Multi-fidelity Geometry-centric Multi-disciplinary Analysis for Design

Edward Alyanak*, Ryan Durscher†

Air Force Research Laboratory, Wright Patterson Air Force Base, Ohio, 45433

Robert Haimes‡

Aerospace Computational Design Laboratory

Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139

John F. Dannenhoffer, III§

Aerospace Computational Methods Laboratory

Syracuse University, Syracuse, New York, 13244

Nitin Bhagat¶

Wright State Research Institute, Beavercreek, Ohio, 45431

Darcy Allison‖

Optimal Flight Sciences, LLC, Dayton Ohio, 45433

Aerospace vehicle design can be described as an evolutionary process of gathering information to make informed decisions. Meticulous application of this process involves numerous simulations covering many disciplines and fidelity levels. A design team needs to be able to easily increase or decrease fidelity as they gather more information about a particular design. To this end a geometry system that can support multi-disciplinary, multi-fidelity analysis from a single source is required. The Computational Aircraft Prototype Syntheses (CAPS), which is a part of the Engineering Sketch Pad (ESP), satisfies the above by combining proven computational geometry, meshing, and analyses model generation techniques into a complete browser-based, client-server environment that is accessible to the entire design team of an aerospace vehicle. CAPS links analysis and meshing disciplines to any ESP geometry model via dynamically-loadable Analysis Interface Module (AIM) plugins. CAPS is accessed from either a browser-based user interface and or Multi-Disciplinary Analysis and Optimization (MDAO) framework through a programing interface. In this paper we describe the fundamental building blocks of CAPS and ESP. The paradigm shift of creating geometry for multi-fidelity design is described in detail and represented in ESP scripts. We then demonstrate the use of this multi-fidelity geometry to support multi-fidelity, multi-physics analysis including discipline coupling.

*Research Engineer, Aerospace Systems Directorate, AIAA Sr. Member.

†Research Engineer, Aerospace Systems Directorate, AIAA Member.

‡Principal Research Engineer, Department of Aeronautics & Astronautics, AIAA Member.

§Associate Professor, Mechanical and Aerospace Engineering, AIAA Associate Fellow.

¶Research Engineer, AFRL Contractor, AIAA Member.

‖Owner and Aerospace Engineer, AFRL Contractor, AIAA Senior Member.

Nomenclature

AIM	Analysis Interface Module
API	Application Programming Interface
BEM	Built Up Element Model
BRep	Boundary Representation
CAD	Computer Aided {Drafting, Drawing, Design, Development}
CAPS	Computational Aircraft Prototype Syntheses
CSG	Constructive Solid Geometry
EGADS	Engineering Geometry Aircraft Design System
ESP	Engineering Sketch Pad
MDAO	Multi-Disciplinary Analysis and Optimization
MSA	Mid-Surface Aero
OML	Outer Mold Line
OpenCSM	Open-source Constructive Solid Modeler
SBO	Solid Boolean Operators
UDC	User-Defined Component
UDP	User-Defined Primitive

I. Introduction

In the design of real configurations, such as aerospace vehicles, geometry plays 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 a 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 is 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 in 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. Parametric geometries allow a designer to not only rapidly modify the geometry, but also allow the possibility of determining the sensitivities of the simulation analyses results with respect to the design parameters. With IGES or STEP one can only do analysis on that particular geometry with no ability to modify it. If IGES or STEP happen to be the required input format by an individual disciplinary analysis code, then these must be generated by a parametric geometry model first before the analysis is performed in order to be able to determine the design sensitivities.

II. Computational Aircraft Prototype Syntheses (CAPS)

Currently, most organizations have often 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 at a conceptual level and can continue to be leveraged throughout preliminary and into detailed design has obvious advantages.

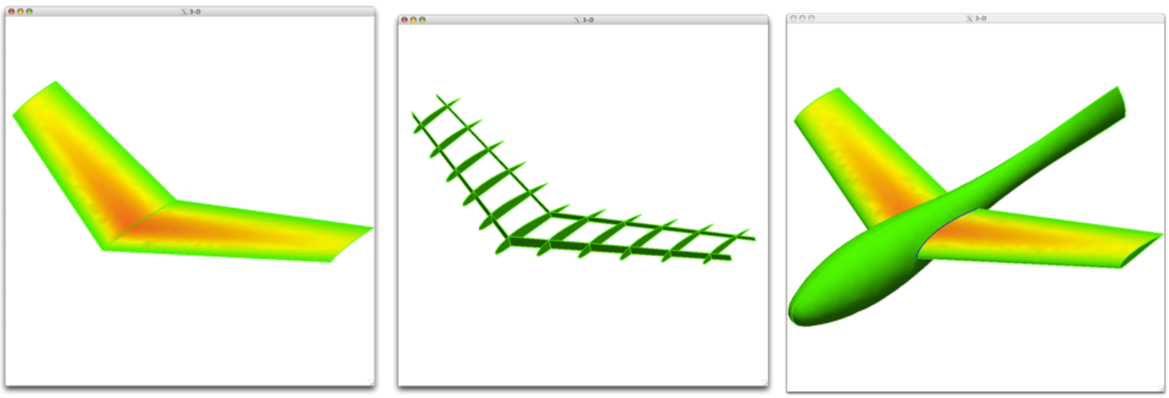


Figure 1: An **ESP** example of the mid-surface-aero (MSA), built-up-element (BEM), and outer-mold-line (OML) models that are generated from a single generic wing/body configuration. The three geometric models, which all have the same driving parameters, are colored based upon the sensitivity of the local surface normal to changes in the wing's camber. Note that the MSA and OML models show the maximum sensitivities near the maximum camber position, as expected; the BEM does not show such sensitivities since the displacements caused by camber changes are not normal to the surfaces of the BEM.

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 driving 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 to develop 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 (see Figure 1). A user accesses this software through a web browser and this complete suite is referred to as the Engineering Sketch Pad (**ESP**),[1] which is a fully-parametric, attributed, feature-based solid-modeling system.

ESP is built both upon the WebViewer[1] and upon **OpenCSM**.[2] **OpenCSM** in turn is built upon **EGADS**[3] and OpenCASCADE. The WebViewer is a WebGL-based visualizer for three-dimensional configurations and data while **OpenCSM** is a constructive solid modeler. There is no absolute requirement to make **ESP** dependent on OpenCASCADE; rather this CAD kernel is chosen because it is open-source and can be distributed with **ESP**. In fact, all of this software is open-source, freely available without licensing restrictions, and is in general use.[4]

Building 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 is not an easy task. This is accomplished in **CAPS** by attacking the process-related bottlenecks head-on. For example, when performing a vortex lattice aerodynamic analysis of a wing with the geometric description of an 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* where, in this case, the mid-surface aerodynamic shape is generated directly. In general, the **CAPS** software allows users (from their web browsers) to

- deal with legacy geometry
- allow for easy assembly of component models into aircraft concepts
- visually exercise their models

Component or subcomponent 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) to:

- change a 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
- setup for the execution of the specific analysis code

A simplified block diagram of the **CAPS** system can be seen in Figure 2. The boxes displayed are individual software components that are either expressed as APIs or dynamically loadable plug-ins. The cylindrically shaped entities refer to information or collections of files found on disk. The arrows reflect the internal movement of specific types of (meta-)data within **CAPS**. Each functional component is described below.

A. CAPS Executive

The primary programmatic access point into **CAPS** is through the **CAPS** Executive. It is envisioned that there will be 3 different approaches to using **CAPS**. One way is to interactively build a model and exercise the build to examine aspects such as the design sensitivities. Another scenario is to run one or more analysis packages interactively. Both of these approaches use an enhanced version of **ESP** (within a Web Browser) to interact with **CAPS** directly. The last approach has **CAPS** driven by an optimizer or by an MDO framework, such as SORCER,[5] ModelCenter[6] or OpenMDAO.[7] The access, under all of these cases, is through the **CAPS** API, which in essence, is the portal to all of the **CAPS** functionality (the lower 2 dotted arrows as seen at the left of Figure 2).

