AIAA SciTech Forum
7-11 January 2019, San Diego, California
AIAA Scitech 2019 Forum

10.2514/6.2019-2226

pyCAPS: A Python Interface to the Computational Aircraft Prototype Syntheses

Ryan Durscher[1]
Air Force Research Laboratory, WPAFB, OH, 45433, USA

Dennis Reedy[2]
Asarian Technologies LLC, Manassas VA, 20111 USA

The Computational Aircraft Prototype Syntheses (CAPS) enables vehicle designers to create multi-disciplinary, multi-fidelity analysis models from a single, parametric geometric source. These capabilities are natively accessed programmatically through an interface written in the C programing language, which can present a significant hurdle for initial users. To lower entry barriers, enable scriptable problem formulation, and improve overall flexibility, a Python-based interface module, pyCAPS, was created. The following paper provides an overview of pyCAPS's capabilities and interface, while also providing simple examples of its use. Furthermore, details and discussions are provided on how pyCAPS can be used to integrate CAPS capabilities into external Multi-Disciplinary Analysis and Optimization (MDAO) frameworks such as OpenMDAO, modeFRONTIER, and MSTC Engineering.

I. Nomenclature

AIM	Analysis Interface Module
API	Application Programing Interface
CAPS	Computational Aircraft Prototype Syntheses
DOE	Design of Experiments
MDAO	Multi-Disciplinary Analysis and Optimization

II. Introduction

The Computational Aircraft Prototype Syntheses, typically referred to as CAPS, has the ability to uniquely support multi-disciplinary, multi-fidelity analysis from a single geometric source for the design of aerospace vehicles by combining geometry, meshing, and analyses model generation into a unified context [1]. Native programmatic access to CAPS is enabled through an API written in the C programming language. This high-level API is 'object-based' which utilizes blind pointers that the API functions internalize and parse to perform the given procedural-based tasks based on the set type for the object. Being written in C, the natural entry point into CAPS would be to setup one's problem within a "main" function, link against the CAPS library, and compile the problem into an executable. From a user's perspective this approach has two immediate disadvantages: 1) inflexibility, to make changes one must re-compile and 2) requires a moderate knowledge of the C programming language. To lower these entry barriers, a Python-based interface, pyCAPS, was developed. The use of Python provides an easy to learn, readable (compared to low-level programming languages), and flexible entry point for CAPS. pyCAPS logically ties together features of the CAPS API to enable rapid generation of problems in the Python environment. Additional functionality not directly available through the CAPS API has also been incorporated into pyCAPS. To date, among the existing CAPS userbase, pyCAPS has become the primary entry point to the CAPS infrastructure.

This paper aims to provide users an overview of pyCAPS's capabilities, show simple examples of its use, and demonstrate and discuss how it can be used to integrate CAPS into external MDAO frameworks such as

[1]Aerospace Research Engineer, Multidisciplinary Science Technology Center, AIAA Member.
[2]Software Research Engineer, Multidisciplinary Science Technology Center, AIAA Member.

OpenMDAO [2], ModeFRONTIER [3], and MSTC Engineering. It is assumed readers have a general familiarity with CAPS's terminology and workflow as outlined by Alyanak et al. [1].

III. Overview for CAPS's Python Interface

pyCAPS is a light weight, Python extension module to interact with the CAPS routines in the Python environment. Written in Cython [4], pyCAPS natively handles all type conversions/casting, while logically grouping CAPS function calls together to simplify a user's experience. The use of Cython inherently implies support for Python versions 2.6, 2.7, and +3.3. Furthermore, pyCAPS is routinely tested on Linux, Mac OSX, and Windows operating systems to verify and ensure architecture independence. pyCAPS makes use of Python's built-in module *unittest* for its unit testing framework with over fifty unit tests routinely executed. These unit tests not only validate proper functionality within pyCAPS, but within CAPS as well. Documentation for the pyCAPS API is embedded within its source code and is generated automatically using Doxygen [5]. For the majority of the classes' methods, source code examples are also provided in the API documentation. In the following sections the overarching API for pyCAPS is discussed, key differences between the Python module and the C API are shown, and select add-ons available only within pyCAPS are highlighted. It should be noted that not all features of the pyCAPS module will be discussed, for which the user is referred to the API documentation.

A. Primary pyCAPS classes

While CAPS's C API is 'object-based', pyCAPS, being Python driven, makes use of object-oriented programing techniques. As such pyCAPS's API is laid out slightly different than its C counterpart that it is based off of (such as the use of classes), however at its core, its use is synonymous with the C API. The diagram in Figure 1 presents a hierarchical outline of a *capsProblem*, the main driving class of the pyCAPS API, and how other pyCAPS classes fit within the problem. It should be noted that for brevity and clarity within the figure, a limited subset of the classes' functions and attributes are shown; readers should refer to pyCAPS's API documentation for a complete list. Within the graphical layout, six class objects are present which are described in greater detail in the following:

