

Engineering Sketch Pad (ESP)



Training Session 10 Airfoil Optimization with CAPS

John F. Dannenhoffer, III

jfdannen@syr.edu
Syracuse University

Marshall Galbraith

galbramc@mit.edu
Massachusetts Institute of Technology
updated for v1.25

- CAPS overview
- Basic Process
- Example pyscripts
 - compute and plot aerodynamic coefficients as a function of angle of attack (`varyAlpha`)
 - compute and plot aerodynamic coefficients as a function of camber (`varyCamber`)
 - minimize drag coefficient by varying angle of attack (`optAlpha`)
- Homework exercises

- Several MDAO frameworks/environments have been developed over the last couple of decades
- These tend to focus on:
 - automating overall analysis process by creating “data flows” between user-supplied analyses
 - scheduling and dispatching of analysis execution
 - generation of suitable candidate designs via DOE, ...
 - visualization of design spaces
 - improvements of designs via optimization
 - techniques for assessing and improving the robustness of designs

- “Data” that current MDAO frameworks handle are “point” quantities (possible in “small” arrays)
 - geometric parameters: length, thickness, camber, ...
 - operating conditions: speed, load, ...
 - performance values: cost, efficiency, range, ...
- No current framework handles “field” data directly:
 - copy (same as for “point” data)
 - interpolate/evaluate
 - integrate
 - supply the derivative
- Multi-disciplinary coupling in current frameworks require that user supplies custom pairwise coupling routines

- Augment/enhance MDAO frameworks
 - Augment MDA with richer geometric information via OpenCSM
 - Enhance automation by tightly coupling analysis with geometry
 - Allow interdisciplinary analysis with “field” data transfer
 - Not replacing optimization algorithms
- Provide the tools & techniques for generalizing analysis coupling
 - multidisciplinary coupling: aeroelastic, FSI
 - multi-fidelity coupling: conceptual and preliminary design
- Provide the tools & techniques for rigorously dealing with geometry (single and multi-fidelity) in a design framework / process
 - OpenCSM connects design parameters to geometry
 - CAPS connects geometry to analysis tools
- Input and attribution driven automated (not automatic)

- The main entry point to **CAPS** system is the C/C++ API
- Direct interface for MDAO framework or User
 - C (Object-based, not object-oriented)
 - **pyCAPS**: Python interface to **CAPS** API
- Facilitates modification of Geometry/Analysis parameters
 - Geometry parameters defined with **OpenCSM**
 - Analysis parameters defined by **AIMs**
- Tracks parameter modification and dependencies
 - Modifying a geometric parameter invalidates analysis outputs

- Python c-types interface to CAPS API
- pyCAPS objects \approx CAPS API objects
 - Nearly 1-to-1 match between interfaces
 - Some aspects “pythonized”
- pyCAPS works with Python 3.3+
 - serveESP requires Python 3.8+
- PreBuilt ESP has Python 3.11 (some AIMs embed Python)
 - Includes minimal packages, e.g. Matplotlib
 - Install additional Python packages with pip

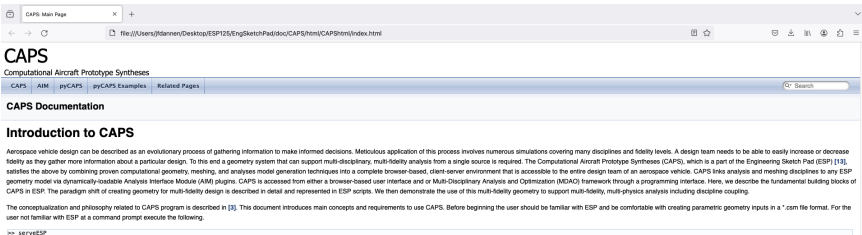
- CAPS API has 6 Object types and ~70 functions
- MDAO framework/User manipulate objects via CAPS API functions

Object	Description
capsProblem	Top-level <i>container</i> for a single mission/geometry
capsValue	Data <i>container</i> for parameters (scalar/vector/matrix)
capsAnalysis	Instance of an AIM
capsBound	Logical grouping of BRep Objects for data transfer
capsVertexSet	Discrete representation of capsBound
capsDataSet	“Field” data related to a capsVertexSet

