DAFoam Documentation

Release v1.0

Ping He

Jul 25, 2020

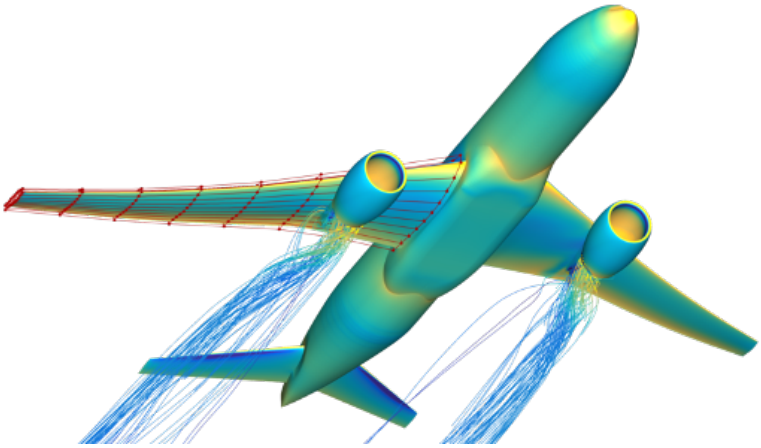
Contents

CHAPTER 1

DAFoam: Discrete Adjoint with OpenFOAM

DAFoam contains a suite of discrete adjoint solvers for OpenFOAM. These adjoint solvers run as standalone executives to compute derivatives. DAFoam also has a Python interface that allows the adjoint solvers to interact with external modules for high-fidelity design optimization using the MACH framework. DAFoam has the following features:

- It implements an efficient discrete adjoint approach with competitive speed, scalability, accuracy, and compatibility.
- It allows rapid discrete adjoint development for any steady-state OpenFOAM solvers with modifying only a few hundred lines of source codes.
- It supports design optimizations for a wide range of disciplines such as aerodynamics, heat transfer, structures, hydrodynamics, and radiation.



The DAFoam repository comprises of five main directories, and the source code is available on GitHub.

- applications: adjoint solvers and utilities
- doc: documentation
- python: python interface to other optimization packages

- src: the core DAFoam libraries
- tutorials: sample optimization setup for each adjoint solver

Contents:

1.1 Download

The current stable version of DAFoam is v1.1. See the changes log from here.

There are two options to run DAFoam: **pre-compiled package** and **source code**. If you are running DAFoam for the first time, we recommend using the pre-compiled version, which supports Linux (Ubuntu, Fedora, CentOS, etc), MacOS, and Windows systems. For production runs on an HPC system, you need to compile DAFoam from the source.

- **Pre-compiled package**

  The pre-compiled package is available on Docker Hub. Before downloading the pre-compiled package, you need to install **Docker**. Follow the installation instructions for Ubuntu, Fedora, CentOS, MacOS, and Windows.

  For example, on Ubuntu, you can install the latest Docker by running this command in the terminal:

```
sudo apt-get remove docker docker-engine docker.io containerd runc && sudo␣
↪apt-get update && sudo apt-get install apt-transport-https ca-certificates␣
↪curl gnupg-agent software-properties-common -y && curl -fsSL https://
↪download.docker.com/linux/ubuntu/gpg | sudo apt-key add - && sudo add-apt-
↪repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
↪$(lsb_release -cs) stable" && sudo apt-get update && sudo apt-get install␣
↪docker-ce -y
```

  Then you need to add your user name to the docker group by:

```
sudo usermod -aG docker $USER
```

  After this, you need to logout and re-login your account to make the usermod command effective. Once done, verify the docker installation by:

```
docker --version
```

  You should be able to see your installed Docker version. Note that different operating systems have very different Docker installation process, refer to the above links for more details.

  Once the Docker is installed and verified, run this command from the terminal:

```
docker run -it --rm -u dafoamuser -v $HOME:/home/dafoamuser/mount -w /home/
↪dafoamuser/mount dafoam/opt-packages:v1.1 bash
```

  It will first download the pre-compiled package (v1.1) from the Docker Hub if it has not been downloaded. Then it will start a Docker container (a light-weight virtual machine), mount your local computer's home directory to the container's `mount` directory, login to `mount` as dafoamuser, and set the relevant DAFoam environmental variables. Now you are ready to run DAFoam tutorials. Refer to *Tutorials* for more details.

- **Source code**

  The DAFoam source code is available at https://github.com/mdolab/dafoam. DAFoam depends on multiple prerequisites and packages. Refer to *Installation* for installation instructions.

1.2 Installation

This section assumes you want to compile the DAFoam optimization package (v1.1) from the source on a Linux system. If you use the pre-compiled version, skip this section.

The DAFoam package can be compiled with various dependency versions. Here we elaborate on how to compile it on Ubuntu 18.04 using the dependencies shown in the following table.

Ubuntu	Compiler	OpenMPI	mpi4py	PETSc	petsc4py	CGNS	python	numpy	scipy	Cython
18.04	gcc/7.5	1.10.7	3.0.2	3.11.0	3.11.0	3.3.0	3.6.5	1.14.3	1.1.0	0.28.2

To compile, you can just copy the code blocks in the following steps and run them on the terminal. **NOTE:** if a code block contains multiple lines, copy all the lines and run them on the terminal. Make sure each step run successfully before going to the next one. The entire compilation may take a few hours, the most time-consuming part is OpenFOAM.

1. **Prerequisites**. Run this on terminal:

```
sudo apt-get update && \
sudo apt-get install -y build-essential flex bison cmake zlib1g-dev libboost-
↪system-dev libboost-thread-dev libreadline-dev libncurses-dev libxt-dev qt5-
↪default libqt5x11extras5-dev libqt5help5 qtdeclarative5-dev qttools5-dev␣
↪libqtwebkit-dev freeglut3-dev libqt5opengl5-dev texinfo  libscotch-dev libcgal-
↪dev gfortran swig wget git vim cmake-curses-gui libfl-dev apt-utils --no-
↪install-recommends
```

2. **Python**. Install Anaconda3-5.2.0:

```
mkdir -p $HOME/packages && \
cd $HOME/packages && \
wget https://repo.anaconda.com/archive/Anaconda3-5.2.0-Linux-x86_64.sh --no-check-
↪certificate && \
chmod 755 Anaconda3-5.2.0-Linux-x86_64.sh && \
./Anaconda3-5.2.0-Linux-x86_64.sh -b -p $HOME/packages/anaconda3 && \
echo '# Anaconda3' >> $HOME/.bashrc && \
echo 'export PATH=$HOME/packages/anaconda3/bin:$PATH' >> $HOME/.bashrc && \
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/packages/anaconda3/lib' >>
↪$HOME/.bashrc && \
. $HOME/.bashrc
```

3. **OpenMPI**. Append relevant environmental variables by running:

```
echo '# OpenMPI-1.10.7' >> $HOME/.bashrc && \
echo 'export MPI_INSTALL_DIR=$HOME/packages/openmpi-1.10.7/opt-gfortran' >> $HOME/
↪.bashrc && \
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MPI_INSTALL_DIR/lib' >> $HOME/.
↪bashrc && \
echo 'export PATH=$MPI_INSTALL_DIR/bin:$PATH' >> $HOME/.bashrc && \
. $HOME/.bashrc
```

Then, configure and build OpenMPI:

```
cd $HOME/packages && \
wget https://download.open-mpi.org/release/open-mpi/v1.10/openmpi-1.10.7.tar.gz --
↪no-check-certificate && \
```

(continues on next page)

(continued from previous page)

```
tar -xvf openmpi-1.10.7.tar.gz && \
cd openmpi-1.10.7 && \
./configure --prefix=$MPI_INSTALL_DIR && \
make all install
```

Append one more relevant environmental variable by running:

```
echo 'export LD_PRELOAD=$MPI_INSTALL_DIR/lib/libmpi.so' >> $HOME/.bashrc && \
. $HOME/.bashrc
```

To verify the installation, run:

```
mpicc -v
```

You should see the version of the compiled OpenMPI.

Finally, install mpi4py-3.0.2:

```
cd $HOME/packages && \
wget https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-3.0.2.tar.gz --no-check-
↪certificate && \
tar -xvf mpi4py-3.0.2.tar.gz && \
cd mpi4py-3.0.2 && \
rm -rf build && \
python setup.py install --user
```

4. **Petsc**. Append relevant environmental variables by running:

```
echo '# Petsc-3.11.0' >> $HOME/.bashrc && \
echo 'export PETSC_DIR=$HOME/packages/petsc-3.11.0' >> $HOME/.bashrc && \
echo 'export PETSC_ARCH=real-opt' >> $HOME/.bashrc && \
echo 'export PATH=$PETSC_DIR/$PETSC_ARCH/bin:$PATH' >> $HOME/.bashrc && \
echo 'export PATH=$PETSC_DIR/$PETSC_ARCH/include:$PATH' >> $HOME/.bashrc && \
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PETSC_DIR/$PETSC_ARCH/lib' >>
↪$HOME/.bashrc && \
echo 'export PETSC_LIB=$PETSC_DIR/$PETSC_ARCH/lib' >> $HOME/.bashrc
. $HOME/.bashrc
```

Then, configure and compile:

```
cd $HOME/packages && \
wget http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-3.11.0.tar.gz --no-
↪check-certificate && \
tar -xvf petsc-3.11.0.tar.gz && \
cd petsc-3.11.0 && \
sed -i 's/ierr = MPI_Finalize();CHKERRQ(ierr);/\/\/ierr = MPI_Finalize();
↪CHKERRQ(ierr);/g' src/sys/objects/pinit.c && \
./configure --PETSC_ARCH=real-opt --with-scalar-type=real --with-debugging=0 --
↪with-mpi-dir=$MPI_INSTALL_DIR --download-metis=yes --download-parmetis=yes --
↪download-superlu_dist=yes --download-fblaslapack=yes --with-shared-
↪libraries=yes --with-fortran-bindings=1 --with-cxx-dialect=C++11 && \
make PETSC_DIR=$HOME/packages/petsc-3.11.0 PETSC_ARCH=real-opt all
```

NOTE: The above `sed` command comments out line 1367 in src/sys/objects/pinit.c to prevent Petsc from conflicting with OpenFOAM MPI_Finalize.