The **CAPS** API is 'object-based' very much in the same manner as both **EGADS** and **OpenCSM**. 'Object-based' (unlike object-oriented) is a technique that allows for the use of *objects* in traditional procedural-based programming. This is done with *opaque* (or blind) pointers that the API functions parse and then perform the desired functions driven by the type of *object* input. The hierarchy of the **CAPS** objects can be seen in Figure 3 and include the following types:

- capsProblem. 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 multi-disciplinary. Various data entries can be connected via *linkage ports* found in all *input* objects. 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.

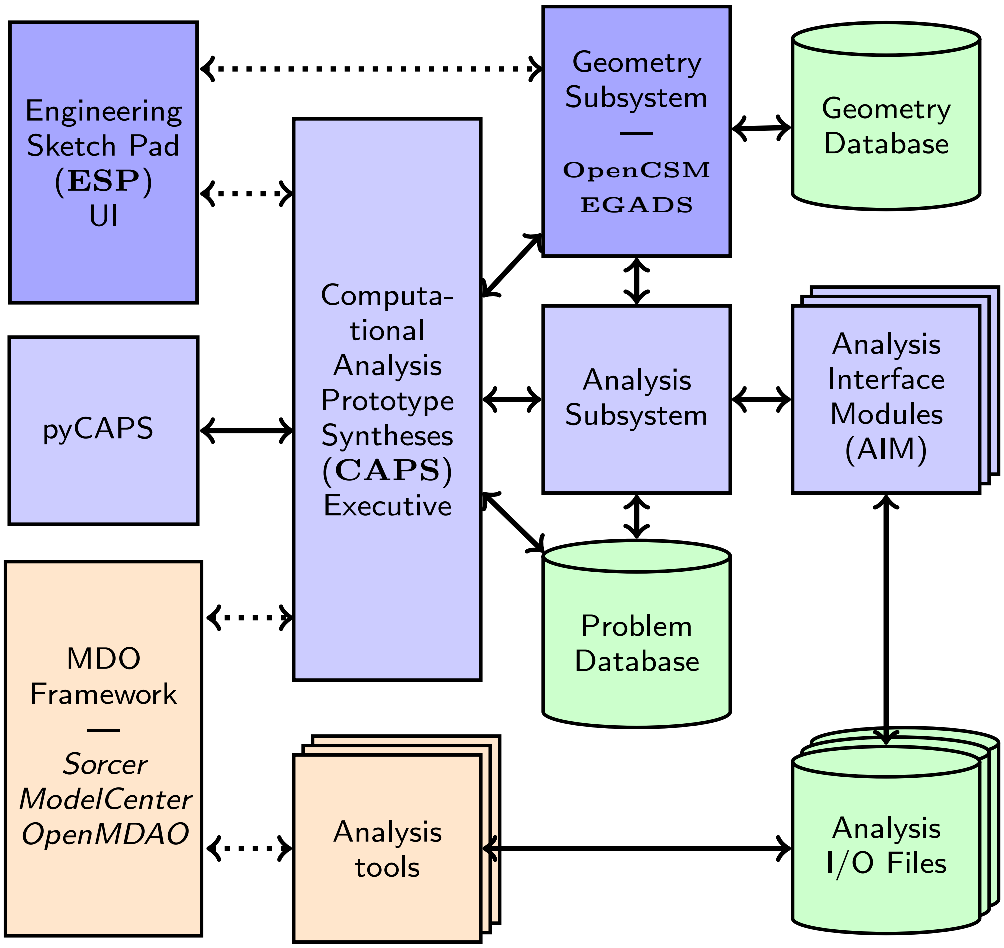


Figure 2: A block diagram of the **ESP** system with **CAPS**. The boxes on the left reflect the access points into the system. The core portions are both the Geometry and Analysis Subsystems. The AIM boxes (like the UDPs of the Geometry Subsystem) use a plugin technology that enhances **CAPS** at run-time. All blue boxes are **ESP** proper where the tan boxes reflect code from elsewhere.

- capsValue. 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.

- capsAnalysis. 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.

- capsBound. A Bound is a logical grouping of BRep Objects that all represent the same entity in an engineering sense (such as the "upper 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).

- capsVertexSet. 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 its 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 geometry itself.

- capsDataSet. 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).

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	capsBound

Figure 3: The hierarchy of **CAPS** objects.

B. Geometry Subsystem

The Geometry Subsystem is responsible for building geometry at the requested level of fidelity using the build recipe. The result is a Boundary Representation (BRep) that is a hierarchical object container that relates geometric entities through a topology. These BRep objects (geometry and topology) are the fundamental geometric model instances that are passed around the **ESP/CAPS** system. This subsystem is comprised of **OpenCSM** (the parametric build component) and **EGADS** (a simple object-based geometry kernel) which depends on OpenCASCADE (an open-source solid-modeling API).

Attribution of the objects found within the BRep can be accomplished during the build, or anytime thereafter using **EGADS** functions. This metadata is persistent when applied to topological objects and saved while writing the geometry (in an **EGADS** native file format).

EGADS also provides the following functions:

- BRep parsing. Given a BRep, the constituent hierarchical objects can be traversed and the overall connectivity understood.

- Tessellation. A complete and watertight (for a solid) triangulation can be requested for any BRep. This tessellation is of high quality and can be adjusted by a number of parameters. This is a good starting point for many meshing schemes (and can be thought of as STL data, but closed). Though currently not automatic, quadrilaterals can be overlaid on top of the triangles in a way that maintains the watertight nature of the tessellation.

- Evaluations & inverse evaluations. Given a geometric object and its parameter(s) (that refer to the position within the object: for example, curves have a t parameter and surfaces have $[u, v]$ parameters) return the physical coordinates. An inverse evaluation is the opposite function, that is given a position in physical space what is the closest position on the object to that point and what are the corresponding geometric parameter(s). These are important functions needed by most grid generation algorithms.

- Import and export of standard files. Both IGES and STEP can be imported to this system or can be exported by utilizing the functionality of **EGADS**.

The parametric engine **OpenCSM** provides a top down perspective on construction, which is based loosely on Constructive Solid Geometry (CSG). **OpenCSM** can easily generate various solid primitives, which can be subsequently used by the Solid Boolean Operators (SBO) to make more complex shapes. Also a 2D sketcher can be utilized to produce closed loops that can be expanded into solid entities by features such as extrudes, revolves, lofts and blends. **OpenCSM** has the ability to apply features such as fillets and chamfers. This is all within the realm of traditional CAD systems, but this system has the following differentiating characteristics:

- Parametric sensitivities. Parametric derivatives can be retrieved for any collections of parameters at specific locations on the geometry; see Figure 1. This is accomplished via a hybrid scheme of analytic differentiation (where possible), compiler-based code differentiation, and point-tracking finite-differences (where analytics are not feasible).

- Multi-fidelity geometric expression. The ability to generate geometry that is at the correct fidelity for the analysis at-hand is critical for an automated process (again, see Figure 1). Commercial CAD generates a model of single fidelity, which is rarely appropriate for anything but manufacturing. The currently supported **CAPS** expressions of fidelity can be see in Table 1.

Table 1: Current **CAPS** supported Fidelities

Fidelity	**Description**	**Fidelity**	**Description**
1	Beam	11	MidSurface Aero
2	ASWING	12	MidSurface w/ Control Surfaces
3	LSM	13	AVL
4	Built-up Element Model	14	OML/Wake – Full Potential
5	Solid Structural Model	15	OML – Euler
		16	OML – RANS

- Custom features. Provides a bottom up construction paradigm that can be fully incorporated into the larger build environment – see UDP (below). This can be thought of as a custom CAD feature, but is more powerful than that. Parametric geometry for full (or sub) components can be constructed in this manner. These features can be used to encapsulate conceptual knowledge and have it expressed in realizable geometry, which can be treated the same as any other **OpenCSM** geometric primitive.