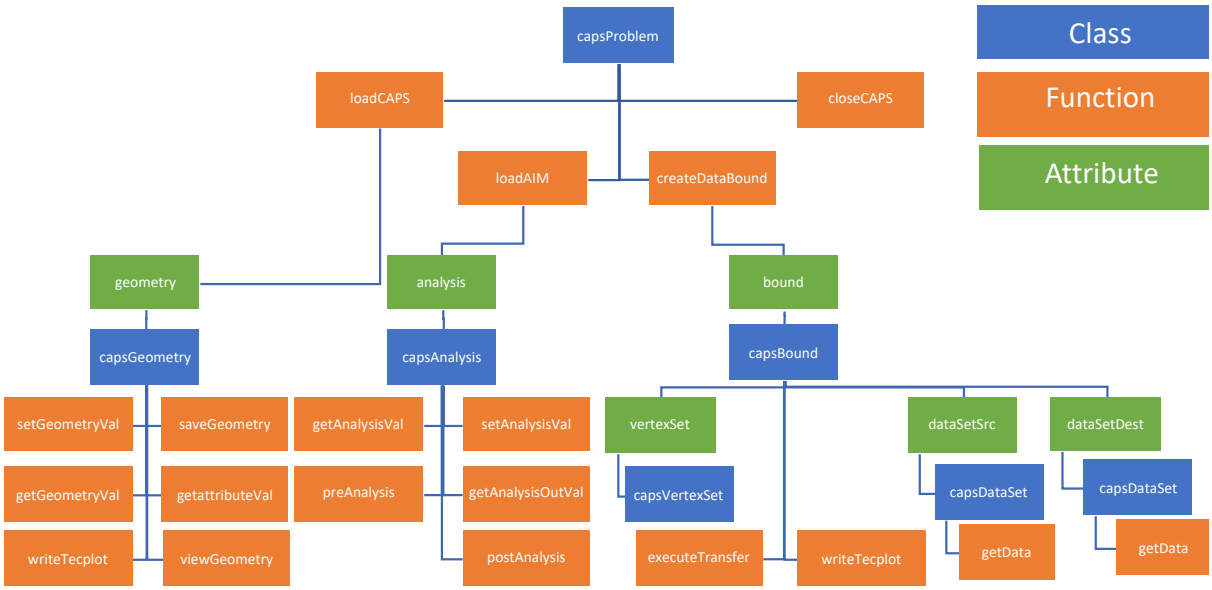

Figure 1. Hierarchy of a *capsProblem* in the pyCAPS public API.

1. *capsProblem* - A *capsProblem* is the top-level class as shown in Figure 1, which contains and is used to describe the entirety of the problem or mission of interest. Conceptually it is similar to the *Problem* object [6] in CAPS's C API. It encompasses a single set of interrelated geometric models (*capsGeometry*), analyses to be executed (*capsAnalysis*), and connectivity and data (*capsBound*).

2. *capsAnalysis* – The *capsAnalysis* class is representative of a CAPS *Analysis* object [6], which corresponds to a specific analysis tool/code. Inputs and outputs to the analysis can be set and retrieved through functions within the class. Any number of instances of the class are kept within the *analysis* attribute of the *capsProblem* class. The *analysis* attribute corresponds to a Python dictionary from which instances of a given analysis are stored as the values in the dictionary's key:value pairs.
3. *capsGeometry* - Functionality and use found in the *capsGeometry* class is wrapped into the *Problem* object [6] within the CAPS C API. Here aspects dealing solely with the geometry are pulled out and packed into the *capsGeometry* class to increase clarity and enhance usability; though admittedly, the majority of the functions within the *capsProblem* class are dependent on the *capsGeometry* class having been initiated. Only a single *capsGeometry* class can be instantiated within a problem, with the instantiation being stored in the *geometry* attribute of the *capsProblem* class. The class provides functionality to set and/or retrieve geometric design and local variables, among other useful utilities.
4. *capsBound* - The *capsBound* class is representative of a CAPS *Bound* object [6] which corresponds to a logical grouping of geometric features such as edges or faces. A *Bound* is primarily used to transfer data between two different bodies/domains, for example the aerodynamics pressure on the outer model line of a wing to the internal structural model. Instances of the class are kept within the *bound* attribute of the *capsProblem* class. The *bound* attribute corresponds to a Python dictionary from which instances of a given bound are stored as the values in the dictionary's key:value pairs.
5. *capsVertexSet* - The *capsVertexSet* class is representative of a CAPS *VertexSet* object [6] which corresponds to discrete locations in which data for a given *capsAnalysis* instance is defined. The CAPS C API supports "unconnected" vertex sets, that is the discretization is not a tied to a given instance of a *capsAnalysis* class, however these are not supported currently within pyCAPS. Instances of the *capsVertexSet* are stored as values in a Python dictionary within the *vertexSet* attribute of the initiating *capsBound*. The names of the *capsAnalysis* instances used to create the bound are used as the dictionary's keys.
6. *capsDataSet* - The *capsDataSet* class is representative of a CAPS *DataSet* object [6] which corresponds to engineering data (i.e. pressure, discplacements, etc.) defined by the *capsVertexSet*. Instances of the *capsDateSet* class are stored in a Python dictionary by variable names declared when initiating the *capsBound*. The dictionaries are stored under the *dataSetSrc* and *dataSetDest* attributes of the *capsBound* class, where *dataSetSrc* corresponds to "source" (or only dataset if not utilizing the *capsBound* to transfer data) and *dataSetDest* corresponds to the "destination" if executing a data transfer.

B. Error handling

Errors in CAPS's C API are handled through an integer return on functions that correspond to marco definitions defined through the *#define* directive. Currently CAPS has thirty-eight predefined error codes, such as -332 = CAPS_IOERR or -317 = CAPS_BADNAME. Within pyCAPS, errors are handled through an exception class, *capsError*, which inherits Python's *Exception* class. Along with CAPS's error codes, error codes for EGADS [7] and OpenCSM [8] are also mapped within the class. All of CAPS's integer function returns are captured and a *capsError* exception is raised if an error code is detected. If a user is trying to capture or check for specific error codes, as opposed to just a *capsError*, the exception can be further dissected using the class's attributes *errorCode* and *errorName*. As the names imply, these attributes correspond to the integer value for the error code (e.g. *errorCode* = -332) and the error's name (e.g. *errorName* = "CAPS_IOERR"), respectively.

C. capsValue class

A core object within CAPS's C API is the *Value* object [6] which acts as the fundamental, general data container for the majority of inputs and outputs to the API functions. Along with storing a data value (e.g. a character string, a double array, etc.) the generic object's structure allows for the assigning of meta-data such as: data type (e.g. Double, Integer, etc.), shape of the data (number of rows and columns), units of the data, and allowable upper and lower bounds of the data. CAPS's C API provides numerous functions to unpack this information from a *Value* object when working with it; information such as data types and number of rows and columns are vital to know when working in a structured programing language such as C. However, this same information in mostly irrelevant in the Python environment as variable declarations and manipulations are much more flexible. As such the decision was made to use Python objects as the default "value" type when setting or retrieving data values through the pyCAPS API, with all the necessary type conversions handled internally and automatically. It should be noted that for the majority of the functions involved with setting or getting data values, units may also be optionally provided or retrieved.

