DAFoam Workshop 2022

v3.0.0

Ping He and Bernardo Pacini

June 8, 2022

Objectives

After this workshop, you should be able to

- Get familiar with the new features and interfaces in DAFoam v3
- Run aerodynamic & aerostructural optimizations with DAFoam v3
- Modify/add DAFoam's C++ and Python codes for a new feature

A few notes

- We assume you are familiar with DAFoam v2.
- This workshop has **hands-on** examples.
- **Stop** us at any time if you have questions.
- The online meeting will be **recorded**.
- All the materials are available at `https://github.com/dafoam/workshops`.

Outline

DAFoam v3 Introduction

What is DAFoam ?

DAFoam : **D**iscrete **A**djoint with Open**FOAM**

DAFoam develops an efficient discrete adjoint method to perform high-fidelity multidisciplinary design optimization. DAFoam has the following features :

- It uses a popular open-source package OpenFOAM (`www.openfoam.com`) for multiphysics analysis
- It implements a Jacobian-free discrete adjoint approach with competitive speed, scalability, and accuracy
- It has a convenient Python interface to couple with OpenMDAO (`www.openmdao.org`) for multidisciplinary design optimization

What is new in DAFoam v3 ?

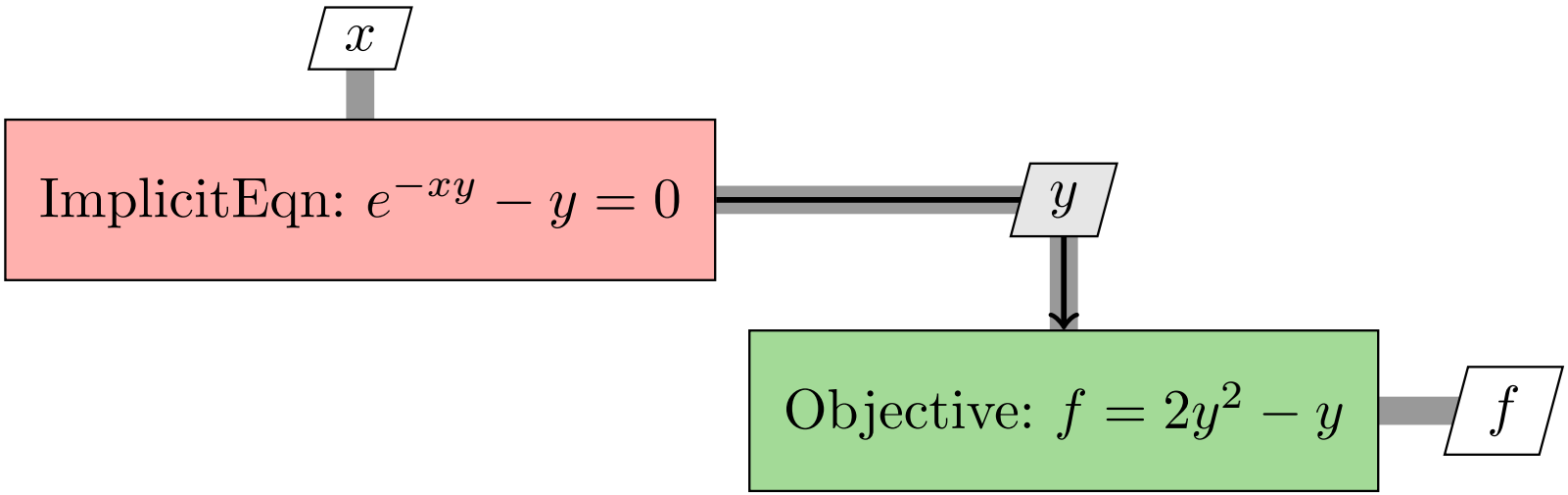
DAFoam v3 is a major release that integrated DAFoam and OpenMDAO for multidisciplinary design optimization (MDO) through the OpenMDAO/Mphys interface

- It developed a new Python interface (mphys/mphys_dafoam.py) to Mphys and OpenMDAO for MDO
- Most of the settings are same as v2, but DAFoam v3 uses very different runScript.py because it is coupled with OpenMDAO.
- You need to update dependency versions for MDO in v3. Check the DAFoam website (`https://dafoam.github.io`).
- DAFoam v3 is compatible with all v2 run scripts.

An OpenMDAO tutorial

Before using DAFoam v3, we need to learn how OpenMDAO works.
Refer to OpenMDAO's documentation for more advanced usage.
`https://openmdao.org/newdocs/versions/latest/main.html`

Optimizing a two-component system



Design structure diagram for the two-component system. The red component is implicit and the green one is explicit. The design variable is x and the objective function is f. y is the solution from the implicit component and is passed to the explicit component as the input to compute f.

Download DAFoam Docker image and examples

The most general way to run the above case is to use the DAFoam Docker image, which has OpenMDAO installed already.

First, install Docker following this website :

`https://dafoam.github.io/mydoc_get_started_download_docker.html`

Once done, verify the installation by running :

```
docker --version
```

Then run this command to download the DAFoam Docker image :

```
docker pull dafoam/opt-packages:v3.0.0
```

Finally, download the workshop examples at :

`https://github.com/dafoam/workshops`

Start a Docker container

If you use Linux or MacOS, open a terminal and use the `cd` command to go this folder on your local computer. If you put the workshops folder in the $HOME directory, the command may look like :

```
cd $HOME/workshops/2022_Summer/examples/openmdao
```

Then, run this command to start a Docker container :

```
docker run -it --rm -u dafoamuser --mount \
"type=bind,src=$(pwd),target=/home/dafoamuser/mount" \
-w /home/dafoamuser/mount dafoam/opt-packages:v3.0.0 bash
```

If you use Windows, open the Prompt Command terminal, use the `cd` command to go to the above folder, and run this command :

```
docker run -it --rm -u dafoamuser --mount \
"type=bind,src=%cd%,target=/home/dafoamuser/mount" \
-w /home/dafoamuser/mount dafoam/opt-packages:v3.0.0 bash
```

Once in a Docker container, you should see something like :

```
dafoamuser@cddb89839078:~/mount$
```

Run the case

Once in the docker container, use the `ls` command to check if you are in the correct directory. You should see something like this :

```
dafoamuser@cddb89839078:~/mount$ ls
runScript.py
```

Finally, you can run the case with this command :

```
python runScript.py
```

Expected output :

```
dafoamuser@cddb89839078:~/mount$ python runScript.py
Optimization terminated successfully (Exit mode 0)
           Current function value: [0.87500003]
           Iterations: 11
           Function evaluations: 11
           Gradient evaluations: 11
Optimization Complete
---------------------------------
f_opt: [0.87500003]
x_opt: [5.54015542]
```

An alternative option without Docker

If you already have a Python 3 on your computer, you can directly run the OpenMDAO case without using the Docker. First, load your Python 3 environment, and run this command in your terminal to install OpenMDAO

```
pip install openmdao==3.16.0
```

Then go to the tutorial folder :

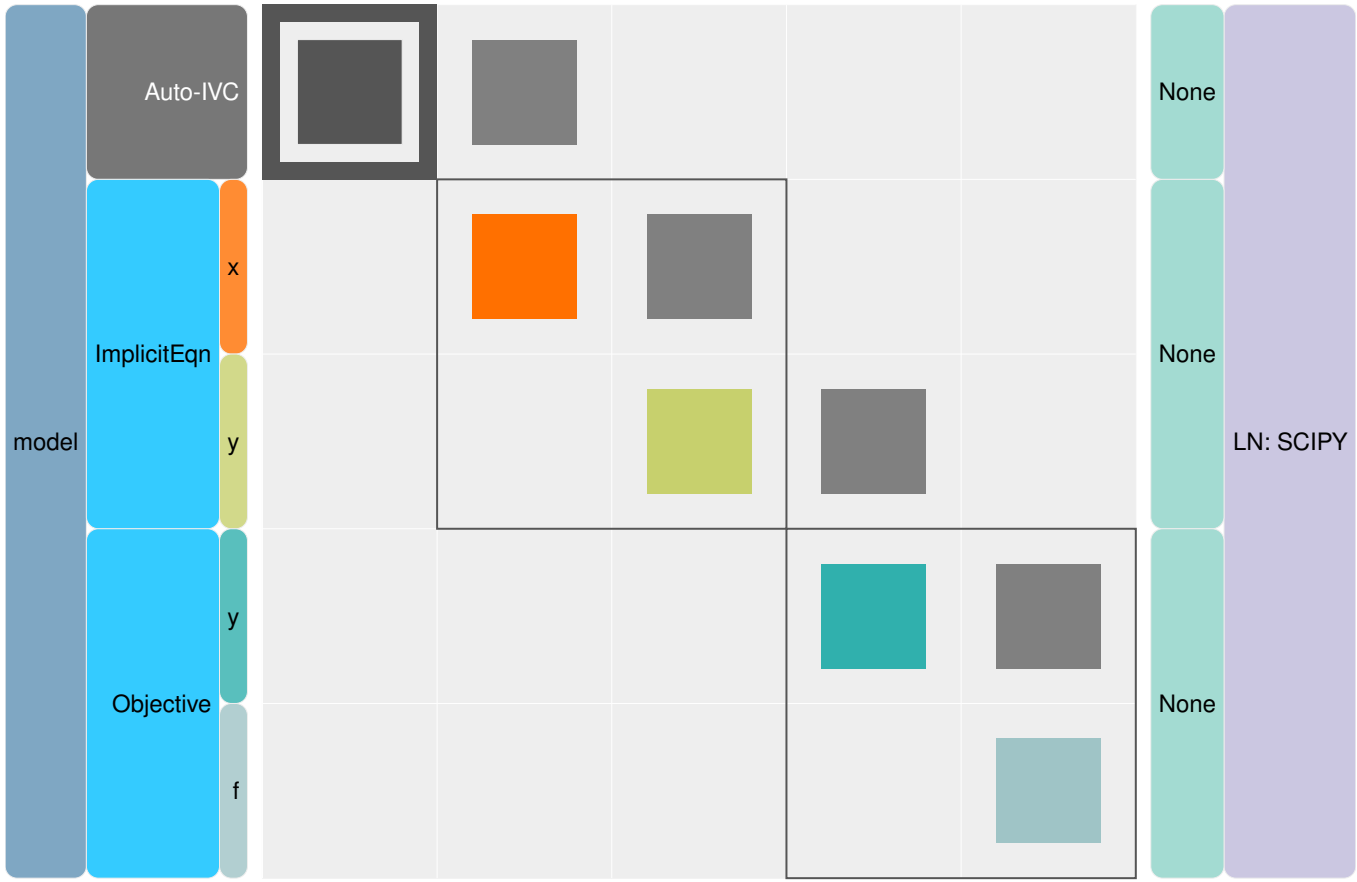
```
cd $HOME/workshops/2022_Summer/examples/openmdao
```

And run the case :

```
python runScript.py
```

N2 diagram

After the case is finished, you should see a N2 diagram (n2.html) generated for this case. The runScript.py file essential sets the components and their data transfer (connection) for the optimization.



