;\Delta(\phi) \tag{80}$$

leaving a_{free}, θ_1, θ_4 and θ_7 as free parameters.

3.3 Downstream Wake Edge BCs:

Assume perpendicular intersection of wake and downstream farfield boundary. Do we need to specify any BCs?

3.4 Kutta Condition:

Kutta condition

$$\Delta(\phi)_{\mathsf{TE}} = -\Gamma_{\mathsf{TE}}, \qquad \text{on } C_{\mathsf{TE}} \tag{81}$$

where $\Delta(\phi)_{\mathsf{TE}}$ is evaluated on the wing in the limit as the TE is approached. In addition, there probably needs to be a BC at the wingtip

$$\Delta(\phi)_{\mathsf{wingtip}} = \Gamma_{\mathsf{wingtip}} = 0, \qquad \text{at } C_{\mathsf{wingtip}} \tag{82}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Along the wing TE, the strong-form primal residual is

$$\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) \quad : \quad \int_{C_{\mathsf{TE}}} w_{\mathsf{TE}}(\psi,\Upsilon)\,\Big(\Delta\left(\phi\right)+\Gamma\Big) \tag{83}$$

with

$$w_{\mathsf{TE}} = a_{\mathsf{TE}}\Sigma\left(\psi\right) + b_{\mathsf{TE}}\Delta\left(\psi\right) + c_{\mathsf{TE}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\mathsf{TE}}\Delta\left(\frac{\partial\psi}{\partial n}\right) + e_{\mathsf{TE}}\Upsilon \tag{84}$$

Strong-form adjoint residual from duality, including terms from wake surface IBP,

$$\begin{aligned}\mathcal{R}^*_s \quad : \quad & \int_{C_{\mathsf{TE}}} \Big\{ -\Big(\theta_1\Sigma\left(\phi\right)\Sigma\left(\psi\right) + \theta_4\Delta\left(\phi\right)\Sigma\left(\psi\right) + \theta_7\Gamma\Sigma\left(\psi\right) + \theta_8\Gamma\Delta\left(\psi\right) + \theta_9\Gamma\Upsilon\Big)\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) \\ & \qquad + \Big(a_{\mathsf{TE}}\Sigma\left(\psi\right) + b_{\mathsf{TE}}\Delta\left(\psi\right) + c_{\mathsf{TE}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\mathsf{TE}}\Delta\left(\frac{\partial\psi}{\partial n}\right) + e_{\mathsf{TE}}\Upsilon\Big)\Big(\Delta\left(\phi\right)+\Gamma\Big)\Big\} \\ & = \int_{C_{\mathsf{TE}}} \mathbf{u}^t\mathbf{M}\mathbf{w}\end{aligned} \tag{85}$$

with

$$\mathbf{u} = \begin{pmatrix}\Sigma\left(\phi\right)\\ \Delta\left(\phi\right)\\ \Sigma\left(\partial\phi/\partial n\right)\\ \Delta\left(\partial\phi/\partial n\right)\\ \Gamma\end{pmatrix}, \qquad \mathbf{w} = \begin{pmatrix}\Sigma\left(\psi\right)\\ \Delta\left(\psi\right)\\ \Sigma\left(\partial\psi/\partial n\right)\\ \Delta\left(\partial\psi/\partial n\right)\\ \Upsilon\end{pmatrix}, \tag{86}$$

$$\mathbf{M} = \begin{pmatrix} -\theta_1\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) & 0 & 0 & 0 & 0 \\ -\theta_4\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) + a_{\mathsf{TE}} & b_{\mathsf{TE}} & c_{\mathsf{TE}} & d_{\mathsf{TE}} & e_{\mathsf{TE}} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -\theta_7\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) + a_{\mathsf{TE}} & -\theta_8\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) + b_{\mathsf{TE}} & c_{\mathsf{TE}} & d_{\mathsf{TE}} & -\theta_9\Big(\vec{U}_{||}\cdot\hat{\mu}\Big) + e_{\mathsf{TE}} \end{pmatrix} \tag{87}$$