- Interface between CAPS framework and analysis tools
 - Hides all of the individual analysis details (and peculiarities)
 - Does not make analysis tool a “black box”
- Shared libraries written in C/C++
 - Loaded at runtime as plugins
- Defines analysis input parameters and outputs
 - Inputs include attributed BRep with geometric-based information
- AIMs inputs/outputs can be linked (create data flow)
 - Transfer simple or rich data (e.g. meshes) between AIMs



Located at `$ESP_ROOT/doc/CAPS/CAPS_Overview.html`



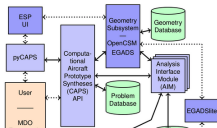
Click on the **Help** button on the upper left of the screen. This will bring up the **Engineering Sketch Pad (ESP)** documentation, that includes manual and tutorials. After completing the tutorials return to this CAPS Overview!

ESP Enhancement: Integrated and Collaborative Design Environment

This release includes an innovative way to develop ESP and CAPS scripts. A few of the highlights are: The scripts developed using this IDE are version controlled, both analysis and geometry scripts can be accessed and modified in the same familiar environment, and can be accessed in a multi-user setup for collaboration [8]. Please refer to Tutorial 6 in the ESP manual, which is a walk-through for this new capability.

Executive Overview

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 MSTC Engr [17], ModelCenter [24] or OpenMDAO [16]. 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 the CAPS Block Diagram Figure).





AIM Listing

Taken from CAPS help

CAPS Introduction x +

file:///Users/jfdannen/Projects/EngSketchPad_v1.25beta_ox64/doc/CAPS/html/aim.htm#index.html

CAPS

Analysis Interface Module (AIM)

CAPS AIM pyCAPS pyCAPS Examples Related Pages

Search

Introduction

AIM Overview

An Analysis Interface Module (AIM) plug in is associated with the Computational Aircraft Prototype Syntheses (CAPS) portion of Engineering Sketch Pad (ESP).

The type of geometric fidelity expected by the plug-in is specified at dynamic load registration (which is be something like: Outer Mode Line, Mid-Surface Aero, Built-up Element Model, Structural Solid Model, etc.). Any inputs (not associated with the BRep) need to be specified at registration. The following functions are a part of any

- AttributeInput 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. Most stand-alone grid generation systems maintain this data internally but do not make it available as output. Any attempt to re-associate this data by inverse evaluations is slow and not robust.
- Analysis Input File(s) Generation: the input values and attributes found on the geometry are used to construct and output 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: 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.

Currently Available AIMS

A table of currently available AIMS, with links to their respective documentation, is outlined below.

Surface/2D Meshing		Volume Meshing		Aerodynamics		Structures	
Name	AIM	Name	AIM	Name	AIM	Name	AIM
AFLR2 (2D Mesh only)	aflr2AIM	AFLR3	aflr3AIM	AVL [5]	avlAIM	Abaqus [4]	abaqusAIM
AFLR4 [11] [12]	aflr4AIM	Pointwise	pointwiseAIM	AWAVE [14]	awaveAIM	Astros	astrosAIM
Deltaundo (2D Mesh only)	deltaundoAIM	refine (metric adapt) [17]	refineAIM	CART3D [1]	cart3dAIM	HSM [6]	hsmAIM
EGADS Tess [9]	egadsTessAIM	TetGen	tetgenAIM	CBAERO [10]	cbaeroAIM	Interference	interferenceAIM
-	-	-	-	FRICITION [13]	frictionAIM	Masstran	masstranAIM
-	-	-	-	FUN3D [2]	fun3dAIM	MYSTRAN [3]	mystranAIM
-	-	-	-	MISES [8]	misesAIM	NASTRAN [18]	nastranAIM
-	-	-	-	SUZ [15] [16]	su2AIM	TACS [10]	tacsAIM
-	-	-	-	TSFOIL	tsfoAIM	-	-
-	-	-	-	XFOIL [7]	xfoAIM	-	-
-	-	-	-	ZERO [20]	zeroAIM	-	-

- Build a configuration with `serveESP`
- Add CAPS-required attributes to the model
- Write and execute a `pyCAPS` script
 - create CAPS problem with geometry model
 - from ESP if running from Pyscript
 - from `.csm` file if running from `python` prompt
 - create AIM(s) for analyses
 - set up required analysis inputs
 - get outputs (which often runs the AIM)