The N2 diagram for the two-component optimization.

runScript.py details (1/3)

```
class ImplicitEqn(om.ImplicitComponent):
    def setup(self):
        # define input
        self.add_input("x", val=1.0)
        # define output
        self.add_output("y", val=1.0)

    def setup_partials(self):
        # Finite difference all partials.
        self.declare_partials("*", "*", method="fd")

    def apply_nonlinear(self, inputs, outputs, residuals):
        # get the input and output and compute the residual
        # R = e^(-x * y) - y
        # NOTE: we use [0] here because OpenMDAO assumes all inputs
        # and outputs are arrays. If the input is a scalar, OpenMDAO
        # will create an array that has size 1, so to get its value
        # we have to use [0]
        x = inputs["x"][0]
        y = outputs["y"][0]
        residuals["y"] = np.exp(-x * y) - y
```

runScript.py details (2/3)

```
class ImplicitEqn(om.ImplicitComponent):
    def setup(self):
        # define input
        self.add_input("x", val=1.0)
        # define output
        self.add_output("y", val=1.0)

    def setup_partials(self):
        # Finite difference all partials.
        self.declare_partials("*", "*", method="fd")

    def apply_nonlinear(self, inputs, outputs, residuals):
        # get the input and output and compute the residual
        # R = e^(-x * y) - y
        # NOTE: we use [0] here because OpenMDAO assumes all inputs
        # and outputs are arrays. If the input is a scalar, OpenMDAO
        # will create an array that has size 1, so to get its value
        # we have to use [0]
        x = inputs["x"][0]
        y = outputs["y"][0]
        residuals["y"] = np.exp(-x * y) - y
```

runScript.py details (3/3)

```
# create an OpenMDAO problem object
prob = om.Problem()
# now add the implicit component defined above to prob
prob.model.add_subsystem("ImplicitEqn", ImplicitEqn(), promotes=["*"])
# add the objective explicit component defined above to prob
prob.model.add_subsystem("Objective", Objective(), promotes=["*"],)
# set the linear/nonlinear equation solution for the implicit component
prob.model.nonlinear_solver = om.NewtonSolver(solve_subsystems=False)
prob.model.linear_solver = om.ScipyKrylov()
# set the design variable and objective function
prob.model.add_design_var("x", lower=-10, upper=10)
prob.model.add_objective("f", scaler=1)
# setup the optimizer
prob.driver = om.ScipyOptimizeDriver()
prob.driver.options["optimizer"] = "SLSQP"
# setup the problem
prob.setup()
# write the n2 diagram
om.n2(prob, show_browser=False, outfile="n2.html")
# run the optimization
prob.run_driver()
```

Summary

- To use OpenMDAO for optimizations, one needs to write classes for each component in the system, define their inputs and outputs, and implement the way to compute the outputs (explicit components) or residuals (implicit components).
- Then one needs to add these components to the optimization problem in the run script, set their connection, set the design variables, objective and constraint functions, and run the optimization.
- A new open-source interface Mphys was recently developed (`https://github.com/openmdao/mphys`) to rewrite the MACH-Aero modules' interfaces (e.g., pyGeo, IDWarp) into the OpenMDAO component format.
- DAFoam v3 has a Python interface to OpenMDAO/Mphys (dafoam/mphys/mphys_dafoam.py). So to run optimizations with DAFoam v3, one needs to use the new run script that sets the components, data transfer, design variables, objective and constraint functions in the optimization (see next section).

Aerodynamic optimization

Summary of the NACA0012 subsonic case

Optimizer	IPOPT
Flow and adjoint solvers	DARhoSimpleFoam
Geometry	NACA0012
Mesh	4 032 cells
Objective function	C_d
Design variables	20 FFDs and α
Constraint	$C_l = 0.5$, thickness, volume, TE/LE
U_∞	100 m/s
Re	6.7$\times 10^6$
Turbulence Model	Spalart–Allmaras

You can find the case settings in workshops/2022_Summer/examples/naca0012.

Run the case

First, use the `cd` command to go the workshops/2022_Summer/examples/naca10012 folder. Then, use the command in Slide 11 to start a Docker container. Next, use the `ls` command to check if you are in the correct directory.

```
dafoamuser@bd114f3f7c94:~/mount$ ls
0.orig      FFD      genAirFoilMesh.py preProcessing.sh runScript.py
Allclean.sh constant paraview.foam    profiles         system
```

Next, run this command to generate the mesh :

```
./preProcessing.sh
```

Finally, run the optimization with 2 cores.

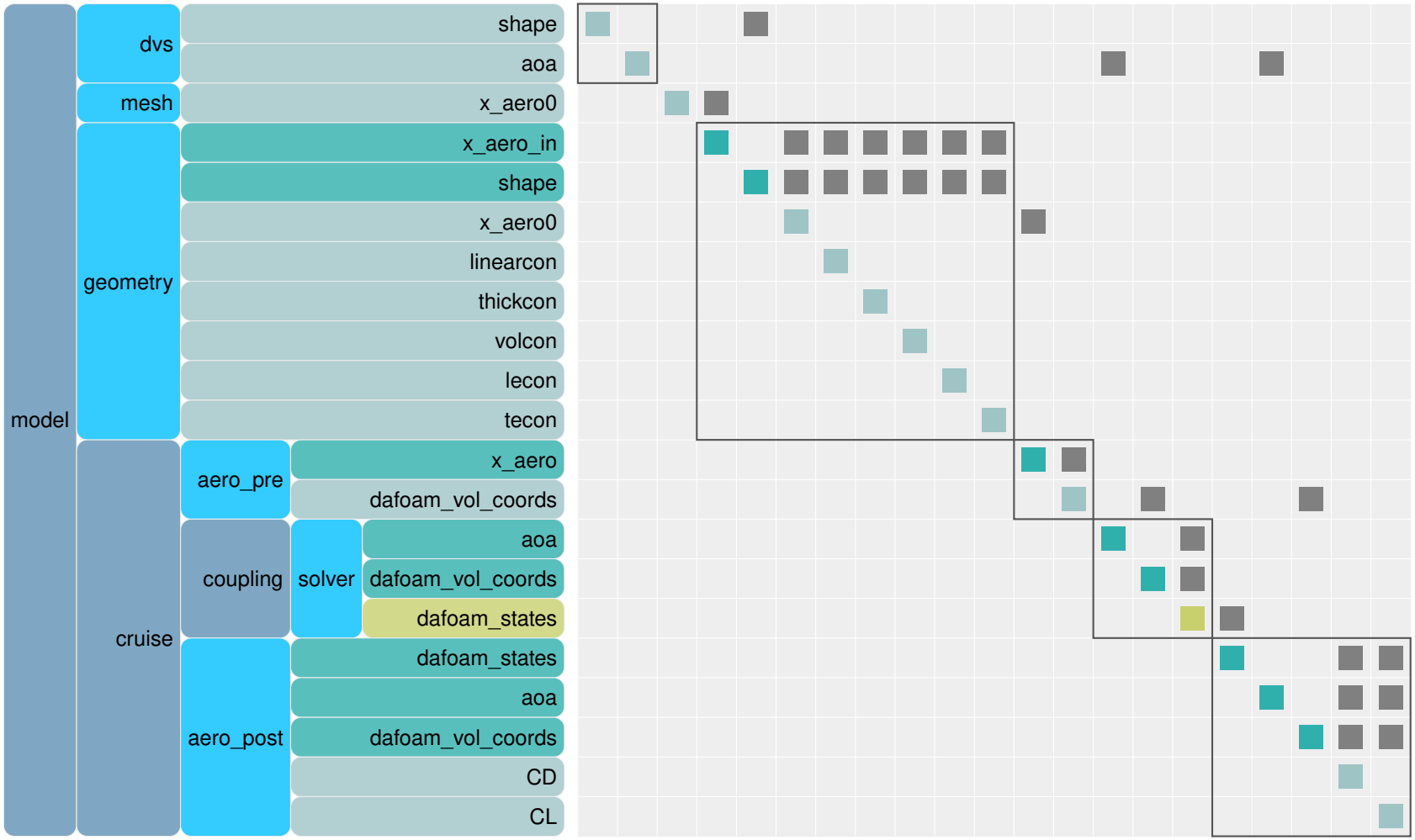
```
mpirun -np 2 python runScript.py 2>&1 | tee logOpt.txt
```

The optimization log will be printed to the screen and saved to `logOpt.txt`. In addition, the optimizer will write a separate log to the disk `opt_IPOPT.txt`.

How to post-process the optimization results ?

This has been covered in the 2021 workshop, refer to Slides 29 to 47 from `https://github.com/DAFoam/workshops/blob/main/2021_Summer/slides/2021_Summer_Workshop.pdf`

NACA0012 N2 diagram



The N2 diagram for the NACA0012 aerodynamic optimization.

runScript.py details (1/10)

Most of the settings in DAFoam v3 are same as v2. The main difference is that DAFoam v3 uses a very different runScript.py

```
# import modules
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

# input arguments for runScript.py
parser = argparse.ArgumentParser()
# which optimizer to use. Options are: IPOPT (default), SLSQP, and SNOPT
parser.add_argument("-optimizer", help="optimizer to use", type=str,
    default="IPOPT")
# which task to run. Options are: opt (default), runPrimal, runAdjoint,
    checkTotals
parser.add_argument("-task", help="type of run to do", type=str, default
    ="opt")
args = parser.parse_args()
```

runScript.py details (2/10)

The input parameters, daOptions, meshOptions are same as DAFoam v2. Next, we need to set a Multipoint Top class. We need to add all the components in the `setup` function.

```
# Top class to setup the optimization problem
class Top(Multipoint):
    def setup(self):
        # create the builder to initialize the DASolvers
        dafoam_builder = DAFoamBuilder(daOptions, meshOptions, scenario="
            aerodynamic")
        dafoam_builder.initialize(self.comm)
        # add the design variable component
        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 and pass the builder to it
        self.mphys_add_scenario("cruise", ScenarioAerodynamic(
            aero_builder=dafoam_builder))
        # need to do manually connection
        self.connect("mesh.x_aero0", "geometry.x_aero_in")
        self.connect("geometry.x_aero0", "cruise.x_aero")
```

runScript.py details (3/10)

Next, we need to do the proper connection and configurations for the above components in the `configure` function. We also need to set the design variables, objective and constraint functions. The overall process is similar to v2 but it needs to use a different syntax to set up.

```
def configure(self):
    # configure and setup perform a similar function, i.e., initialize
    # the optimization. The configure will be run after setup
    # 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 geo component for geo constraints
    tri_points = self.mesh.mphys_get_triangulated_surface()
    self.geometry.nom_setConstraintSurface(tri_points)
```

runScript.py details (4/10)

Next, we define an angle of attack (aoa) function to change the far field velocity, and add it to the "cruise" scenario as the design variable.

```
# 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)
```

runScript.py details (5/10)

We can then add the shape as the design variable by choosing all the FFD points. The setup is similar to v2.

```
# 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
    )
```

runScript.py details (6/10)

Because it is a symmetric case, we need to impose the symmetry constraint for the k=0 and k=1 level FFD points. The setup is very similar to v2.

```
# 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)
```

runScript.py details (7/10)

Similar to v2, we need to set leList and teList for the volume and thickness constraint. We also use the LETE constraint to fix the leading edge and trailing edge.

```
# 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")
```

runScript.py details (8/10)

Now, we can add the shape and aoa variables as the output for the "dvs" component and use them as the design variables. Note that we need to manually connect dvs' output to the cruise and geometry component. See the N2 diagram.

```
# 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([aoa0]))
# 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)
```

runScript.py details (9/10)

Finally, we add the objective and constraint functions.

```
# 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)
```

runScript.py details (10/10)