Zero eigenvector has multiplicity of 4 ($5-1=4$), so set one x_i in terms of all others. Obvious choice is $x_1 = 0$. Zero coefficients for all other x_i gives,

$$b_{\mathsf{TE}} = c_{\mathsf{TE}} = d_{\mathsf{TE}} = e_{\mathsf{TE}} = \theta_8 = \theta_9 = 0 \tag{88}$$

giving the strong-form adjoint residual

$$\mathcal{R}^*_s(\psi,\Upsilon;\,\phi,\Gamma) \quad : \quad \int_{C_{\mathsf{TE}}} \Big[-\theta_1\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Sigma\left(\phi\right) + \Big(a_{\mathsf{TE}} - \theta_4\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Big)\,\Delta\left(\phi\right) + \Big(a_{\mathsf{TE}} - \theta_7\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Big)\,\Gamma\Big]\,\Big[\Sigma\left(\psi\right)\Big] \tag{89}$$

and strong-form primal residual

$$\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) \quad : \quad \int_{C_{\mathsf{TE}}} a_{\mathsf{TE}}\Sigma\left(\psi\right)\,\Big(\Delta\left(\phi\right)+\Gamma\Big) \tag{90}$$

leaving a_{TE}, θ_1, θ_4 and θ_7 as free parameters.

The wingtip has no additional IBP terms in the adjoint residual, so the most general primal weighting should work,

$$\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) \quad : \quad \int_{C_{\mathsf{wingtip}}} w_{\mathsf{wingtip}}(\psi,\Upsilon)\,\Delta\left(\phi\right), \tag{91}$$

$$w_{\mathsf{wingtip}} = a_{\mathsf{wingtip}}\Sigma\left(\psi\right) + b_{\mathsf{wingtip}}\Delta\left(\psi\right) + c_{\mathsf{wingtip}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\mathsf{wingtip}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + e_{\mathsf{wingtip}}\Upsilon \tag{92}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3.4.1 Kutta Condition: Linearized Pressure Jump

Alternate Kutta condition: zero linearized pressure jump at TE

$$0 = \Delta\left(\vec{U}\cdot\nabla\phi\right)_{\mathsf{TE}} = \Delta\left(\vec{U}_{||}\cdot\widetilde{\nabla}\phi + U_n\frac{\partial\phi}{\partial n}\right)_{\mathsf{TE}} = \Delta\left(\vec{U}_{||}\cdot\widetilde{\nabla}\phi - U_n^2\right)_{\mathsf{TE}} \tag{93}$$

where the last form incorporates the wall BC. Along the wing TE, the strong-form primal residual is

$$\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) \quad : \quad \int_{C_{\mathsf{TE}}} w_{\mathsf{TE}}(\psi,\Upsilon)\,\Delta\left(\vec{U}_{||}\cdot\widetilde{\nabla}\phi - U_n^2\right)_{\mathsf{TE}} \tag{94}$$

with

$$w_{\mathsf{TE}} = a_{\mathsf{TE}}\Sigma(\psi) + b_{\mathsf{TE}}\Delta(\psi) + c_{\mathsf{TE}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\mathsf{TE}}\Delta\left(\frac{\partial\psi}{\partial n}\right) + e_{\mathsf{TE}}\Upsilon + f_{\mathsf{TE}}\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\right] + g_{\mathsf{TE}}\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\psi)\right] \tag{95}$$