While Python objects are the default data types used in pyCAPS, pyCAPS does allow for the creation and return of the equivalent of a *Value* object [6] through the *capsValue* class. The *capsValue* class has attributes which store meta-data such as the objects value (e.g. 50.0, "string", etc.), units assigned, and data type (e.g. Double, Integer, etc.) similar to what can be retrieved for a *Value* object.

D. Core differences between pyCAPS and CAPS

While minimal, there are few notable differences between the pyCAPS API and CAPS's C API. These differences are outlined as follows:

1. Manipulating the "owner" information for CAPS objects isn't currently supported. See CAPS's C API [6] documentation for more information.
2. CAPS does not natively support an array of string values, an array of strings is viewed by CAPS as a single concatenated string; pyCAPS does, however. If a list of strings is provided this list is concatenated, separated by a ';' and provided to CAPS as a single string. The number of rows and columns on the *Value* object [6] are correctly set to match the original list for the value. If a string is received from CAPS by pyCAPS and the rows and columns are set correctly it will be unpacked correctly considering entries are separated by a ';'. If the rows and columns aren't set correctly and the string contains a ';', the data will likely be unpacked incorrectly or raise an indexing error.

E. Select add-ons available through pyCAPS

Additional add-ons available through the pyCAPS API, which are not explicitly a part of the C API are described below.

1. *Unit conversions* – Units play an important role in data values entered into and coming out of CAPS, with unit conversions happening automatically (when needed) to reduce this error prone burden on the user. Within CAPS, unit conversions are handled using the UDUNITS-2 package [9]. Now while there are numerous Python packages dedicated to unit conversions, to allow pyCAPS users a consistent syntax in specify units, a wrapping function, *capsConvert()*, to the UDUNITS-2 package is provided with pyCAPS. The *capsConvert()* function accepts a value (either a single or list of float(s)/double(s) and integer(s)), a string indicating the value's current units, and a string indicating the requested units. A simple use case is as follows:

```
from pyCAPS import capsConvert
convert = capsConvert(2, "in", "m") # Convert "inches" to "meters"
print("Value in meters =", convert) # Result: Value in meters  = 0.0508
```

2. *Dendrogram visualization* – Within the *capsProblem*, *capsGeometry*, *capsAnalysis*, and *capsBound* classes resides a function *createTree()* which can be used to generate an interactive dendrogram of the current state of the object. Written to an HTML file, and viewed with a web browser, the dendrogram utilizes the open-source JavaScript library, *D3 (Data Driven Documents)* [10], to visualize the attributes and information in the class. This library is freely available from https://d3js.org/ and is dynamically loaded within the HTML file. If running on a machine without internet access, a (miniaturized) copy of the library in written out alongside the generated HTML file by setting a given keyword during the function invocation. These dendrograms are interactive in which the tree's branches can be expanded and collapsed to explore the object(s). Users of pyCAPS to date, have found the dendrogram visualization to be useful for (1) record keeping and (2) debugging. First, the dendrogram created on an object represents an instantaneous snapshot of data contained within the object, which can be used to bookkeep user settings as a script progresses. Furthermore, the data used to generate the dendrogram may be output as a JSON (JavaScript Object Notation) string in a separate file for additional, independent processing. The second use reported by pyCAPS users of the dendrogram feature, is its ability to aid in debugging by providing a quick, condensed visualization for lengthy or complex scripts. Along with trouble shooting user inputs such as the values set for a particular analysis (e.g. the Mach number set for an aerodynamic solver), attributes set on the geometry, a cornerstone of CAPS capabilities [1], can also be visualized with the dendrogram. Dendrograms can be created to provide detailed breakdowns at the body, face, edge and node level to display the attributes and their values for all geometric models loaded in the problem. A representative example dendrogram is provided in Figure 2 for a given instance of a *capsProblem* class. The subclass instances stored in the attributes of problem (*geometry*, *analysis*, *bound* –

Figure 1) are presented and expanded further in Appendix I. In this example, under the *Geometry* branch, information such as the design parameters can be viewed for the given geometry file loaded into the problem. Similarly, the *Analysis* branch can be expanded out to visualize information such as the input values set for a given analysis instance.

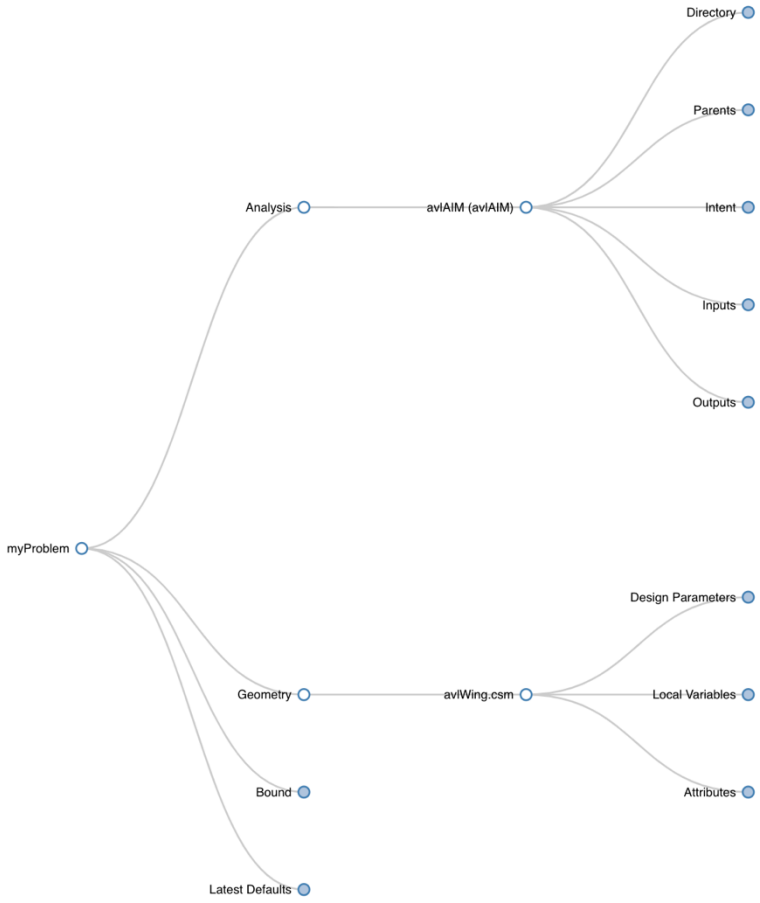

Figure 2. Representative dendrogram for an instance of a *capsProblem*. An expanded version of the dendrogram is provided in Appendix I.