Once the Top class is created, we pass it as the OpenMDAO problem's model. Then, we can run the OpenMDAO's `driver` for optimization.

```
prob = om.Problem()
prob.model = Top()
prob.setup(mode="rev") # reverse mode AD
om.n2(prob, show_browser=False, outfile="mphys.html")
# initialize the optimization function
optFuncs = OptFuncs(daOptions, prob)
# use pyOptSparse to run optimization
prob.driver = om.pyOptSparseDriver()
prob.driver.options["optimizer"] = args.optimizer
# ........... pyOptSparse optimizer setup .........
# set the output option
prob.driver.options["debug_print"] = ["nl_cons", "objs", "desvars"]
# prob.driver.options["print_opt_prob"] = True
prob.driver.hist_file = "OptView.hst"
# select task to run
if args.task == "opt":
    # solve CL by changing aoa
    optFuncs.findFeasibleDesign(["cruise.aero_post.CL"], ["aoa"],
        targets=[CL_target])
    # run the optimization
    prob.run_driver()
```

Summary

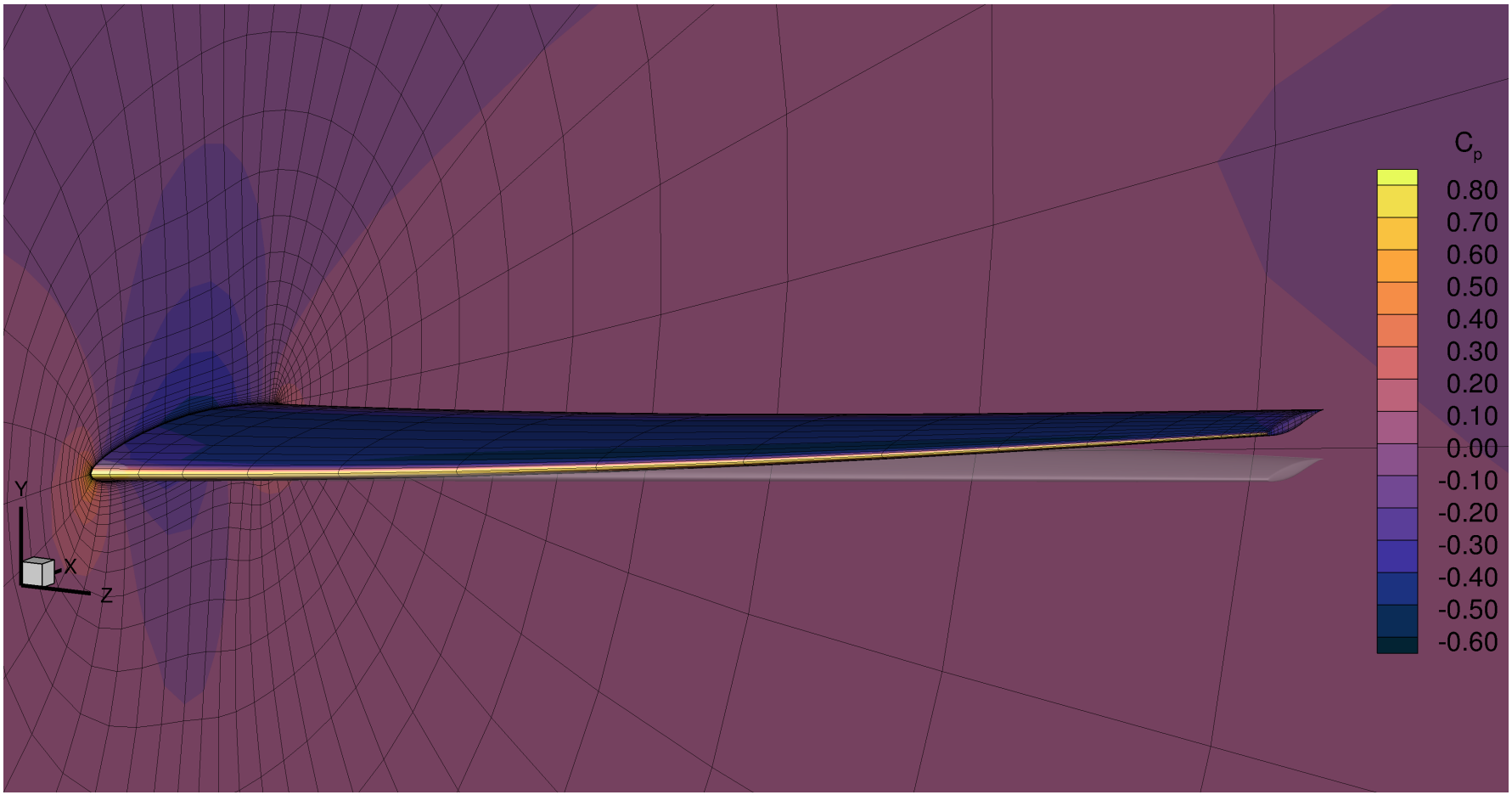
- The runScript.py is essentially an OpenMDAO run script. So we suggest you first learn how OpenMDAO works by going through the OpenMDAO's documentation (`https://openmdao.org/newdocs/versions/latest/main.html`).
- We can use the above script to run any airfoil aerodynamic optimization with DAFoam v3. If you want to change the flight conditions, FFD points, airfoil profiles, refer to the DAFoam FAQ. `https://dafoam.github.io/mydoc_get_started_faq.html`. Note that these changes are for v2 but they also work for v3.
- For 3D wing aerodynamic optimization, refer to the run script `https://github.com/DAFoam/tutorials/blob/main/MACH_Tutorial_Wing/runScript_Aero.py`
- For multipoint optimization, refer to the run script `https://github.com/DAFoam/tutorials/blob/main/NACA0012_Airfoil/multipoint/runScript.py`

Aerostructural optimization

Aerostructural Optimization Tutorial

- What is aerostructural analysis and optimization?
- How are the disciplines and solvers coupled?
- How do we setup the aerodynamic and structural problems?
- What does the runscript look like?
- What does this problem look like represented in OpenMDAO?
- What do aerostructural analysis and optimization results look like?

What is aerostructural analysis ?



Results of a converged aerostructural analysis, with the undeformed shape shown in translucent gray while the updated, converged wing is shown as the resulting mesh with contours of coefficient of pressure.

Discipline and Solver Coupling

eXtended Design Structure Matrix :

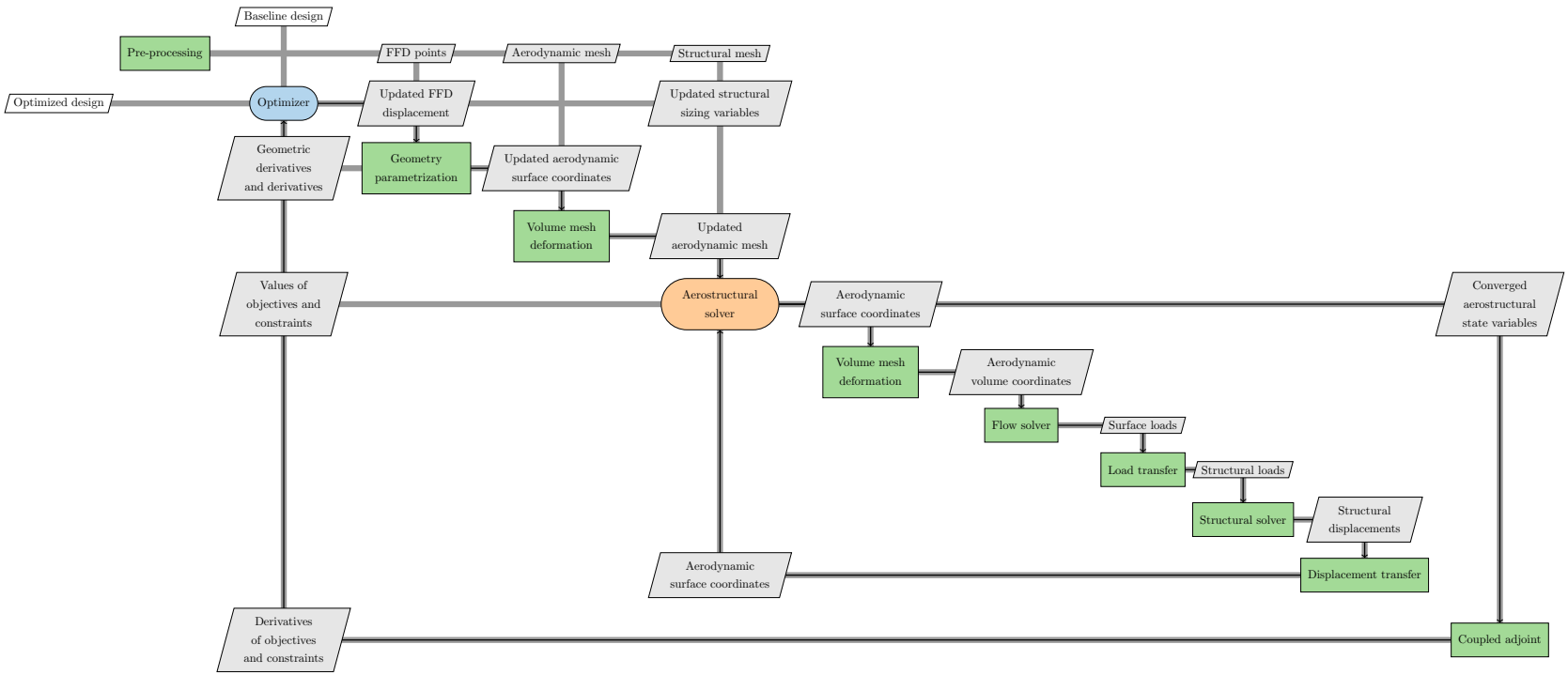
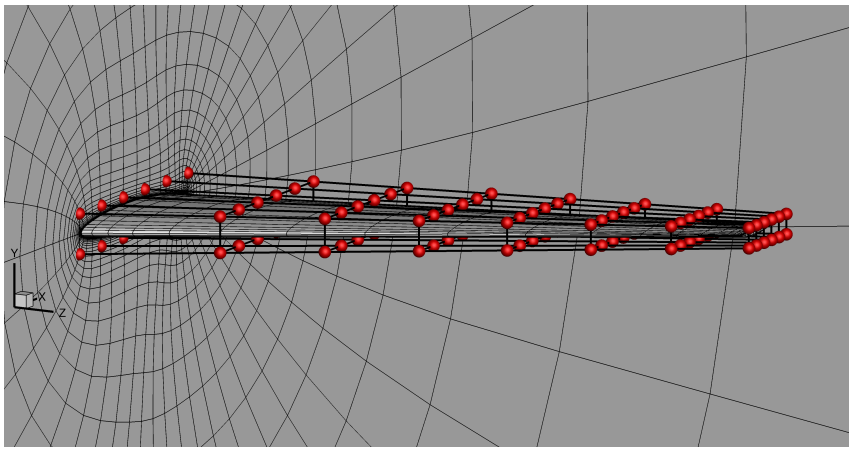


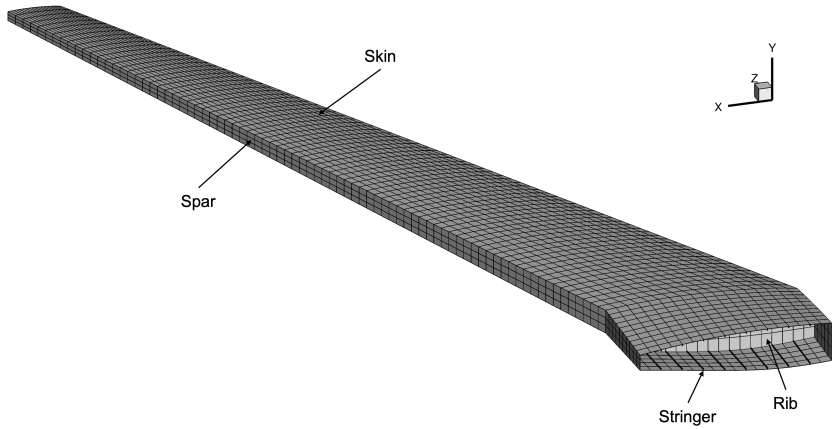
Diagram for coupled aerostructural analysis and optimization. The diagonal elements specify computational blocks where green signifies analysis, yellow signifies solvers, and blue signifies the optimizer.