Strong-form adjoint residual from duality, including terms from wake-surface IBP,

$$\begin{aligned}\mathcal{R}_s^* \quad : \quad & \int_{C_{\mathsf{TE}}}\Big\{-\Big(\theta_1\Sigma(\phi)\Sigma(\psi) + \theta_4\Delta(\phi)\Sigma(\psi) + \theta_7\Gamma\Sigma(\psi) + \theta_8\Gamma\Delta(\psi) + \theta_9\Gamma\Upsilon\Big)\left(\vec{U}_{||}\cdot\hat{\mu}\right) \\ & \qquad + \left(a_{\mathsf{TE}}\Sigma(\psi) + b_{\mathsf{TE}}\Delta(\psi) + c_{\mathsf{TE}}\Sigma\left(\frac{\partial\psi}{\partial n}\right) + d_{\mathsf{TE}}\Delta\left(\frac{\partial\psi}{\partial n}\right) + e_{\mathsf{TE}}\Upsilon \right. \\ & \qquad\qquad \left. + f_{\mathsf{TE}}\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\right] + g_{\mathsf{TE}}\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\right]\right)\left(\vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\phi)\right)\Big\} \\ & = \int_{C_{\mathsf{TE}}}\mathbf{u}^t\mathbf{M}\mathbf{w} \end{aligned} \tag{96}$$

with

$$\mathbf{u} = \begin{pmatrix}\Sigma(\phi)\\ \Delta(\phi)\\ \Sigma(\partial\phi/\partial n)\\ \Delta(\partial\phi/\partial n)\\ \Gamma\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\phi)\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\phi)\end{pmatrix}, \qquad \mathbf{w} = \begin{pmatrix}\Sigma(\psi)\\ \Delta(\psi)\\ \Sigma(\partial\psi/\partial n)\\ \Delta(\partial\psi/\partial n)\\ \Upsilon\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Sigma(\psi)\\ \vec{U}_{||}\cdot\widetilde{\nabla}\Delta(\psi)\end{pmatrix}, \tag{97}$$

$$\mathbf{M} = \begin{pmatrix} -\theta_1\left(\vec{U}_{||}\cdot\hat{\mu}\right) & 0 & 0 & 0 & 0 & 0 & 0\\ -\theta_4\left(\vec{U}_{||}\cdot\hat{\mu}\right) & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0\\ -\theta_7\left(\vec{U}_{||}\cdot\hat{\mu}\right) & -\theta_8\left(\vec{U}_{||}\cdot\hat{\mu}\right) & 0 & 0 & -\theta_9\left(\vec{U}_{||}\cdot\hat{\mu}\right) & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0\\ a_{\mathsf{TE}} & b_{\mathsf{TE}} & c_{\mathsf{TE}} & d_{\mathsf{TE}} & e_{\mathsf{TE}} & f_{\mathsf{TE}} & g_{\mathsf{TE}} \end{pmatrix} \tag{98}$$

Zero eigenvector has multiplicity of 6 ($7-1=6$), so set one x_i in terms of all others. Obvious choice is $x_1 = 0$. Zero coefficients for all other x_i gives,

$$b_{\mathsf{TE}} = c_{\mathsf{TE}} = d_{\mathsf{TE}} = e_{\mathsf{TE}} = f_{\mathsf{TE}} = g_{\mathsf{TE}} = \theta_8 = \theta_9 = 0 \tag{99}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

giving the strong-form adjoint residual

$$\mathcal{R}_s^*(\psi, \Upsilon;\, \phi, \Gamma) \quad : \quad \int_{C_{\mathsf{TE}}} \Big[-\theta_1\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Sigma\left(\phi\right) - \theta_4\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Delta\left(\phi\right) - \theta_7\Big(\vec{U}_{||}\cdot\hat{\mu}\Big)\Gamma + a_{\mathsf{TE}}\Big[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\phi\right)\Big]\Big]\,\Big[\Sigma\left(\psi\right)\Big] \tag{100}$$

and strong-form primal residual

$$\mathcal{R}_s(\phi, \Gamma;\, \psi, \Upsilon) \quad : \quad \int_{C_{\mathsf{TE}}} a_{\mathsf{TE}}\Sigma\left(\psi\right)\,\Delta\left(\vec{U}_{||}\cdot\widetilde{\nabla}\phi - U_n^2\right)_{\mathsf{TE}} \tag{101}$$

leaving a_{TE}, θ_1, θ_4 and θ_7 as free parameters. Units: $a_{\mathsf{TE}} \sim \mathsf{L}/\mathsf{V}$; θ_1, θ_4, $\theta_7 \sim 1/\mathsf{V}$.