3. *Saving and viewing geometry* – pyCAPS users have found when updating or changing a parametrized geometric model through CAPS during an optimization or design of experiments study it is often desirable to make a static copy of the modified geometry at each iteration for later reference or use. Using the method *saveGeometry()* within the *capsGeometry* class, users can save a static copy of the current geometry to a file. Various file formats (i.e. *.egads, *.iges, *.stp, and *.brep) are supported. Similarly, it has been found useful to have a picture of the current geometry automatically generated as a design is evolving. The ability to automatically create images, which can be saved or viewed or both, is enabled through the *viewGeometry()* function within the *capsGeometry* class. This capability is enabled through the *matplotlib* [11] Python module which must be installed separately on the system (an *ImportError* is raised otherwise).

IV. Getting started with pyCAPS

The following provides an overview of pyCAPS's use, with the intention being to emphasis and focus on basic, core functionality; as such not at all functions will be discussed. Readers are encouraged to individually explore each classes documentation within the pyCAPS API for a complete list of options, attributes, and methods available.

A. Setting up the problem

The *capsProblem* class is the front end of pyCAPS. All other classes are intended to be initiated through the problem class as outlined in Figure 1. The following code details the primary function calls and uses when creating and setting up a new problem.

1. *Initialization and termination*

The first step to create a new *capsProblem* is to import the pyCAPS module; on Linux and OSx this is the pyCAPS.so file, while on Windows it is the pyCAPS.pyd file.

```
import pyCAPS
```

After the module is loaded, a new *capsProblem* class object should be instantiated. Note that multiple problems may be simultaneously loaded and exist in a single script as CAPS was designed to be thread safe allowing for multi-threading.

```
myProblem = pyCAPS.capsProblem()
```

When using CAPS's C API it is necessary to "close" the problem, after all desired operations are finished, through a dedicated function call to ensure all memory allocated by CAPS is cleaned up properly. A synonymous function is included as part of the *capsProblem* class (as shown below), however its invocation is optional (though recommend) as it is automatically invoked during garbage collection on the *capsProblem* object. An example explicit function call is as follows:

```
myProblem.closeCAPS()
```

2. *Loading the geometry*

A geometry file is loaded into the problem using the *loadCAPS()* function. A previously saved CAPS problem (*.caps) file, a specified OpenCSM file (*.csm), or a static EGADS geometry file (*.egads) may be used. In the example below an OpenCSM file, "inputGeom.csm", is loaded into our created problem from above. The project name "basicTest" may be optionally set here; if no argument is provided, the CAPS file provided is used as the project name. A reference to the newly created *capsGeometry* class is stored and accessed through the *geometry* attribute (e.g. myProblem.geometry), which is optional returned by the method as well.

```
myGeometry = myProblem.loadCAPS("inputGeom.csm", "basicTest")
```

3. *Loading an analysis*

Interfaces to various analysis tools, denoted as Analysis Interface Modules (AIMs) in CAPS's vernacular, are loaded into the problem using the *loadAIM()* function in the *capsProblem* class. In the code sample below, the "exampleAIM" is loaded into the problem with a specified working director.

```
myAnalysis = myProblem.loadAIM(aim = "exampleAIM",
                               analysisDir = "./work")
```

The instance of the loaded AIM specified in the above snippet will be referenced in the problem's *analysis* attribute (which is a Python dictionary – Section III.A.2) as "exampleAIM" and can be retrieved at any point such as:

```
myAnalysis = myProblem.analysis["exampleAIM"]
```

B. Working with geometry

Once the geometry is loaded various functions are provided to interact with it within the *capsGeometry* class. The following sections highlight a few of the more common ones.

1. *Setting and getting design parameters*

Geometric design parameters may be set using the *setGeometryVal()* function, while the current value of the parameter may be retrieved using *getGeometryVal()*. In the following example the current value for the parameter "sweep" is first obtained. The value is then reset and increased by 5.0.

```
value = myGeometry.getGeometryVal("sweep")

myGeometry.setGeometryVal("sweep", value + 5.0)
```

2. *Viewing geometry*

As described in Section III.E.3 it is often desirable to quickly view and/or automatically save images for the updated geometry during a design of experiments or optimization study. This is achieved using the *viewGeometry()* method in *capsGeometry* class. A representative example of the function is as follows,

```
myGeometry.viewGeometry(filename = "GeomViewDemo_NewSweep",
                        title="DESPMTR: lesweep = " + str(value),
                        showImage = True,
                        combineBodies = True,
                        viewType = "fourview")
```

Upon execution of the above code, an image of the current geometry is displayed on the screen (showImage = True) as shown if Figure 3, where all the bodies are combined (combineBodies = True) into a single image with four different viewpoints (viewType = "fourview"). Since a filename is also provided, the image displayed on the screen is also saved.

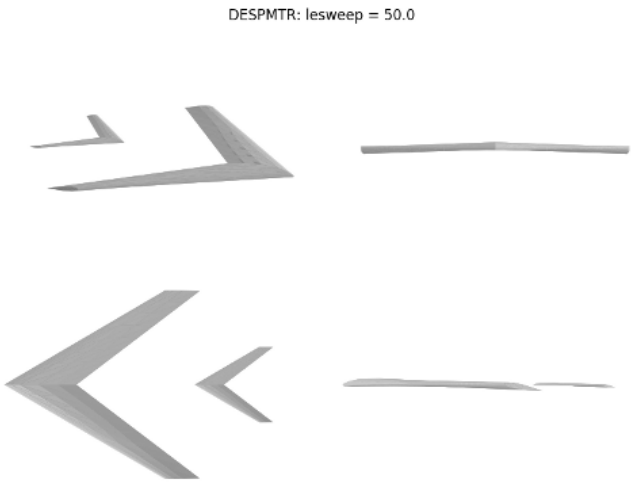

Figure 3. Representative result from an invocation of the *viewGeometry()* function.