Problem Setup

Aerodynamic Setup



Structural Setup



- The aerodyamic mesh and FFD are no different than a typical aerodynamic optimization case setup.
- The structural mesh consists of a finite element wingbox structure, with shell elements representing the ribs, spars, stringers, and skins.
- The aerodynamic and structural meshes should be constructed to correspond in the global coordinate frame of the optimization.

Summary of the aerostructural optimization case

Optimizer	SLSQP
Flow and adjoint solvers	DARhoSimpleFoam
Geometry	MACH Tutorial Wing
Objective function	C_D
Design variables	7 twist, 96 FFDs, and α
Constraint	$C_L = 0.3$, Von Mises stress failure, thickness, volume, TE/LE
U_∞	100 m/s
Turbulence Model	Spalart–Allmaras

You can find the case settings in workshops/2022_Summer/examples/aerostructural.

runScript.py Details (1/21) - Package Imports

The required packages are similar to the ones needed for aerodynamic analysis, with the addition of the load and displacement transfer tool (MELD) and the finite element analysis code (TACS).

```
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 tacs.mphys import TacsBuilder
from mphys.solver_builders.mphys_meld import MeldBuilder
from mphys.scenario_aerostructural import ScenarioAeroStructural
from pygeo.mphys import OM_DVGEOCOMP
from pygeo import geo_utils

parser = argparse.ArgumentParser()
# which optimizer to use. Options are: IPOPT (default), SLSQP, and SNOPT
parser.add_argument("-optimizer", help="optimizer to use", type=str, default="
    IPOPT")
# which task to run. Options are: opt (default), runPrimal, runAdjoint,
    checkTotals
parser.add_argument("-task", help="type of run to do", type=str, default="opt")
args = parser.parse_args()
```

runScript.py Details (2/21) - DAFoam Solver Settings

The DAFoam solver options remain unchanged, except for the addition of a pressure correction term.

```
U0 = 100.0
p0 = 101325.0
nuTilda0 = 4.5e-5
T0 = 300.0
CL_target = 0.5
aoa0 = 2.0
rho0 = p0 / T0 / 287.0
A0 = 45.5

daOptions = {
  "designSurfaces": ["wing"],
  "solverName": "DARhoSimpleFoam",
  "primalMinResTol": 1.0e-8,
  "fsi": {"pRef": p0}, # set the ref pressure for computing force for FSI
  "primalBC": {
    "U0": {"variable": "U", "patches": ["inout"], "value": [U0, 0.0, 0.0]},
    "p0": {"variable": "p", "patches": ["inout"], "value": [p0]},
    "T0": {"variable": "T", "patches": ["inout"], "value": [T0]},
    "nuTilda0": {"variable": "nuTilda", "patches": ["inout"], "value": [nuTilda0]},
    "useWallFunction": True,
  },
```

runScript.py Details (3/21) - DAFoam Objective Functions

Define the aerodynamic functions of interest in the same way as for the aerodynamic optimization case.

```
"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,
    }
  },
},
```

runScript.py Details (4/21) - DAFoam Adjoint Settings

When setting the adjoint options, consider that the DAFoam adjoint will be solved repeatedly when solving for coupled derivatives. Use a loose relative tolerance that will slowly converge to a much tighter absolute tolerance with each coupled iteration.

```
"adjEqnOption": {
  "gmresRelTol": 1.0e-2, # set relative tolerance for block Gauss-Seidel adjoint
  "pcFillLevel": 1,
  "jacMatReOrdering": "rcm",
  "useNonZeroInitGuess": True,
},
"normalizeStates": {
  "U": U0,
  "p": p0,
  "T": T0,
  "nuTilda": 1e-3,
  "phi": 1.0,
},
```

runScript.py Details (5/21) - DAFoam CheckMesh Settings

Aerostructural analysis and optimization require significant mesh deformation, both when updating the design itself and when displacing the geometry under load. In some cases, relax the mesh check thresholds to give DAFoam more flexibility when deforming the design.

```
"checkMeshThreshold": {
  "maxAspectRatio": 1000.0,
  "maxNonOrth": 70.0,
  "maxSkewness": 5.0,
},
```

runScript.py Details (6/21) - DAFoam Design Variables

When using MPHYS and OpenMDAO, all of the DAFoam design variables *must* be defined in the design variable dictionary. Otherwise, they may not be properly included in the problem setup.

```
"designVar": {
  "aoa": {"designVarType": "AOA", "patches": ["inout"], "flowAxis": "x", "
       normalAxis": "y"},
  "twist": {"designVarType": "FFD"},
  "shape": {"designVarType": "FFD"},
},
```

runScript.py Details (7/21) - Mesh Settings

Finalize the DAFoam mesh setup as is done for the aerodynamic analysis case.

Import the TACS setup file and specify the finite element mesh to begin configuring the structural solver.

```
# Mesh deformation setup
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]]],
}

# TACS Setup
import tacsSetup
tacsOptions = {
    "element_callback": tacsSetup.element_callback,
    "problem_setup": tacsSetup.problem_setup,
    "mesh_file": "./wingbox.bdf",
}
```

runScript.py Details (8/21) - TACS Setup

This file is separate from the runscript and is used to specify the TACS setup.

Import the TACS packages and define the material and element properties.

```
import numpy as np
from tacs import elements, constitutive, functions

# Material properties
rho = 2780.0 # density, kg/m^3
E = 73.1e9 # elastic modulus, Pa
nu = 0.33 # poisson's ratio
kcorr = 5.0 / 6.0 # shear correction factor
ys = 324.0e6 # yield stress, Pa

# Shell thickness
t = 0.003 # m
tMin = 0.002 # m
tMax = 0.05 # m
```

runScript.py Details (9/21) - TACS Element Function

This file is separate from the runscript and is used to specify the TACS setup.

Define the element callback function, used in TACS to define the element type and properties.

```
def element_callback(dvNum, compID, compDescript, elemDescripts, specialDVs, **
    kwargs):
  # Setup (isotropic) property and constitutive objects
  prop = constitutive.MaterialProperties(rho=rho, E=E, nu=nu, ys=ys)
  # Set one thickness dv for every component
  con = constitutive.IsoShellConstitutive(prop, t=t, tNum=dvNum, tlb=tMin, tub=
      tMax)

  # For each element type in this component,
  # pass back the appropriate tacs element object
  transform = None
  elem = elements.Quad4Shell(transform, con)

  return elem
```

runScript.py Details (10/21) - TACS Problem Setup

This file is separate from the runscript and is used to specify the TACS setup.

Set up the TACS problem, defining the functions of interest and any additional structural loads.

```
def problem_setup(scenario_name, fea_assembler, problem):
  """
␣␣Helper␣function␣to␣add␣fixed␣forces␣and␣eval␣functions
␣␣to␣structural␣problems␣used␣in␣tacs␣builder
␣␣"""
  # Add TACS Functions
  # Only include mass from elements that belong to pytacs components (i.e. skip
      concentrated masses)
  problem.addFunction("mass", functions.StructuralMass)
  problem.addFunction("ks_vmfailure", functions.KSFailure, safetyFactor=1.0,
      ksWeight=50.0)

  # Add gravity load
  g = np.array([0.0, 0.0, -9.81]) # m/s^2
  problem.addInertialLoad(g)
```

runScript.py Details (11/21) - Solver Setup

Set up the MPHYS components by defining the solvers : DAFoam (aerodynamic analysis), TACS (structural analysis), and MELD (load and displacement transfer).

```
class Top(Multipoint):
  def setup(self):
    # create the builder to initialize the DASolvers
    aero_builder = DAFoamBuilder(daOptions, meshOptions, scenario="aerostructural")
    aero_builder.initialize(self.comm)

    # add the aerodynamic mesh component
    self.add_subsystem("mesh_aero", aero_builder.get_mesh_coordinate_subsystem())

    # create the builder to initialize TACS
    struct_builder = TacsBuilder(tacsOptions)
    struct_builder.initialize(self.comm)

    # add the structure mesh component
    self.add_subsystem("mesh_struct", struct_builder.get_mesh_coordinate_subsystem
        ())

    # load and displacement transfer builder (meld), isym sets the symmetry plan
        axis (k)
    xfer_builder = MeldBuilder(aero_builder, struct_builder, isym=2, check_partials
        =True)
    xfer_builder.initialize(self.comm)
```

runScript.py Details (12/21) - Scenario Setup

Define the design variable component and geometry handler, as well as the aerostructural solvers and MPHYS scenario. The initialized solvers are responsible for converging the aerostructural analysis blocks. The scenario combines all of the solvers to configure the aerostructural problem.

```
# add the design variable component to keep the top level design variables
dvs = self.add_subsystem("dvs", om.IndepVarComp(), promotes=["*"])

# add the geometry component (FFD)
self.add_subsystem("geometry", OM_DVGEOCOMP(ffd_file="FFD/wingFFD.xyz"))

# primal and adjoint solution options, i.e., nonlinear block Gauss-Seidel for
    aerostructural analysis
# and linear block Gauss-Seidel for the coupled adjoint
nonlinear_solver = om.NonlinearBlockGS(maxiter=25, iprint=2, use_aitken=True, rtol
    =1e-8, atol=1e-8)
linear_solver = om.LinearBlockGS(maxiter=25, iprint=2, use_aitken=True, rtol=1e-6,
     atol=1e-6)
# add the coupling aerostructural scenario
self.mphys_add_scenario(
  "cruise",
  ScenarioAeroStructural(
    aero_builder=aero_builder, struct_builder=struct_builder, ldxfer_builder=
        xfer_builder
  ),
  nonlinear_solver,
  linear_solver,
)
```

runScript.py Details (13/21) - Aerostructural Connections

Connect the aerodynamic and structural design variables to ensure the required data is provided to all components.

```
# need to manually connect the vars in the geo component to cruise
for discipline in ["aero", "struct"]:
  self.connect("geometry.x_%s0" % discipline, "cruise.x_%s0" % discipline)

# add the structural thickness DVs
ndv_struct = struct_builder.get_ndv()
dvs.add_output("dv_struct", np.array(ndv_struct * [0.01]))
self.connect("dv_struct", "cruise.dv_struct")

# more manual connection
self.connect("mesh_aero.x_aero0", "geometry.x_aero_in")
self.connect("mesh_struct.x_struct0", "geometry.x_struct_in")
```

