DAFoam Workshop 2025

v4.0.2

Ping He

August 14, 2025

Objectives

After this workshop, you should be able to

- Identify the new APIs in runScript.py in DAFoam v4
- Get familiar with the new code structure in DAFoam v4
- Run tutorials and add new features to DAFoam v4

Outline

Part 1 : User-Focused Features

- Overview of new run scripts, APIs, and solver options
- Walkthrough of selected tutorials

Part 2 : Developer-Focused Updates

- Introduction to the updated code structure
- Examples of extending DAFoam (e.g., adding new design variables)

Note : We assume you are familiar with DAFoam v3 and will mainly focus on the new features in v4.

Part 1 : User-Focused Features

Objective : let the DAFoam APIs be more consistent with the OpenMDAO standard.

New interfaces in v4 runScript.py

DAFoam v2 scripts are completely deprecated, but v3 scripts can be used with minor changes. Download the DAFoam workshop repo [Link] and go to workshops/2025_Summer/examples/NACA0012. Let us compare runScript_v3.py and runScript_v4.py.

- The `-task` flag is changed to be consistent with OpenMDAO
- `objFunc` is replaced with `function` in `daOption`.
- `addToAdjoint` and `parts` are no longer needed in `function`.
- `alphaName` is replaced with `patchVelocityInputName` for the angle of attack definition for force (e.g., CD and CL) `function`.
- `designVar` is replaced with `inputInfo` in `daOption`.
- The aoa function is defined in `inputInfo` instead of `configure`
- A new way to define shape for 2D airfoil using `non_addShapeFunctionDV`.

Details of the new `inputInfo` key in `daOptions`

`inputInfo` defines the input variables for a component in OpenMDAO. Let us check runScript_test_inputInto.py in examples/NACA0012

- Each key in `inputInfo` defines an input variable for a component, specified by the `components` key. Note : one input can be connected to multiple components.
- The name of the key in `inputInfo` is the name of the input for that component. Run the script and check the N2 diagram in mphys_n2.html. If you have permission issues, run : `chmod -R 750 *`
- The `type` key defines the type of this input. Check all available input types from examples/dafoam-4.0.2/src/adjoint/DAInput/*.H
- Each input type has its own customized keys, such as `patches`, `fieldName`, and `distributed`.
- Once an input is defined in `inputInfo`, we can connect dvs.output to it. Check line 163 in runScript_test_inputInto.py
- NOTE : Carefully define the inputs to avoid conflicts !

New interfaces for MDO problems

Let us comopare runScript_v3.py and runScript_v4.py in examples/MACH_Tutorial_Wing

- The `couplingInfo` key is deprecated, and the MDO coupling is now defined in `inputInfo` and `outputInfo`.
- The aerostructural problem has a `forceCoupling` component, which replaces the `getForce` component in v3. This component uses the volume coordinates and states as input and outputs the surface force (f_aero). So we need to define this f_aero as the output in `outputInfo`.
- The `forceCouplingOutput` type in `outputInfo` will tell mphys_dafoam that we have an aerostructural problem, and it will then trigger the aero-structural related codes in mphys_dafoam.py.
- The `forceCoupling` component will automatically get the volume coordinates and states from the solver component, so we don't need to manually define these inputs in `inputInfo`.

New interfaces for unsteady problems

Let us check runScript.py in examples/Cylinder

- We treat the unsteady flow/adjoint solver as an explicit component in OpenMDAO (line 116).
- The inputs for this explicit component are defined by `add_design_var` and the outputs are defined in `unsteadyCompOutput`. Here, each key (e.g., obj) in `unsteadyCompOutput` is the actual function used in optimization. Their sub-keys (e.g., CD and CL) are the functions defined in `daOption-function`. If one defines more than one sub-key, these functions will be added together.
- Define unsteady adjoint-related parameters in `unsteadyAdjoint`.
- All components are promoted, so you should directly use `shape`, instead of `dvs.shape`.