C. Working with an analysis

Similar to the geometry, once an analysis or AIM has been loaded, various functions are provided to interact with it within the *capsAnalysis* class. A few examples of some of the more commonly used functionality are outlined below.

1. *Setting and getting analysis inputs and outputs*

Analysis inputs may be set using the *setAnalysisVal()* function, while the current value of the input may be retrieved using *getAnalysisVal()*. In the following example the current value for the input "Mach" is first obtained and is then reset.

```
value = myAnalysis.getAnalysisVal("Mach")
```

```
myAnalysis.setAnalysisVal("Mach", value + 0.2)
```

Similar to *getAnalysisVal()*, the *getAnalysisOutVal()* function returns analysis output variables.

2. *Analysis pre- and post- analysis*

For a given analysis, the CAPS pre- and post- analysis [6] functions are executed in pyCAPS using the *preAnalysis()* and *postAnalysis()* functions.

```
myAnalysis.preAnalysis()

myAnalysis.postAnalysis()
```

V. Interfacing with MDAO Frameworks

The flexibility provided within pyCAPS allows it to be a natural entry point into the CAPS infrastructure. This flexibility also makes it an easy means to integrate CAPS into existing multi-disciplinary analysis and optimization frameworks. The following sections describe some of the frameworks pyCAPS has been used with and how the interaction was achieved.

A. OpenMDAO

The opensource MDAO framework, OpenMDAO [2], can be directly interfaced and linked to CAPS through pyCAPS. This connection is achieved through the *capsAnalysis* method, *createOpenMDAOComponent*. Upon being called, this function populates the public, class attribute *openMDAOComponent* (within the *capsAnalysis* object) with a reference to an OpenMDAO "component" object that can be used within its internal framework. This component provides OpenMDAO a direct connection to CAPS, for which its design of experiment or optimization drivers can change or evaluate analysis interface input and output variables or geometric design parameters. As such all of the analysis interfaces, AIMs, currently available for CAPS may be used within OpenMDAO. For example, Pankonien et al. [12] utilized OpenMDAO's DOE driver coupled to a parametric geometry model of a trailing edge control surface for a design space study using CAPS's Nastran [13] analysis interface. A representative, simplified optimization script is provided in the Appendix II, in which a simple lift optimization, subject to a moment constraint is executed by varying the angle of attack, airfoil thickness and area using the vortex lattice code, AVL [14] as the analysis tool. It is important to note that this functionality is currently tied to version 1.7.3 of OpenMDAO, use of version 2.x will result in an import error.

B. modeFRONTIER

Esteco's commercial MDAO framework, modeFRONTIER [3], can also interface CAPS through the use of pyCAPS. Using modeFRONTIER's scripting node for Python, CAPS problems can be used as a main project or a subproject. An example of this is shown in Figure 4 where two CAPS problems utilizing the analysis interfaces for TSFoil [15] and XFoil [16] are setup to implement a DOE sweep over a range of Mach numbers (analysis input) and airfoil cambers (geometry input). Design variables (Mach number and airfoil camber) are linked automatically within modeFrontier and propagated through the scripting node into CAPS.

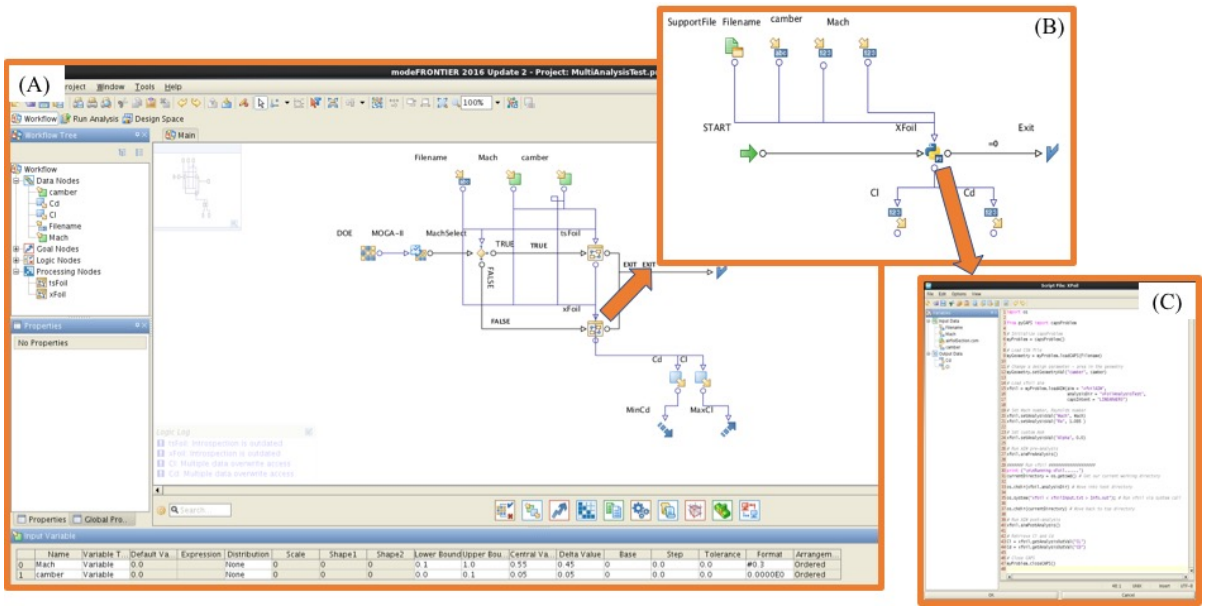

Figure 4: Example implementation of pyCAPS within a modeFRONTIER project: A) Overall project, B) subproject using modeFRONTIER's Python component, and C) Python script utilizing pyCAPS.