4 Energy Stability: BC Formulations without Lagrange Multipliers

Primal weak-form residual

$$\mathcal{R}_w(\phi, \Gamma;\, \psi, \Upsilon) = \iiint_\Omega \Big[\nabla\psi\cdot\nabla\phi\Big] - \oiint_{\partial\Omega} \psi\frac{\partial\phi}{\partial n} + \{\text{BC's}\} \tag{102}$$

Energy

$$\mathcal{E} = \mathcal{R}_w(\phi, \Gamma;\, \phi, \Gamma)\Big|_{U_n=0} = \iiint_\Omega \Big[\nabla\phi\cdot\nabla\phi\Big] - \oiint_{\partial\Omega} \phi\frac{\partial\phi}{\partial n} + \{\text{BC's}\}_{U_n=0} \tag{103}$$

4.1 Wake BCs

IBP term on wake

$$\begin{aligned}\oiint_{\partial\Omega} \phi\frac{\partial\phi}{\partial n} &= \iint_{\partial\Omega_b} + \iint_{\partial\Omega_f} + \iint_{\partial\Omega_{wu}} + \iint_{\partial\Omega_{wl}} \phi\frac{\partial\phi}{\partial n} \\ &= \ldots + \iint_{\partial\Omega_w} \left[\left(\phi\frac{\partial\phi}{\partial n}\right)_u - \left(\phi\frac{\partial\phi}{\partial n}\right)_l\right] = \ldots + \iint_{\partial\Omega_w} -\Delta\left(\phi\frac{\partial\phi}{\partial n}\right)\end{aligned} \tag{104}$$

Wake contribution to energy

$$\begin{aligned}\mathcal{E}_w \quad : \quad \iint_{\partial\Omega_w} \Bigg\{ &\frac{1}{2}\left[\Sigma\left(\phi\right)\Delta\left(\frac{\partial\phi}{\partial n}\right) + \Delta\left(\phi\right)\Sigma\left(\frac{\partial\phi}{\partial n}\right)\right] \\ &+ w_a(\phi, \Gamma)\,\Sigma\left(\frac{\partial\phi}{\partial n}\right) + w_b(\phi, \Gamma)\left[\Delta\left(\phi\right) + \Gamma\right] + w_c(\phi, \Gamma)\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\right]\Bigg\}\end{aligned} \tag{105}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

where

$$
\begin{aligned}
w_a(\phi, \Gamma) &= a_1 \Sigma\left(\phi\right) - \frac{1}{2} \Delta\left(\phi\right) + a_3 \Sigma\left(\frac{\partial \phi}{\partial n}\right) + a_4 \Delta\left(\frac{\partial \phi}{\partial n}\right) + 2 a_3 \theta_8\, \widetilde{\nabla} \cdot \left[\vec{U}_{||} \Delta\left(\phi\right)\right] \\
&\quad + 2 a_3 \theta_9\, \widetilde{\nabla} \cdot \left[\vec{U}_{||} \Gamma\right] - 2(a_4 \theta_1 - a_3(\theta_7 - \theta_4)) \left[\vec{U}_{||} \cdot \widetilde{\nabla} \Sigma\left(\phi\right)\right] && (106) \\
w_b(\phi, \Gamma) &= b_1 \Sigma\left(\phi\right) + b_3 \Sigma\left(\frac{\partial \phi}{\partial n}\right) + b_4 \Delta\left(\frac{\partial \phi}{\partial n}\right) \\
&\quad - (1 - 2 b_3) \left(\theta_8\, \widetilde{\nabla} \cdot \left[\vec{U}_{||} \Delta\left(\phi\right)\right] + \theta_9\, \widetilde{\nabla} \cdot \left[\vec{U}_{||} \Gamma\right]\right) \\
&\quad - (\theta_7 - 2 b_3 (\theta_7 - \theta_4) + 2 b_4 \theta_1) \left[\vec{U}_{||} \cdot \widetilde{\nabla} \Sigma\left(\phi\right)\right] && (107) \\
w_c(\phi, \Gamma) &= c_1 \Sigma\left(\phi\right) + c_3 \Sigma\left(\frac{\partial \phi}{\partial n}\right) + c_4 \Delta\left(\frac{\partial \phi}{\partial n}\right) - \theta_8 \Delta\left(\phi\right) - \theta_9 \Gamma \\
&\quad + 2 c_3 \left(\theta_8\, \widetilde{\nabla} \cdot \left[\vec{U}_{||} \Delta\left(\phi\right)\right] + \theta_9\, \widetilde{\nabla} \cdot \left[\vec{U}_{||} \Gamma\right]\right) \\
&\quad + 2 \left(c_3 (\theta_7 - \theta_4) - c_4 \theta_1\right) \left[\vec{U}_{||} \cdot \widetilde{\nabla} \Sigma\left(\phi\right)\right] && (108)
\end{aligned}
$$

