# Details of run scripts

We need to set OpenFOAM configurations, preProcessing.sh, and runScript.py for DAFoam optimization.

内容目录：

[OpenFOAM configurations](https://dafoam.github.io/mydoc_get_started_runscript.html#openfoam-configurations)

[preProcessing.sh](https://dafoam.github.io/mydoc_get_started_runscript.html#preprocessingsh)

[runScript.py](https://dafoam.github.io/mydoc_get_started_runscript.html#runscriptpy)

## OpenFOAM configurations

As mentioned in [Overview](https://dafoam.github.io/index.html), DAFoam uses OpenFOAM for multiphysics analysis. So before running DAFoam optimizations, one needs to set up an OpenFOAM run case for NACA0012 airfoil. tutorials-main/NACA0012\_Airfoil/incompressible has the following folder structure:

NACA0012\_Airfoil/incompressible

|-- 0.orig *# initial fields and boundary conditions (OpenFOAM essentials)*

|-- constant *# flow and turbulence property definition (OpenFOAM essentials)*

|-- profiles *# NACA0012 profile coordinate for mesh generation*

|-- system *# flow discretization, setup, time step, etc (OpenFOAM essentials)*

|-- Allclean.sh *# script to clean up the simulation and optimization results*

|-- preProcessing.sh *# generate mesh, copy the initial and boundary conditions to 0*

|-- genAirFoilMesh.py *# mesh generation script called by preProcessing.sh*

|-- paraview.foam *# dummy file for paraview post-processing*

|-- runScript.py *# main run script for DAFoam*

Here we assume you are familiar with the OpenFOAM setup. Otherwise, refer to the [OpenFOAM tutorial guide](https://www.openfoam.com/documentation/tutorial-guide) for setting up the configuration files (i.e., the files in the 0, constant, and system folders).

## preProcessing.sh

Once the OpenFOAM configuration files are properly set, we run the preProcessing.sh script to generate the mesh. This script first runs the genAirFoilMesh.py script to generate hyperbolic mesh using [pyHyp](https://github.com/mdolab/pyhyp), save the mesh to the plot3D format (volumeMesh.xyz).

Then it uses the OpenFOAM’s built-in utility plot3dToFoam to convert the plot3D mesh to the OpenFOAM mesh and save it to constant/polyMesh (plot3dToFoam -noBlank volumeMesh.xyz).

Since the plot3D mesh does not have boundary information, the converted OpenFOAM mesh has only one boundary patch, so we need to use the autoPatch utility to split boundaries (autoPatch 30 -overwrite). Here 30 is the feature angle between two surface mesh faces. The utility will split patches if the feature angle is larger than 30 degree.

The above split patches will have names such as auto0, auto1, auto2, we need to rename them to wing, sym, inout, etc. This is done by running createPatch -overwrite. The definition of boundary name is in system/createPatchDict.

Next we call renumberMesh -overwrite to renumber the mesh points to reduce the bandwidth of Jacobians and reduce the memory usage. Finally, we copy the boundary condition files 0.orig to 0.

#!/bin/bash

# Check if the OpenFOAM environments are loaded

if [ -z "$WM\_PROJECT" ]; then

echo "OpenFOAM environment not found, forgot to source the OpenFOAM bashrc?"

exit

fi

# generate mesh

echo "Generating mesh.."

python genAirFoilMesh.py &> logMeshGeneration.txt

plot3dToFoam -noBlank volumeMesh.xyz >> logMeshGeneration.txt

autoPatch 30 -overwrite >> logMeshGeneration.txt

createPatch -overwrite >> logMeshGeneration.txt

renumberMesh -overwrite >> logMeshGeneration.txt

echo "Generating mesh.. Done!"

# copy initial and boundary condition files

cp -r 0.orig 0

## runScript.py

Once the OpenFOAM mesh is generated, we run mpirun -np 4 python runScript.py 2>&1 | tee logOpt.txt for optimization and save the detailed progress of optimization to logOpt.txt.

We first elaborate on the runScript.py script, which contains configurations for primal, adjoint, and optimization solutions.

The first section imports modules for DAFoam.