C. MSTC Engineering

MSTC Engineering is a MDAO framework developed by the Air Force Research Lab's Multidisciplinary Science and Technology Center (MSTC). The framework is used as a testbed to investigate distributed, service-oriented computing technology to enable modeling for multi-disciplinary engineering systems. At the core of MSTC Engineering is the capability to distribute the individual (or coupled) analyses through the network dynamically. This network centric approach allows for easy problem scaling and is achieved through a "grid" architecture which provides continuous operation allowing the distributed system to operate without interruption during the addition or removal of compute resources, and ultimately providing self-healing of the grid through dynamic and adaptive processing capabilities. Compute resources within the grid are virtualized allowing grid components to run on any machine that supports the Java Virtual Machine. Additional details on the capabilities and architectural layout of MSTC Engineering will be published in the future.

Incorporating the CAPS capabilities into the distributed MSTC Engineering grid allows MSTC Engineering providers access to CAPS, providing a single geometric source for the design of aerospace vehicles. This merger is currently enabled through pyCAPS to make use of its flexible problem formation ability. In order to bridge the runtime divide of the pyCAPS Python accessible requirements to clients running in the MSTC Engineering grid, a Python server was developed that takes inbound JSON requests, decodes them, and translates them to pyCAPS invocations. Results are encoded into JSON replies and returned to the invoking client[3]. The Python server runs in a multi-threaded environment, creating handlers for geometry and AIM creation. Once a geometry is loaded it stays available for future requests to create analyses from, or to change geometry design parameters. This results in a stateful, networked geometry service, that multiple requesting clients can use to perform multi-disciplinary engineering analyses through the network.

The pyCAPS Python server can be accessed by a client in a variety of ways:

1. It can be launched as a "tethered" process, available to the client directly.
2. It can be started as a shared network service, and dynamically discovered, and used remotely.

[3]This approach was chosen over using JNA (or JNI), which was shown to be unstable across different architectures. Additionally, it provides pyCAPS access to heterogeneous clients, albeit the capability to encode and decode JSON requests exists.

The client framework has been developed using a combination of Java and Groovy. The API follows the semantics of the pyCAPS API. The code which follows has been written using Groovy.

1. *Getting access to the pyCAPS server*

The client first creates an instance of the PyCAPSManager (to discover the pyCAPS network service, simply remove the *.launch()* method):

```
PyCAPSManager pyCAPSManager = new PyCAPSManager().launch()
PyCAPS pyCAPS = pyCAPSManager.getPyCAPS()
```

2. *Creating a geometry*

When working over a network, the ability to load geometry files remotely needs to be addressed. This is addressed within the pyCAPS server as the following example demonstrates how to declare a geometry file using a http URL and have the pyCAPS server load the file.

```
String geometryURL = "http://${address}:${port}/ feaAGARD445.csm"
Geometry myGeometry = pyCAPS.getOrCreateGeometry(geometryURL)
```

3. *Creating an analysis*

Analyses are created using the Geometry object. The Geometry objects has references to the pyCAPS server that created it and also contains a unique identifier for its Geometry Handler

```
CAPSAnalysis myAnalysis = myGeometry.createAnalysis( "exampleAIM")
```

Using the created analysis, one can configure analysis values using semantics similar to those found in pyCAPS:

```
def Mach = 0.25
myAnalysis.setAnalysisVal("Mach", Mach)
```

Furthermore, one can declare what outputs are desired,

```
myAnalysis.addOutputKey("Cd")
```

4. *Running the analysis*

There are two ways to run the configured analysis. First, the analysis can be directly submitted to the pyCAPS service one has been working with. For example, create an AnalysisDispatcher, dispatch it and get the results:

```
AnalysisDispatcher myAnalysisDispatcher = new AnalysisDispatcher(myAnalysis)
Map result = myAnalysisDispatcher.dispatch()
```

Second the analysis can be submitted to the MSTC Engineering Grid. This is typically the approach taken when one is solving a larger problem built up of analysis linked together. Using this approach, the analysis needs to be added to a Workflow object. The Workflow object then creates a MSTC Engineering Model which is submitted to the MSTC Engineering grid for processing:

```
Workflow myWorkflow = new Workflow("myExample")
myWorkflow.addAnalysis(myAnalysis)
ResponseContext response = Model.runModel(myWorkflow.createModel())
```

Additional details regarding MSTC Engineering and its use of CAPS/pyCAPS will be outlined in greater detail in follow on work.

VI.Conclusions and Future Work

While the Computational Aircraft Prototype Syntheses, CAPS, program acts as an enabler for multi-disciplinary, multi-fidelity analysis generation from a single, parametric geometric source, its entry-point, a C API, may prove to be too great of hurdle for some users. To lower this entry barrier, a Python interface module, pyCAPS, has been developed to allow greater flexibility and wider-spread use. Core functionality of select features from pyCAPS's API have been presented along with simple examples. Additional features, found only within pyCAPS, were also described in full. Furthermore, details and discussions were provided on how pyCAPS can be used to integrate CAPS capabilities into external multi-disciplinary analysis and optimization frameworks.