C. User Defined Primitive (UDP) Plug-ins

A dynamic-load run-time plug-in architecture is used to ensure that one can add custom features to **OpenCSM** without changing the software. Parameters that influence the construction are registered when the plug-in is first loaded. The plug-in must also respond to these requests:

- Adjust a parameter value: the value for a parameter can be modified.

- Build: the request for a build. The current suite of parameter values is used to generate the geometry. **EGADS** functions must be invoked to build the BRep that is returned to **OpenCSM**.

- Sensitivity: if possible return the parametric sensitivities for the selected set of parameters. If this is not computed within the UDP, **OpenCSM** forces finite-differences to be applied to compute the derivatives for any parameter that is part of the UDP.

D. Analysis Subsystem

The Analysis Subsystem is where analysis data is prepared, held, queried, and where meshing is either accomplished or inputs are prepared for the use of a stand-alone grid generation package. A BRep at the proper level of fidelity (optionally with its tessellation) can be retrieved from the Geometry Subsystem, which is used for the meshing process. Again, to maintain the flexibility of **CAPS**, all specific analysis packages have (at least) one associated plug-in to perform the associated analysis. Multiple plug-ins may be attached to a single analysis package if the analysis has different execution modes (with different input requirements) and/or requiring differing levels of geometric fidelity. See the discussion of the **AIM** plug-ins below.

This subsystem is responsible for parsing the analysis output and aggregating the results to be passed to the **CAPS** Executive or maintained within this subsystem to pass on to another analysis module (in a multi-disciplinary setting). The plugin also handles retrieving the salient outputs (that can be constructed as objective functions) from the analysis.

E. Analysis Interface Module (AIM) Plug-ins

The type of geometric fidelity expected by the plug-in is specified at dynamic load registration (which is something like: Outer Mode Line, Mid-Surface Aero, Built-up Element Model, Structural Solid Model, etc.) as seen in Table 1. Any inputs (not associated with the BRep) need to be specified at registration. 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 and performing conservative data fitting.[8] 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.
- Quilting: after meshing and the association of the mesh vertices back to the geometry, this function specifies collections of BRep objects that may act as a single engineering "surface". This function also specifies element types so that the solver-based discretization is fully described.
- Analysis Input File(s) Generation: the input values and attributes found on the geometry are used to construct and generate the input file(s) required to run the analysis.
- Output file 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).
- Conservative Data Transfer Functions:[8] in order to perform the interdisciplinary coupling in a conservative manner, functions that compute interpolation within a surface element, integration of quantities over an element (and their backward or dual variants) are needed.

F. pyCAPS

As previous detailed, **CAPS** is designed to be incorporated into a larger MDO infrastructure; as such **pyCAPS** provides a light weight, Python-based framework that logically ties together features of the **CAPS**

"C" API to enable rapid generation of problems in the Python environment. To date not all features of the **CAPS** API have been utilized in **pyCAPS**.

In its current state, **pyCAPS** consists of a single parent class, 'capsProblem', and three children classes: 'capsAnalysis', 'capsGeometry', and 'capsDataTransfer'. The capsProblem class controls the overall workflow for a given problem (similar to the object defined in Section II.A of the same name) and contains functions to load the geometry, load AIMs, setup data transfers, etc. The children classes provide additional functionality to the capsProblem class for dealing with specific aspects of their respective domains. Multiple analysis (capsAnalysis class) and data transfer (capsDataTransfer class) objects may be loaded into a single capsProblem object and are book-kept by the class in a dictionary format for easy reference. The following outlines additional details for the children classes:

- capsAnalysis - A single instance of the capsAnalysis class is synonymous with an instance of a loaded AIM (Section II.E). This class provides the user the ability to set input values (e.g., Mach number, angle of attack), retrieve output variables (e.g., lift or drag coefficient), and other functionality outlined in Section II.D for a given analysis plug-in.
- capsGeometry - This class consists of functions to interact with the geometry model (e.g., set a geometric design parameter) loaded into a given capsProblem.
- capsDataTransfer - A single instance of the capsDataTransfer class corresponds to an instance of a capsBound (Section II.A), with capsDataSets (Section II.A) being referenced in a dictionary format. The primary function of the capsDataTransfer class is to execute the **CAPS** routines needed to initiate a data transfer to take place for given variable (i.e., capsDataSet).

III. Example Problems

To highlight the multi-fidelity, geometric parametrization provided by **ESP** and **CAPS**, a generic glider is chosen as an example problem. Some effort is taken to ensure that this example is grounded in reality and is representative of an 18 (m) span class glider. The global quantities defining the glider geometry are provided in Table 2 with the corresponding model shown in Figure 4. Please note that the intention here is to demonstrate capability, not to design a glider.

Table 2: Global Glider Values

Variable	Value
Aspect Ratio	27.5
Wing Taper Ratio	0.8
Wing Area	127 (ft^2)
Wing Sweep	0.0 ($^\circ$)
Flight Weight	1000 (lbs)

A. Geometry Construction

In a multi-discipline, multi-fidelity environment, multiple instances of geometry (such as shown in Figure 6) must be made from a single set of design parameters. This is done here through the use of **OpenCSM**,[2] which is a parametric, constructive solid modeling system that was designed specifically to be used as the front-end for analysis and design systems (including adjoint-based design systems).

Configurations in **OpenCSM** are defined in an ASCII file, which can be prepared either by hand, interactively through the **ESP** user interface, or can be written externally by an editor or any program that can write an ASCII file.

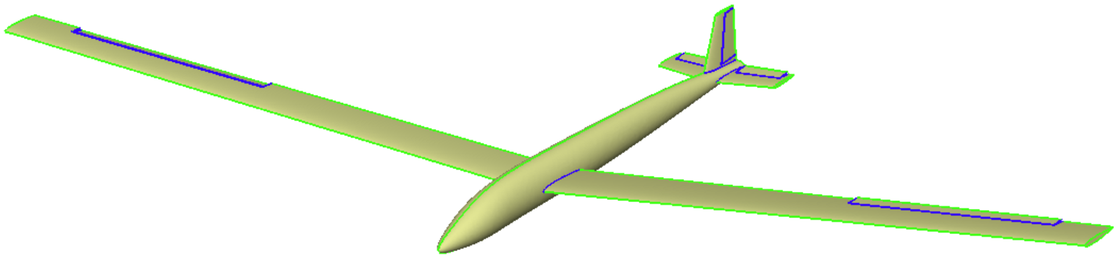


Figure 4: Solid Geometry Representation of Glider Example – **CAPS** fidelity 15/16.

The configuration construction process in **OpenCSM** follows the traditional construction process available in most other modeling systems. It works by first constructing the base geometry; this can be done either with simple primitives (such as a box, cylinder, cone, sphere, or torus), by extruding, rotating, ruling, or blending two-dimensional sketches, or via user-defined primitives — which turn out to be very useful when creating aerospace vehicles. Once the base geometry is made, it can be moved (through translation, rotation, scaling, or mirroring operations), various base geometries can be combined (with Boolean unions, intersections, and differences), and any geometry can be modified (for example, by adding fillets or chamfers). The order in which these operations are done defines the relationship amongst the various operations in a natural way through the use of a stack of operations.

To understand this process, consider the various models that are required for a wing; a similar process is used for the fuselage (and engines for non-glider configurations).