Getting More Help for CAPS and pyCAPS

- Materials for previous CAPS training can be found at acd1.mit.edu/ESP/Training/CAPStraining_2023
 - Expanded discussion about pyCAPS and the various AIMs
 - Includes lectures, sample data files, and solution files
- On-line help documentation can be found at ESP125/EngSketchPad/doc/CAPS/CAPS_Overview.html

- `naca.csm` — geometry definition
- `varyAlpha.py` — compute and plot variations of aerodynamic coefficients as a function of angle of attack
- `varyCamber.py` — compute and plot variations of aerodynamic coefficients as a function of camber
- `optAlpha.py` — find the angle of attack that minimizes the drag coefficient

- ATTRIBUTE needed on Body so that it is accessible to AIM in CAPS
- OUTPMTRs needed to make ESP variables accessible to CAPS

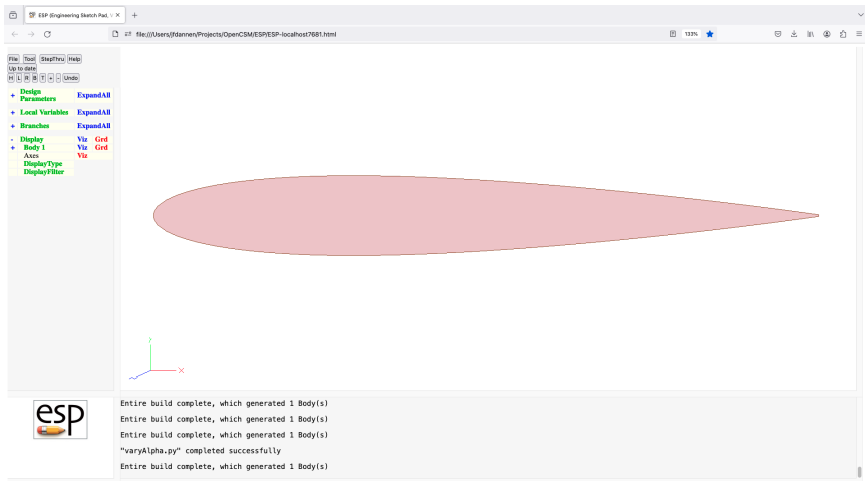
```
# naca
# written by Marshall Galbraith

# NACA design paramters
DESPMTR   thick      0.12      # frac of local chord
DESPMTR   camber     0.00      # frac of local chord

# Construct the airfoil and tell CAPS that mses will use it
UDPRIM    naca      Thickness thick   Camber   camber
          ATTRIBUTE capsAIM $msesAIM

# output the area
OUTPMTR    Area
SET        Area      @area

END
```