# =============================================================================

# Imports

# =============================================================================

import os

import argparse

import numpy as np

from mpi4py import MPI

import openmdao.api as om

from mphys.multipoint import Multipoint

from dafoam.mphys import DAFoamBuilder, OptFuncs

from mphys.scenario\_aerodynamic import ScenarioAerodynamic

from pygeo.mphys import OM\_DVGEOCOMP

from pygeo import geo\_utils

In the next section, we define the optimizer to use in “-optimizer”. We use [pyOptSparse](https://github.com/mdolab/pyoptsparse) to set optimization problems. pyOptSparse supports multiple open-source and commercial optimizers. However, in runScript.py, we only provide optimizer setup for [IPOPT](https://coin-or.github.io/Ipopt) (default), [SLSQP](http://www.pyopt.org/reference/optimizers.slsqp.html), and [SNOPT](https://ccom.ucsd.edu/~optimizers/solvers/snopt). Refer to [pyOptSparse documentation](https://mdolab-pyoptsparse.readthedocs-hosted.com/en/latest/) for all supported optimizers.

The “-task” argument defines the task to run, which includes “opt”: run optimization, “runPrimal”: run the primal analysis, “runAdjoint”: run the adjoint derivative computation, “checkTotals”: verify the adjoint accuracy against the finite-difference method.

We then define some global parameters such at “U0”: the far field velocity, “p0”: the far field pressure, “nuTilda0”: the far field turbulence variables, “CL\_target”: the target lift coefficient, “alpha0”: the initial angle of attack, “A0” and “rho0”: the reference area and density to normalize drag and lift coefficients.

parser = argparse.ArgumentParser()

parser.add\_argument("-optimizer", help="optimizer to use", type=str, default="IPOPT")

parser.add\_argument("-task", help="type of run to do", type=str, default="opt")

args = parser.parse\_args()

# =============================================================================

# Input Parameters

# =============================================================================

U0 = 10.0

p0 = 0.0

nuTilda0 = 4.5e-5

CL\_target = 0.5

alpha0 = 5.0

A0 = 0.1

rho0 = 1.0

|  |
| --- |
|  |

Next, the “daOptions” dictionary contains all the DAFoam parameters for primal and adjoint solvers. For a full list of input parameters in daOptions, refer to [here](https://dafoam.github.io/doxygen/html/classdafoam_1_1pyDAFoam_1_1DAOPTION.html).

“designSurfaces” is a list of patch names for the design surface to change during optimization. Here “wing” is a patch in constant/polyMesh/boundary and it needs to be of **wall** type.

“DASimpleFoam” is an incompressible solver that uses the SIMPLE algorithm, and it is derived from the OpenFOAM’s built-in solver simpleFoam with modification to compute adjoint derivatives.

The “primalMinResTol” parameter is the residual convergence tolerance for the primal solver (DASimpleFoam).

The “primalBC” dictionary defines the boundary conditions for primal solution. Note that if primalBC is defined, it will overwrite the values defined in the 0 folder. Here we need to provide the variable name, patch names, and value to set for each variable. If “primalBC” is left blank, we will use the BCs defined in the 0 folder.

The “objFunc” dictionary defines the objective functions. Taking “CD” as an example, we need to give a name to the objective function, e.g., “CD” or any other preferred name, and the information for each part of the objective function. Most of the time, the objective has only one part (in this case “part1”), but one can also combine two parts of objectives, e.g., we can define a new objective that is the sum of force and moment. For each part, we need to define the type of objective (e.g., “force”, “moment”; we need to use the reserved type names), how to select the discrete mesh faces to compute the objective (e.g., we select them from the name of a patch “patchToFace”), and the name of the patch (wing) for “patchToFace”. Since it is a force objective, we need to project the force vector to a specific direction. Here we defines that “CD” is the force that is parallel to the flow direction (“parallelToFlow”). Alternative, we can also use “fixedDirection” and provide a “direction” key for force, i.e., “directionMode”: “fixedDirection”, “direction”: [1.0, 0.0, 0.0]. Since we select “parallelToFlow”, we need to prescribe the name of angle of attack (aoa) design variable to determine the flow direction. NOTE: if no aoa is defined as the design variables, we can NOT use “parallelToFlow”. For this case, we have to use “directionMode”: “fixedDirection” instead. The “scale” parameter is scaling factor for this objective “CD”, i.e., CD = force / (0.5 \* U0 \* U0 \* A0). Finally, if “addToAdjoint” is “True”, the adjoint solver will compute the derivative for this objective. Otherwise, it will only calculate the objective value and print it to screen when solving the primal, no adjoint will be computed for this objective. The definition of “CL” is similar to “CD” except that we use “normalToFlow” for “directionMode”.