The process begins with the definition of the user's design parameters. An important feature of **OpenCSM** is that the meanings of the design parameters are defined by the writer of the model. So for a wing, one may define design parameters such as:

```
# wing design parameters
despmtr   xloc        6.0     # x-location of wing in aircraft model
despmtr   zloc        0.2     # z-location of wing in aircraft model
despmtr   area        127     # planform area
despmtr   aspect      27.5    # aspect ratio
despmtr   taper       0.8     # taper ratio
despmtr   twist       0.0     # tip washout (deg)
despmtr   sweep       0.0     # leading edge sweep (deg)
despmtr   dihedral    5.0     # diherdal (deg)
despmtr   camber      0.04    # max camber/chord of NACAxxxx
despmtr   thickness   0.12    # thickness/chord  of NACAxxxx
despmtr   sharpte     1       # =1 to modify thickness to close TE
despmtr   spar1       0.25    # fraction of chord for forward spar
despmtr   spar2       0.75    # fraction of chord for rearward spar
despmtr   nrib        11      # number of ribs
```

To begin the construction process for a wing, one needs to compute geometric quantities from the design parameters. This is done with statements such as:

```
set       span        sqrt(area*aspect)
set       span2       span/2
set       cbar        area/span
set       croot       2*cbar/(1+taper)
set       ctip        taper*croot
set       dxtip       span2*tand(sweep)
set       dztip       span2*tand(dihedral)
```

Now one can construct the various geometric models.

The first model to be constructed is a simple beam model, which one may use to compute simple deflections and aerodynamic twists of the wing. The very simple model in constructed of two line segments at the quarter-chord location and halfway between the upper and lower surfaces. The **OpenCSM** statements that do this are:

```
# build wing beam Body (at quarter-chord)  (fidelity==1)
ifthen fidelity eq 1
   ifthen mirror ne 1
      skbeg     croot/4  0  0
         linseg dxtip+ctip/4 +span2 +dztip
      skend
         attribute ID       !ID
         attribute Fidelity 1  # simple 1D model (aka, beam)
   else
      skbeg     dxtip+ctip/4 -span2 +dztip
         linseg croot/4  0  0
         linseg dxtip+ctip/4 +span2 +dztip
      skend
         attribute ID       !ID
         attribute Fidelity 1  # simple 1D model (aka, beam)
   endif
endif
```

Here `fidelity` is set to 1 to select the simple beam model, and `mirror` is a flag to tell if this wing consists of only the right wing (`mirror` $\neq 1$) or should be mirrored to generate both wings. The various `attribute` statements put a unique `ID` and `Fidelity` on the geometry so that it can be identified through the rest of the **CAPS** system. This model, together with the beam model of the fuselage, is shown in Figure 6(a).

The second model to be constructed is an outer mold line (OML) for the wing. The **OpenCSM** script used to build this is:

```
# build wing outer-mold line  (fidelity==4 or 15)
ifthen fidelity eq 4 or fidelity eq 15
   mark
      ifthen mirror eq 1
         udprim  naca  thickness  thickness  sharpte  sharpte  camber  camber
         rotatex  90  0  0
         scale    ctip
         rotatey  -twist 0 0
         translate dxtip -span2 +dztip
      endif

      udprim  naca  thickness  thickness  sharpte  sharpte  camber  camber
      rotatex  90  0  0
      scale    croot

      udprim  naca  thickness  thickness  sharpte  sharpte  camber  camber
      rotatex  90  0  0
      scale    ctip
      rotatey  -twist 0 0
      translate dxtip +span2 +dztip
   rule

   ifthen fidelity eq 4
      store wingOML
   endif
endif
```

The OML gets built whenever the user wants the OML directly (`fidelity` = 15) or during the process of generating the built-up-element model (`fidelity` = 4) (described below). One of **OpenCSM**'s unique capabilities is used here: the user-defined primitive (UDP). **OpenCSM** ships with several pre-built UDPs that have been found to be useful for the construction of aerospace vehicles, including various airfoil generators, super-ellipses, and a “waffle” (also described below). In the above code snippet, one can see the use of the “naca” UDP, which generates a NACA 4-digit airfoil section with a given thickness and camber; since the published definition of the NACA 4-digit series has a blunt trailing edge, an option has been added to modify the x^4 coefficient so as to generate a sharp trailing edge (this option has been used here). The actual construction of the configuration is composed of the solid body that is generated when the various airfoil sections are connected with "rule"d surfaces. Note that if `fidelity` = 4, then the resulting model is stored (into `wingOML`) for latter use, instead of being left on the stack.

The third model is the mid-surface aerodynamic model, which is made of the wing's camber sheet. The "naca" UDP has been written so that if the `thickness` = 0 the camberline is generated. This is exploited in the following:

```
# build wing mid-surface aero  (fidelity==11)
ifthen fidelity eq 11
   mark
      ifthen mirror eq 1
         ifthen camber eq 0
            skbeg     0  0  0
               linseg 1  0  0
            skend
         else
            udprim  naca  thickness  0  camber  camber
         endif
         rotatex  90  0  0
         scale    ctip
         rotatey  -twist 0 0
         translate dxtip -span2 +dztip
      endif

      ifthen camber eq 0
         skbeg     0  0  0
            linseg 1  0  0
         skend
      else
         udprim  naca  thickness  0  camber  camber
      endif
      rotatex  90  0  0
      scale    croot

      ifthen camber eq 0
         skbeg     0  0  0
            linseg 1  0  0
         skend
      else
         udprim  naca  thickness  0  camber  camber
      endif
      rotatex  90  0  0
      scale    ctip
      rotatey  -twist 0 0
      translate dxtip +span2 +dztip
   rule
      attribute ID       !ID
      attribute Fidelity 11  # 2D model (aka, mid-surface aero)
endif
```

This model, together with a simple model of the wake sheet, are shown in Figure 6(b).

The most complex of the models associated with a wing is the built-up-element (BEM) model, which consists of the wing's ribs and spars, as well as the various skin panels between these ribs and spars. The **OpenCSM** code for generating such a BEM is:

```
# build BEM model  (fidelity==4)
ifthen fidelity eq 4 and nrib gt 0
   set nwaffle nrib+ifpos(spar1,2,0)+ifpos(spar2,2,0)
   dimension waffle nwaffle 4 0

   # ribs
   patbeg  i nrib
      set  waffle[i,1]     -0.10*croot
      set  waffle[i,2]     span2*(2*i-nrib-1)/nrib
      set  waffle[i,3]     1.1*max(croot,dxtip+ctip)
      set  waffle[i,4]     waffle[i,2]
   patend

   # spars
   ifthen spar1 gt 0
      set  nrib   nrib+1
      set  waffle[nrib,1]   spar1*ctip+dxtip
      set  waffle[nrib,2]  -1.01*span2
      set  waffle[nrib,3]   spar1*croot
      set  waffle[nrib,4]   0

      set  nrib   nrib+1
      set  waffle[nrib,1]   spar1*ctip+dxtip
      set  waffle[nrib,2]   1.01*span2
      set  waffle[nrib,3]   spar1*croot
      set  waffle[nrib,4]   0
   endif
   ifthen spar2 gt 0
      set  nrib   nrib+1
      set  waffle[nrib,1]   spar2*ctip+dxtip
      set  waffle[nrib,2]  -1.01*span2
      set  waffle[nrib,3]   spar2*croot
      set  waffle[nrib,4]   0

      set  nrib   nrib+1
      set  waffle[nrib,1]   spar2*ctip+dxtip
      set  waffle[nrib,2]   1.01*span2
      set  waffle[nrib,3]   spar2*croot
      set  waffle[nrib,4]   0
   endif

   udprim  waffle    Depth     +4*croot      Segments  waffle
      attribute name $wing_structure
   translate 0         0         -2*croot
   store     wingWaffle

   # keep part of wingWaffle inside wingOML
   restore   wingWaffle
   restore   wingOML
   intersect

   # break the wingOML with the wingWaffle and extract the shell
   restore   wingOML
   restore   wingWaffle
   subtract
   extract   0

   # sew the two SheetBodies together
   union     0  0  1e-5

   # set the spacing along Edges
   udprim    createBEM filename  $BEM.dat  space  0.02   imin   3   imax   99
endif
```

As can be seen, the construction of this model uses the `wingOML` that was constructed previously. Key to the generation of this model is another UDP, named “waffle”, which takes a series of interconnected lines segments and then extrudes them for a given `Depth`, creating a waffle-like non-manifold group of Faces, as shown in Figure 5. In this case, the interconnected line segments represent the footprint of the ribs and spars, which are generated on the fly by the above code; here the `patbeg` and `patend` statements are used to denote a looping structure, called a “pattern” in the CAD world.