Appendix I: Extended representative dendrogram

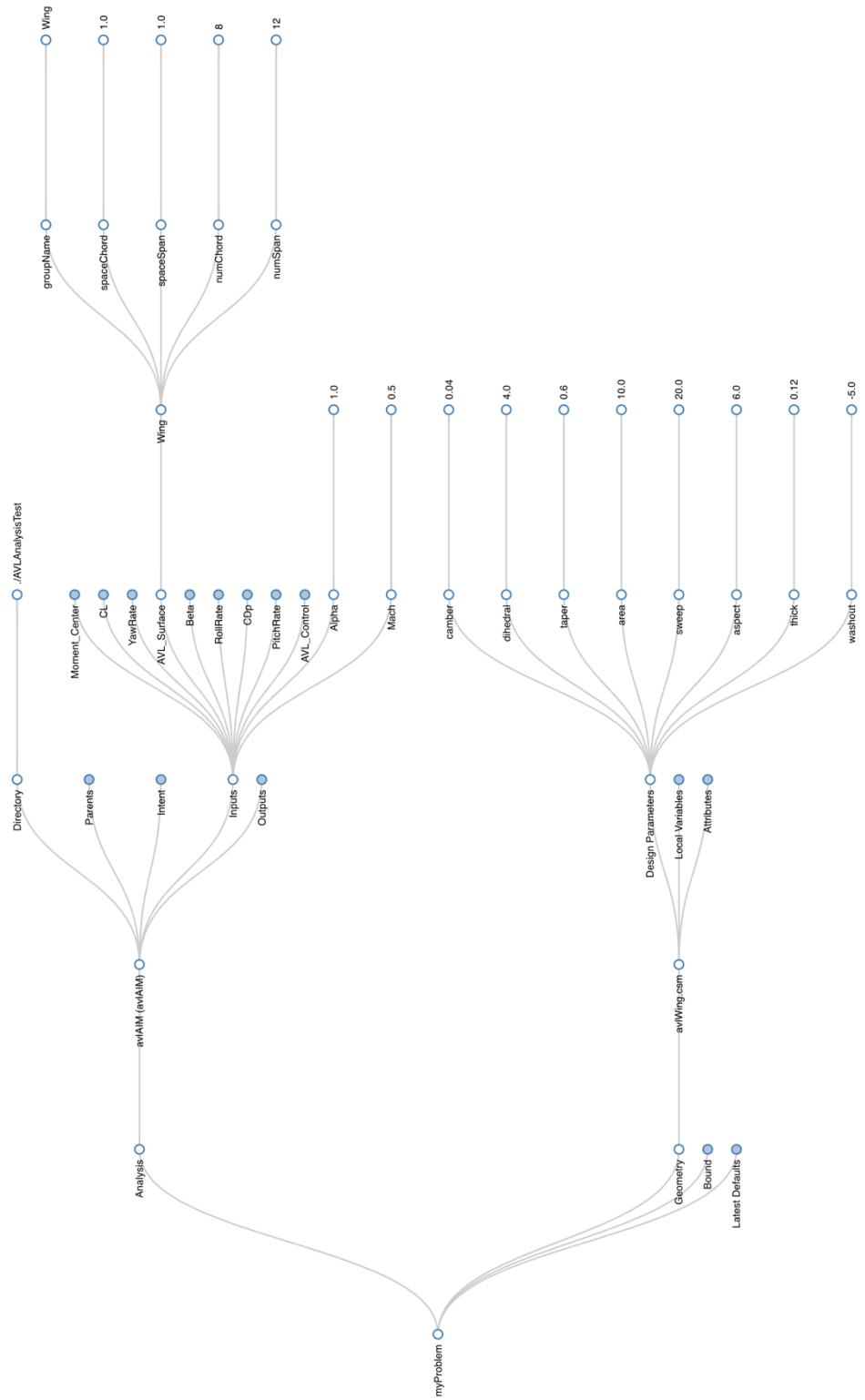

Figure A.1. Extended representative dendrogram for an instance of a *capsProblem*, with branches of the dendrogram expanded out to visualize specific inputs to the given analysis loaded into the problem, along with design variables (and their values) present for the geometry.

Appendix II: Example pyCAPS script using OpenMDAO

A representative example script coupling pyCAPS and OpenMDAO is outlined below:

```
from __future__ import print_function

# Import pyCAPS module (Linux and OSx = pyCAPS.so file; Windows = pyCAPS.pyd file)
import pyCAPS

# Import OpenMDAO module - currently tested against version 1.7.3
from openmdao.api import Problem, Group, IndepVarComp, Component, ScipyOptimizer

# Instantiate our CAPS problem "myProblem"
myProblem = pyCAPS.capsProblem()

# Load a *.csm file into our newly created problem.
myGeometry = myProblem.loadCAPS("avlWing.csm")

# Load AVL aim
myAnalysis = myProblem.loadAIM(aim = "avlAIM",
                               analysisDir = "OpenMDAOExample")

# Create OpenMDAOComponent - ExternalCode
avlComponent = myAnalysis.createOpenMDAOComponent(["Alpha",              # Analysis inputs parameters
                                                   "thick", "area"],     # Geometry design variables
                                                  ["Cmtot", "CLtot"], # Output parameters
                                                  executeCommand = ["avl", "caps"],
                                                  stdin  = "avlInput.txt",     # Modify stdin and stdout
                                                  stdout = "avlOutput.txt", # for avl execution
                                                  setSensitivity = {"type": "fd",    # Set sensitivity
                                                                    "form": "central",
                                                                    "step_size" : 1.0E-3})
# Setup AVL surfaces
wing = {"numChord"   : 8,
        "numSpan"    : 12}

myAnalysis.setAnalysisVal("AVL_Surface", ("Wing", wing))

myAnalysis.setAnalysisVal("Mach", 0.25)

# Setup and run OpenMDAO model

# Create a Group class for our design problem
class MaxLift(Group):
   def __init__(self):
      super(MaxLift, self).__init__()

      # Add design variables
      self.add('dvAlpha', IndepVarComp('Alpha', float(3.0)))
      self.add('dvAirfoil_Thickness', IndepVarComp('Thick', float(0.13)))
      self.add('dvAirfoil_Area', IndepVarComp('Area', float(10)))

      # Add AVL component
      self.add("AVL", avlComponent)

      # Make connections between design variables and inputs to AVL and geometry
```

```
        self.connect("dvAlpha.Alpha", "AVL.Alpha")
        self.connect("dvAirfoil_Thickness.Thick", "AVL.thick")
        self.connect("dvAirfoil_Area.Area" , "AVL.area")

# Initiate the OpenMDAO problem
openMDAOProblem = Problem()
openMDAOProblem.root = MaxLift ()

# Set design driver parameters
openMDAOProblem.driver = ScipyOptimizer()
openMDAOProblem.driver.options['optimizer'] = 'SLSQP'
openMDAOProblem.driver.options['tol'] = 1.0e-8
openMDAOProblem.driver.options['maxiter'] = 500 # Maximum number of solver iterations

# Set bounds on design variables
openMDAOProblem.driver.add_desvar('dvAlpha.Alpha', lower=-5.0, upper=10) # Analysis design values

openMDAOProblem.driver.add_desvar('dvAirfoil_Thickness.Thick', lower=0.1, upper=0.5) # Geometry design
values
openMDAOProblem.driver.add_desvar('dvAirfoil_Area.Area', lower=5.0, upper=20.0)

# Set objective
openMDAOProblem.driver.add_objective('AVL.CLtot')

# Set Cm constaint
openMDAOProblem.driver.add_constraint('AVL.Cmtot', equals= -0.2)

# Setup and run
openMDAOProblem.setup()
openMDAOProblem.run()

# Print the output
print("Design Alpha = ", openMDAOProblem["dvAlpha.Alpha"])
print("Design Thickness = ", openMDAOProblem["dvAirfoil_Thickness.Thick"])
print("Design Area = " , openMDAOProblem["dvAirfoil_Area.Area"])

print("Lift Coeff  = " , myAnalysis.getAnalysisOutVal("CLtot"))
print("Drag Coeff  = " , myAnalysis.getAnalysisOutVal("CDtot"))
print("Moment (y) Coeff  = ", myAnalysis.getAnalysisOutVal("Cmtot"))

# Close our problems
myProblem.closeCAPS()
```