The “adjEqnOption” dictionary contains the adjoint linear equation solution options. If the adjoint does not converge, increase “pcFillLevel” to 2. Or try “jacMatReOrdering” : “nd”. By default, we require the adjoint equation to drop six orders of magnitudes.

“normalizeStates” contains the state normalization values. Here we use the far field values as reference. NOTE: since “p” is relative, we use the dynamic pressure “U0 \* U0 / 2”. For compressible flow, we can just use p0. Also, the face flux variable phi will be automatically normalized by its surface area so we can set “phi”: 1.0. We also need to normalize the turbulence variables, such as nuTilda, k, omega, and epsilon.

Finally, we define the design variables in the “designVar” dictionary. We need to set the “designVarType” to let DAFoam knows what type of total derivatives to compute.

daOptions = {

"designSurfaces": ["wing"],

"solverName": "DASimpleFoam",

"primalMinResTol": 1.0e-8,

"primalBC": {

"U0": {"variable": "U", "patches": ["inout"], "value": [U0, 0.0, 0.0]},

"p0": {"variable": "p", "patches": ["inout"], "value": [p0]},

"nuTilda0": {"variable": "nuTilda", "patches": ["inout"], "value": [nuTilda0]},

"useWallFunction": True,

},

"objFunc": {

"CD": {

"part1": {

"type": "force",

"source": "patchToFace",

"patches": ["wing"],

"directionMode": "parallelToFlow",

"alphaName": "aoa",

"scale": 1.0 / (0.5 \* U0 \* U0 \* A0 \* rho0),

"addToAdjoint": True,

}

},

"CL": {

"part1": {

"type": "force",

"source": "patchToFace",

"patches": ["wing"],

"directionMode": "normalToFlow",

"alphaName": "aoa",

"scale": 1.0 / (0.5 \* U0 \* U0 \* A0 \* rho0),

"addToAdjoint": True,

}

},

},

"adjEqnOption": {"gmresRelTol": 1.0e-6, "pcFillLevel": 1, "jacMatReOrdering": "rcm"},

"normalizeStates": {

"U": U0,

"p": U0 \* U0 / 2.0,

"nuTilda": nuTilda0 \* 10.0,

"phi": 1.0,

},

"designVar": {

"aoa": {"designVarType": "AOA", "patches": ["inout"], "flowAxis": "x", "normalAxis": "y"},

"shape": {"designVarType": "FFD"},

},

}

Next, we need to define the mesh deformation option. Users need to manually provide the point and normal of all symmetry planes for “symmetryPlanes” in meshOptions.

# mesh warping parameters

meshOptions = {

"gridFile": os.getcwd(),

"fileType": "OpenFOAM",

# point and normal for the symmetry plane

"symmetryPlanes": [[[0.0, 0.0, 0.0], [0.0, 0.0, 1.0]], [[0.0, 0.0, 0.1], [0.0, 0.0, 1.0]]],

}

|  |
| --- |
|  |

Next, we use the [Mphys](https://github.com/OpenMDAO/mphys) interface to set up the optimization problem. Mphys is based on [OpenMDAO](https://github.com/OpenMDAO/mphys) and allows us to develop multidisciplinary optimization capability in a modular way. If you are not familiar with OpenMDAO, check [OpenMDAO documentation](https://openmdao.org/newdocs/versions/latest/getting_started/getting_started.html). We suggest you read the “Getting Started”, “Basic user guide”, and “Advanced user guide” (optional).

The following is the OpenMDAO N2 diagram that illustrates the inputs and outputs of components and their interaction in an optimization problem. The Top class will essentially create the components and sets the proper connection.