- pyCAPS initialization
 - `capsProblem = pyCAPS.Problem()`
- AIM (MSES) initialization
 - `mSES = capsProblem.analysis.create()`
- Setting variables for the AIM
 - `mSES.input.varName = value`
 - `mSES.input.varName = [value1, value2, ...]`
 - `mSES.input["varName"].value = value`
- Getting variables from the AIM
 - `value = mSES.output.varName`
 - `value = mSES.output["varName"].value`
- Getting the derivative of a variable from the AIM
 - `value = mSES.output["varName"].deriv("varName")`

```
#####
#                                                                 #
# varyAlpha --- MSES airfoil analysis a a function of alpha    #
#                                                                 #
#           Written by John Dannenhoffer @ Syracuse University #
#           and Marshall Galbraith @ MIT                        #
#                                                                 #
#####

# import pyCAPS module
import pyCAPS
from pyOCSM import esp

#-----

# make a semi-colon-separated string from a list
def makeString(array):
    out = ""
    for i in array:
        out += str(i) + ";"
    return out
```

```
# load geometry [.csm] file or link to model in ESP
capsProblem = pyCAPS.Problem(problemName = "varyAlpha",
                             capsFile    = "naca.csm",
                             outLevel    = 1)

# setup AIM for MSES
mSES = capsProblem.analysis.create(aim  = "mSESAIM",
                                  name = "mSES")

# set flow condition
mSES.input.Alpha = 0.0
mSES.input.Mach  = 0.1
mSES.input.Re    = 5e6

# set meshing parameters
mSES.input.GridAlpha      = 0
mSES.input.Airfoil_Points = 201

# trip the flow near the leading edge to get smooth gradient
mSES.input.xTransition_Upper = 0.1
mSES.input.xTransition_Lower = 0.1
```

```
# plot the functional and gradient
Alpha    = []
CL       = []
CD       = []
CD_Alpha = []

# run MSES and generate the data
for i in range(19):
    alpha = i - 9
    print("--> alpha", alpha)

    try:
        # asking for the outputs below run MSES
        mses.input.Alpha = alpha

        CL.append(      mses.output["CL"].value)
        CD.append(      mses.output["CD"].value)
        CD_Alpha.append(mses.output["CD"].deriv("Alpha"))
        Alpha.append(alpha)

    except pyCAPS.CAPSError:
        print("    *** did not converge ***")
```

```
# load the plotter
esp.TimLoad("plotter", esp.GetEsp("pyscript"), "")

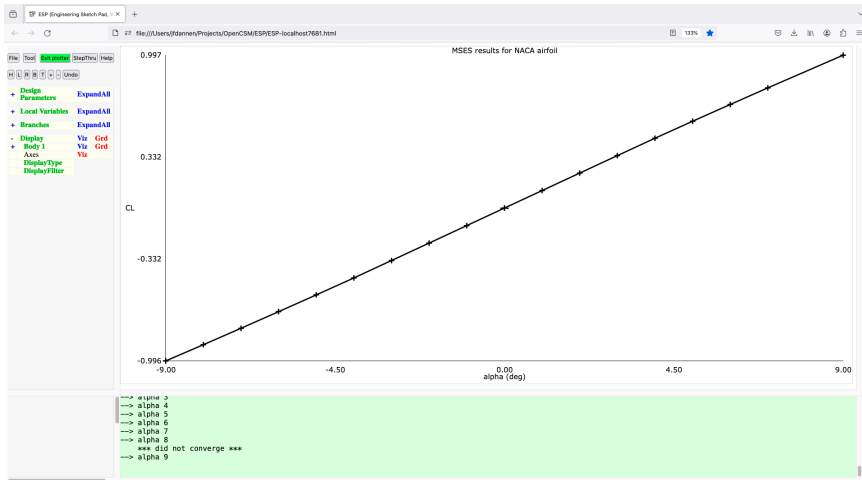
# plot the lift curve
esp.TimMsg("plotter", "new|MSES results for NACA airfoil|alpha (deg)|CL|")
esp.TimMsg("plotter", "add|"+makeString(Alpha)+"|"+makeString(CL)+"|k-+|")
esp.TimMsg("plotter", "add|-.1;+.1|0;0|k:|")
esp.TimMsg("plotter", "show")

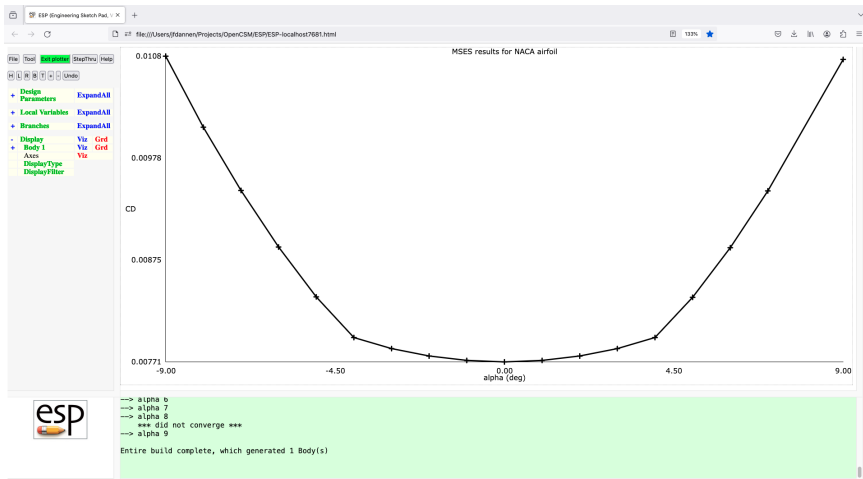
# plot the drag polar
esp.TimMsg("plotter", "new|MSES results for NACA airfoil|alpha (deg)|CD|")
esp.TimMsg("plotter", "add|"+makeString(Alpha)+"|"+makeString(CD)+"|k-+|")
esp.TimMsg("plotter", "show")

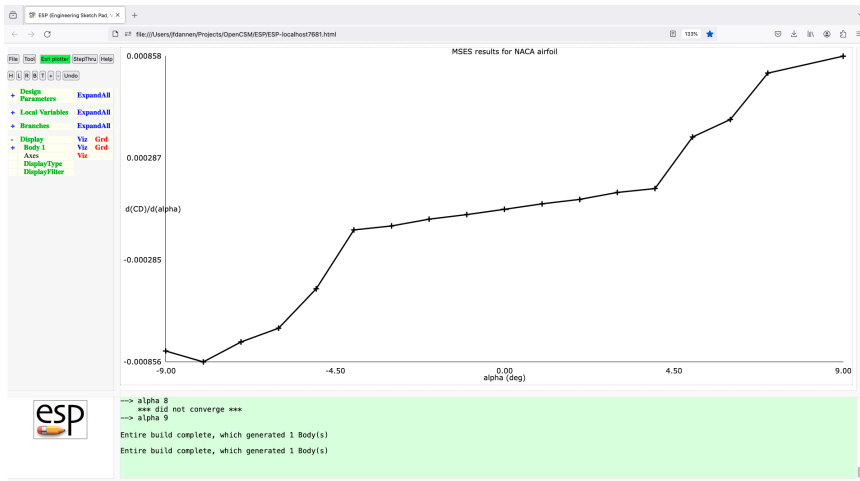
# plot the derivative of the lift curve
esp.TimMsg("plotter", "new|MSES results for NACA airfoil|alpha (deg)|d(CD)/d(alpha)|")
esp.TimMsg("plotter", "add|"+makeString(Alpha)+"|"+makeString(CD_Alpha)+"|k-+|")
esp.TimMsg("plotter", "add|-.1;+.1|0;0|k:|")
esp.TimMsg("plotter", "show")

# exit the plotter
esp.TimQuit("plotter")
```

```
# close the capsProblem (required if you want to run another pyscript)
capsProblem.close()
```