runScript.py Details (14/21) - Problem Configuration

Configure the optimization problem, setting the functions of interest and retrieving the initialized computational meshes.

```
def configure(self):
  # call this to configure the coupling solver
  super().configure()

  # 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_aero.mphys_get_surface_mesh()

  # add pointset for both aero and struct
  self.geometry.nom_add_discipline_coords("aero", points)
  self.geometry.nom_add_discipline_coords("struct")

  # set the triangular points to the geometry component for geometric constraints
  tri_points = self.mesh_aero.mphys_get_triangulated_surface()
  self.geometry.nom_setConstraintSurface(tri_points)
```

runScript.py Details (15/21) - FFD Variable Definitions

Define the FFD-based design variables in the same way as for the aerodynamic optimization case.

```
# Create reference axis for the twist variable
nRefAxPts = self.geometry.nom_addRefAxis(name="wingAxis", xFraction=0.25,
      alignIndex="k")

# Set up global design variables. We dont change the root twist
def twist(val, geo):
  for i in range(1, nRefAxPts):
    geo.rot_z["wingAxis"].coef[i] = -val[i - 1]

# add twist variable
self.geometry.nom_addGeoDVGlobal(dvName="twist", value=np.array([0] * (nRefAxPts -
       1)), func=twist)

# add shape variable
pts = self.geometry.DVGeo.getLocalIndex(0)
indexList = pts[:, :, :].flatten()
PS = geo_utils.PointSelect("list", indexList)
nShapes = self.geometry.nom_addGeoDVLocal(dvName="shape", pointSelect=PS)
```

runScript.py Details (16/21) - Angle of Attack Definition

Define the angle of attack design variable in the same way as for the aerodynamic optimization case, and add it as a function in DAFoam.

```
# 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.aero.solver.add_dv_func("aoa", aoa)
self.cruise.aero_post.add_dv_func("aoa", aoa)
```

runScript.py Details (17/21) - Geometric Constraints

Configure the geometric constraints, for example leading edge / trailing edge constraints, thickness constraints, and volume constraints.

```
# setup the volume and thickness constraints
leList = [[0.1, 0, 0.01], [7.5, 0, 13.9]]
teList = [[4.9, 0, 0.01], [8.9, 0, 13.9]]
self.geometry.nom_addThicknessConstraints2D("thickcon", leList, teList, nSpan=10,
     nChord=10)
self.geometry.nom_addVolumeConstraint("volcon", leList, teList, nSpan=10, nChord
     =10)
# add the LE/TE constraints
self.geometry.nom_add_LETEConstraint("lecon", volID=0, faceID="iLow")
self.geometry.nom_add_LETEConstraint("tecon", volID=0, faceID="iHigh")
```

runScript.py Details (18/21) - Design Variables

Initialize the design variables in OpenMDAO and connect them to the respective values in the optimization problem. Specify any bounds or variable scaling as desired.

```
# add the design variables to the dvs component's output
self.dvs.add_output("twist", val=np.array([0] * (nRefAxPts - 1)))
self.dvs.add_output("shape", val=np.array([0] * nShapes))
self.dvs.add_output("aoa", val=np.array([aoa0]))

# manually connect the dvs output to the geometry and cruise
self.connect("twist", "geometry.twist")
self.connect("shape", "geometry.shape")
self.connect("aoa", "cruise.aoa")

# define the design variables
self.add_design_var("twist", lower=-10.0, upper=10.0, scaler=1.0)
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)
```

runScript.py Details (19/21) - Objective and Constraints

Define the objective and constraint functions. Specify the lower / upper / equivalent values for constraints as well as any function scaling.

```
# add constraints and the objective
self.add_objective("cruise.aero_post.CD", scaler=1.0)
self.add_constraint("cruise.aero_post.CL", equals=0.3, scaler=1.0)
self.add_constraint("cruise.ks_vmfailure", lower=0.0, upper=1.0, 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)
```

runScript.py Details (20/21) - Optimization Settings

Finalize the OpenMDAO problem definition and specify the optimizer, along with its options.

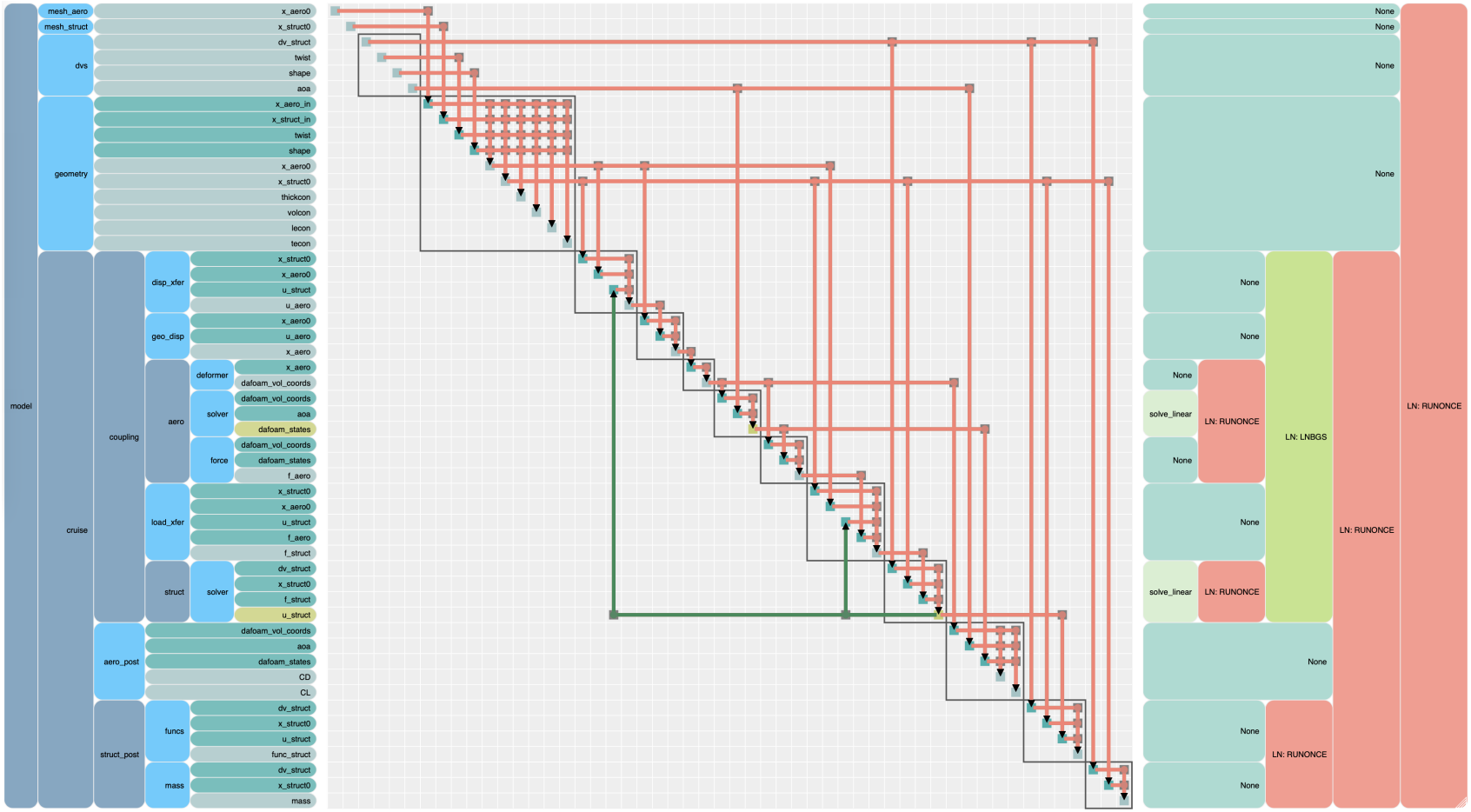
```
# OpenMDAO setup
prob = om.Problem()
prob.model = Top()
prob.setup(mode="rev")
om.n2(prob, show_browser=False, outfile="mphys_aero_struct.html")

# initialize the optimization function
optFuncs = OptFuncs(daOptions, prob)

# use pyoptsparse to setup optimization
prob.driver = om.pyOptSparseDriver()
prob.driver.options["optimizer"] = args.optimizer

prob.driver.opt_settings = {
  "ACC": 1.0e-5,
  "MAXIT": 100,
  "IFILE": "opt_SLSQP.txt",
}

prob.driver.options["debug_print"] = ["nl_cons", "objs", "desvars"]
prob.driver.options["print_opt_prob"] = True
prob.driver.hist_file = "OptView.hst"
```

runScript.py Details (21/21) - Task Runner

Run the case. Using the built in OpenMDAO features one can run the analysis, run the optimization, run the adjoint, or check the total derivatives.

```
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)
```

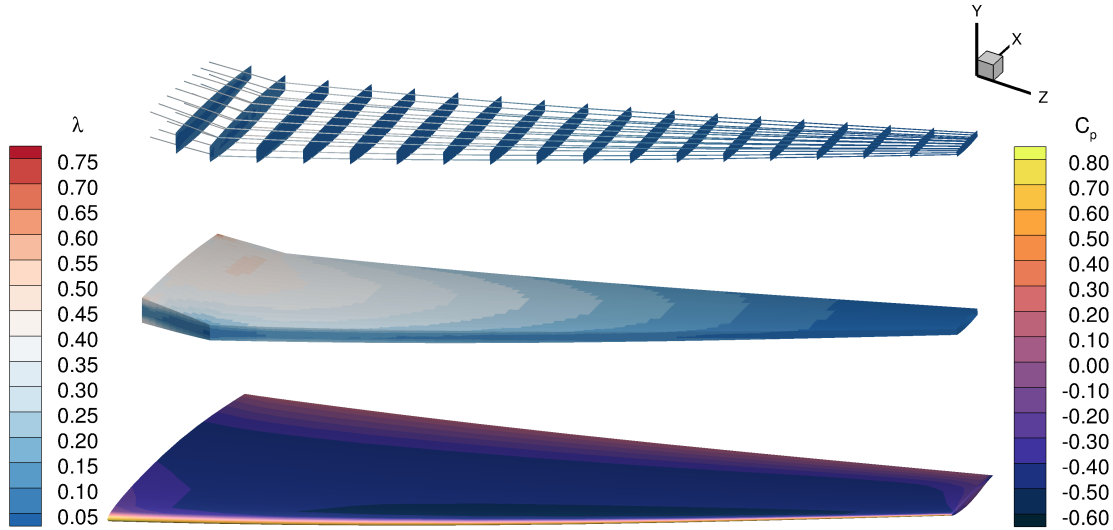
OpenMDAO Representation - N2 Diagram



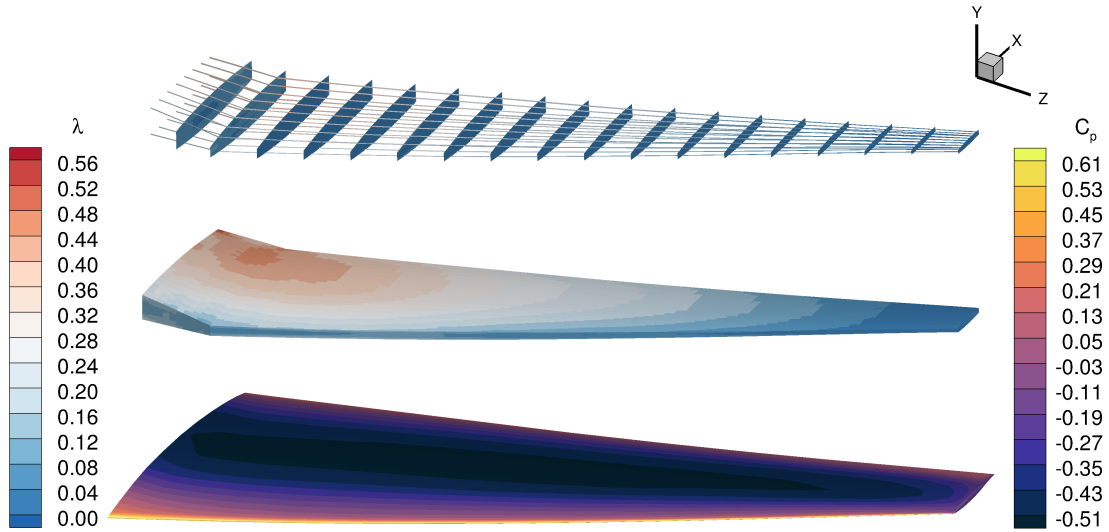
An N2 diagram showing aerostructural optimization. The diagram shows each component and how data is passed between them, as well as the associated solvers used for each part of the aerostructural analysis and optimization.