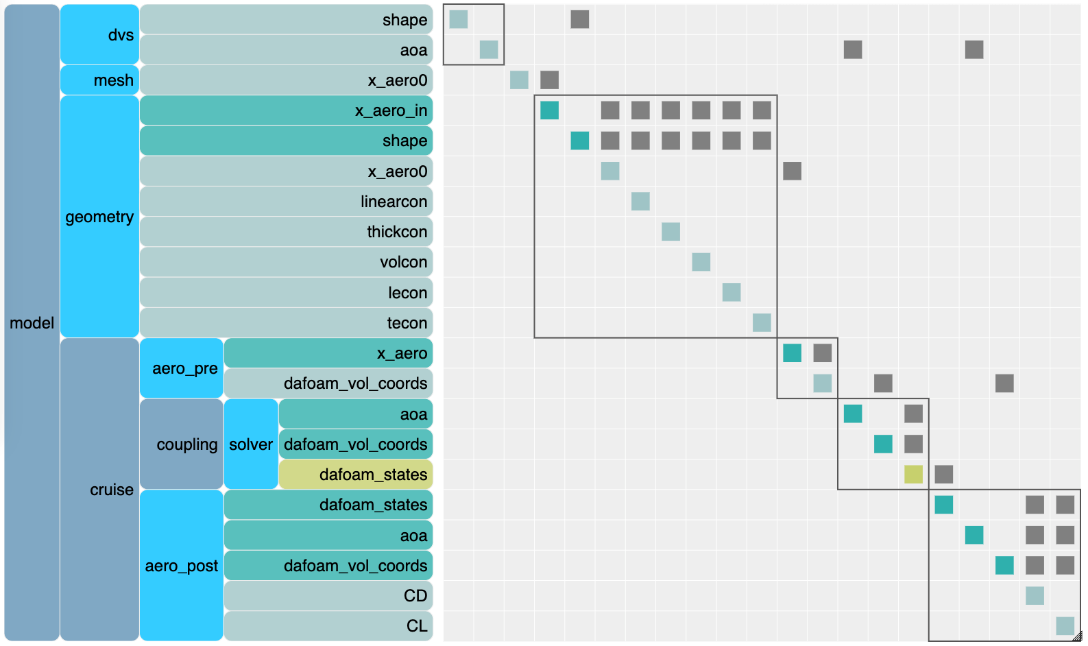


Fig. 1. The OpenMDAO N2 diagram for the airfoil aerodynamic optimization problem.

In the above diagram, blue is the groups, cyan is the components, light green is the outputs, and dark green is the inputs. The yellow denotes the outputs computed implicitly.

In the setup(self) function, we need to create all components for this problem. We first create the builder to initialize the DASolvers. This step will initialize the memory for all the variables in the OpenFOAM layer.

dafoam\_builder = DAFoamBuilder(daOptions, meshOptions, scenario="aerodynamic")

dafoam\_builder.initialize(self.comm)

Then, we call the self.add\_subsystem interface to add components (dvs, mesh, geometry) for the optimization problem.

The dvs component is a special component that has only outputs. We usually use its outputs as the design variables (shape and aoa).

The mesh component has the airfoil surface mesh coordinates as the output. It is used as the initial design surface coordinates.

The geometry component is based on [pyGeo](https://github.com/mdolab/pygeo). pyGeo uses the free-form deformation (FFD) approach to manipulate the design surface geometry. Here pyGeo takes the baseline surface mesh and the shape variables as the input and outputs the deformed surface mesh x\_aero0, as well as all the geometry constraints (defined later in this script). The FFD point file (wingFFD.xyz) is loaded to the geometry (pyGeo) component. NOTE: the FFD volume should completely contain the design surface (see the red dots from [this page](https://dafoam.github.io/mydoc_get_started_run.html)). The FFD file wingFFD.xyz is generated by running “python genFFD.py” in the FFD folder.

We also add an aerodynamic scenario called “cruise”, which is essentially a pre-defined OpenMDAO group (a group of components defined in Mphys). Refer to the above N2 diagram. We pass the dafoam\_builder to the cruise scenario and allows it to manipulate the variables in the OpenFOAM layer to compute flow and derivatives.

In the cruise group, we have the aero\_pre component which is based on [IDWarp](https://github.com/mdolab/idwarp). IDWarp uses an inverse-distance weighted algorithm to deform the volume mesh coordinates based on the surface mesh deformation, computed by pyGeo. Once the volume mesh is deformed, IDWarp outputs it as dafoam\_vol\_coords.

Then, the dafoam\_vol\_coords is passed to the solver component, along with the aoa from the dvs component. The solver (DAFoam) will compute the state variables dafoam\_states.