- Setting DESPMTRs

- `capsProblem.geometry.despmtr.varName = value`
- `capsProblem.geometry.despmtr.varName = [value1, value2, ...]`
- `capsProblem.geometry.despmtr["varName"].value = value`
- `capsProblem.geometry.despmtr["varName"].value = [value1, value2, ...]`

- Setting CFGPMTRs

- `capsProblem.geometry.cfgpmtr.varName = value`
- `capsProblem.geometry.cfgpmtr.varName = [value1, value2, ...]`
- `capsProblem.geometry.cfgpmtr["varName"].value = value`
- `capsProblem.geometry.cfgpmtr["varName"].value = [value1, value2, ...]`

- Getting OUTPMTRs
 - value = capsProblem.geometry.outpmtr.varName
 - value =
capsProblem.geometry.outpmtr["varName"].value
- Getting the derivative of an OUTPMTR
 - value = capsProblem.geometry
.outpmtr["varName"].deriv("varName")

```
#####  
#                                                                 #  
# varyCamber --- MSES airfoil analysis a a function of camber  #  
#                                                                 #  
#           Written by John Dannenhoffer @ Syracuse University #  
#           and Marshall Galbraith @ MIT                        #  
#                                                                 #  
#####  
  
# import pyCAPS module  
import pyCAPS  
from pyOCSM import esp  
  
#-----  
  
# make a semi-colon-separated string from a list  
def makeString(array):  
    out = ""  
    for i in array:  
        out += str(i) + " ;"  
    return out
```

```
# load geometry [.csm] file or link to model in ESP
capsProblem = pyCAPS.Problem(problemName = "varyCamber",
                             capsFile    = "naca.csm",
                             outLevel    = 1)

# setup AIM for MSES
mSES = capsProblem.analysis.create(aim = "mSESAIM",
                                   name = "mSES")

# set flow condition
mSES.input.Alpha = 0.0
mSES.input.Mach  = 0.1
mSES.input.Re    = 5e6

# set meshing parameters
mSES.input.GridAlpha      = 0
mSES.input.Airfoil_Points = 201