Analysis and Optimization Results

Baseline



Optimized



Summary

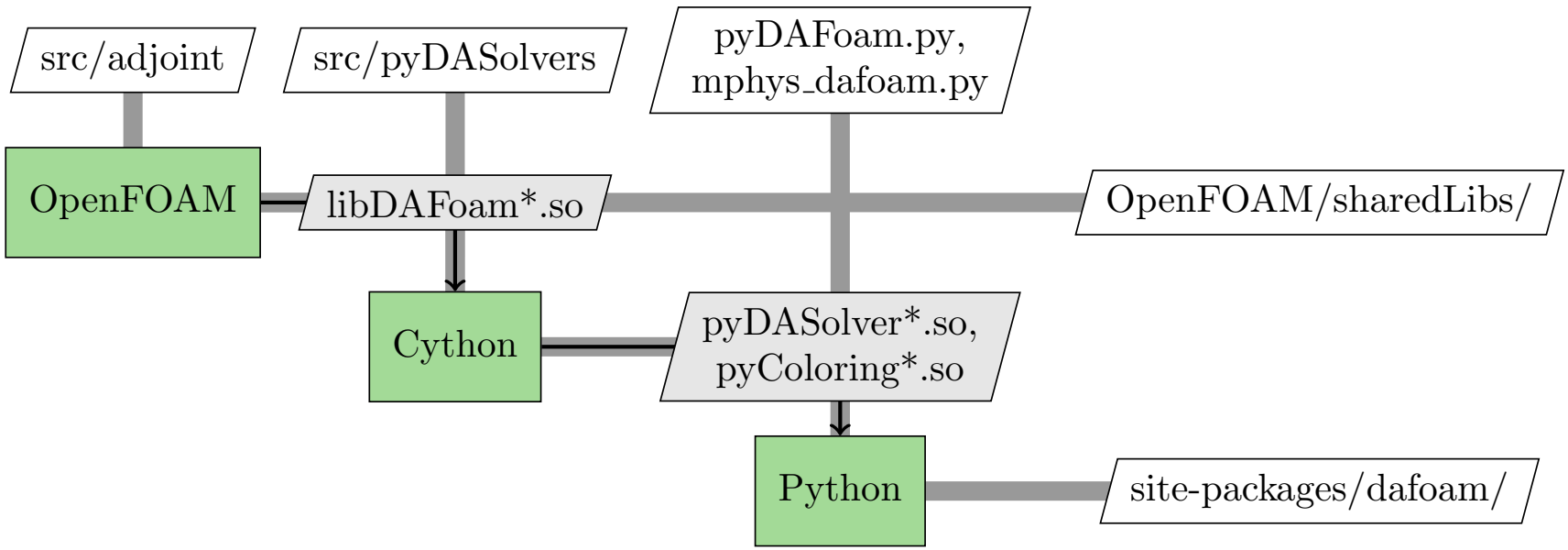
- For aerostructural analysis or optimization, we couple DAFoam with TACS, a flexible, opensource, finite element solver available at `https://github.com/smdogroup/tacs`
- OpenMDAO manages the solver coupling and data passage, streamlining the configuration for the user.
- The runscript is largely an OpenMDAO component, defining the optimizaion scenario(s) and problem options.

Add new features to DAFoam

- DAFoam source code structure : overview
- OpenFOAM layer details
- Cython layer details
- Python layer details
- Add a new parameter
- Add a new OpenFOAM layer function & expose it to Python layer
- Add a new objective/constraint function
- Add a new model or boundary condition
- Add a new design variable
- Add a new solver

DAFoam source code structure : overview

Three code layers : OpenFOAM, Cython, and Python



There are three code layers in DAFoam : OpenFOAM, Cython, and Python. Each layer takes source code or libraries as the input in the vertical direction and output libraries to the horizontal direction.

OpenFOAM layer overview

The OpenFOAM layer contains C++ classes for the OpenFOAM primal solvers and the discrete adjoint classes to compute derivatives. The OpenFOAM layer's source codes are in dafoam/src/adjoint.

- When DAFoam is compiled with the original mode, it will generate dynamic libraries for three different conditions : incompressible, compressible, and solid. In other words, it will compile three libDAFoam*.so files. Refer to dafoam/src/adjoint/Make/files_Incompressible for which files are compiled for the incompressible case
- When DAFoam is compiled with the ADR (reverse-mode AD) or ADF (forward-mode AD) modes, it will generate six more libDAFoam*.so libraries.
- These libraries will be saved to dafoam/OpenFOAM/sharedLibs
- To avoid file IO during optimization, we rewrite all the OpenFOAM solvers (binary executives) as libraries such that we can call them from Python

Cython layer overview

The compiled libDAFoam*.so from the OpenFOAM layer are pure C++ libraries, so they cannot be directly called from Python. We need to wrap these libraries using Cython. The Cython layer source codes are in src/pyDASolvers

- The Cython layer solver will use the libDAFoam*.so libraries from the OpenFOAM layer
- When running the `Allmake` command, both OpenFOAM and Cython libraries will be compiled
- Similar to the OpenFOAM layer, we will compile three sets of Cython libraries for both pyDASolver*.so and pyColoringSolver*.so. These libraries will be saved to dafoam/dafoam folder.
- pyDASolver*.so and pyColoringSolver*.so are the Cython libraries callable from the Python layer

Python layer overview

When running `pip install .`, all the files (pyDASolver*.so, pyDAFoam.py, mphys_dafoam.py, etc.) in the dafoam/dafoam folder will be copied to miniconda3/lib/python3.8/site-packages/dafoam, and they become importable from runScript.py

- runScript.py interacts with the Python layer only, it can not directly access the OpenFOAM or Cython libraries.
- The Python layer's main entry point is the `DAFoamBuilder` class from mphys_dafoam.py. So in runScript.py we import DAFoamBuilder. DAFoamBuilder then call other Python interfaces from pyDAFoam.py
- pyDAFoam.py actually imports pyDASolvers*.so (solve primal and adjoint) and pyColoringSolver*.so (calculate coloring) in its `__initSolver()` function
- Other moudles (e.g., pyGeo, IDWarp) have their own Python interfaces written in Mphys, so runScript.py imports them too

OpenFOAM layer details

Folder structure in dafoam/src/adjoint

The source codes for the OpenFOAM layer are in src/adjoint. Each subfolder defines a class.

```
src/adjoint
  Allclean          DAIndex           DAOption          DAUtility
  Allmake           DAJacCon          DAPartDeriv       Make
  DACheckMesh       DALinearEqn       DARegDb           boundaryConditions
  DAColoring        DAModel           DAResidual        lnInclude
  DAField           DAMotion          DASolver          models
  DAFvSource        DAObjFunc         DAStateInfo
```

If a class has no child classes, it reads

```
src/adjoint/DAIndex
  DAIndex.C DAIndex.H
```

Otherwise, it reads.

```
src/adjoint/DAObjFunc
  DAObjFunc.C      DAObjFuncForce.C  DAObjFuncMoment.C
  DAObjFunc.H      DAObjFuncForce.H  DAObjFuncMoment.H
  ...........      ................  ................
```

Here DAObjFuncForce is one of the child classes for DAObjFunc.

OpenFOAM layer's main entry point : DASolver class

The DASolver class is the main interface for the OpenFOAM layer.

```
src/adjoint/DASolver
  DALaplacianFoam       DARhoSimpleFoam           DASolver.C
  DAPimpleDyMFoam       DAScalarTransportFoam     DASolver.H
  DAPimpleFoam          DASimpleFoam              DATurboFoam
  DAPisoFoam            DASimpleTFoam
  DARhoSimpleCFoam      DASolidDisplacementFoam
```

In DASolver.H, we define primal and adjoint functions to interface with the Python layer, e.g., initSolver, solvePrimal, calcdRdWT, solveLinearEqn. The adjoint functions are universal to all solvers. solvePrimal() is a virtual function that needs to be defined by each child class/solver.

For example, the child class DASimpleFoam defines its own solvePrimal function that follows the exact same solution process as OpenFOAM's original simpleFoam solver.

```
src/adjoint/DASolver/DASimpleFoam
  DASimpleFoam.C     UEqnSimple.H       createRefsSimple.H
  DASimpleFoam.H     createFieldsSimple.H pEqnSimple.H
```

DASolver—DASimpleFoam example (1/4)

`initSolver()` and `solvePrimal()` are the main virtual functions implemented in DASimpleFoam.C.

```
void DASimpleFoam::initSolver(){
  Info << "Initializing␣fields␣for␣DASimpleFoam" << endl;
  // get the reference for runTime and mesh objects these two variables
  // have been initialized in DASolver constructor: DASolver.C line 48
  Time& runTime = runTimePtr_();
  fvMesh& mesh = meshPtr_();
  // initialize field variables
  #include "createSimpleControlPython.H"
  #include "createFieldsSimple.H"
  // initialize the adjoint classes, e.g., DAIndex, DAOption, DATurbo
  // NOTE: this file is in dafoam/src/adjoiont/include
  #include "createAdjointIncompressible.H"
  // initialize checkMesh object
  daCheckMeshPtr_.reset(new DACheckMesh(daOptionPtr_(), runTime, mesh));
  // initialize adjoint linear equation solution object
  daLinearEqnPtr_.reset(new DALinearEqn(mesh, daOptionPtr_()));
  // initialize the objective function
  this->setDAObjFuncList();

  .... }
```

DASolver−DASimpleFoam example (2/4)

`solvePrimal()` uses xvVec (volume coordinates) as the input and calculates wVec (state variables). xvVec and wVec are Petsc vectors.

```
void DASimpleFoam::solvePrimal(const Vec xvVec, Vec wVec){
  // create the references for the pointer objects
  #include "createRefsSimple.H"
  #include "createFvOptions.H"
  // change the run status, this is needed for the fvSource object
  daOptionPtr_->setOption<word>("runStatus", "solvePrimal");
  // call correctNut, this is equivalent to turbulence->validate();
  daTurbulenceModelPtr_->updateIntermediateVariables();
  Info << "\nStarting␣time␣loop\n" << endl;
  // deform the mesh based on the xvVec, we essentially assign the
       values
  // in xvVec to the OpenFOAM's mesh coordinates
  this->pointVec2OFMesh(xvVec);
  // check mesh quality, if the mesh quality fails, do not run the
       primal
  label meshOK = this->checkMesh();
  if (!meshOK)
  {
    this->writeFailedMesh();
    return 1;
  }
  .... }
```