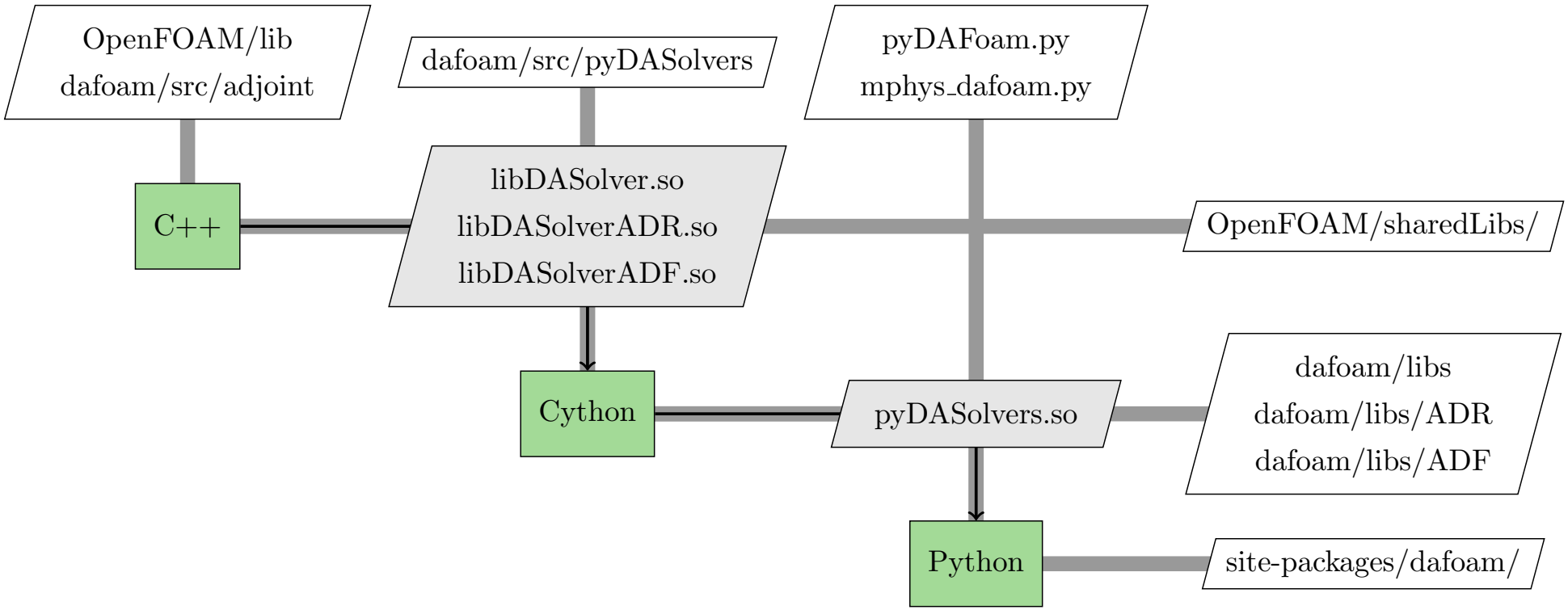
Part 2 : Developer-Focused Updates

Objective : make the DAFoam development easier.

Improvements in DAFoam v4 development

	v3	v4
First build time	$>$30 mins	$\sim$10 mins
Re-build time	$>$30 mins	$\ll$10 mins
Build command	Multiple commands	One command
Parallel build	Fixed 4 cores	Max available cores
DAFoam libs	Three libs (e.g., incomp)	One lib
Add new DVs	Multiple new functions	One new function
Add new partials	Ad-hoc	Standardized
Add new in/out	Ad-hoc	Standardized

DAFoam layers : C++, Cython, and Python



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

New features in v4 layer interaction

- We can directly pass a Python array from Python to the C++ layer. There is no need to use PETSc vectors as inputs/outputs. [Link to an example].
- The Python layer can call the C++ layer's members, AND the C++ layer can also call the Python layer's members. [Link to an example].
- The C++ layer no longer separate incompressible, compressible, and solid libs. All these flow conditions are now compiled into one library.

Structure of the DAFoam Python layer (1/2)

- The main entry point in the Python layer is the DAFoamBuilder class in dafoam/mphys/mphys_dafoam.py. The self.DASolver = PYDAFOAM call [link] initializes the DASolver, in which all the OpenFOAM flow fields and meshes are initialized.
- In addition to initializing OpenFOAM fields, DAFoamBuilder also exposes standard functions to MPhys for MDO coupling, such as get_solver(), get_coupling_group_subsystem().
- The get_solver() call in DAFoamBuilder just returns the DASolver object initialized above.
- The get_coupling_group_subsystem() call in DAFoamBuilder initializes the DAFoamGroup (an OpenMDAO group), which includes DAFoamSolver, DAFoamWarper, and optionally other MDO coupling components such as DAFoamForces.
- DAFoamSolver is an implicit component that solves the primal nonlinear equations (e.g., flow simulations), the adjoint linear system, and partial derivatives such as dRdX.