# trip the flow near the leading edge to get smooth gradient
mSES.input.xTransition_Upper = 0.1
mSES.input.xTransition_Lower = 0.1
```

```
# use camber as the design variable  
mses.input.Design_Variable = {"camber":{}}
```

```
# new
```

```
# plot the functional and gradient
Camber = [] # changed
CL = []
CD = []
CD_Alpha = []

# run MSES and generate the data
for i in range(21): # changed
    camber = 0.01 * (i - 10) # changed
    print("--> camber", camber) # changed

    try:
        # asking for the outputs below run MSES
        capsProblem.geometry.despmtr.camber = camber # changed

        CL.append(mses.output["CL"].value)
        CD.append(mses.output["CD"].value)
        CD_Alpha.append(mses.output["CD"].deriv("camber")) # changed
        Camber.append(camber) # changed

    except pyCAPS.CAPSError:
        print(" *** did not converge ***")
```



```
# load the plotter
esp.TimLoad("plotter", esp.GetEsp("pyscript"), "")

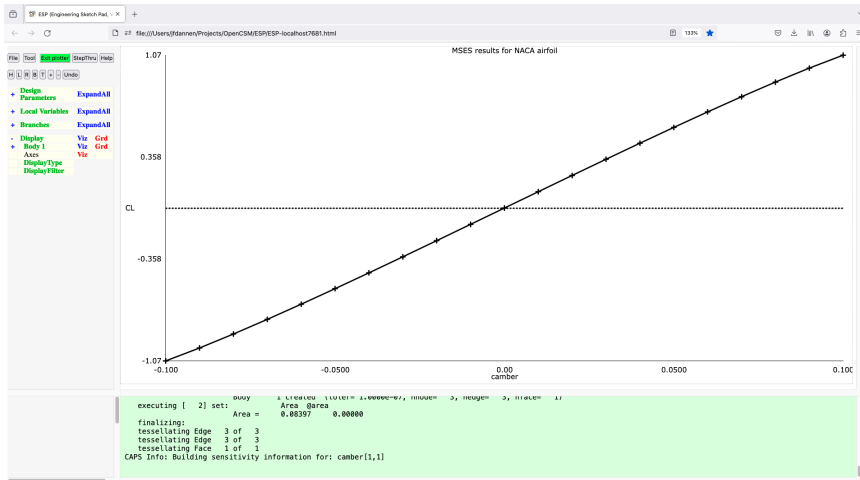
# plot the lift curve
esp.TimMesg("plotter", "new|MSES results for NACA airfoil|camber|CL|")
esp.TimMesg("plotter", "add|"+makeString(Camber)+"|"+makeString(CL)+"|k-+|")
esp.TimMesg("plotter", "add|-.1;+.1|0;0|k:|")
esp.TimMesg("plotter", "show")

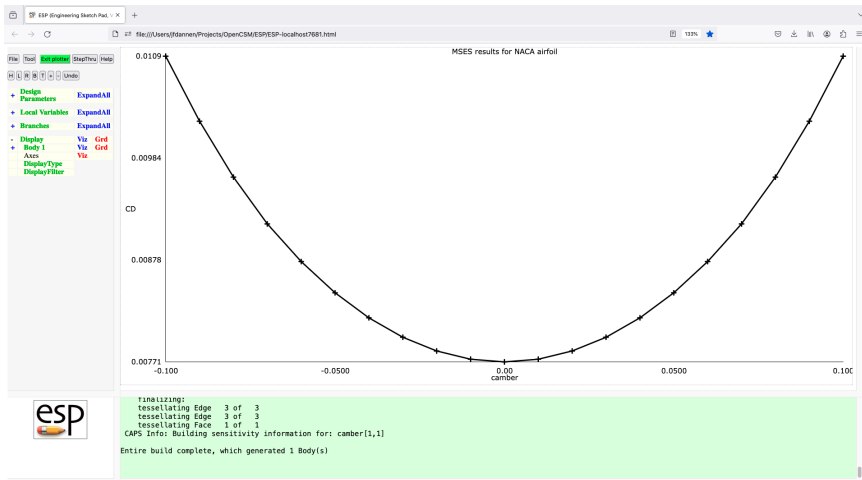
# plot the drag polar
esp.TimMesg("plotter", "new|MSES results for NACA airfoil|camber|CD|")
esp.TimMesg("plotter", "add|"+makeString(Camber)+"|"+makeString(CD)+"|k-+|")
esp.TimMesg("plotter", "show")

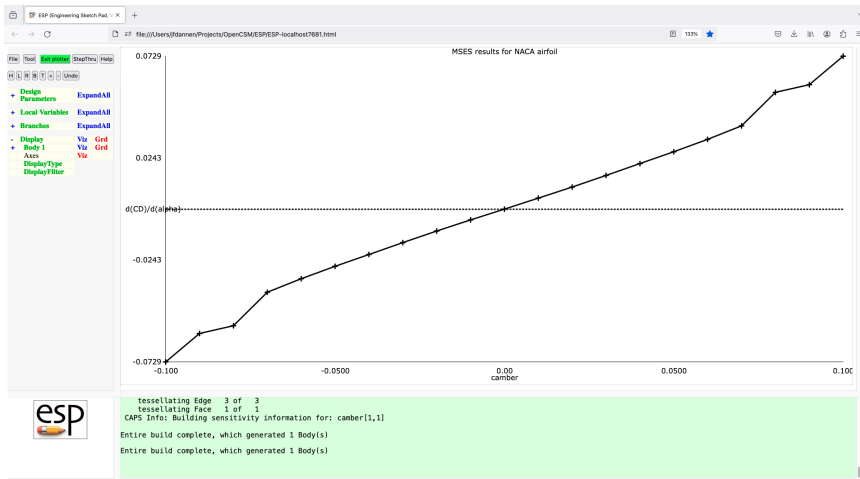
# plot the derivative of the lift curve
esp.TimMesg("plotter", "new|MSES results for NACA airfoil|camber|d(CD)/d(alpha)|")
esp.TimMesg("plotter", "add|"+makeString(Camber)+"|"+makeString(CD_Alpha)+"|k-+|")
esp.TimMesg("plotter", "add|-.1;+.1|0;0|k:|")
esp.TimMesg("plotter", "show")

# exit the plotter
esp.TimQuit("plotter")
```

```
# close the capsProblem (required if you want to run another pyscript)
capsProblem.close()
```