Finally, install petsc4py-3.11.0:

```
cd $HOME/packages && \
wget https://bitbucket.org/petsc/petsc4py/downloads/petsc4py-3.11.0.tar.gz --no-
↪check-certificate && \
tar -xvf petsc4py-3.11.0.tar.gz && \
cd petsc4py-3.11.0 && \
rm -rf build && \
python setup.py install --user
```

5. **CGNS**. Append relevant environmental variables by running:

```
echo '# CGNS-3.3.0' >> $HOME/.bashrc && \
echo 'export CGNS_HOME=$HOME/packages/CGNS-3.3.0/opt-gfortran' >> $HOME/.bashrc &&
↪ \
echo 'export PATH=$PATH:$CGNS_HOME/bin' >> $HOME/.bashrc && \
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CGNS_HOME/lib' >> $HOME/.bashrc &&␣
↪\
. $HOME/.bashrc
```

Then, configure and compile:

```
cd $HOME/packages && \
wget https://github.com/CGNS/CGNS/archive/v3.3.0.tar.gz --no-check-certificate &&␣
↪\
tar -xvaf v3.3.0.tar.gz && \
cd CGNS-3.3.0 && \
mkdir -p build && \
cd build && \
cmake .. -DCGNS_ENABLE_FORTRAN=1 -DCMAKE_INSTALL_PREFIX=$CGNS_HOME -DCGNS_BUILD_
↪CGNSTOOLS=0 && \
make all install
```

6. **MACH framework**. First create a `repos` folder and setup relevant environmental variables:

```
echo '# Python Path' >> $HOME/.bashrc && \
echo 'export PYTHONPATH=$PYTHONPATH:$HOME/repos' >> $HOME/.bashrc
. $HOME/.bashrc && \
mkdir -p $HOME/repos
```

Then run:

```
cd $HOME/repos && \
git clone https://github.com/mdolab/baseclasses -b v1.1.0 && \
cd $HOME/repos && \
git clone https://github.com/mdolab/pygeo -b v1.1.0 && \
cd $HOME/repos && \
git clone https://github.com/mdolab/multipoint -b v1.1.0 && \
cd $HOME/repos && \
git clone https://github.com/mdolab/pyspline -b v1.1.0 && \
cd pyspline && \
cp config/defaults/config.LINUX_GFORTRAN.mk config/config.mk && \
make && \
cd $HOME/repos && \
git clone https://github.com/mdolab/pyhyp -b v2.1.0 && \
cd pyhyp && \
cp -r config/defaults/config.LINUX_GFORTRAN_OPENMPI.mk config/config.mk && \
make && \
cd $HOME/repos && \
```

(continues on next page)

(continued from previous page)

```
git clone https://github.com/mdolab/cgnsutilities -b v2.1.0 && \
cd cgnsutilities && \
cp config.mk.info config.mk && \
make && \
echo '# cgnsUtilities' >> $HOME/.bashrc && \
echo 'export PATH=$PATH:$HOME/repos/cgnsutilities/bin' >> $HOME/.bashrc && \
cd $HOME/repos && \
git clone https://github.com/mdolab/idwarp && \
cd idwarp && \
git checkout f854b65 && \
cp -r config/defaults/config.LINUX_GFORTRAN_OPENMPI.mk config/config.mk && \
make && \
cd $HOME/repos && \
git clone https://github.com/mdolab/pyoptsparse -b v2.1.0 && \
cd pyoptsparse && \
pip install -r requirements.txt && \
rm -rf build && \
python setup.py install --user
```

7. **OpenFOAM**. Compile OpenFOAM-v1812 by running:

```
mkdir -p $HOME/OpenFOAM && \
cd $HOME/OpenFOAM && \
wget https://sourceforge.net/projects/openfoamplus/files/v1812/OpenFOAM-v1812.tgz/
↪download  --no-check-certificate -O OpenFOAM-v1812.tgz && \
wget https://sourceforge.net/projects/openfoamplus/files/v1812/ThirdParty-v1812.
↪tgz/download  --no-check-certificate -O ThirdParty-v1812.tgz && \
tar -xvf OpenFOAM-v1812.tgz && \
tar -xvf ThirdParty-v1812.tgz && \
cd $HOME/OpenFOAM/OpenFOAM-v1812 && \
wget https://github.com/mdolab/dafoam/releases/download/v1.1.0/UPstream.C --no-
↪check-certificate && \
mv UPstream.C src/Pstream/mpi/UPstream.C && \
. etc/bashrc && \
export WM_NCOMPPROCS=4 && \
./Allwmake
```

NOTE: In the above command, we replaced the OpenFOAM-v1812's built-in UPstream.C file with a customized one because we need to prevent OpenFOAM from calling the MPI_Finialize function when wrapping OpenFOAM functions using Cython.

NOTE: The above command will compile OpenFOAM using 4 CPU cores. If you want to compile OpenFOAM using more cores, change the `WM_NCOMPPROCS` parameter before running `./Allwmake`

Finally, verify the installation by running:

```
simpleFoam -help
```

It should see some basic information of OpenFOAM

8. **DAFoam** and **pyOFM**. First compile pyOFM:

```
cd $HOME/repos && \
git clone https://github.com/mdolab/pyofm -b v1.1.2 && \
cd pyofm && \
. $HOME/OpenFOAM/OpenFOAM-v1812/etc/bashrc && \
make
```

Then, compile DAFoam by running:

```
cd $HOME/repos && \
git clone https://github.com/mdolab/dafoam -b v1.1.2 && \
. $HOME/OpenFOAM/OpenFOAM-v1812/etc/bashrc && \
cd $HOME/repos/dafoam && \
make
```

Finally, run the regression test:

```
cd $HOME/repos/dafoam/python/reg_tests && \
rm -rf input.tar.gz && \
wget https://github.com/mdolab/dafoam/raw/master/python/reg_tests/input.tar.gz --
↪no-check-certificate && \
tar -xvf input.tar.gz && \
python run_reg_tests.py
```

The regression tests should take less than 30 minutes. You should see something like:

```
dafoam buoyantBoussinesqSimpleDAFoam: Success!
dafoam buoyantSimpleDAFoam: Success!
dafoam calcDeltaVolPointMat: Success!
dafoam calcSensMap: Success!
dafoam rhoSimpleCDAFoam: Success!
dafoam rhoSimpleDAFoam: Success!
dafoam simpleDAFoam: Success!
dafoam simpleTDAFoam: Success!
dafoam solidDisplacementDAFoam: Success!
dafoam turboDAFoam: Success!
```

You should see the first "Success" in less than 5 minute. If any of these tests fails or they take more than 30 minutes, check the error in the generated dafoam_reg_* files. Make sure all the tests pass before running DAFoam.

In summary, here is the folder structures for all the installed packages:

```
$HOME
  - OpenFOAM
    - OpenFOAM-v1812
    - ThirdParty-v1812
  - packages
    - anaconda3
    - CGNS-3.3.0
    - mpi4py-3.0.2
    - petsc-3.11.0
    - petsc4py-3.11.0
  - repos
    - baseclasses
    - cgnsutilities
    - dafoam
    - idwarp
    - multipoint
    - pygeo
```

(continues on next page)

(continued from previous page)

```
    - pyhyp
    - pyofm
    - pyoptsparse
    - pyspline
```

Here is the DAFoam related environmental variable setup that should appear in your bashrc file:

```
# OpenMPI-1.10.7
export MPI_INSTALL_DIR=$HOME/packages/openmpi-1.10.7/opt-gfortran
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MPI_INSTALL_DIR/lib
export PATH=$MPI_INSTALL_DIR/bin:$PATH
export LD_PRELOAD=$MPI_INSTALL_DIR/lib/libmpi.so
# PETSC
export PETSC_DIR=$HOME/packages/petsc-3.11.0
export PETSC_ARCH=real-opt
export PATH=$PETSC_DIR/$PETSC_ARCH/bin:$PATH
export PATH=$PETSC_DIR/$PETSC_ARCH/include:$PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PETSC_DIR/$PETSC_ARCH/lib
export PETSC_LIB=$PETSC_DIR/$PETSC_ARCH/lib
# CGNS-3.3.0
export CGNS_HOME=$HOME/packages/CGNS-3.3.0/opt-gfortran
export PATH=$PATH:$CGNS_HOME/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CGNS_HOME/lib
# Python Path
export PYTHONPATH=$PYTHONPATH:$HOME/repos
# cgnsUtilities
export PATH=$PATH:$HOME/repos/cgnsutilities/bin
```

1.3 Tutorials

There are multiple optimization cases in the **tutorials** folder.