Finally, the converged state dafoam\_states, aoa, and dafoam\_vol\_coords will be passed to the aero\_post component to compute the objective and constraint functions (CD and CL).

Note that we need to only setup the component and data transfer, and OpenMDAO will automatically compute the total derivatives using the adjoint method.

# add the design variable component to keep the top level design variables

self.add\_subsystem("dvs", om.IndepVarComp(), promotes=["\*"])

# add the mesh component

self.add\_subsystem("mesh", dafoam\_builder.get\_mesh\_coordinate\_subsystem())

# add the geometry component (FFD)

self.add\_subsystem("geometry", OM\_DVGEOCOMP(ffd\_file="FFD/wingFFD.xyz"))

# add a scenario (flow condition) for optimization, we pass the builder

# to the scenario to actually run the flow and adjoint

self.mphys\_add\_scenario("cruise", ScenarioAerodynamic(aero\_builder=dafoam\_builder))

Next, we manually connect the surface mesh coordinates between the mesh, geometry, and cruise components.

# need to manually connect the x\_aero0 between the mesh and geometry components

# here x\_aero0 means the surface coordinates of structurally undeformed mesh

self.connect("mesh.x\_aero0", "geometry.x\_aero\_in")

# need to manually connect the x\_aero0 between the geometry component and the cruise

# scenario group

self.connect("geometry.x\_aero0", "cruise.x\_aero")

In the configure(self) function, we need to set up more details for each component (e.g., set the input and output). The following lines set the objective function defined in daOptions to Mphys, add the triangulated surface point to pyGeo (geometry component) for defining surface mesh deformation and also geometry constraints. For aerodynamic cases, users usually don’t need to change this.

# add the objective function to the cruise scenario

self.cruise.aero\_post.mphys\_add\_funcs()

# get the surface coordinates from the mesh component

points = self.mesh.mphys\_get\_surface\_mesh()

# add pointset to the geometry component

self.geometry.nom\_add\_discipline\_coords("aero", points)

# set the triangular points to the geometry component for geometric constraints

tri\_points = self.mesh.mphys\_get\_triangulated\_surface()

self.geometry.nom\_setConstraintSurface(tri\_points)

Next, we define a angle of attack (aoa) function to update the boundary velocity defined in daOptions. Note that before each primal solution, the OpenFOAM solver will use the values defined in “primalBC” and overwrite the values in the OpenFOAM flow boundaries. So by changing the “primalBC” in the aoa function, we can change the flow angle. Once the aoa function is defined, we need to pass it to the cruise group such that it will be called every time the primal solver is run.

# define an angle of attack function to change the U direction at the far field

def aoa(val, DASolver):

aoa = val[0] \* np.pi / 180.0

U = [float(U0 \* np.cos(aoa)), float(U0 \* np.sin(aoa)), 0]

# we need to update the U value only

DASolver.setOption("primalBC", {"U0": {"value": U}})

DASolver.updateDAOption()

# pass this aoa function to the cruise group

self.cruise.coupling.solver.add\_dv\_func("aoa", aoa)

self.cruise.aero\_post.add\_dv\_func("aoa", aoa)

Next, we select all the FFD points and use them as the design variables. The FFD file supports multi block meshes, but in this case we have only one block in the FFD, so we select “getLocalIndex(0)”. We allow all the points to move so we set “pts[:, :, :]” for “indexList”. Alternatively, we can select a subset of indices to move by setting a range for pts to move, e.g., indexList = pts[1:2, 3, 5:6].flatten().

# select the FFD points to move

pts = self.geometry.DVGeo.getLocalIndex(0)

indexList = pts[:, :, :].flatten()

PS = geo\_utils.PointSelect("list", indexList)

nShapes = self.geometry.nom\_addGeoDVLocal(dvName="shape", pointSelect=PS)

Next, we setup the symmetry constraints, which link the FFD point movement in k=0 and k=1.

Next, we use the nom\_addLinearConstraintsShape interface to create linear constraints to link the shape changes between k=0 and k=1 so that the shape changes are same in the spanwise direction, this is needed only for the airfoil case where we have two symmetry planes. Here “nFFDs\_x = pts.shape[0]” is the number of FFD points in the x direction. Here we impose: “lower <= factorA \* dy\_{k=0} + factorB \* dy\_{k=1} <= upper” with “dy” being the displacement of FFD point in the y direction. Substituting the parameters into the above equation, we have: “0 <= dy\_{k=0} - dy\_{k=1} <= 0”. In other words: “dy\_{k=0} = dy\_{k=1}”.