Rewritten in matrix-vector form,

$$
\mathcal{E}_w \quad : \quad \iint_{\partial \Omega_w} \left\{ \frac{1}{2} \mathbf{u}^t \mathbf{M} \mathbf{u} \right\} \tag{109}
$$

with...

We can scrape terms from the volume

$$
\begin{aligned}
\iiint_{\Omega} \nabla \phi \cdot \nabla \phi &= \ldots + \iint_{\partial \Omega_{wu}} h_u \left(\frac{\partial \phi}{\partial n}\right)_u^2 + \iint_{\partial \Omega_{wl}} h_l \left(\frac{\partial \phi}{\partial n}\right)_l^2 = \ldots + \iint_{\partial \Omega_w} -\Delta\left(h \left(\frac{\partial \phi}{\partial n}\right)^2\right) \\
&= \ldots - \iint_{\partial \Omega_w} \frac{1}{2} \left[\Sigma\left(h\right) \Delta\left(\left(\frac{\partial \phi}{\partial n}\right)^2\right) + \Delta\left(h\right) \Sigma\left(\left(\frac{\partial \phi}{\partial n}\right)^2\right)\right] \\
&= \ldots - \iint_{\partial \Omega_w} \frac{1}{2} \left\{\Sigma\left(h\right) \Sigma\left(\frac{\partial \phi}{\partial n}\right) \Delta\left(\frac{\partial \phi}{\partial n}\right) + \frac{1}{2} \Delta\left(h\right) \left[\left(\Sigma\left(\frac{\partial \phi}{\partial n}\right)\right)^2 + \left(\Delta\left(\frac{\partial \phi}{\partial n}\right)\right)^2\right]\right\} && (110)
\end{aligned}
$$

giving quadratic terms in the normal derivatives. Without further quadratics, all cross terms in $\Sigma\left(\phi\right)$ and $\Delta\left(\phi\right)$ must vanish; but this is not possible since the first term due to the volume IBP is always present (coefficient is 1/2).

5 BC Formulations with Lagrange Multipliers: *mit-Lagrange*

5.1 Wake BCs: Mass Flux, Potential Jump and Circulation Gradient

Primal BCs

$$
\begin{aligned}
&\text{mass flux:} && \Sigma\left(\frac{\partial \phi}{\partial n}\right) = 0 && (111) \\
&\text{potential jump:} && \Delta\left(\phi\right) = -\Gamma && (112) \\
&\text{circulation:} && \vec{U}_{||} \cdot \widetilde{\nabla} \Gamma = 0 && (113)
\end{aligned}
$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

From *sans-Lagrange*, adjoint BCs are

$$\Delta\left(\frac{\partial\psi}{\partial n}\right) - 2\theta_1\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\psi\right)\right] = \mathrm{rhs}_1, \tag{114}$$

$$\Sigma\left(\psi\right) = \mathrm{rhs}_2, \tag{115}$$

$$\Sigma\left(\frac{\partial\psi}{\partial n}\right) + \frac{\theta_7-\theta_4}{\theta_1}\Delta\left(\frac{\partial\psi}{\partial n}\right) + 2\theta_8\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta\left(\psi\right) + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\psi\right)\right]\right) + 2\theta_9\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\right]\right) = \mathrm{rhs}_3 \tag{116}$$