Structure of the DAFoam Python layer (2/2)

- The DAFoamWarper (IDWarp) in DAFoamGroup is an explicit component for volume mesh deformation. It does in-solver mesh warping for aero-structural coupling.
- The geometry parameterization, e.g., pyGeo, is defined in the runScript.py, instead of in DAFoamGroup
- The get_mesh_coordinate_subsystem() call in DAFoamBuilder just returns the design surface coordinates, which will be used as the initial design geometry
- The get_pre_coupling_subsystem() call in DAFoamBuilder calls DAFoamPrecouplingGroup, which is also the DAFoamWarper component. The DAFoamPrecouplingGroup is mainly used to conduct geometry design variable changes, e.g., twist change.
- The get_post_coupling_subsystem() call in DAFoamBuilder returns the DAFoamFunction component, which computes the values and derivatives of functions defined in runScript.py

How to add new features in DAFoam v4 ?

Outline

- Change existing functions/variables in the C++ layer
- Add a new parameter in DAOption
- Add a new C++ function and expose it to the Python layer
- Add a new objective/constraint function
- Add a new design variable or input
- Add new coupling variables for MDO
- Add a new boundary condition
- Add a new turbulence model
- Add a new solver

Let us build DAFoam

- Open a terminal and cd to workshops/2025_Summer
- Create a DAFoam Docker container v4.0.2. Refer to [this page] for the commands.
- In the container, cd to /home/dafoam/repos/dafoam, and run

```
./Allmake
```

- That is it. The Docker image has a pre-compiled dafoam v4.0.2, so the above command will only verify the pre-compiled version and will take less than 30 seconds. At the end of the terminal output, you should see : "Build Successful !"
- NOTE : If you modify DAFoam and want to rebuild it, simply run the above command again. DO NOT run Allclean. DAFoam will automatically compile **only** the new files you change. This saves a lot of time !

Change existing functions/variables in the C++ layer

- cd into $HOME/dafoam/repos/dafoam/
- Use the vi command to open the src/adjoint/DASolver/DASimpleFoam/DASimpleFoam.C file
- Add a print statement at the end of the solverPrimal function

```
Info << "This is my first change!" << endl;
```

- Recompile DAFoam (make sure you see "Build Successful !") :

```
./Allmake
```

- cd into $HOME/mount/examples/NACA0012
- Run :

```
chmod -R 750 *
./preProcessing.sh
python runScript_v4.py -task=run_model
```

- Verify if you see the new print statement in the terminal output

Add a new parameter in DAOption (1/2)

- cd into $HOME/dafoam/repos/dafoam/
- Use the vi command to open the dafoam/pyDAFoam.py file and add this to the __init__ function of the DAOPTION class :

```
self.testOption = 3.0
```

- NOTE : We need to give default values for str, int, float, and list options defined here (this helps DAFoam determine the datatype). For dict options, we can leave it blank.
- Open src/adjoint/DASolver/DASimpleFoam/DASimpleFoam.C and add this line at the end of the solverPrimal function

```
scalar testOption =
    daOptionPtr_->getOption<scalar>("testOption");
Info << "My test option is: " << testOption << endl;
```

- Recompile DAFoam

Add a new parameter in DAOption (2/2)

- In $HOME/mount/examples/NACA0012, run

```
python runScript_v4.py -task=run_model
```

- Verify if you see the new testOption value (3.0 ; default value) in the terminal output
- You can also change the default value in the runScript.py. For example, you can add the following line to the daOption dict in $HOME/mount/examples/NACA0012/runScript_v4.py.

```
"testOption": 15.2,
```

- Run the following again and see if the testOption's value is changed to 15.2 in the terminal output.

```
python runScript_v4.py -task=run_model
```

- NOTE : The daOption defined in runScript.py will be passed to the C++ layer **only once** (when the DAFoam is initialized). If you need to pass the daOption from Python to C++ again, you need to manually call updateDAOption(). Here is [an example].