1.3.1 Aerodynamics

List of cases:

NACA0012 airfoil incompressible

This is an aerodynamic shape optimization case for an airfoil at low speed. The summary of the case is as follows:

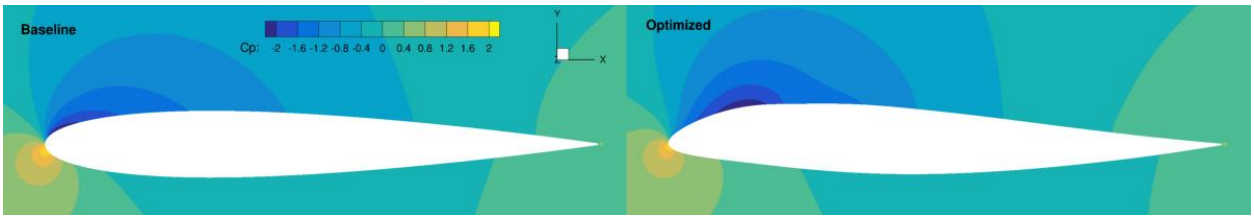
Case: Airfoil aerodynamic optimization
Geometry: NACA 0012
Objective function: Drag coefficient
Design variables: 40 FFD points moving in the y direction, one angle of attack
Constraints: Symmetry, volume, thickness, and lift constraints (total number: 123)
Mach number: 0.1
Reynolds number: 2.3 million
Mesh cells: 8.6K
Adjoint solver: simpleDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

```
./Allrun.sh 1
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 6 steps, see the following figure. The baseline design has C_D=0.01316, C_L=0.3750, and the optimized design has C_D=0.01273, C_L=0.3750.



Now we elaborate on the details of optimization configurations. The Allrun.sh script has three sections. In the first section, we check if the OpenFOAM environments are loaded:

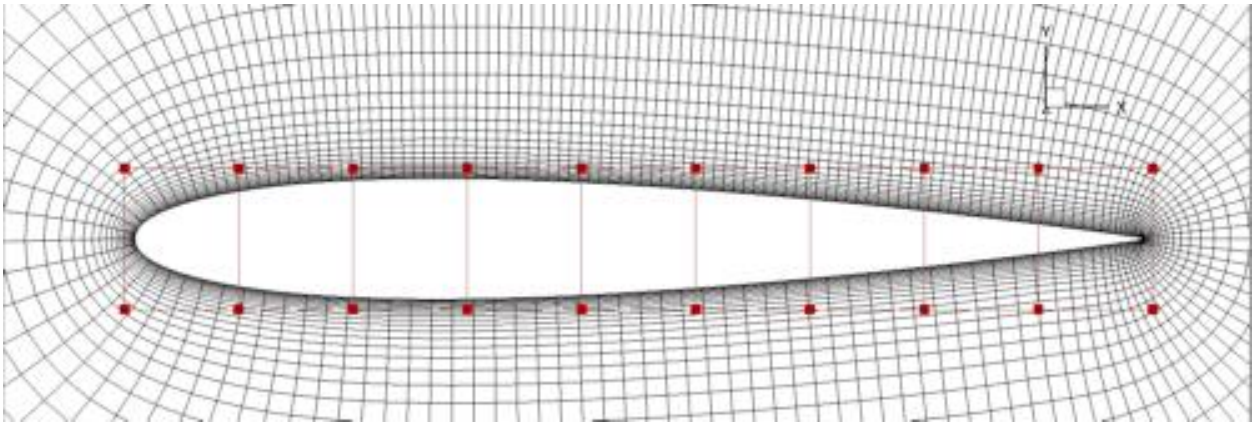
```
if [ -z "$WM_PROJECT" ]; then
  echo "OpenFOAM environment not found, forgot to source the OpenFOAM bashrc?"
  exit
fi

if [ -z "$1" ]; then
  echo "No argument supplied!"
  echo "Example: ./Allrun.sh 1"
  echo "This will run the case using 1 CPU core."
  exit
fi
```

In the second section, we generate a structured mesh using the pyHyp package:

```
# generate mesh
echo "Generating mesh.."
python genAirFoilMesh.py > log.meshGeneration
plot3dToFoam -noBlank volumeMesh.xyz >> log.meshGeneration
autoPatch 30 -overwrite >> log.meshGeneration
createPatch -overwrite >> log.meshGeneration
renumberMesh -overwrite >> log.meshGeneration
echo "Generating mesh.. Done!"
```

Here the genAirFoilMesh.py script reads the NACA 0012 profile, generates a surface mesh, and calls pyHyp to generate a volume mesh. DAFoam does not support pure 2D cases, so we use one cell in the spanwise (z) direction and impose the symmetry boundary condition. The pyHyp will output the volume mesh in plot3D format (.xyz). We convert it to OpenFOAM meshes using the **plot3dToFoam**, **autoPatch**, **createPatch**, and **renumberMesh** utilities in OpenFOAM. Refer to Mesh Generation in OpenFOAM for detailed instructions. The mesh is as follows:



Here the red squares are the FFD control points to morph the airfoil shape. **NOTE**: make sure the design surfaces are completely within the FFD volume, otherwise, you will see errors. The FFD file is in plot3D format and is located in FFD/wingFFD.xyz. You can use the genFFD.py script to generate this FFD file by running `python genFFD.py`. Alternatively, you can use advance software such as ICEM for more complex FFD point generation. You can use Paraview to view the plot3D files (remember to uncheck **Binary File** and check **Multi Grid**), or you can convert the plot3D mesh to OpenFOAM format by using the plot3dToFoam utility (see the example above).

In the third section, we run these commands to start the optimization:

```
# these are the actually commands to run the case
./foamRun.sh $1 &
sleep 1
echo "Running the optimization. Check the log.opt file for the progress."
mpirun -np $1 python runScript.py &> log.opt
```

DAFoam has two major layers: OpenFOAM and Python, and they interact through file IO. The first command `./foamRun.sh $1 &` runs a bash script for the OpenFOAM layer and put it to background. This bash script will detect file output from the Python layer and run the corresponding executives, i.e., run the coloring, check the mesh quality, simulate the flow, and compute the adjoint derivatives. You need to change the names of the executives in foamRun.sh if you want to use different primal and adjoint solvers.

The second command `mpirun -np $1 python runScript.py &> log.opt` runs the Python layer and outputs the optimization log to log.opt. All the optimization configurations are defined in runScript.py. As mentioned in *Tutorials*, the runScript.py has seven sections. We need to modify the **Input Parameters**, **DVGeo**, and **DVCon** sections for different optimization cases. Taking the runScript.py in tutorials/Aerodynamics/NACA0012_Airfoil_Incompressible as an example, we first define the input argument as follow:

```
parser.add_argument("--output", help='Output directory', type=str,default='../
↪optOutput/')
parser.add_argument("--opt", help="optimizer to use", type=str, default='slsqp')
parser.add_argument("--task", help="type of run to do", type=str, default='opt')
parser.add_argument('--optVars',type=str,help='Vars for the optimizer',default="[
↪'shape']")
```

Here the –output argument defines where the intermediate results are stored, the default is ../optOutput. The intermediate results contain the OpenFOAM 3D flow fields, design variables, objective and derivatives values, flow and adjoint solution log files, check mesh quality logs for each optimization iteration. We do not store the adjoint vectors but they are available in the `run` folder with names `psi_*`. The adjoint vectors are stored in the same order as the state variables, either in state-by-state or cell-by-cell order. Refer to this paper for details. Note that the adjoint vectors are scaled based on the values defined by `stateScaling` in `adjointDict`. The –opt argument defines which optimizer to use. pyOptsparse supports multiple optimizers, here we pre-define some parameters for SNOPT, SLSQP, IPOPT, and PSQP. SLSQP (default), IPOPT, and PSQP are open souce while SNOPT is not. The –task argument defines what task to perform, the options are: `run` (just run the flow and adjoint once), `opt` (optimization),

`testsensshape` (test the accuracy of sensitivity wrt to shape variables), `testsensuin` (test the accuracy of sensitivity wrt to the inlet boundary conditions), `solvecl` (solve for CL; this is for wing cases only), `plotsensmap` (plot the sensitivity map).

After this, we define the boundary conditions and reference values:

```
pRef        = 0.0               # reference pressure, set it to 0 for incompressible␣
↪cases
rhoRef      = 1.0               # reference density, set it to 1 for incompressible␣
↪cases
UmagIn      = 35.0              # magnitude of far field velocity
LRef        = 1.0               # reference length used in momentum coefficient (CM)␣
↪calculation
ARef        = 1.0*0.1           # reference area used in drag or lift coefficients (CD,␣
↪CL) calculations
CofR        = [0.25,0,0]        # center of rotation used in momentum coefficient (CM)␣
↪calculation
CL_star     = 0.375             # the target lift coefficient (lift constraint)
alpha0      = 3.579107          # initial angle of attack
```

Then we define a function to compute far field velocity components and drag and lift directions, given the value of angle of attack and far field velocity magnitude:

```
def calcUAndDir(UIn,alpha1):
    dragDir = [ np.cos(alpha1*np.pi/180),np.sin(alpha1*np.pi/180),0]
    liftDir = [-np.sin(alpha1*np.pi/180),np.cos(alpha1*np.pi/180),0]
    inletU = [float(UIn*np.cos(alpha1*np.pi/180)),float(UIn*np.sin(alpha1*np.pi/180)),
↪0]
    return inletU, dragDir, liftDir

inletu0, dragdir0, liftdir0 = calcUAndDir(UmagIn,alpha0)
```

Next, we define the input parameters for optimization in the aeroOptions dictionary. The explanation of these input parameters is in Python layer. Refer to classes-python-pyDAFoam-PYDAFOAM-aCompleteInputParameterSet(). For this specific case, we have:

```
# output options
'casename':                 'NACA0012_'+task+'_'+optVars[0],    # name of the case
'outputdirectory':          outputDirectory,                    # path to store the␣
↪intermediate shapes and flow fields
'writesolution':            True,                               # write intermediate␣
↪shapes and flow fields to outputdirectory
# design surfaces and cost functions
'designsurfacefamily':      'designSurfaces',                   # group name of design␣
↪surface, no need to change
'designsurfaces':           ['wing','wingte'],                  # names of design␣
↪surface to morph, these patch names should be in constant/polyMesh/boundary
'objfuncs':                 ['CD','CL'],                        # names of the␣
↪objective functions
'objfuncgeoinfo':           [['wing','wingte'],                 # for each object␣
↪function, what are their patch names to integrate over
                             ['wing','wingte']],
'referencevalues':          {'magURef':UmagIn,                  # these are reference␣
↪values for computing CD, CL, etc.
                             'ARef':ARef,
                             'LRef':LRef,
                             'pRef':pRef,
                             'rhoRef':rhoRef},
```

(continues on next page)

(continued from previous page)

```
'liftdir':                  liftdir0,                               # drag, lift␣
↪directions and center of rotation
'dragdir':                  dragdir0,
'cofr':                     CofR,
# flow setup
'adjointsolver':            'simpleDAFoam',                         # which flow/adjoint␣
↪solver to use, for incompressible we use simpleDAFoam
'rasmodel':                 'SpalartAllmarasFv3',                   # which turbulence␣
↪model to use
'flowcondition':            'Incompressible',                       # flow condition,␣
↪either Incompressible or Compressible
'maxflowiters':             800,                                    # how many steps to␣
↪run the flow
'writeinterval':            800,                                    # how many steps to␣
↪write the flow fields to disks
'setflowbcs':               True,                                   # whether to set/
↪update boundary conditions
'inletpatches':             ['inout'],                              # names of the␣
↪farfield or inlet/outlet patches
'outletpatches':            ['inout'],
'flowbcs':                  {'bc0':{'patch':'inout',                # we can set boundary␣
↪condition in the python layer, this will overwrite
                                    'variable':'U',                 # the boundary␣
↪conditions in the '0' folder in the OpenFOAM cases.
                                    'value':inletu0},
                             'useWallFunction':'true'},             # we use wall function
# adjoint setup
'adjgmresmaxiters':         1000,                                   # how many steps to␣
↪solve the adjoint equations
'adjgmresrestart':          1000,                                   # how many Krylov␣
↪subspace to keep, always set it to adjgmresmaxiters
'adjgmresreltol':           1e-6,                                   # adjoint GMRE␣
↪convergence tolerance
'adjdvtypes':               ['FFD'],                                # types of derivatives,
↪ can be FFD (shape variables), UIn (boundary condition)
'epsderiv':                 1.0e-6,                                 # the finite-
↪difference step size for state variables in the partial derivative computation for␣
↪the adjoint
'epsderivffd':              1.0e-3,                                 # the finite-
↪difference step size for shape variables (FFD displacement)
'adjpcfilllevel':           1,                                      # number of incomplete-
↪LU preconditioning fill-in, set it to higher if you have convergence problems
'adjjacmatordering':        'cell',                                 # how to order the␣
↪states can be either state or cell
'adjjacmatreordering':      'natural',                              # how to reorder the␣
↪states, options are: natural, rcm, nd
'statescaling':             {'UScaling':UmagIn,                     # give reference␣
↪values to scale the states
                             'pScaling':UmagIn*UmagIn/2,
                             'nuTildaScaling':1e-4,
                             'phiScaling':1},
########## misc setup ##########
'mpispawnrun':              False,                                  # if you want to run␣
↪this script without the mpirun command (in serial), set it to False, otherwise, True
'restartopt':               False,                                  # whether to restart␣
↪the optimization
'meshmaxnonortho':          70.0,                                   # these are some␣
↪thresholds for mesh quality check
```

(continues on next page)

(continued from previous page)

```
'meshmaxskewness':          10.0,
'meshmaxaspectratio':       2000.0,
```

Next, we need to define the mesh warping parameters:

```
# mesh warping parameters, users need to manually specify the symmetry plane
meshOptions = {'gridFile':                os.getcwd(),
               'fileType':                'openfoam',
               # point and normal for the symmetry plane
               'symmetryPlanes':          [[[0.,0., 0.],[0., 0., 1.]],[[0.,0., 0.1],
↪[0., 0., 1.]]]}
```

Here we need to manually define the symmetry planes.

Next, we can define some parameters for optimizers, check pyOptSparse for a complete set of parameters for each optimizer:

```
# options for optimizers
outPrefix = outputDirectory+task+optVars[0]
if args.opt == 'snopt':
    optOptions = {
        'Major feasibility tolerance':  1.0e-6,    # tolerance for constraint
        'Major optimality tolerance':   1.0e-6,    # tolerance for gradient
        'Minor feasibility tolerance':  1.0e-6,    # tolerance for constraint
        'Verify level':                 -1,        # do not verify derivatives
        'Function precision':           1.0e-6,
        'Major iterations limit':       20,
        'Nonderivative linesearch':     None,
        'Major step limit':             2.0,
        'Penalty parameter':            0.0, # initial penalty parameter
        'Print file':                   os.path.join(outPrefix+'_SNOPT_print.out'),
        'Summary file':                 os.path.join(outPrefix+'_SNOPT_summary.out')}
elif args.opt == 'psqp':
    optOptions = {
        'TOLG':                         1.0e-6,    # tolerance for gradient
        'TOLC':                         1.0e-6,    # tolerance for constraint
        'MIT':                          20,        # max optimization iterations
        'IFILE':                        os.path.join(outPrefix+'_PSQP.out')}
elif args.opt == 'slsqp':
    optOptions = {
        'ACC':                          1.0e-5,    # convergence accuracy
        'MAXIT':                        20,        # max optimization iterations
        'IFILE':                        os.path.join(outPrefix+'_SLSQP.out')}
elif args.opt == 'ipopt':
    optOptions = {
        'tol':                          1.0e-6,    # convergence accuracy
        'max_iter':                     20,        # max optimization iterations
        'output_file':                  os.path.join(outPrefix+'_IPOPT.out')}
else:
    print("opt arg not valid!")
    exit(0)
```

Now we can define the design variable in the DVGeo section:

```
FFDFile = './FFD/wingFFD.xyz'
DVGeo = DVGeometry(FFDFile)

```

(continues on next page)

(continued from previous page)

```
# ref axis
x = [0.25,0.25]
y = [0.00,0.00]
z = [0.00,0.10]
c1 = pySpline.Curve(x=x, y=y, z=z, k=2)
DVGeo.addRefAxis('bodyAxis', curve = c1,axis='z')

def alpha(val, geo=None):
    inletu, dragdir, liftdir = calcUAndDir(UmagIn,np.real(val))

    flowbcs=CFDSolver.getOption('flowbcs')
    for key in flowbcs.keys():
        if key == 'useWallFunction':
            continue
        if flowbcs[key]['variable'] == 'U':
            flowbcs[key]['value'] = inletu
    CFDSolver.setOption('setflowbcs',True)
    CFDSolver.setOption('flowbcs',flowbcs)
    CFDSolver.setOption('dragdir',dragdir)
    CFDSolver.setOption('liftdir',liftdir)

# select points
pts=DVGeo.getLocalIndex(0)
indexList=pts[:,:,:].flatten()
PS=geo_utils.PointSelect('list',indexList)
DVGeo.addGeoDVLocal('shapey',lower=-1.0, upper=1.0,axis='y',scale=1.0,pointSelect=PS)
DVGeo.addGeoDVGlobal('alpha', alpha0,alpha,lower=0, upper=10., scale=1.0)
```

Here we first load the wingFFD.xyz file and create a DVGeo object. Then we add a reference axis defined by the x, y, and z lists. The reference axis can be used to define twist design variables. The wing sections will then rotate wrt to the reference axis (see the *Aircraft wing-body-tail configuration* and *UAV wing multipoint* cases for reference). Next, we define a function `def alpha` and use it as the design variable (angle of attack). This function will basically change the far field velocity components, drag and lift directions for a given angle of attack. Finally, we select the design variable points. We first select the first block of the plot3D FFD file `pts=DVGeo.getLocalIndex(0)`. We then select all the points in this block `indexList=pts[:,:,:]`. Base on your case setup, you may want to select only parts of the points. This can be done by giving a range of indices, e.g., `pts[1:3,:,:]` will select points with i=1 to 3, and all j and k indices. **NOTE**: for this case, we have only one block for the plot3D file, but you can create multiple blocks. For example, if your plot3D file has two blocks and you want to select the 2nd block, do `pts=DVGeo.getLocalIndex(1)`. We then call `DVGeo.addGeoDVLocal` to add these FFD points as the shape variable, and allow them to move in the y direction with lower and upper bounds -1.0 m and +1.0 m. Similarly, `DVGeo.addGeoDVGlobal` adds the angle of attack as the design variable. See the instructions in pyGeo for more details.

After the design variables are set, we need to impose the relevant constraints:

```
# no need to change this block
DVCon = DVConstraints()
DVCon.setDVGeo(DVGeo)
[p0, v1, v2] = CFDSolver.getTriangulatedMeshSurface(groupName=CFDSolver.getOption(
↪'designsurfacefamily'))
surf = [p0, v1, v2]
DVCon.setSurface(surf)

# define a 2D plane for 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]]
```

(continues on next page)

(continued from previous page)

```
DVCon.addVolumeConstraint(leList, teList, nSpan=2, nChord=50,lower=1.0,upper=3,␣
↪scaled=True)
DVCon.addThicknessConstraints2D(leList, teList,nSpan=2,nChord=50,lower=0.8, upper=3.0,
↪scaled=True)

#Create a linear constraint so that the curvature at the symmetry plane is zero
pts1=DVGeo.getLocalIndex(0)
indSetA = []
indSetB = []
for i in range(10):
    for j in [0,1]:
        indSetA.append(pts1[i,j,1])
        indSetB.append(pts1[i,j,0])
DVCon.addLinearConstraintsShape(indSetA,indSetB,factorA=1.0,factorB=-1.0,lower=0.0,
↪upper=0.0)

#Create a linear constraint so that the leading and trailing edges do not change
pts1=DVGeo.getLocalIndex(0)
indSetA = []
indSetB = []
for i in [0,9]:
    for k in [0]: # do not constrain k=1 because it is linked in the above symmetry␣
↪constraint
        indSetA.append(pts1[i,0,k])
        indSetB.append(pts1[i,1,k])
DVCon.addLinearConstraintsShape(indSetA,indSetB,factorA=1.0,factorB=1.0,lower=0.0,
↪upper=0.0)
```

Here we first define a 2D plane for volume and thickness constraints by giving `leList` and `teList`. The thickness constraint function will project the points in the 2D plane to the upper and lower surfaces of the wing, the distance will be the thickness. Similarly, the volume constraint function will project and form a 3D volume. Then, we define linear constraints to link the displacements for the FFD points. Because we use a symmetry plane, we need to link all the y displacement magnitudes between k=0 and k=1. In addition, we want to fix the leading and trailing edges. To do this, we set the y displacements at j=0 and j=1 to have the same magnitudes but opposite signs. We do this for both i=0 (leading) and i=9 (trailing). Note that for wing cases, the fixed leading and trailing edge constraints can be easily done by calling:

```
# Le/Te constraints
DVCon.addLeTeConstraints(0, 'iHigh')
DVCon.addLeTeConstraints(0, 'iLow')
```

See *Aircraft wing-body-tail configuration* and *UAV wing multipoint* cases for reference. Also refer to the instructions in pyGeo for more details.

Next, we define a function to compute objective functions and constraints `def aeroFuncs(xDV):`. Similarly, we define a function to compute derivatives `def aeroFuncsSens(xDV,funcs):`. These two functions will be given to pyOptSparse for optimization, i.e., `optProb = Optimization('opt', aeroFuncs, comm=gcomm)` and `sol = opt(optProb, sens=aeroFuncsSens, storeHistory=histFile)`. For optimization, we also need to define the objective function and add physical constraints:

```
# Add objective
optProb.addObj('CD', scale=1)
# Add physical constraints
optProb.addCon('CL',lower=CL_star,upper=CL_star,scale=1)
```

We can add only one objective function, but multiple physical constraints (call `optProb.addCon` multiple times).

NOTE: the geometric constraints have been added in DVGeo.

The above are the basic configurations for DAFoam. Good luck!

NACA0012 airfoil compressible

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is an aerodynamic shape optimization case for an airfoil at transonic conditions. The summary of the case is as follows:

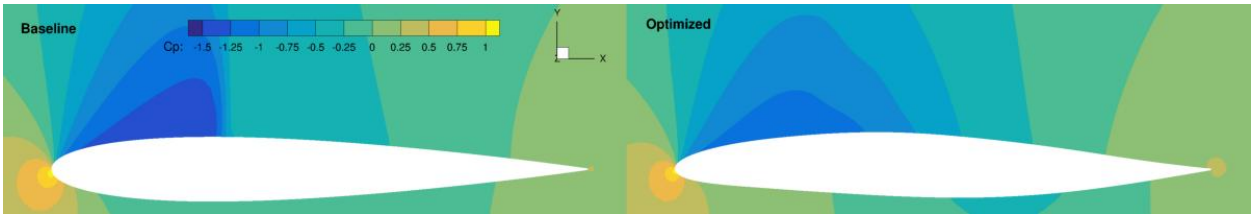
Case: Airfoil aerodynamic optimization
Geometry: NACA0012
Objective function: Drag coefficient
Design variables: 40 FFD points moving in the y direction, one angle of attack
Constraints: Symmetry, volume, thickness, and lift constraints (total number: 81)
Mach number: 0.7
Reynolds number: 2.3 million
Mesh cells: 8.6K
Adjoint solver: rhoSimpleCDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

```
./Allrun.sh 1
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 17 steps, see the following figure. The baseline design has C_D=0.01777, C_L=0.5000, and the optimized design has C_D=0.01205, C_L=0.5000.



In this case, we need to use rhoSimpleCDAFoam, a compressible solver that uses the SIMPLEC algorithm. The case setup is similar to *NACA0012 airfoil incompressible*. The major difference is in the `aeroOptions` dictionary where we need to define different `divschemes`, `fvrelaxfactors`, and `simplecontrol`. These parameters are critical to ensure robust flow simulations for transonic conditions.

Aircraft wing-body-tail configuration

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is a trimmed aerodynamic shape optimization case for an aircraft wing-body-tail configuration at transonic conditions. The summary of the case is as follows:

Case: Aircraft aerodynamic optimization
Geometry: CRM wing, body, and tail
Objective function: Drag coefficient

Design variables: 216 FFD points moving in the z direction, 9 wing twists, one tail rotation, one angle of attack

Constraints: Volume, thickness, LE/TE, and lift constraints (total number: 771)

Mach number: 0.85

Reynolds number: 5 million

Mesh cells: 100K

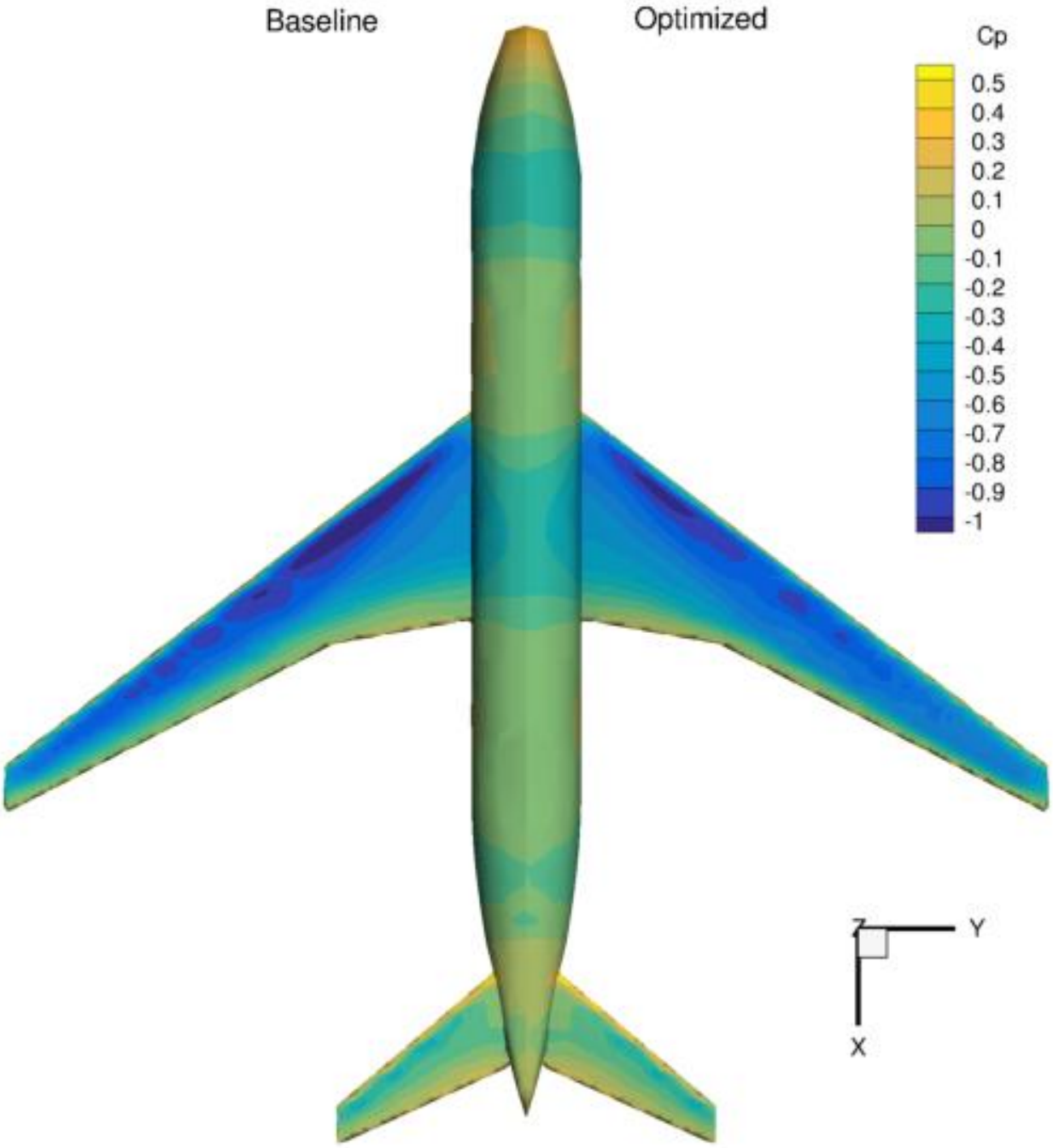
Adjoint solver: rhoSimpleCDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