# setup the symmetry constraint to link the y displacement between k=0 and k=1

nFFDs\_x = pts.shape[0]

nFFDs\_y = pts.shape[1]

indSetA = []

indSetB = []

for i in range(nFFDs\_x):

for j in range(nFFDs\_y):

indSetA.append(pts[i, j, 0])

indSetB.append(pts[i, j, 1])

self.geometry.nom\_addLinearConstraintsShape("linearcon", indSetA, indSetB, factorA=1.0, factorB=-1.0)

We can also add the volume/thickness constraints. We need to first define the leading (“leList”) and trailing (“teList”) edges. For example, the leList includes two points that defines a straight line that is parallel to the leading edge. The straight line in leList should be close to the leading edge and completely within the wing surface mesh.

Then we call “nom\_addVolumeConstraint” to add a volume constraint with bounds [1.0:3.0]. Here we use relative upper and lower bound values with respect to the initial volume (default). To compute the volume, pyGeo first constructs a 2D mesh from the “leList” and “teList”. Here “nSpan = 2” and “nChord = 10” mean we use two points in the spanwise (z) and 10 points in the chordwise (x) to construct the 2D mesh. Then pyGeo projects this 2D mesh upward and downward to the wing surface mesh and form 3D trapezoid volumes to approximate the wing volume. The more the leList and teList are close to the actual leading and trailing edges of the airfoil mesh, the better the volume approximation will be. Also, increasing the nSpan and nChord gives a better volume approximation. We recommend nSpan and nChord be similar to the number of FFD points in the spanwise and chordwise directions.

Similarly, we call “nom\_addThicknessConstraints2D” to add thickness constraints with bounds [0.8:3.0]. Again, we use relative value with respect to the initial thickness (default). Similar to the volume constraint, pyGeo first construct a 2 by 10 mesh from the leList and teList and and projects the 2D mesh points upward and downward to the wing surface to compute the thickness at these 20 locations; we have 20 thickness constraints in total.

We can also add the LE/TE constraint. This is done by requiring the upper and lower FFD point on the leading and trailing edges to move in opposite directions. This constraint is needed because we do not want the shape variable to change the pitch and therefore the angle of attack. Instead, we want to change the far field velocity direction for the angle of attack. Here we have only one FFD block so volID=0. We also need to set the topID, which is the direction for the spanwise. This is needed only for airfoil case. For wing case, topID is automatically determined so no input is needed. “iLow” means i=0 (leading edge), and “iHigh” means i=max (trailing edge).

For more detailed explanation of constraint setup, refer to [MACH-Aero-Tutorials](https://mdolab-mach-aero.readthedocs-hosted.com/en/latest/).

# setup the volume and thickness constraints

leList = [[1e-4, 0.0, 1e-4], [1e-4, 0.0, 0.1 - 1e-4]]

teList = [[0.998 - 1e-4, 0.0, 1e-4], [0.998 - 1e-4, 0.0, 0.1 - 1e-4]]

self.geometry.nom\_addThicknessConstraints2D("thickcon", leList, teList, nSpan=2, nChord=10)

self.geometry.nom\_addVolumeConstraint("volcon", leList, teList, nSpan=2, nChord=10)

# add the LE/TE constraints

self.geometry.nom\_add\_LETEConstraint("lecon", volID=0, faceID="iLow", topID="k")

self.geometry.nom\_add\_LETEConstraint("tecon", volID=0, faceID="iHigh", topID="k")

Next, we set the outputs for the dvs component and use them as the design variables. We also need to connect the output of dvs component to the cruise and geometry component. Check the above N2. Once done, we can choose the shape and aoa in the dvs component as the design variables with proper lower and upper bounds. Note that the outputs for the dvs component has been “promoted” so we can directly use without the dvs. prefix, i.e., no need to use “dvs.shape”. Check the OpenMDAO’s documentation for promoting variable names.