Add a C++ func and expose it to the Python layer (1/2)

- cd into $HOME/dafoam/repos/dafoam/, open src/adjoint/DASolver/DASolver.H, and add a public member :

```
void printMeshSize()
{
    label nCells = meshPtr_->nCells();
    Info << "Mesh cells: " << nCells << endl;
}
```

- Open src/pyDASolvers/DASolvers.H, and add a public member :

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

- Open src/pyDASolvers/pyDASolvers.pyx, and add this to `cppclass DASolvers` :

```
void printMeshSize()
```

Add a C++ func and expose it to the Python layer (2/2)

- Open src/pyDASolvers/pyDASolvers.pyx, and add this to `cdef class pyDASolvers:`

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

- Open dafoam/pyDAFoam.py and add this line to the end of the __call__ function

```
self.solver.printMeshSize()
```

- Recompile DAFoam
- In $HOME/mount/examples/NACA0012, run the following, and you should see the printed statement from the terminal output

```
python runScript_v4.py -task=run_model
```

- Alternatively, you can call this new function in runScript_v4.py by adding this line after `prob.run_model()`

```
print("Print in the runScript")
prob.model.scenario1.coupling.solver.DASolver.solver.
    printMeshSize()
```

Add a new objective/constraint function (1/2)

- cd into src/adjoint/DAFunction
- Copy DAFunctionPatchMean.C into DAFunctionTestFunc.C and copy DAFunctionPatchMean.H into DAFunctionTestFunc.H
- Open DAFunctionTestFunc.C and replace all "DAFunctionPatchMean" with "DAFunctionTestFunc". Do the same for DAFunctionTestFunc.H
- (Important !) Replace TypeName("patchMean") with TypeName("testFunc") in DAFunctionTestFunc.H. Here, TypeName defines the function name we will use in runScript.py
- Modify DAFunctionTestFunc.C and DAFunctionTestFunc.H as needed to compute and return the function value in calcFunction(). Ensure that the function value is summed across all processors. (Call the `reduce` command).
- Add a print statement at the end of the calcFunction() function, which will be used to verify the implementation.

```
Info << "This is my test function " << endl;
```

- Add DAFunction/DAFunctionTestFunc.C to src/adjoint/Make/files

Add a new objective/constraint function (2/2)

- Recompile DAFoam
- In $HOME/mount/examples/NACA0012/runScript_v4.py, add these lines to the `function` dict from daOption.

```
"Test": {
    "type": "testFunc",
    "source": "patchToFace",
    "patches": ["wing"],
    "varName": "p",
    "varType": "scalar",
    "index": 0,
    "scale": 1.0},
```

- Then, run the following, and you should see the function value and custom print statement in the terminal output.

```
python runScript_v4.py -task=run_model
```

- NOTE : DAFoam will take care of the derivative computation for this new function automatically.

Add a new design variable or input (1/2)

- cd into src/adjoint/DAInput
- Copy DAInputPatchVelocity.C into DAInputPatchVx.C and copy DAInputPatchVelocity.H into DAInputPatchVx.H
- Open DAInputPatchVx.C and replace all "DAInputPatchVelocity" with "DAInputPatchVx". Do the same for DAInputPatchVx.H
- (Important !) Replace TypeName("patchVelocity") with TypeName("patchVx") in DAInputPatchVx.H.
- Modify DAInputPatchVx.C and DAInputPatchVx.H as needed to assign the Vx value from input[0] to the OpenFOAM's U field.
- Change "return 2" to "return 1" for the size() func in DAInputPatchVx.H
- Add DAInput/DAInputPatchVx.C to src/adjoint/Make/files
- Recompile DAFoam
- NOTE : that is all the changed needed to add a new design variable, the rest of the operations, such as computing the partial derivatives dF/dX or dR/dX, will be automatically handled

Add a new design variable or input (1/2)

- In $HOME/mount/examples/NACA0012/runScript.py, add these lines to the `inputInfo` dict from daOption.

```
"vx_in": {
   "type": "patchVx",
   "patches": ["inlet"],
   "components": ["solver", "function"]},
```

- In the same runScript, add these lines to the `configure` function.

```
self.dvs.add_output("vx_in", val=np.array([10.0]))
self.connect("vx_in", "scenario1.vx_in")
self.add_design_var("vx_in", upper=50.0, scaler=1.0)
```

- Finally, run the following, and verify the inlet velocity in Paraview

```
python runScript.py -task=run_model
```

- You can also change the default value from 10 to 20 in the `self.dvs.add_output` call, run the script again, and then verify the new inlet value in Paraview

Add a new boundary condition

- Copy a template of the boundary condition .C and .H files from OpenFOAM and paste them to src/adjoint/DAMisc/.
- Modify the .C and .H file as needed. Remember to change TypeName("your_new_bc_name") in the .H file to a custom name that has not been used in OpenFOAM.
- Add the .C file's path to src/adjoint/Make/files
- Recompile DAFoam
- You can use your new boundary condition now. Remember to use your custom name.