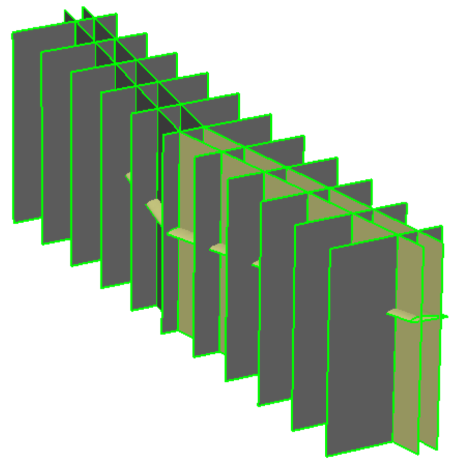


Figure 5: Result of the “waffle” UDP, together with the wing OML. These are used in the generation of the built-up-element model that can be seen in Figure 6(d).

The actual BEM is constructed by the following sequence:

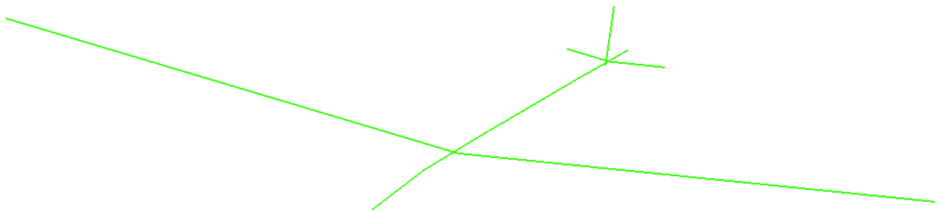
- build the OML (stored in `wingOML`)
- build the waffle (stored in `wingWaffle`)
- keep the part of waffle that is inside the OML by using a Boolean intersection
- scribe the OML at the ribs and spars by subtracting the waffle from the OML
- keep only the outer skin of the scribed body by extracting its external Faces
- finally, combine (union) the ribs/spars that are internal to the OML with the OML’s skin panels to generate the final BEM, which is shown in Figure 6(d).

Since the full glider is composed of three wing-like bodies (the wing, the horizontal tail, and the vertical tail), it makes sense to encapsulate the above into macro-like scripts, which are called user-defined components (UDCs) in **OpenCSM**. In the present example, a file named *wing.udc* contains all of the above.

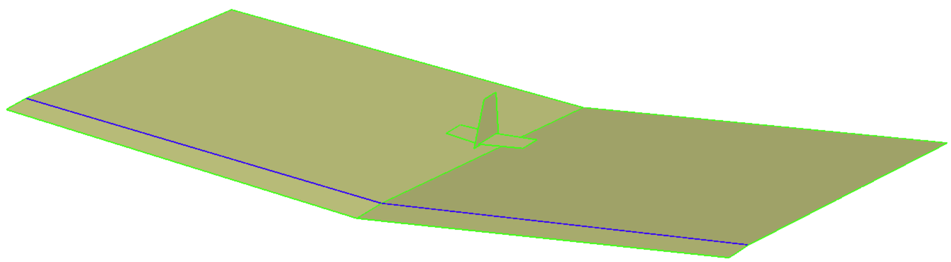
Not shown here (because of space limitations) is the ability to cut flap-like objects into any lifting surface. This is done through the use of the `flapz.udc`. The arguments to this UDC are the projection of the shape of the flap (in the z direction), the size of the gap between the flap and the wing, and the flap’s deflection. When executed, this UDC modified the body on the top of the stack so as to generate the ailerons, elevators, and rudder that are shown in Figure 4.

The fuselage is generated in a similar way; in this case, the cross-sectional shape is defined by super-ellipses that are constructed by the “supell” UDP. Like the wing, the fuselage is encapsulated into a UDC (named *fuselage.udc*) so that is can be easily reused.

The final assembly of the glider models, shown in Figure 4, is accomplished by combining the outputs of the UDCs.

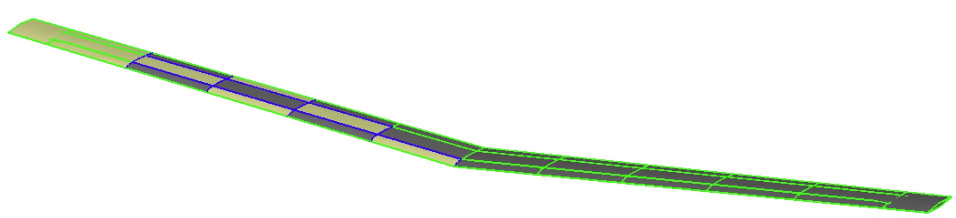


(a) Beam model – **CAPS** fidelity 1



(b) Mid-plane aerodynamics with wake sheet – **CAPS** fidelity 11

(c) Lifting surfaces cross sections – **CAPS** fidelity 13



(d) Wing structural model – **CAPS** fidelity 4

Figure 6: Example geometric fidelities of *myGlider.csm* which includes the OML as seen in Figure 4.

B. Low-Fidelity Aerodynamic Trim Optimization

A trim optimization problem is performed using an **AIM** that accesses the Athena Vortex Lattice (**AVL**) aerodynamic solver.[9] Currently the **AIM** does not expose all of **AVL**'s capabilities; however what is exposed is the ability to perform analysis at a fixed Mach number, angle of attack, sideslip angle, roll rate, pitch rate, yaw rate, and control deflection. The outputs include the static and dynamic stability derivatives about the analysis input condition.

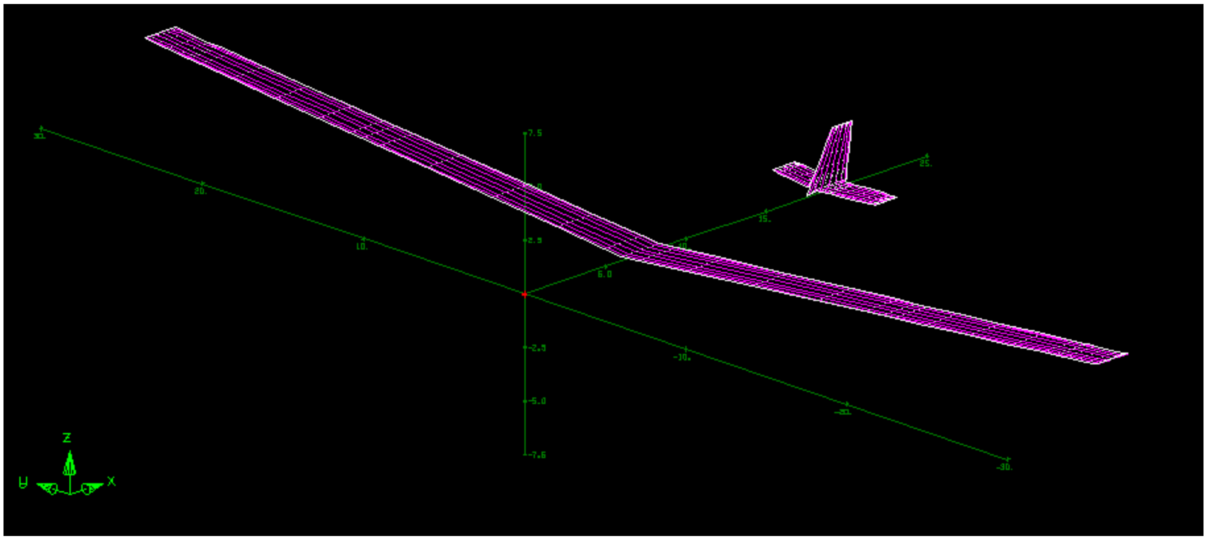


Figure 7: AVL Analysis Model

The **AVL** model shown in Figure 7 is generated from the geometry fidelity shown in Figure 6c. The **AIM** takes the attributed *myGlider.csm* file and constructs the lifting-surface meshes shown. The mesh resolution is controlled through the attributes defined on the geometry.