Accompanying geometry file for OpenMDAO example – avlWing.csm:

```
# Example AVL input file to create a simple wing model
# ------------------------------------------------------------------
# Define the analysis that the geometry is intended support
# ------------------------------------------------------------------
attribute capsAIM     $avlAIM
# ------------------------------------------------------------------
# Design parameters to define the wing cross section and planform
# ------------------------------------------------------------------
despmtr   thick     0.12      frac of local chord
despmtr   camber    0.04      frac of loacl chord
despmtr   area      10.0      Planform area of the full span wing
despmtr   aspect    6.00      Span^2/Area
despmtr   taper     0.60      TipChord/RootChord
despmtr   sweep     20.0      1/4 Chord Sweep
despmtr   washout   -5.00     deg (negative is down at tip)
despmtr   dihedral  4.00      deg
# ------------------------------------------------------------------
# set parameters for use internally to create geometry
# ------------------------------------------------------------------
set       span      sqrt(aspect*area)
set       croot     2*area/span/(1+taper)
set       ctip      croot*taper
set       dxtip     (croot-ctip)/4+span/2*tand(sweep)
set       dztip     span/2*tand(dihedral)

# ------------------------------------------------------------------
# Reference quantities must exist on any body, otherwise AVL defaults
# to 1.0 for Area, Span, Chord and 0.0 for X,Y,Z moment References
# ------------------------------------------------------------------

# left tip
udprim    naca      Thickness thick     Camber    camber
attribute capsGroup    $Wing
attribute capsReferenceArea  area
attribute capsReferenceSpan  span
attribute capsReferenceChord croot
attribute capsReferenceX     croot/4
scale     ctip
rotatex   90        0         0
rotatey   washout   0         ctip/4
translate dxtip     -span/2   dztip

# root
udprim    naca      Thickness thick     Camber    camber
attribute capsGroup    $Wing
rotatex   90        0         0
scale     croot

# right tip
udprim    naca      Thickness thick     Camber    camber
attribute capsGroup    $Wing
scale     ctip
rotatex   90        0         0
rotatey   washout   0         ctip/4
translate dxtip     span/2    dztip
```

VII.Acknowledgements

The second author's contribution to the effort was fundamental research sponsored by the Air Force Research Laboratory's Multidisciplinary Science and Technology Center (MSTC) through the University of Dayton Research Institute (UDRI). This paper has been cleared for public release, case number 88ABW-2018-5919.

VIII.References

[1] Alyanak, E., Durscher, R., Haimes, R., Dannenhoffer, J., Bhagat, N., and Allison, D., "Multi-fidelity Geometry-centric Multi-disciplinary Analysis for Design," AIAA Modeling and Simulation Technologies Conference, AIAA Aviation Forum (2016) [doi: 10.2514/6.2016-4007].

[2] 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) (2012). [doi: 10.2514/6.2012-1673]

[3] Esteco, https://www.esteco.com/modefrontier/, Accessed: May 2018.

[4] Bradshaw, R., Behnel, S., Seljebotn, D. S., Ewing, G., et. al., "The Cython compiler", http://cython.org

[5] Van Heesch, D., "Doxygen: Source code documentation generator tool." http://www. doxygen.org/, Accessed May 2018.

[6] Haimes, R., "Computational Aircraft Prototype Syntheses: The CAPS API" 2018.

[7] Haimes, R. and Drela, M., "On The Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design", 2012, 50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, Aerospace Sciences Meetings.

[8] Dannenhoffer, J., "OpenCSM: An Open-Source Constructive Solid Modeler for MDAO", 2013, 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition. Grapevine, Texas.

[9] UDUNITS 2.2.26 Manual, https://www.unidata.ucar.edu/software/udunits/, Accessed November 2018.

[10] Bostock, M., Ogievetsky, V., and Heer, J., "D^3 Data-Driven Documents", 2011, IEEE Transactions on Visualization and Computer Graphics, Vol. 17, Issue 12.

[11] Hunter, J., "Matplotlib: A 2D graphics environment", Computing In Science & Engineering, Vol. 9, No. 3, 2007

[12] Pankonien, A., Durscher, R., Bhagat, N., Reich, G., "Multi-material Printed Control Surface for an Aeroservoelastic Wind Tunnel Model", AIAA Aviation Forum (2018).

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

[14] 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/

[15] TSFoil, http://www.dept.aoe.vt.edu/~mason/Mason_f/MRsoft.html#TSFOIL2, Accessed November 2018

[16] Drela, M., "XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils", Low Reynolds Number Aerodynamics: Proceedings of the Conference Notre Dame, Indiana, USA, 5--7 June, 1989