Add a new turbulence model (1/3)

- Copy the turbulence model .C and .H files from OpenFOAM and paste (use custom names like mySA.C and mySA.H) them to src/newTurbModels/models.
- Replace the old turbulence class name with your new turbulence name. Don't forget to assign a custom name for the TypeName in the .H file.
- The turbulence model here is used to initialize the parent class and turbulence variable. So we need to delete all intermediate functions or variables and keep only these things : (1) The initialization of eddyViscosity and turbulence variables, such as nuTilda_, k_, omega_. (2) All the virtual function, (3) An empty correct() function.
- In src/newTurbModels/compressible, copy makeSpalartAllmarasFv3Compressible.C and paste it into sth like makeMySAModelCompressible.C. Then, in makeMySAModelCompressible.C, replace SpalartAllmarasFv3 with your custom turbulence name. Finally, add makeMySAModelCompressible.C to src/newTurbModels/compressible/Make/files

Add a new turbulence model (2/3)

- Do the same to create makeMySAModelIncompressible.C file in src/newTurbModels/incompressible
- In addition to the dummy turbulence initialization class above, we need to add the actual turbulence model computation to the DATurbulenceModel class.
- First, in /src/adjoint/DAModel/DATurbulenceModel, copy DASpalartAllmaras.C and DASpalartAllmaras.H and paste them to custom names.
- Replace the old turbulence class name with your new turbulence name. Don't forget to assign a custom name for the TypeName in the .H file.
- Modify the the custom .C and .H files as needed. Follow the code structure from existing model implementations. The actual solution of the turbulence model should be implemented in the calcResiduals() function.
- Add your custom .C file's path to src/adjoint/Make/files
- Recompile DAFoam, and your new turbulence model is ready to use.

Add a new turbulence model (3/3)

A few notes :

- If a turbulence model is already in OpenFOAM, you don't need to add it to src/newTurbulenceModel ; however you still need to add it to src/adjoint/DAModel/DATurbulenceModel.
- The actual computation of the turbulence model variable is done by the DATurbulenceModel class, check here for [an example].

Add new coupling variables for MDO (1/2)

- In MDO, we often need to extract boundary variables from the solver and pass these variables to another solver. Examples include the surface force for aero-structural coupling and the surface temperature for aero-thermal coupling.
- Here we go over the process of extracting surface nodal force for aero-structural coupling.
- First, we need to add a child class (DAOutputForceCoupling) to src/adjoint/DAOutput.
- In DAOutputForceCoupling.H, we need to define the size of the surface nodal force, and set distributed to 1, meaning each processor owns and extracts their own surface nodal force values.
- In DAOutputForceCoupling.C, we need to implement how to extract the surface nodal force values, and then assign them to the `output` variable in the `run` function.
- This output is ready to use in the Python layer.

Add new coupling variables for MDO (2/2)

- Next, we need to add a new explicit component called DAFoamForces to dafoam/mphys/mphys_dafoam.py. This component's inputs are the states and volume coordinate, and the output is the surface nodal forces.
- The compute method in the DAFoamForces component calls the calcOutput() function to extract the nodal forces and assign them to the forces array. The outputName and outputType are defined in outputInfo in runScript.py
- The compute_jacvec_product method in DAFoamForces component calls the calcJacTVecProduct() function to compute partial derivatives of all the outputs wrt all the inputs.
- We then need to add this new coupling component to the DAFoamGroup group in dafoam/mphys/mphys_dafoam.py. Check here for [an example] of force component in aero-structural coupling.
- You also need to implement how the output (e.g., force) is connected to other solver components in mphys

Add a new solver

There are three main steps in adding a new solver

- Create a new child class in srd/adjoint/DAStateInfo, which defines all the state variables and their connection.
- Create a new child class in srd/adjoint/DAResidual, which implements the residual computation for this new solver
- Create a new child class in srd/adjoint/DASolver, which implement the primal solution, the solve_nonlinear() call in DAFoamSolver

You can take the existing solver's .C and .H files as reference to add a new solver.