An example Python script using **pyCAPS** to define and execute the **AVL AIM** is presented in Figure 9. This Python script is easily modified for use as an objective function in an optimization problem, where the problem is solved using one of the available optimizers in the SciPy library. The trim optimization problem described by Equations 1 and 2 is solved in this manner. Each iteration of the optimization process passes a new set of control surface deflections and angles of attack until the optimal solution is found and the constraints are met.

$$minimize\,\{C_{D,i}\} \tag{1}$$

such that

$$C_L = C_{L,target} \text{ and } C_M = C_{M,target} \tag{2}$$

where, $C_{D,i}$ is the induced drag coefficient, C_L is the lift coefficient, and C_M is the moment coefficient.

In addition to the trim optimization problem, an untrimmed ($C_M \neq C_{M,target}$) angle of attack sweep is also performed through a series of calls to the Python script in Figure 9, where the angle of attack is updated each time. The resulting lift and drag coefficients are retained for comparison with the trimmed solutions.

To define a realistic trim vs. untrimmed condition for this example, **AVL** is used to estimate the aerodynamic center location of the glider. It was found to be located 6.956 (ft) from the nose of the vehicle. For simplicity the center of gravity assumed to be at the quarter chord location of the wing root is 6.597 (ft) from the nose. Using a reference chord of 2.388 (ft) this becomes a 15% positive static margin.

$$C_M = \frac{M_y}{q * C_{ref} * S_{ref}} \tag{3}$$

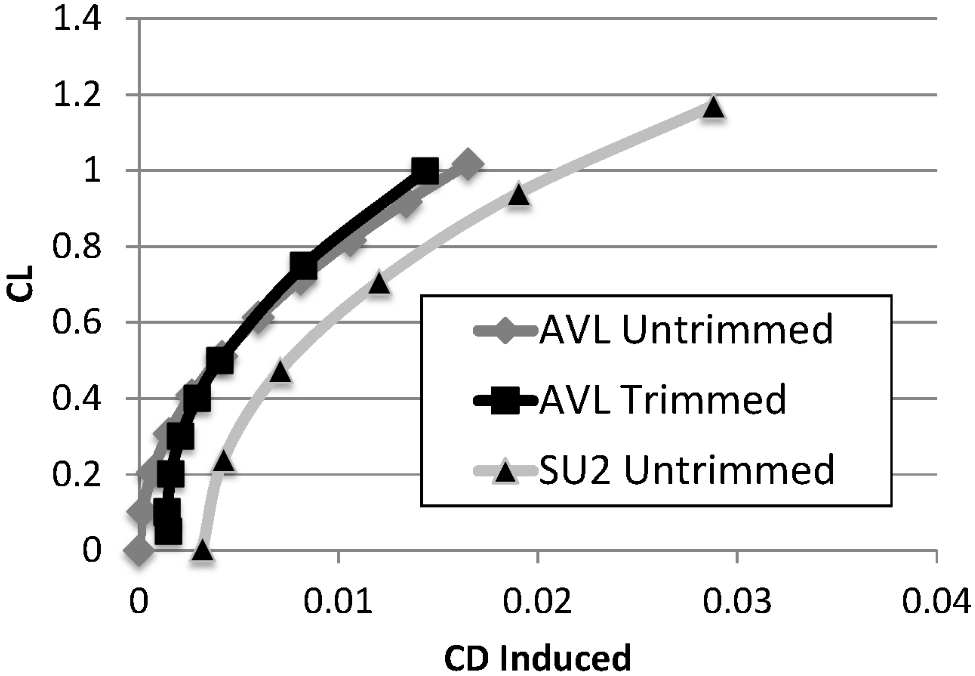


Figure 8: AVL Trimmed and Untrimmed Drag Polars

With a weight of 1000 (lbs) the nose down or negative moment caused by the positive static margin becomes -359.12 (ft-lbs). Using Equation 3, where M_y is the moment, q is the dynamic pressure, C_{ref} is the reference chord and S_{ref} is the reference span, the moment coefficient caused by the static margin is -0.185. Therefore, for a trimmed solution $C_{M,target}$ must be 0.185 in Equation 2 measured about the aerodynamic center location.

A drag polar comparison between an untrimmed and trimmed configuration over a range of target lift coefficients is shown in Figure 8. This problem is relatively easily achieved using **ESP** geometry, with **CAPS** enabled access to AVL driven by the **pyCAPS** interface. For comparison an untrimmed Euler evaluation of the drag polar using the SU2[10,11] flow solver is presented. This analysis is accomplished using the same geometry. For AVL the geometry fidelity level only produced wing cross sections. For SU2 the complete solid geometry representation is created.

C. Loosely Coupled Aeroelastic Analysis

An example that illustrates the multi-disciplinary capability of CAPS, using the same geometry outlined in Section III.A, is now presented by executing a loosely coupled static aeroelastic analysis. This example demonstrates two analysis AIMs: a structural solver and a fluid solver. In addition, the interpolation and passing of data (i.e. pressure from the fluid to the structure, and displacements from the structure to the fluid) is accomplished internally by **CAPS**.

To enable a structural or aerodynamic solver, a properly attributed geometry is required. The major attribution entity required for this task by **CAPS** is *capsGroup*. In the example a half span wing model is created. The entire OML of the wing is attributed as a single *capsGroup*. In this case it is assigned the string *Skin*:

```
attribute capsGroup $Skin
```

```
# Import modules
import os
from pyCAPS import capsProblem

# Initiate a CAPS Problem
myProblem = capsProblem(libDir = "/ESP/Root/lib/dir")

# Load ESP input file describing the attributed geometry
myProblem.loadCAPS("./csmData/myGlider.csm")

# Load the AVL AIM and define an analysis directory for execution
myProblem.loadAIM(aim = "avlAIM", analysisDir = "avl_Aviation_Test")

# Define the flight conditions for and AVL analysis point
myProblem.analysis["avlAIM"].setAnalysisVal("Mach", 0.0667)
myProblem.analysis["avlAIM"].setAnalysisVal("Alpha", 1.0)
myProblem.analysis["avlAIM"].setAnalysisVal("Beta", 0.0)
myProblem.analysis["avlAIM"].setAnalysisVal("CntrlDef", [0.0,0.0,0.0]) # Flap, Elevator, Rudder

# Create the AVL Input files required for analysis
myProblem.analysis["avlAIM"].aimPreAnalysis()

# Run AVL
print "Running AVL"
currentDirectory = os.getcwd() # Get our current working directory
os.chdir(myProblem.analysis["avlAIM"].analysisDir) # Move into test directory
os.system("avl caps < avlInput.txt > avlOutput.txt");
os.chdir(currentDirectory) # Move back to working directory

# Run AIM post-analysis to retrieve the output data
myProblem.analysis["avlAIM"].aimPostAnalysis()

# Report desired data back from the analysis
print "Cmtot  " + str(myProblem.analysis["avlAIM"].getAnalysisOutVal("Cmtot"));
print "CLtot  " + str(myProblem.analysis["avlAIM"].getAnalysisOutVal("CLtot"));
print "CDtot  " + str(myProblem.analysis["avlAIM"].getAnalysisOutVal("CDtot"));
```

Figure 9: Example **pyCAPS** script to execute the AVL AIM

The root of the wing is then assigned a different *capsGroup*. This allows access to the root only to apply boundary conditions to this location.

```
attribute capsGroup $Rib_Root
```