- Set up class needed by OpenMDAO
 - setup arguments for the component
 - assign initial values to the variables
 - compute functionals (objective functions)
 - compute functional partial derivatives
- Initialize problem
- Call the optimizer

```
#####  
#                                                                 #  
# optAlpha --- find alpha that minimized CD using OpenMDAO      #  
#                                                                 #  
#           Written by John Dannenhoffer @ Syracuse University  #  
#                   and Marshall Galbraith @ MIT                 #  
#                                                                 #  
#####  
  
# import pyCAPS module  
import pyCAPS  
from    pyOCSM import esp  
  
# import OpenMDAO v3 module  
import openmdao  
import openmdao.api as om  
#-----  
  
# make a semi-colon-separated string from a list  
def makeString(array):  
    out = ""  
    for i in array:  
        out += str(i) + ";"  
    return out
```

```
# global storage for convergence history
Alpha_call = []

# class definitions to be used by OpenMDAO
class msesAnalysis(om.ExplicitComponent):

    def initialize(self):
        # setup arguments for the component.

        # CAPS Problem input
        self.options.declare('capsProblem', types=object)
```



```
def setup(self):  
    # assign initial values to the variables.  
  
    capsProblem = self.options['capsProblem']  
    mses = capsProblem.analysis['mses']  
  
    # attach parameters to OpenMDAO object  
    self.add_input('Alpha', val=mses.input.Alpha)  
  
    # add output metric  
    self.add_output('CD')  
  
    # declare and attach partials to self  
    self.declare_partials('CD', 'Alpha')
```

```
def compute(self, inputs, outputs):
    # compute functionals

    capsProblem = self.options['capsProblem']
    mses = capsProblem.analysis['mses']

    print("\n--> Alpha    ", inputs['Alpha'])

    # update input values if changed
    if mses.input.Alpha != inputs['Alpha']:
        mses.input.Alpha = inputs['Alpha']

    # grab objective and attach as an output
    outputs['CD'] = mses.output.CD
    print("--> CD        ", outputs['CD'])

    Alpha_call.append(mses.input.Alpha)
```

```
def compute_partials(self, inputs, partials):  
    # compute functional partial derivatives  
  
    capsProblem = self.options['capsProblem']  
    mses = capsProblem.analysis['mses']  
  
    print("\n--> Alpha    ", inputs['Alpha'])  
  
    # update input values if changed  
    if mses.input.Alpha != inputs['Alpha']:  
        mses.input.Alpha = inputs['Alpha']  
  
    # get derivatives and set partials  
    partials['CD', 'Alpha'] = mses.output["CD"].deriv("Alpha")  
    print("--> CD_alpha", partials['CD', 'Alpha'])
```

```
# initiate CAPS problem
capsProblem = pyCAPS.Problem(problemName = "optAlpha",
                             capsFile    = "naca.csm",
                             outLevel    = 0)

# setup AIM for MSES
mSES = capsProblem.analysis.create(aim = "mSESAIM",
                                   name = "mSES")

# set flow condition
mSES.input.Alpha = 3.0    # Initial guess away from solution Alpha == 0
mSES.input.Mach   = 0.5
mSES.input.Re     = 5e6

# set meshing parameters
mSES.input.GridAlpha = 0
mSES.input.Airfoil_Points = 201

# trip the flow near the leading edge to get smooth gradient
mSES.input.xTransition_Upper = 0.1
mSES.input.xTransition_Lower = 0.1
```

```
# setup OpenMDAO problem

# setup the openmdao problem object
omProblem = om.Problem()

# create the OpenMDAO component
mSESSystem = mSESAnalysis(capsProblem = capsProblem)

# add subsystem to model
omProblem.model.add_subsystem('mSESSystem', mSESSystem)

# add design variables to model
omProblem.model.add_design_var('mSESSystem.Alpha', lower=-5, upper=5)

# add objective to minimize CD
omProblem.model.add_objective('mSESSystem.CD')
```

```
# setup the optimization
omProblem.driver = om.ScipyOptimizeDriver()
omProblem.driver.options['optimizer'] = "L-BFGS-B"
omProblem.driver.options['tol'] = 1.e-9
omProblem.driver.options['disp'] = True

# execute optimization
print ("\n==> Starting Optimization...")
omProblem.setup()
omProblem.run_driver()
omProblem.cleanup()

print("--> Optimized alpha:", omProblem.get_val("msesSystem.Alpha"))
```