Primal weights are,

$$\begin{aligned} w_a &= a_1\Sigma\left(\psi\right) - \frac{1}{2}\Delta\left(\psi\right) + a_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + a_4\Delta\left(\frac{\partial\psi}{\partial n}\right) + 2a_3\theta_8\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta\left(\psi\right) + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\psi\right)\right]\right) \\ &\quad + 2a_3\theta_9\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\right]\right) - 2(a_4\theta_1 - a_3(\theta_7-\theta_4))\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\psi\right)\right] && (117) \\ w_b &= b_1\Sigma\left(\psi\right) + b_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + b_4\Delta\left(\frac{\partial\psi}{\partial n}\right) \\ &\quad - (1-2b_3)\left(\theta_8\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta\left(\psi\right) + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\psi\right)\right]\right) + \theta_9\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\right]\right)\right) \\ &\quad - (\theta_7 - 2b_3(\theta_7-\theta_4) + 2b_4\theta_1)\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\psi\right)\right] && (118) \\ w_c &= c_1\Sigma\left(\psi\right) + c_3\Sigma\left(\frac{\partial\psi}{\partial n}\right) + c_4\Delta\left(\frac{\partial\psi}{\partial n}\right) - \theta_8\Delta\left(\psi\right) - \theta_9\Upsilon \\ &\quad + 2c_3\left[\theta_8\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta\left(\psi\right) + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\psi\right)\right]\right) + \theta_9\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Upsilon + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Upsilon\right]\right)\right] \\ &\quad + 2\left(c_3(\theta_7-\theta_4) - c_4\theta_1\right)\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\psi\right)\right] && (119) \end{aligned}$$

Adjoint weights are (note reordering vs *sans-Lagrange* section),

$$\begin{aligned} \omega_1 &= -\frac{1}{2}\Sigma\left(\phi\right) + \frac{\theta_7-\theta_4}{2\theta_1}\Delta\left(\phi\right) + \left(b_4 - b_3\frac{\theta_7-\theta_4}{\theta_1}\right)\left(\Delta\left(\phi\right)+\Gamma\right) + \left(a_4 - a_3\frac{\theta_7-\theta_4}{\theta_1}\right)\Sigma\left(\frac{\partial\phi}{\partial n}\right) \\ &\quad + \left(c_4 - c_3\frac{\theta_7-\theta_4}{\theta_1}\right)\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\right] && (120) \\ \omega_2 &= \theta_1\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Sigma\left(\phi\right) + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Sigma\left(\phi\right)\right]\right) + \theta_4\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Delta\left(\phi\right) + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Delta\left(\phi\right)\right]\right) + b_1(\Delta\left(\phi\right)+\Gamma) \\ &\quad + a_1\Sigma\left(\frac{\partial\phi}{\partial n}\right) + \frac{1}{2}\Delta\left(\frac{\partial\phi}{\partial n}\right) + \theta_7\left((\widetilde{\nabla}\cdot\vec{U}_{||})\Gamma + \left[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\right]\right) + c_1\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\right] && (121) \\ \omega_3 &= -\frac{1}{2}\Delta\left(\phi\right) + b_3(\Delta\left(\phi\right)+\Gamma) + a_3\Sigma\left(\frac{\partial\phi}{\partial n}\right) + c_3\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\right] && (122) \end{aligned}$$

giving the strong-form primal residual

$$\mathcal{R}_s(\phi,\Gamma;\,\psi,\Upsilon) \quad : \quad \iint_{\partial\Omega_w}\left\{w_a\left[\Sigma\left(\frac{\partial\phi}{\partial n}\right)\right] + w_b\Big[\Delta\left(\phi\right)+\Gamma\Big] + w_c\left[\vec{U}_{||}\cdot\widetilde{\nabla}\Gamma\right]\right\} \tag{123}$$