An outer box to represent the flow solution domain is also attributed, with the a single Face in plane with the wing root as *SymmPlane* and the remaining five sides as *Farfield*.

```
box  farfield:xmin  0  farfield:zmin  dx  dy  dz
attribute capsGroup $Farfield
...
face 2
attribute capsGroup $SymmPlane
```

Information directly relevant to a given analysis is assigned directly to the *capsGroup* attributes. For example, aerodynamic boundary conditions are directly assigned in the following **pyCAPS** script:

```
# Set boundary conditions
inviscid = {"bcType" : "Inviscid", "wallTemperature" : 1.0}
myProblem.analysis["fun3d"].setAnalysisVal("Boundary_Condition", [("Skin", inviscid),
                                                                  ("SymmPlane", "SymmetryY"),
                                                                  ("Farfield","farfield")])
```

Additionally, materials and properties for a structural solver are assigned to specific *capsGroups* in a similar manner. For example, a generic material is assigned to a shell element that is modeling the wing skin in the following **pyCAPS** script:

```
# Populate the variable mymaterial with material properties
mymaterial = { "materialType" : "isotropic",
               "youngModulus" : 1.5E9,
               "poissonRatio" : 0.33,
               "density"      : 5.5 }

# Add the material "MaterialName" to the "Material" information inside CAPS with the mymaterial properties
myProblem.analysis["mystran"].setAnalysisVal("Material", ("MaterialName", mymaterial))

# Define shell element properties in the variable skin
skin  = {"propertyType" : "Shell",
         "membraneThickness"   : 0.021, # ft - 0.25 in
         "material"            : "MaterialName",
         "bendingInertiaRatio" : 1.0,
         "shearMembraneRatio"  : 5.0/6.0}

# Assign the shell element properties skin to the "Skin" capsGroup
myProblem.analysis["mystran"].setAnalysisVal("Property", ("Skin", skin))
```

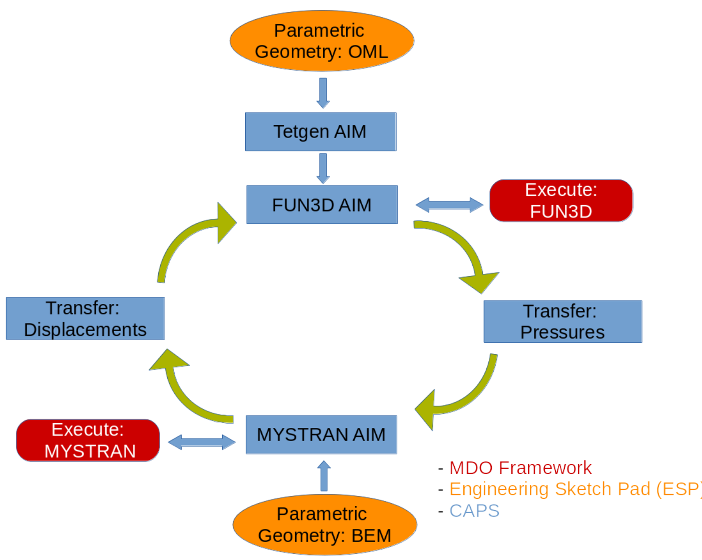
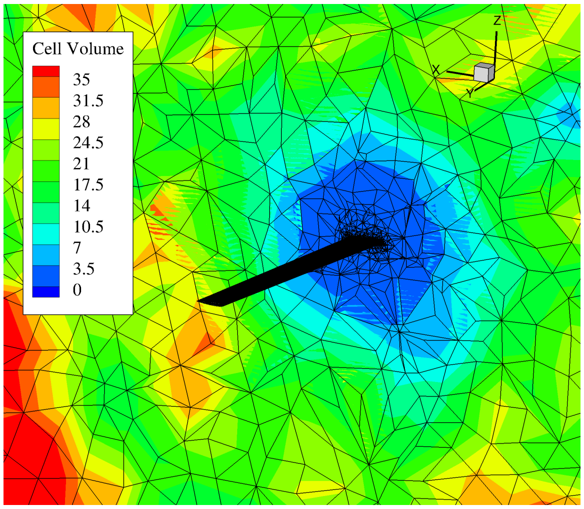


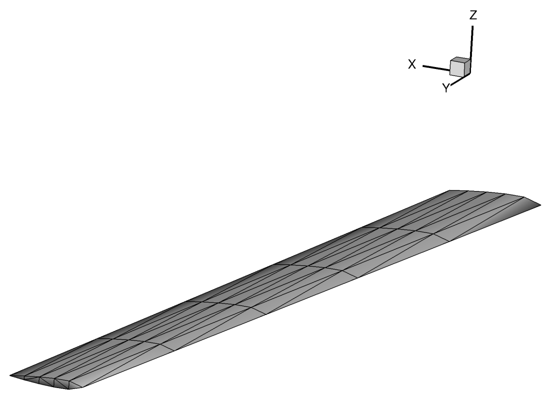
Figure 10: Aeroelastic workflow using **CAPS**.

An additional attribute on the geometry is required to enable data transfer between two disciplines. In this case the fluid and structure solvers need to be able to transfer pressures and displacements back and forth between two different surface meshes. This is enabled through the attribute *capsTransfer*. For example, recall that the entire wing surface is a single *capsGroup* assigned the name *Skin*. In the data transfer case the wing is broken into three boundaries.

```
select face 1
attribute capsTransfer $Skin_Top
select face 2
attribute capsTransfer $Skin_Bottom
select face 4
attribute capsTransfer $Skin_Tip
```



(a) Fluid mesh used in CFD analysis



(b) Structural mesh

Figure 11: Sample meshes used in loosely coupled, static aeroelastic demonstration.

Because this attribute is assigned at the geometry level, it is inherited by both the fluid and structure (or any applications) analysis meshes that are generated in the **CAPS** environment. Then when data is transferred, the disparate mesh locations are automatically mapped together.

After defining the attributes above, the general work flow for this loosely coupled aeroelastic analysis example is outlined in Figure 10. NASA Langley Research Center's unstructured, mixed element fluid solver, FUN3D[12] is used to solve for the flow field and corresponding pressures on the glider's wing from a tetrahedral volumetric mesh generated using the open-source mesh generator TetGen.[13] The open-source, general purpose finite element, structural analysis tool, MYSTRAN,[14] is used to solve for the resulting structural displacements. Both the surface mesh (143,176 triangular elements) used in the volumetric grid generation (552,024 tetrahedral) and the structural mesh (116 triangular elements) were generated using the **EGADS**[3] body tessellation, as shown in Figures 11a and 11b respectively.

The aeroelastic iterations depicted in Figure 10 were carried out in a Python environment using **py-CAPS**. Tables 3 and 4 outline the inputs into the aforementioned aerodynamic and structural **AIM**s used to generated inputs for the execution of the respective analyses. Results of the aeroelastic analysis are presented in Figure 12.

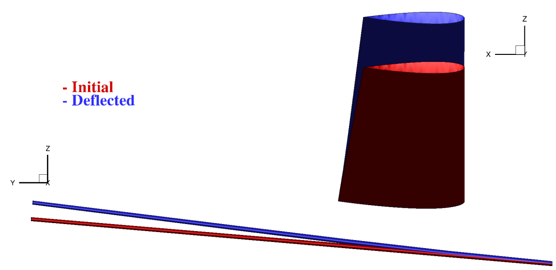
Table 3: Input parameters for fluid analysis.