DASolver−DASimpleFoam example (3/4)

The main loop, which is same as the origional simpleFoam solver

```
while (this->loop(runTime))
{
    printToScreen = this->isPrintTime(runTime, printInterval);
    if (printToScreen){
        Info << "Time = " << runTime.timeName() << nl << endl;}
    p.storePrevIter();
    // --- Pressure-velocity SIMPLE corrector
    #include "UEqnSimple.H"
    #include "pEqnSimple.H"
    laminarTransport.correct();
    daTurbulenceModelPtr_->correct();
    // we print the residual to screen every 100 steps
    if (printToScreen)
    {
        daTurbulenceModelPtr_->printYPlus();
        this->printAllObjFuncs();
        Info << "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
             << "  ClockTime = " << runTime.elapsedClockTime() << " s"
                  << nl << endl;
    }
    runTime.write();
}
```

DASolver−DASimpleFoam example (4/4)

After the simulation is done, we need to do the following

```
// write the intermediate fields
this->writeAssociatedFields();
// calculate the converged primal residual norm
this->calcPrimalResidualStatistics("print");
// primal converged, assign the OpenFoam fields to the state vec wVec
this->ofField2StateVec(wVec);
// write the deformed mesh to files
mesh.write();
Info << "End\n"<< endl;

// check if the primal residual drops to the prescribed tolerance
// if not, return failed for this primal run
return this->checkResidualTol();
```

Adjoint functions in DASolver

Essentially, any adjoint functions we want to call in the Python layer need to be defined in DASolver.C and DASolver.H. Here are just a few examples.

```
// function to compute dF/dW for the adjoint equation rhs
void DASolver::calcdFdWAD(
  const Vec xvVec,
  const Vec wVec,
  const word objFuncName,
  Vec dFdW)
{
  .....
}

// function to extract force from the wing surface for aerostructural
void DASolver::getForces(Vec fX, Vec fY, Vec fZ, Vec pointList)
{
  .....
}

// return the objective function value givne the function name
scalar DASolver::getObjFuncValue(const word objFuncName)
{
  ......
}
```

Original, reverse- and forward-AD OpenFOAM codes

- To faciliate the optimization, we have used an operator-overloading automatic differentiation (AD) tool called CoDiPack (`https://github.com/SciCompKL/CoDiPack`) to differentiate the OpenFOAM's source code. The AD version of OpenFOAM is avaiable at `https://github.com/DAFoam/OpenFOAM-v1812-AD`
- When differentiated, the OpenFOAM calculation becomes $\sim$3 times slower than the original version. Therefore, we use the AD version for the adjoint but not for running the primal.
- To enable this, we have compiled OpenFOAM into three versions : original, ADR (reverse AD), and ADF (forward AD).
- The forward- and reverse-AD versions of OpenFOAM libraries have names end with ADR and ADF. For example, we will have finiteVolume.so, finiteVolumeADR.so, and finiteVolumeADF.so. With this treatment, we can allow loading both original and ADR libraries during the optimization in the Python layer.

Cython layer details

Folder structure in dafoam/src/pyDASolvers

As mentioned above, to wrap the C++ libraries from the OpenFOAM layer, we define a C++ class called DASolvers (DASolvers.H and DASolvers.C), and its corresponding Cython class is called pyDASolvers (pyDASolver.pyx).

```
src/pyDASolvers
  Allclean                DASolvers.H             setup_Incompressible.py
  Allmake                 pyDASolvers.pyx         setup_Solid.py
  DASolvers.C             setup_Compressible.py
```

A few notes :

- DASolver (OpenFOAM layer) and DASolvers (Cython layer) are two different C++ classes.
- We do not directly wrap the DASolver class with Cython, instead, we create the DASolvers class to call the DASolver class. This treatment allows DASolver to become a pure C++ library that can easily use the runtime selction mechanism (e.g., child class for turbulence models) in OpenFOAM.
- The DASolvers and pyDASolvers are pure wrapper classes so they do not handle child classes or virtual functions

DASolvers class

As mentioned above, what DASolvers class does is to call the DASolver's methods implemented in the OpenFOAM layer. So DASolvers.H will receive the DASolver object and call DASolver's methods

```
class DASolvers
{
  // DASolverPtr_ is initialized in DASolvers.C
  // initialize fields and variables
  void initSolver()
  {
      // call DASolverPtr's initSolver method
      DASolverPtr_->initSolver();
  }
  // solve the primal equations
  label solvePrimal(
      const Vec xvVec,
      Vec wVec)
  {
      // call DASolverPtr's solvePrimal method
      return DASolverPtr_->solvePrimal(xvVec, wVec);
  }
}
```

pyDASolvers class

pyDASolvers is written in Cython format. We need to wrap all the DASolvers' member and expose them to the Python layer. To to so, we need to first define them in `cppclass DASolvers:`. Then, we need to define them in `cdef class pyDASolvers:`.

See the following for how initSolver and solverPrimal are wrapped.

```
# declare cpp functions
cdef extern from "DASolvers.H" namespace "Foam":
    cppclass DASolvers:
        DASolvers(char *, object) except +
        void initSolver()
        int solvePrimal(PetscVec, PetscVec)

# create python wrappers that call cpp functions
cdef class pyDASolvers:
    # wrap all the other member functions in the cpp class
    def initSolver(self):
        self._thisptr.initSolver()

    def solvePrimal(self, Vec xvVec, Vec wVec):
        return self._thisptr.solvePrimal(xvVec.vec, wVec.vec)
```

A few notes about the Cython layer

- We can not directly pass a Python array to the OpenFOAM layer. We have to pass them as Petsc Vector or Matrix.
- When passing a char type variable to the OpenFOAM layer, we need to need to use var_name.encode() in the Python layer.
- The Python layer can call OpenFOAM layer's member ; however, the OpenFOAM layer can not call Python layer's members.

Python layer details

Folder structure in dafoam

The Python layer codes are in the dafoam folder, see below. You may also see a few *.so libraries after compiling DAFoam. The two main Python layer files are pyDAFoam.py and mphys_dafoam.py

```
dafoam
  __init__.py   mphys/mphys_dafoam.py   optFuncs.py
  pyDAFoam.py   scripts
```

The scripts folder contains some useful utility python functions, e.g., to convert the plot3d to tecplot format, to transform or rotate a stl file.

pyDAFoam.py overview : PYDAFOAM

PYDAFOAM is the main entry point to pyDAFoam.py, and it will call initSolver() method from the OpenFOAM layer to initialize variables for both primal and adjoint solvers.

```
class PYDAFOAM(object):
  def __init__(self, comm=None, options=None):
      """
      Initialize class members
      """
      self.version = __version__
      # name
      self.name = "PYDAFOAM"
      # initialize options for adjoints
      self._initializeOptions(options)
      # check if the combination of options is valid.
      self._checkOptions()
      # initialize comm for parallel communication
      self._initializeComm(comm)
      ......
      # register solver names and set their types
      self._solverRegistry()
      # initialize the pySolvers
      self.solverInitialized = 0
      self._initSolver()
```

pyDAFoam.py overview : DAOPTION

The DAOPTION class in pyDAFoam.py defines the parameters for DAFoam optimization and assign inital values to them.

```
class DAOPTION(object):
  def __init__(self, comm=None, options=None):
      self.solverName = "DASimpleFoam"
      self.primalMinResTol = 1.0e-8
      self.primalBC = {}
```

Then it will be used by the PYDAFOAM class from :

```
def _getDefOptions(self):
    # initialize the DAOPTION object
    daOption = DAOPTION()
    defOpts = {}
    # assign all the attribute of daOptoin to defOpts
    for key in vars(daOption):
        value = getattr(daOption, key)
        defOpts[key] = [type(value), value]
    return defOpts
```

Note that the above `getDefOptions` will be called by `initializeOptions` from the PYDAFOAM's `init()` function

mphys_dafoam.py overview : DAFoamBuilder

The DAFoamBuilder is the main entry point for mphys_dafoam.py. It will call PYDAFOAM class from pyDAFoam.py to initialize OpenFOAM layer variables, and also return a DAFoamGroup (see next slide)

```
class DAFoamBuilder(Builder):
  # api level method for all builders
  def initialize(self, comm):
      # initialize the PYDAFOAM class, defined in pyDAFoam.py
      self.DASolver = PYDAFOAM(options=self.options, comm=comm)
      # always set the mesh
      mesh = USMesh(options=self.mesh_options, comm=comm)
      self.DASolver.setMesh(mesh)
  # api level method for all builders
  def get_coupling_group_subsystem(self, scenario_name=None):
      dafoam_group = DAFoamGroup(
          solver=self.DASolver,
          use_warper=self.warp_in_solver,
          struct_coupling=self.struct_coupling,
          prop_coupling=self.prop_coupling,
      )
      return dafoam_group
```

mphys_dafoam.py overview : DAFoamGroup

The DAFoamGroup will add relevant component for analysis and optimization. It will also add the DAFoamSolver component (see next slide).

```
class DAFoamGroup(Group):
  def setup(self):
      # add the IDWarp component for mesh deformation
      if self.use_warper:
          self.add_subsystem(
              "deformer",
              DAFoamWarper(solver=self.DASolver),
              promotes_inputs=["x_aero"],
              promotes_outputs=["dafoam_vol_coords"],
          )
      # add the solver implicit component from DAFoamSolver
      self.add_subsystem(
          "solver",
          DAFoamSolver(solver=self.DASolver, prop_coupling=self.
               prop_coupling),
          promotes_inputs=["*"],
          promotes_outputs=["dafoam_states"],
      )
```

mphys_dafoam.py overview : DAFoamSolver

The DAFoamSolver implements the way to compute the adjoint matrix-vector product using AD and solve the adjoint equation. DAFoamSolver will call the OpenFOAM layer functions.

```
class DAFoamSolver(ImplicitComponent):
  def apply_linear(self, inputs, outputs, d_inputs, d_outputs,
       d_residuals, mode):
      # calculate the matrix vector product: [dR/dW]^T * psi
      prodVec = DASolver.xvVec.duplicate()
      prodVec.zeroEntries()
      DASolver.solverAD.calcdRdXvTPsiAD(DASolver.xvVec, DASolver.wVec,
           resBarVec, prodVec)
      xVBar = DASolver.vec2Array(prodVec)
      d_inputs["dafoam_vol_coords"] += xVBar

  def solve_linear(self, d_outputs, d_residuals, mode):
      # solve adjoint equation using DASolver's solveLinearEqn func
      DASolver.dRdWTPC = PETSc.Mat().create(self.comm)
      DASolver.solver.calcdRdWT(DASolver.xvVec, DASolver.wVec, 1,
           DASolver.dRdWTPC)
      DASolver.ksp = PETSc.KSP().create(self.comm)
      DASolver.solverAD.createMLRKSPMatrixFree(DASolver.dRdWTPC,
           DASolver.ksp)
      DASolver.solverAD.solveLinearEqn(DASolver.ksp, dFdW, self.psi)
```

mphys_dafoam.py overview : DAFoamFunctions

The partial derivatives for the objective functions, e.g., dF/dW or dF/dXv are computed in the DAFoamFunctions class

```
class DAFoamFunctions(ExplicitComponent):
  def compute_jacvec_product(self, inputs, d_inputs, d_outputs, mode):
      dFdXv.zeroEntries()
      # call OpenFOAM layer's function to compute dF/dXv
      DASolver.solverAD.calcdFdXvAD(
          DASolver.xvVec, DASolver.wVec, objFuncName.encode(), "dummy".
              encode(), dFdXv
      )
      xVBar = DASolver.vec2Array(dFdXv)
      d_inputs["dafoam_vol_coords"] += xVBar
```

Developing new features for DAFoam

General steps to add a new feature and test it

To faciliate the compiling and testing the new feature, we created a lightweight version of DAFoam in examples/dafoam_lite. We also have a simple primal-only case in examples/curve_cube.