and the corresponding strong-form adjoint residual

$$\mathcal{R}^*_s(\psi,\Upsilon;\,\phi,\Gamma) \quad : \quad \iint_{\partial\Omega_w}\left\{\omega_1\Big[\mathrm{BC}^*_1 - \mathrm{rhs}_1\Big] + \omega_2\Big[\mathrm{BC}^*_2 - \mathrm{rhs}_2\Big] + \omega_3\Big[\mathrm{BC}^*_3 - \mathrm{rhs}_3\Big]\right\} \tag{124}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

The *mit-Lagrange* formulation has strong-form primal residual

$$\begin{aligned}\mathcal{R}_s(\phi, \Gamma, \lambda;\, \psi, \Upsilon, \mu) \quad &: \quad \iint_{\partial\Omega_w} \Bigg\{ \mu_1 \left[\Sigma \left(\frac{\partial \phi}{\partial n} \right) \right] + \mu_2 \Big[\Delta\left(\phi\right) + \Gamma \Big] + \mu_3 \left[\vec{U}_{||} \cdot \widetilde{\nabla} \Gamma \right] \\ &\qquad + \Big[\mathrm{BC}_1^* \Big] \Big[\lambda_1 - \omega_1 \Big] + \Big[\mathrm{BC}_2^* \Big] \Big[\lambda_2 - \omega_2 \Big] + \Big[\mathrm{BC}_3^* \Big] \Big[\lambda_3 - \omega_3 \Big] \Bigg\} \end{aligned} \tag{125}$$

and strong-form adjoint residual

$$\begin{aligned}\mathcal{R}_s^*(\psi, \Upsilon, \mu;\, \phi, \Gamma, \lambda) \quad &: \quad \iint_{\partial\Omega_w} \Bigg\{ \lambda_1 \Big[\mathrm{BC}_1^* - \mathrm{rhs}_1 \Big] + \lambda_2 \Big[\mathrm{BC}_2^* - \mathrm{rhs}_2 \Big] + \lambda_3 \Big[\mathrm{BC}_3^* - \mathrm{rhs}_3 \Big] \\ &\qquad + \left[\Sigma \left(\frac{\partial \phi}{\partial n} \right) \right] \Big[\mu_1 - w_a \Big] + \Big[\Delta\left(\phi\right) + \Gamma \Big] \Big[\mu_2 - w_b \Big] + \left[\vec{U}_{||} \cdot \widetilde{\nabla} \Gamma \right] \Big[\mu_3 - w_c \Big] \Bigg\} \end{aligned} \tag{126}$$

Without altering duality, we can add arbitrarily to λ_i and μ_i

$$\lambda_i \quad \rightarrow \quad \lambda_i + A_i \Sigma\left(\phi\right) + B_i \Delta\left(\phi\right) + C_i \Sigma \left(\frac{\partial \phi}{\partial n} \right) + D_i \Delta \left(\frac{\partial \phi}{\partial n} \right) + E_i \Gamma + \ldots \tag{127}$$

$$\mu_i \quad \rightarrow \quad \mu_i + \bar{A}_i \Sigma\left(\psi\right) + \bar{B}_i \Delta\left(\psi\right) + \bar{C}_i \Sigma \left(\frac{\partial \psi}{\partial n} \right) + \bar{D}_i \Delta \left(\frac{\partial \psi}{\partial n} \right) + \bar{E}_i \Upsilon + \ldots \tag{128}$$

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

BIBLIOGRAPHY

Robert Haimes and Mark Drela, “On the Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design”, AIAApaper2012-0683.

John Dannenhoffer, “OpenCSM: An Open-Source Constructive Solid Modeler for MDAO”, AIAApaper2013-0701.

Bridget Dixon and John Dannenhoffer, “Geometric Sketch Constraint Solving with User Feedback”, AIAApaper2013-0702.

Robert Haimes and John Dannenhoffer, “The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry”, AIAApaper2013-3073.