Reference Variable	Value
Analysis Type	Euler
Mach number, Ma	0.3
Reference temperature, T_{ref}	491.4 (R)
Reference density, ρ_{ref}	0.07518 (slug/ft^3)
Specific heat ratio, γ	1.4
Angle of attack, α	10.0 (^{o})
Side-slip angle, β	0.0 (^{o})

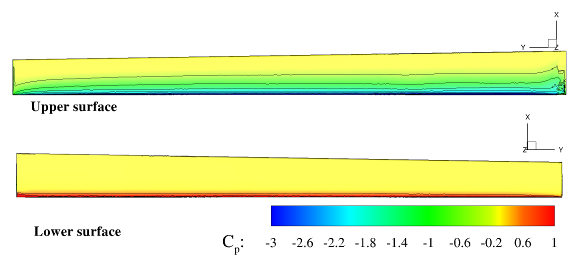
Table 4: Input parameters for structural analysis.

Property Name	Membrane Thickness (ft)	Material	Bending Inertia Ratio	Shear Membrane Ratio
Skin	0.021	Madeupium	1.0	0.833
Ribs & Spars	0.170	Unobtainium	1.0	0.833

Material Name	E (lbf/ft^2)	ν	ρ (slug/ft^3)
Madeupium	1.5 x 10^9	0.33	5.5
Unobtainium	1.0 x 10^{10}	0.33	6.0



(a) Initial and deflected wing shapes during aeroelastic iterations



(b) Pressure distribution on aeroelastically defected wing

Figure 12: Results from loosely coupled, aeroelastic analysis.

IV. Conclusions

The **CAPS** program set out with the goal of having a single source geometry capability that could provide inputs to multi-fidelity, multi-physics solvers. This capability is intended for use in conceptual design, multi-disciplinary design optimization, and high fidelity physics simulations (notably absent is CAD assemblies and three view drawings). The program outlines specific objectives to deal with legacy geometry (not covered in this paper), allow easy assembly of component models, and visual interaction with the models. In addition, the models are required to be parametric, attributable and have geometric sensitivities associated with the parameters. Finally the geometry model is to be constructed in such a way as to support multi-fidelity analysis from a single source.

This paper has demonstrated the parametric attributable geometry by introducing the overall **CAPS** infrastructure that has be built upon **ESP**. The ability to generate component models has been enabled through the application of UDP and UDC model generation techniques. These methods of model construction allow the user to tailor the component models to their specific needs instead of relying on a one-size-fits-all library.

Finally, this paper presented why multi-fidelity, attributed geometry is necessary to enable future designs. The first example demonstrated a low fidelity aerodynamic trim optimization problem that requires a very low fidelity level, but a large number of runs. Using the same geometry source the second example demonstrates a loosely coupled aeroelastic analysis where the user-intensive generation of the computational meshes and interface connections are automated through the **CAPS** infrastructure.

To enable these examples and many other analysis efforts a collection of **AIM** plugins has been created. An outline of **AIM**s in active development is provided in Table 5.

Table 5: Actively developed **AIM**s.

Surface Meshing	Volume Meshing	Aerodynamics	Structures
EGADS Tessellation[3]	TetGen[13]	FRICTION[15]	MYSTRAN[14]
AFLR4[16,17]	AFLR3[16,17]	AWAVE[18]	NASTRAN[19]
		AVL[9]	Abaqus[20]
		CART3D[21]	
		SU2[10,11]	
		FUN3D[12]	

ESP and **CAPS** are open-source and are available from `http://acdl.mit.edu/ESP`.

References

[1] Haimes, R. and Dannenhoffer, J., "The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry," *21st AIAA Computational Fluid Dynamics Conference*, No. 2013-3073, American Institute of Aeronautics and Astronautics, Jun. 2013.

[2] Dannenhoffer, J., "OpenCSM: An Open-Source Constructive Solid Modeler for MDAO," *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, No. 2013-0701, American Institute of Aeronautics and Astronautics, Jan. 2013.

[3] Haimes, R. and Drela, M., "On The Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design," *50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, No. 2012-0683, American Institute of Aeronautics and Astronautics, Jan. 2012.

[4] Bhagat, N. D. and Alyanak, E. J., "Computational Geometry for Multi-fidelity and Multi-disciplinary Analysis and Optimization," *52nd Aerospace Sciences Meeting*, No. 2014-0188, American Institute of Aeronautics and Astronautics, Jan. 2014.

[5] Kolonay, R. M., "A physics-based distributed collaborative design process for military aerospace vehicle development and technology assessment," *International Journal of Agile Systems and Management*, Vol. 7, No. 3-4, 2014, pp. 242–260.

[6] Phoenix Integration, http://www.phoenix-int.com/, Accessed: May 2016.

[7] Heath, C. M. and Gray, J. S., "OpenMDAO: Framework for Flexible Multidisciplinary Design, Analysis and Optimization Methods," *8th AIAA Multidisciplinary Design Optimization Specialist Conference (MDO)*, Honolulu, Hawaii, 2012, pp. 1–13.

[8] Dannenhoffer, J. and Haimes, R., "Conservative Fitting for Multi-Disciplinary Analysis," *52nd Aerospace Sciences Meeting*, No. 2014-0294, American Institute of Aeronautics and Astronautics, Jan. 2014.

[9] Drela, M. and Youngren, H., *AVL (Athena Vortex Lattice) 3.30 User Primer*, Aug. 2010, Available from http://web.mit.edu/drela/Public/web/avl/.

[10] Palacios, F., Colonno, M. R., Aranake, A. C., Campos, A., Copeland, S. R., Economon, T. D., Lonkar, A. K., Lukaczyk, T. W., Taylor, T. W. R., and Alonso, J. J., "Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design," *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, No. 2013-0287, American Institute of Aeronautics and Astronautics, Jan. 2013.

[11] Palacios, F., Economon, T. D., Aranake, A. C., Copeland, S. R., Lonkar, A. K., Lukaczyk, T. W., Manosalvas, D. E., Naik, K. R., Padron, A. S., Tracey, B., Variyar, A., and Alonso, J. J., "Stanford University Unstructured (SU2): Open-source analysis and design technology for turbulent flows," *52nd Aerospace Sciences Meeting*, No. 2014-0243, American Institute of Aeronautics and Astronautics, Jan. 2014.

[12] Biedron, R. T., Carlson, J.-R., Derlaga, J. M., Gnoffo, P. A., Hammond, D. P., Jones, W. T., Kleb, B., Lee-Rausch, E. M., Nielsen, E. J., Park, M. A., Rumsey, C. L., Thomas, J. L., and Wood, W. A., *FUN3D Manual: 12.7*, May 2015.

[13] Si, H., "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator," *ACM Trans. Math. Softw.*, Vol. 41, No. 2, Feb. 2015, pp. 11:1–11:36.

[14] Case, W., *MYSTRAN General Purpose Finite Element Structural Analysis Computer Program (Linux Version 6.35) [Software Manual]*, Nov. 2011, Available from http://www.MYSTRAN.com.

[15] Mason, W. H., *FRICTION - Skin Friction and Form Drag Program*, Jan. 2006, Available from http://www.dept.aoe.vt.edu/ mason/Mason_f/MRsoft.html.

[16] Marcum, D. L. and Weatherill, N. P., "Unstructured grid generation using iterative point insertion and local reconnection," *AIAA Journal*, Vol. 33, No. 9, Sep. 1995, pp. 1619–1625.

[17] Marcum, D. L., "Unstructured grid generation using automatic point insertion and local reconnection," *The Handbook of Grid Generation*, 1998, pp. 18–1.

[18] McCullers, L. A., *AWAVE: User's Guide for the Revised Wave Drag Analysis Program*, Apr. 1992.

[19] Reymond, M. and Miller, M., *MSC NASTRAN Quick Reference Guide Version 68*, 1996.

[20] Dassault Systmes, http://www.3ds.com/products-services/simulia/products/abaqus/, Accessed: May 2016.

[21] Aftomis, M., Berger, M., and Adomavicius, G., "A Parallel Multilevel Method for Adaptively Refined Cartesian Grids with Embedded Boundaries," No. AIAA-2008-0808, American Institute of Aeronautics and Astronautics, Reno, NV, Jan. 2008.