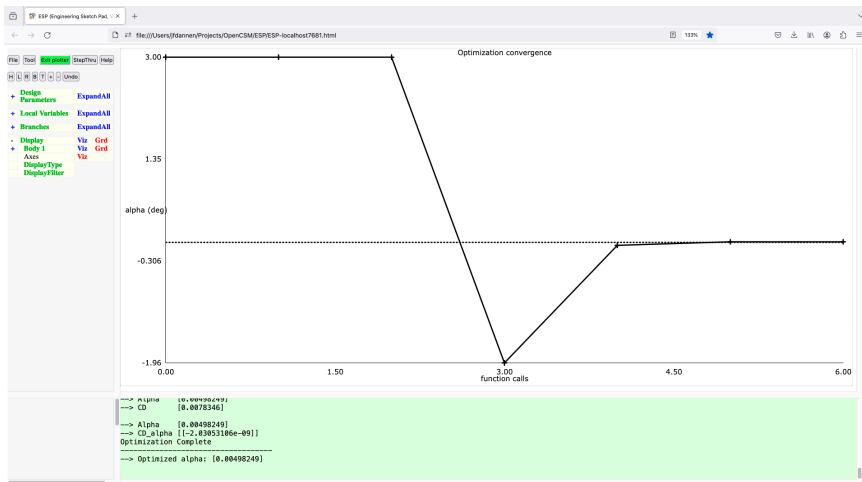
```
# load the plotter
esp.TimLoad("plotter", esp.GetEsp("pyscript"), "")

Iters = range(len(Alpha_call))
Zeros = len(Alpha_call) * [0]

# plot the convergence history
esp.TimMesg("plotter", "new|Optimization convergence|function calls|alpha (deg)|")
esp.TimMesg("plotter", "add|"+makeString(Iters)+"|"+makeString(Alpha_call)+"|k-+|")
esp.TimMesg("plotter", "add|"+makeString(Iters)+"|"+makeString(Zeros      )+"|k:|")
esp.TimMesg("plotter", "show")

# exit the plotter
esp.TimQuit("plotter")

# close the capsProblem (required if you want to run another pyscript)
capsProblem.close()
```



- Compute and plot finite-difference approximations to $\frac{\partial C_D}{\partial \alpha}$ in `varyAlpha.py` and $\frac{\partial C_D}{\partial \text{camber}}$ in `varyCamber.py`
- Find the camber that minimizes C_D , when you start with a camber of 0.1