Nitin Bhagat and Edward Alyanak, “Computational Geometry for Multifidelity and Multidisciplinary Analysis and Optimization”, AIAApaper2014-0188.

John Dannenhoffer and Robert Haimes, “Conservative Fitting for Multi-Disciplinary Analysis”, AIAApaper2014-0294.

John Dannenhoffer and Robert Haimes, “Design Sensitivity Calculations Directly on CAD-based Geometry”, AIAApaper2015-1370.

John Dannenhoffer and Robert Haimes, “Generation of Multi-fidelity, Multi-discipline Air Vehicle Models with the Engineering Sketch Pad”, AIAApaper2016-1925.

Pengcheng Jia and John Dannenhoffer, “Generation of Parametric Aircraft Models from a Cloud of Points”, AIAApaper2016-1926.

Edward Alyanak, Ryan Durscher, Robert Haimes, John Dannenhoffer, Nitin Bhagat and Darcy Allison, “Multi-fidelity Geometry-centric Multi-disciplinary Analysis for Design”, AIAApaper2016-4007.

John Dannenhoffer, “The Creation of a Static BRep Model Given a Cloud of Points”, AIAApaper2017-0138.

John Dannenhoffer and Robert Haimes, “Using Design-Parameter Sensitivities in Adjoint-Based Design Environments”, AIAApaper2017-0139.

Robert Haimes and John Dannenhoffer, “EGADSlite: A Lightweight Geometry Kernel for HPC”, AIAApaper2018-1401.

Zachary Eager and John Dannenhoffer, “Flends: Generalized Fillets via B-splines”, AIAApaper2019-1717.

Steve Karman and Nick Wyman, “Automatic Unstructured Mesh Generation with Geometry Attribution”, AIAApaper2019-1721.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Julia Docampo-Sánchez and Robert Haimes, “Towards Fully Regular Quad Mesh Generation”, AIAApaper2019-1988.

Philip Caplan, Robert Haimes, David Darmofal and Marshall Galbraith, “Extension of local cavity operators to 3d + t spacetime mesh adaptation”, AIAApaper2019-1992.

Dean Bryson, Robert Haimes and John Dannenhoffer, “Toward the Realization of a Highly Integrated, Multidisciplinary, Multifidelity Design Environment”, AIAApaper2019-2225.

Ryan Durscher and Dennis Reedy, “pyCAPS: A Python Interface to the Computational Aircraft Prototype Syntheses”, AIAApaper2019-2226.

Mark Drela, Marshall Galbraith and Steven Allmaras, “Hybrid Shell Model for Aeroelastic Modeling”, AIAApaper2019-2227.

Robert Canfield, Suood Alnaqbi, Ryan Durscher, Dean Bryson and Raymond Kolonay, “Shape Continuum Sensitivity Analysis using ASTROS and CAPS”, AIAApaper2019-2228.

John Joe, Viraj Gandhi, John Dannenhoffer and Hamid Dalir, “Rapid Generation of Parametric Aircraft Structural Models”, AIAApaper2019-2229.

Christopher Meckstroth, “Parameterized, Multi-fidelity Aircraft Geometry and Analysis for MDAO Studies using CAPS”, AIAApaper2019-2230.

Justin Clough, Assad Oberai and Andrew Zakrajsek, “Automated Wing Internal Structure Placement Guided by Finite Element Analysis”, AIAApaper2019-3369.

William Chan, Shishir Pandya, and Robert Haimes, “Automation of Overset Structured Mesh Generation on Complex Geometries”, AIAApaper2019-3671.

Julia Docampo-Sánchez and Robert Haimes, “A Regularization Approach for Automatic Quad Mesh Generation”, Presented at the 28th Interntional Meshing Roundtable.

Philip Caplan, Robert Haimes and David Darmofal, “Four-dimentional anisotropic mesh adaptation”, Submitted to *Computer-Aided Design*.

Note that most of these papers can be found at http://acdl.mit.edu/ESP/Publications

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.