# add the design variables to the dvs component's output

self.dvs.add\_output("shape", val=np.array([0] \* nShapes))

self.dvs.add\_output("aoa", val=np.array([alpha0]))

# manually connect the dvs output to the geometry and cruise

self.connect("aoa", "cruise.aoa")

self.connect("shape", "geometry.shape")

# define the design variables to the top level

self.add\_design\_var("shape", lower=-1.0, upper=1.0, scaler=1.0)

self.add\_design\_var("aoa", lower=0.0, upper=10.0, scaler=1.0)

Finally, we setup the objective and constraint functions. Note that we can set any output variables as the objective and constraints. Check the N2 diagram. We also need to prescribe the bounds for the constraints.

# add objective and constraints to the top level

self.add\_objective("cruise.aero\_post.CD", scaler=1.0)

self.add\_constraint("cruise.aero\_post.CL", equals=CL\_target, scaler=1.0)

self.add\_constraint("geometry.thickcon", lower=0.5, upper=3.0, scaler=1.0)

self.add\_constraint("geometry.volcon", lower=1.0, scaler=1.0)

self.add\_constraint("geometry.tecon", equals=0.0, scaler=1.0, linear=True)

self.add\_constraint("geometry.lecon", equals=0.0, scaler=1.0, linear=True)

self.add\_constraint("geometry.linearcon", equals=0.0, scaler=1.0, linear=True)

Once the Top class is defined, we pass it to the OpenMDAO problem and write a N2 diagram for this problem.

# OpenMDAO setup

prob = om.Problem()

prob.model = Top()

prob.setup(mode="rev")

om.n2(prob, show\_browser=False, outfile="mphys.html")

The optimizer parameters are defined next. We use the [pyOptSparse](htths://github.com/mdolab/pyoptsparse) interface to optimizers.

# use pyoptsparse to setup optimization

prob.driver = om.pyOptSparseDriver()

prob.driver.options["optimizer"] = args.optimizer

# options for optimizers

if args.optimizer == "SNOPT":

prob.driver.opt\_settings = {

"Major feasibility tolerance": 1.0e-5,

"Major optimality tolerance": 1.0e-5,

"Minor feasibility tolerance": 1.0e-5,

"Verify level": -1,

"Function precision": 1.0e-5,

"Major iterations limit": 100,

"Nonderivative linesearch": None,

"Print file": "opt\_SNOPT\_print.txt",

"Summary file": "opt\_SNOPT\_summary.txt",

}

elif args.optimizer == "IPOPT":

prob.driver.opt\_settings = {

"tol": 1.0e-5,

"constr\_viol\_tol": 1.0e-5,

"max\_iter": 100,

"print\_level": 5,

"output\_file": "opt\_IPOPT.txt",

"mu\_strategy": "adaptive",

"limited\_memory\_max\_history": 10,

"nlp\_scaling\_method": "none",

"alpha\_for\_y": "full",

"recalc\_y": "yes",

}

elif args.optimizer == "SLSQP":

prob.driver.opt\_settings = {

"ACC": 1.0e-5,

"MAXIT": 100,

"IFILE": "opt\_SLSQP.txt",

}

else:

print("optimizer arg not valid!")

exit(1)

Finally, we select the proper task to run. By default, the script will run findFeasibleDesign to find the correct aoa to get the target lift, and then start the optimization. You can also do runPrimal (solve the primal once), runAdjoint (run the primal and adjoint once), or checkTotals (compare the adjoint derivatives with the finite-difference references.)

if args.task == "opt":

*# solve CL*

optFuncs.findFeasibleDesign(["cruise.aero\_post.CL"], ["aoa"], targets=[CL\_target])

*# run the optimization*

prob.run\_driver()

elif args.task == "runPrimal":

*# just run the primal once*

prob.run\_model()

elif args.task == "runAdjoint":

*# just run the primal and adjoint once*

prob.run\_model()

totals = prob.compute\_totals()

if MPI.COMM\_WORLD.rank == 0:

print(totals)

elif args.task == "checkTotals":

*# verify the total derivatives against the finite-difference*

prob.run\_model()

prob.check\_totals(

of=["CD", "CL"], wrt=["shape", "aoa"], compact\_print=True, step=1e-3, form="central", step\_calc="abs"

)

else:

print("task arg not found!")

exit(1)

In the [next section](https://dafoam.github.io/mydoc_get_started_faq.html), we will elaborate on some common modification for this case.