To add a new feature and test it, follow these general steps :

- Go to workshops/2022_Summer/examples, and start the Docker container
- Add/modify the source codes in dafoam_lite
- Compile dafoam_lite by running

```
Allmake && pip install .
```

- Add/modify curved_cube/runScript.py
- In curved_cube, run :

```
python runScript.py
```

- From the screen output, check if the new feature is implemented correctly

Add a new parameter

Steps to add a parameter

If we want to add a new float parameter called newPar, we need to

- Add this parameter to the DAOPTION's `init()` function in pyDAFoam.py, e.g., line 68 in dafoam_lite/dafoam/pyDAFoam.py

```
self.newPar = 0.0
```

- Get the new parameter's value from the OpenFOAM layer. For instance, we can call this function from the solvePrimal function in src/adjoint/DASolver/DASimpleFoam/DASimpleFoam.C, e.g., add the following line to line 192 in DASimpleFoam.C. **Then, recompile DAFoam.**

```
scalar newPar = daOptionPtr_->getOption<scalar>("newPar");
Info << "newPar " << newPar << endl;
```

- Add this new parameter to daOptions in runScript.py, e.g., line 24 in runScript.py

```
"newPar": 12.3
```

- Finally, run it to see newPar's value printed to the screen

```
python runScript.py
```

A few notes about adding a parameter

- Make sure to give an initial value for a new option, this help PYDAFOAM determine the type of this option. If it is a list, give at least one default value. If it is a dict, you can leave it blank, e.g., {}
- The parameters will be passed from the Python layer to the OpenFOAM layer through the DAOption object. This is done ONCE when the DAOptoin object is constructed. Therefore, if one needs to update a parameter during the optimization, one needs to call `DASolver.updateDAOption()` to pass the updated Python layer parameters to the OpenFOAM layer again
- The OpenFOAM layer won't be able to change the parameters from the Python layer

Add a new OpenFOAM layer function and expose it to the Python layer

Steps to add a function (1/2)

If we want to add a new function to compute the total mesh volume (`calcVol()`) in the OpenFOAM layer and expose it to the Python layer, we need to

- Add this function to the DASolver calss in the OpenFOAM layer, e.g., add this to line 241 in dafoam_lite/src/adjoint/DASolver/DASolver.H

```
void calcVol(){
  scalar vol = 0.0;
  forAll(meshPtr_->V(), cellI){
    vol += meshPtr_->V()[cellI];
  }
  Info << "Mesh␣total␣volume:␣" << vol <<endl;
}
```

- Next, we add it to the Cython layer. First add this same function to the DASolvers class in the Cython layer, e.g., add this to line 70 in dafoam_lite/src/pyDASolvers/DASolvers.H

```
void calcVol(){
  DASolverPtr_->calcVol();
}
```

Steps to add a function (2/2)

- In addition to DASolvers, we need to add this function to the pyDASolvers class, e.g., we need to add this to line 95 in dafoam_lite/src/pyDASolvers/pyDASolvers.pyx

```
void calcVol()
```

- In addition, we need to add this to line 350 in dafoam_lite/src/pyDASolvers/pyDASolvers.pyx. The Cython layer is done and this function is calable in the Python layer

```
def calcVol(self):
    self._thisptr.calcVol()
```

- To verify the the new function, we can add this line to line 32 in examples/curved_cube/runScript.py. **Recompile DAFoam.**

```
DASolver.solver.calcVol()
```

- Run the case, you will see the total volume printed to the screen

```
python runScript.py
```

A few notes about adding a function

- We can add a new function to any classes in the OpenFOAM layer. But eventually, we need to write a access function in the DASolver class because only functions in this class is accessable from the Cython layer's DASolvers class.
- Note that when calling this OpenFOAM layer function from the Python layer, we use `DASolver.solver.calcVol()`. Here the `solver` is a member in the PYDAFOAM class for the original version of OpenFOAM. PYDAFOAM also has another variable call `solverAD` which contains the variables for the AD version of OpenFOAM. As mentioned above, `solver` and `solverAD` objects contain independent variables that allow us to run the primal using the original OpenFOAM and run adjoint using the AD version of OpenFOAM.

Add a new objective/constraint function

Steps to add a objective/constraint function (1/3)

If we want to add a new objective function `MyObj` that computes the volume weighted pressure over the entire field, we need to

- First, we copy DAObjFuncForce.C and DAObjFuncForce.H in src/adjoint/DAObjFuncas and rename them as DAObjFuncMyObj.H and DAObjFuncMyObj.C
- Next, replace all "DAObjFuncForce" with "DAObjFuncMyObj" for both DAObjFuncMyObj.H and DAObjFuncMyObj.C
- In DAObjFuncMyObj.H, use `TypeName("myObj");`. Note that this TypeName defines the objFunc "type" in runScript.py. So if you get the "Unknown DAObjFunc type xxxx" error, double check if the TypeName is set for your new objective function.

Steps to add a objective/constraint function (2/3)

- Then, we need to implement the volume weighted pressure computation in the calcObjFunc function in DAObjFuncMyObj.C. This function calculates objFuncVal.

```
// initialize the objective function value
objFuncValue = 0.0;
// get the pressure variable from the mesh database (memory)
const objectRegistry& db = mesh_.thisDb();
const volScalarField& p = db.lookupObject<volScalarField>("p");
// calculate the volume weighted pressure
forAll(p, cellI){
  objFuncValue += p[cellI] * mesh_.V()[cellI];
}
// need to reduce the sum of force across all processors
reduce(objFuncValue, sumOp<scalar>());
```

- Add `DAObjFunc/DAObjFuncMyObj.C` file to line 29 in src/adjoint/Make/files_Incompressible. **Recompile DAFoam.**

Steps to add a objective/constraint function (3/3)

- Add this new objective function to the objFunc key in daOptions from runScript.py

```
"F1": {
    "part1": {
        "type": "myObj",
        "source": "patchToFace",
        "patches": ["walls"],
        "addToAdjoint": True,
    }
},
```

- Run the case with `python runScript.py`, you should see the objective function printed to the screen when solving the primal.
- NOTE : Examples of DAObjFuncMyObj.H and DAObjFuncMyObj.C are in 2022_Summer/examples/codes

Add a new model or boundary condition

Steps to add a new model or boundary condition

If we want to add a new model or boundary condition, we need to

- Put the source codes in to src/adjoint/models
- Include its .C file in src/adjoint/Make/files_Incompressible
- **Recompile DAFoam** and this new model/boundary condition is ready to use in the OpenFOAM layer
- Refer to the existing models for templates

Add a new design variable

Steps to add a new design variable (1/2)

If we want to add a new design variable called `NewDV`, we need to

- Implement calcdRdNewDVTPsi and calcdFdNewDVAD functions in the DASolver class (src/adjoint/DASolver/DASolver.C) to compute the matrix-vector product and partial derivatives for this new design variables. Rerfer to the functions for other design variables for templates
- Follow the steps in Slide 98 and expose this function to the Python layer. **Recompile DAFoam.**
- Add this new design variable as the input to the DAFoamSolver class in mphys_dafoam.py, e.g., follow other design variables in line 240 of mphys_dafoam.py
- Call calcdRdNewDVTPsi in DAFoamSolver class in mphys_dafoam.py, e.g., follow other design variables in line 442 of mphys_dafoam.py

Steps to add a new design variable (2/2)

- Add this new design variable as the input to the DAFoamFunctions class in mphys_dafoam.py, e.g., follow other design variables in line 659 of mphys_dafoam.py
- Call calcdFdNewDVAD in DAFoamFunctions class in mphys_dafoam.py, e.g., follow other design variables in line 787 of mphys_dafoam.py
- Set this new design variables in runScript.py, following other design variables.

Add a new solver

Steps to add a new solver (1/3)

If we want to add a new solver called `DASimpleFoamNew`, we need to

- Copy the DASimpleFoam folder and rename it to DASimpleFoamNew, put the DASimpleFoamNew folder to src/adjoint/DASolver
- Rename DASimpleFoam.C to DASimpleFoamNew.C, rename UEqnSimple.H to UEqnSimpleNew.H. Do the same renaming for all the files in the DASimpleFoamNew folder.
- Replace `DASimpleFoam` with `DASimpleFoamNew` in DASimpleFoamNew.H and DASimpleFoamNew.C. Remember to change `TypeName("DASimpleFoam")` to `TypeName("DASimpleFoamNew")` in DASimpleFoamNew.H.
- Replace `#include "createFieldsSimple.H"` with `#include "createFieldsSimpleNew.H"` in DASimpleFoamNew.C. Do the same renaming for other include files.
- Do some changes to the DASimpleFoamNew.C file to implement your new features

Steps to add a new solver (2/3)

- Copy DAStateInfoSimpleFoam.C and DAStateInfoSimpleFoam.H and rename them to DAStateInfoSimpleFoamNew.C and DAStateInfoSimpleFoamNew.H in src/adjoint/DAStateInfo
- Replace `DAStateInfoSimpleFoam` with `DAStateInfoSimpleFoamNew` in DAStateInfoSimpleFoam.C and DAStateInfoSimpleFoam.H. Remember to change `TypeName("DASimpleFoam")` to `TypeName("DASimpleFoamNew")` in DAStateInfoSimpleFoam.H.
- If the new solver has different state variables and connectivity, change it according in DAStateInfoSimpleFoam.C. Otherwise, keep DAStateInfoSimpleFoam.C unchanged.
- Copy DAResidualSimpleFoam.C and DAResidualSimpleFoam.H and rename them to DAResidualSimpleFoamNew.C and DAResidualSimpleFoamNew.H in src/adjoint/DAResidual

Steps to add a new solver (3/3)

- Replace `DAResidualSimpleFoam` with `DAResidualSimpleFoamNew` in DAResidualSimpleFoam.C and DAResidualSimpleFoam.H. Remember to change `TypeName("DASimpleFoam")` to `TypeName("DASimpleFoamNew")` in DAResidualSimpleFoam.H.
- If you change the primal implementation in DASimpleFoamNew.C, make sure you do the same change for DAResidualSimpleFoamNew.C to make sure the residual is consistent with the primal.
- Add `DASolver/DASimpleFoamNew/DASimpleFoamNew.C`, `DAStateInfo/DAStateInfoSimpleFoam.C`, and `DAResidual/DAResidualSimpleFoam.C` to src/adjoint/Make/files_Incompressible. **Recompile DAFoam.**
- Append the new solver's name to line 801 of pyDAFoam.py
- Set the solverName to `DASimpleFoamNew` in runScript.py to run the case with the new solver

Thank you !