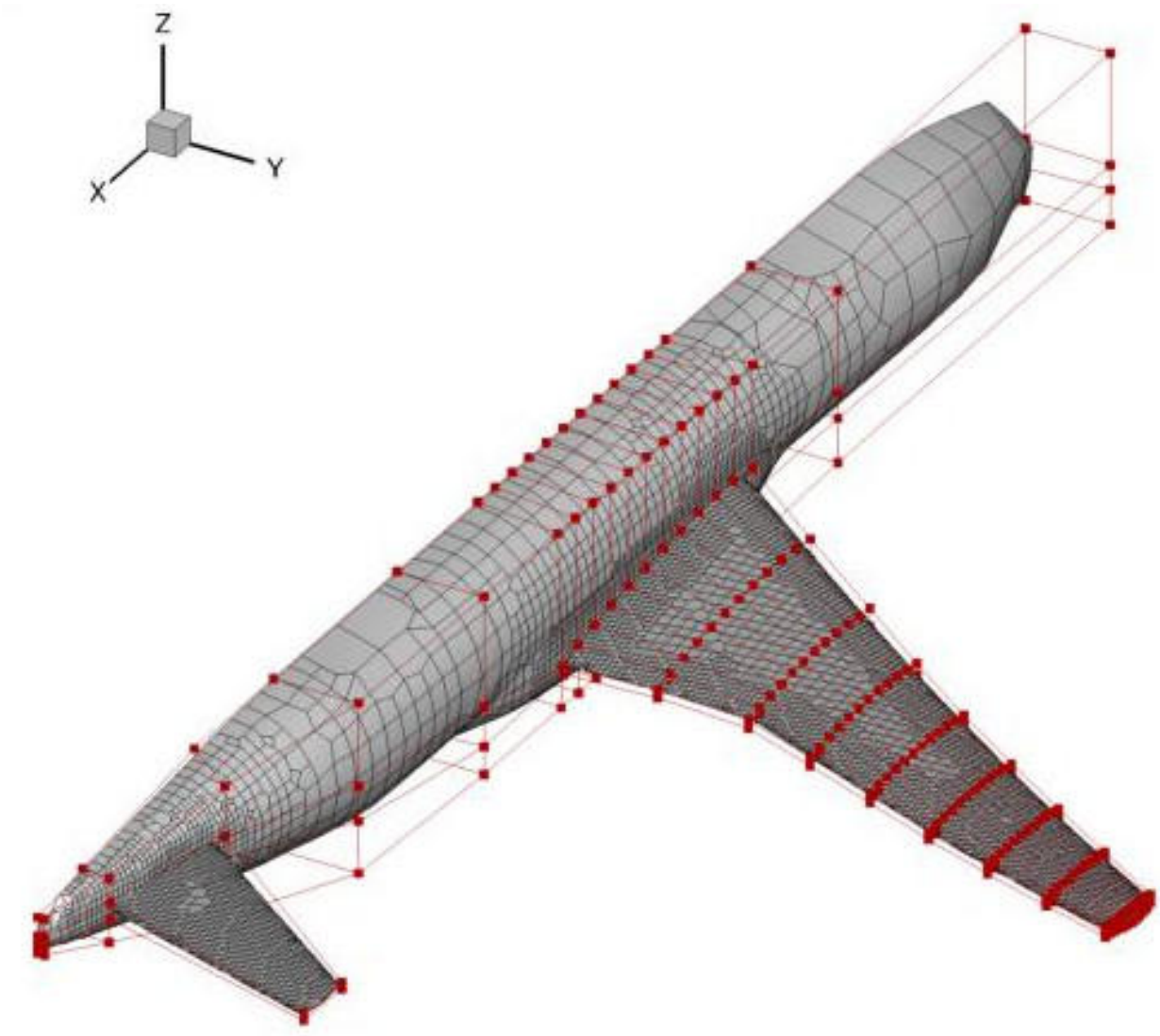
```
./Allrun.sh 4
```

The optimization progress will then be written in the **log.opt** file. **NOTE**, we recommend running this case on an HPC system using at least 4 CPU cores.

For this case, the optimization converges in 20 steps, see the following figure. The baseline design has C_D=0.05242, C_L=0.5000, C_M=-0.02611 and the optimized design has C_D=0.04997, C_L=0.4999, C_M=0.00015.



In this case, we need to use rhoSimpleCDAFoam, a compressible solver that uses the SIMPLEC algorithm. The case setup is similar to the *NACA0012 airfoil compressible* except that we have more design variables and constraints. The mesh and FFD points are as follows.



We use the OpenFOAM's built-in mesh tool `snappyHexMesh` to generate the unstructured hexa mesh. We use ICEM to generate the body-fitted FFD points. We define two more global design variables: twist and tailTwist:

```
def twist(val, geo):
    # Set all the twist values
    for i in xrange(nTwist):
        geo.rot_y['wing'].coef[i+1] = val[i]

    # Also set the twist of the root to the SOB twist
    geo.rot_y['wing'].coef[0] = val[0]

def tailTwist(val, geo):
    # Set one twist angle for the tail
    geo.rot_y['tail'].coef[:] = val[0]
```

We then add them into the DVGeo object:

```
DVGeo.addGeoDVGlobal('twist', 0*np.zeros(nTwist), twist,lower=lower, upper=upper,␣
↪scale=0.1)
DVGeo.addGeoDVGlobal('tail', 0*np.zeros(1), tailTwist,lower=-10, upper=10, scale=0.1)
```

UAV wing multipoint

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is a multipoint aerodynamic shape optimization case for a low-speed UAV wing. The summary of the case is as follows:

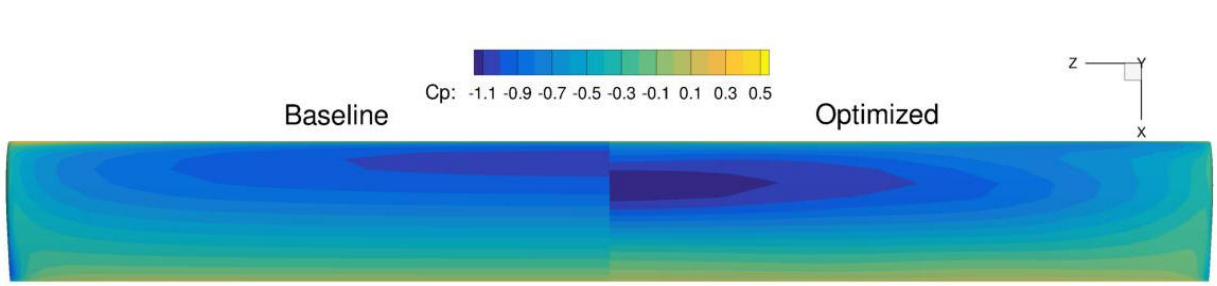
Case: UAV wing multipoint aerodynamic optimization
Geometry: Rectangular wing with the Eppler214 profile
Objective function: Weighted drag coefficient at CL=0.6 and 0.75
Design variables: 120 FFD points moving in the y direction, 6 twists, two angle of attack
Constraints: Volume, thickness, LE/TE, and lift constraints (total number: 414)
Mach number: 0.07
Reynolds number: 0.9 million
Mesh cells: 25K
Adjoint solver: simpleDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **MultiPointMain** folder and run:

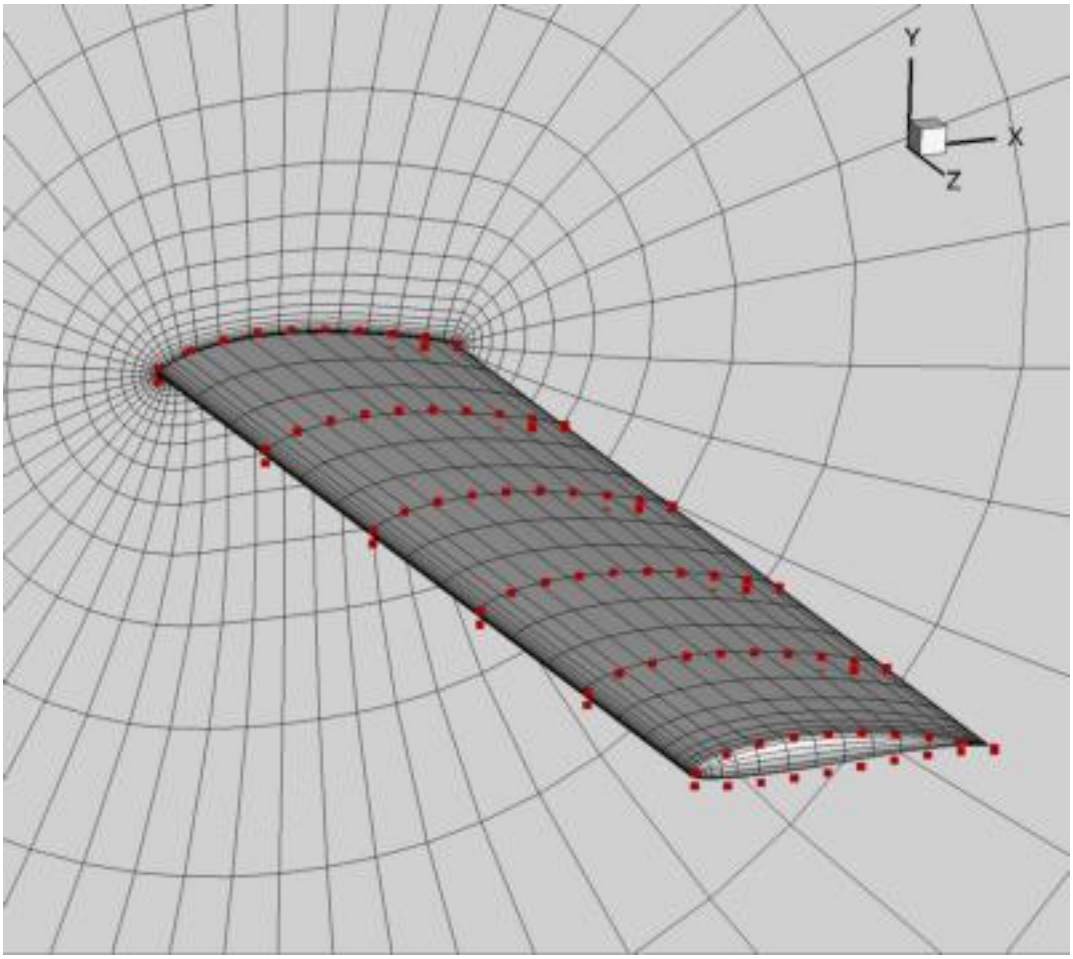
```
./Allrun.sh 2
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 20 steps, see the following figure. The baseline design has C_D=0.04019, C_L=0.7500 and the optimized design has C_D=0.03811, C_L=0.7446.



In this case, we use simpleDAFoam. The case setup is similar to *NACA0012 airfoil incompressible* except that we have more design variables and constraints and use a multipoint setup `'multipointopt':True`. The mesh and FFD points are as follows. We use ICEM to generate the body-fitted FFD points.



In the beginning of the runScript.py script, we define multipoint parameters:

```
nProcs      = args.nProcs
nFlowCases  = 2
CL_star     = [0.6,0.75]
alphaMP     = [1.768493,3.607844]
MPWeights   = [0.3,0.7]
UmagIn      = 24.8
ARef        = 1.2193524
```

Here we need to prescribe the number of CPU cores and then provide it later to the multipoint module `multiPointSparse`. We setup two flow conditions at C_L=0.6 and 0.75 and their weights are 0.3 and 0.7. We need to create two folders FlowConfig0 and FlowConfig1 for multipoint runs. If you have more flow conditions, add more accordingly.

For the multipoint runs, we no longer use a foamRun.sh script, instead, we use foamRunMultiPoint.sh, which has more complex IO interaction calls. In addition, instead of using `aeroFuncs(xDV):` and `aeroFuncsSens(xDV, funcs):`, we define `aeroFuncsMP(xDV):` and `aeroFuncsSensMP(xDV,funcs):`. We also define a function `objCon(funcs, printOK):` to combine the objective functions and derivatives of these flow conditions.

Wind turbine

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is an aerodynamic shape optimization case for the NREL6 wind turbine. The summary of the case is as follows:

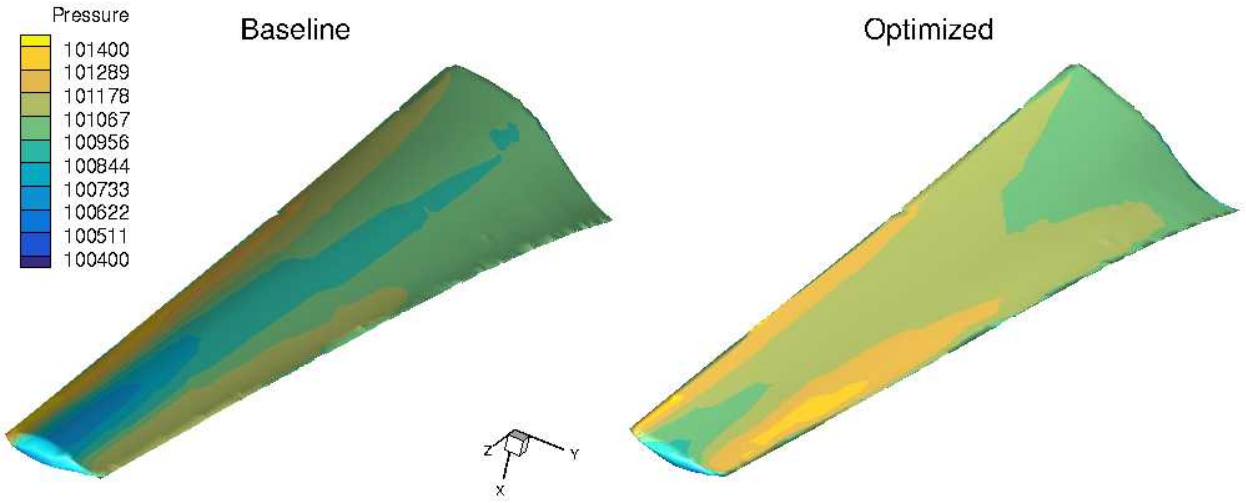
Case: Wind turbine aerodynamic optimization
Geometry: NREL6
Objective function: Torque
Design variables: 100 FFD points moving in the x and y directions
Constraints: None
Inlet velocity: 7 m/s
Rotation speed: 7.5 rad/s
Mesh cells: 60K
Adjoint solver: turboDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

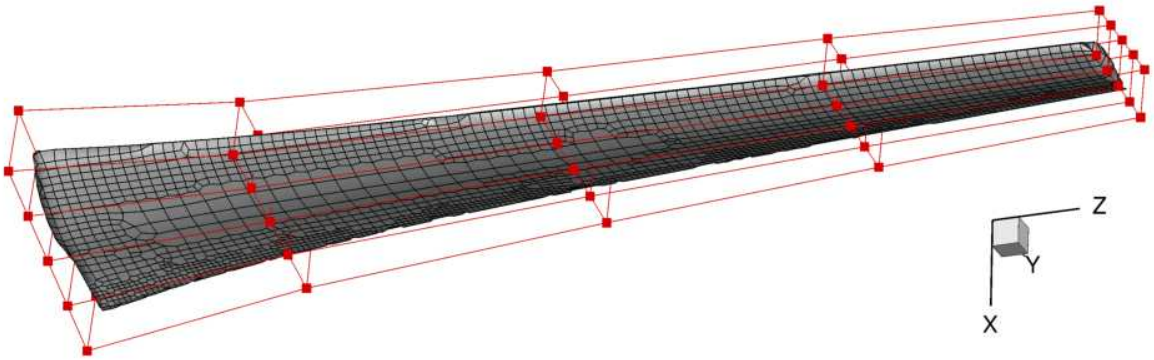
```
./Allrun.sh 8
```

The optimization progress will then be written in the **log.opt** file. **NOTE**, we recommend running this case on an HPC system using at least 8 CPU cores.

For this case, the optimization was run for 5 steps, see the following figure. The baseline design has a torque of 631.1 N m, and the optimized design has a torque of 1352.9 N m.



The mesh and FFD points are as follows. We use snappyHexMesh to generate the meshes.



Axial compressor rotor

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is an aerodynamic shape optimization case for the Rotor67 axial compressor. The summary of the case is as follows:

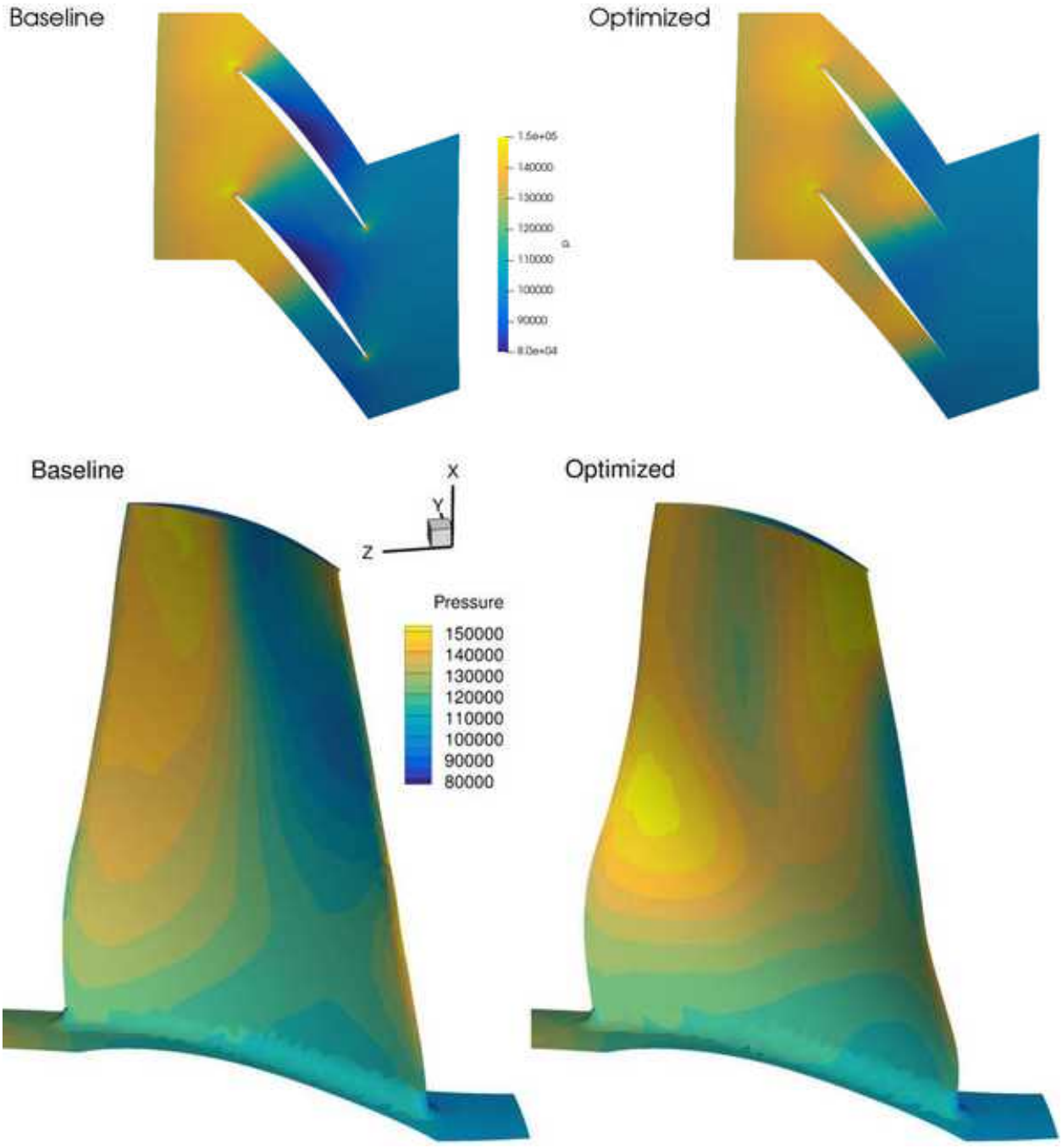
Case: Axial compressor aerodynamic optimization at transonic conditions
Geometry: Rotor67
Objective function: Torque
Design variables: 80 FFD points moving in the y and z directions
Constraints: Constant mass flow rate and total pressure ratio
Tip Mach number: 1.38
Rotation speed: -1680 rad/s
Mesh cells: 60K
Adjoint solver: turboDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

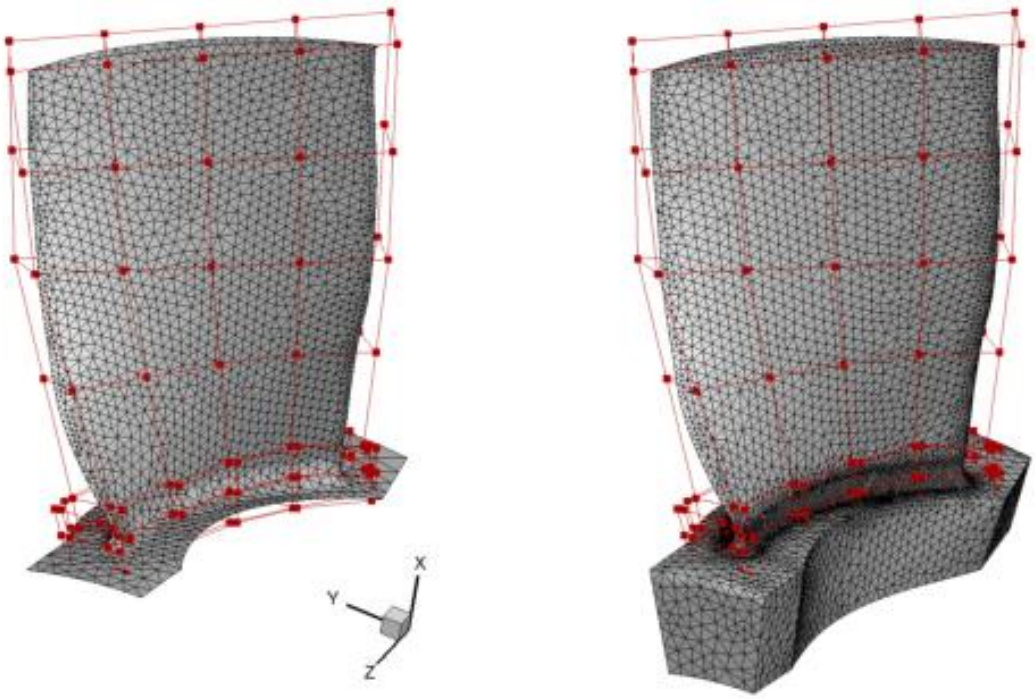
```
./Allrun.sh 8
```

The optimization progress will then be written in the **log.opt** file. **NOTE**, we recommend running this case on an HPC system using at least 8 CPU cores.

For this case, the optimization was run for 20 steps, see the following figure. The baseline design has C_M=0.08574, m=1.733 m^3/s, p1/p0=1.463, and the optimized design has C_M=0.0782, m=1.625 m^3/s, p1/p0=1.460.



The mesh and FFD points are as follows. We use ICEM to generate the triangular meshes and FFD points.



1.3.2 HeatTransfer

List of cases:

U-bend internal cooling channel

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is a heat transfer optimization case for a U-bend internal cooling channel. The summary of the case is as follows:

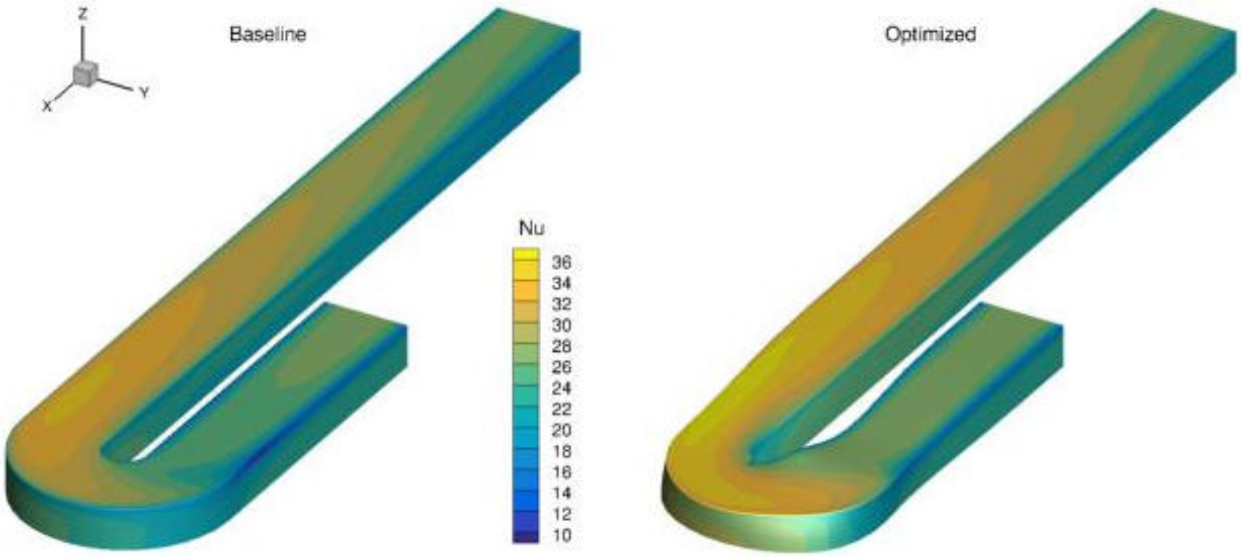
Case: Heat transfer optimization for U bend cooling channels
Geometry: von Karman U bend duct
Objective function: Nusselt number
Design variables: 114 FFD points moving in the x, y, and z directions
Constraints: Symmetry constraint (total number: 38)
Mach number: 0.02
Reynolds number: 4.2e4
Mesh cells: 4.8K
Adjoint solver: simpleTDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

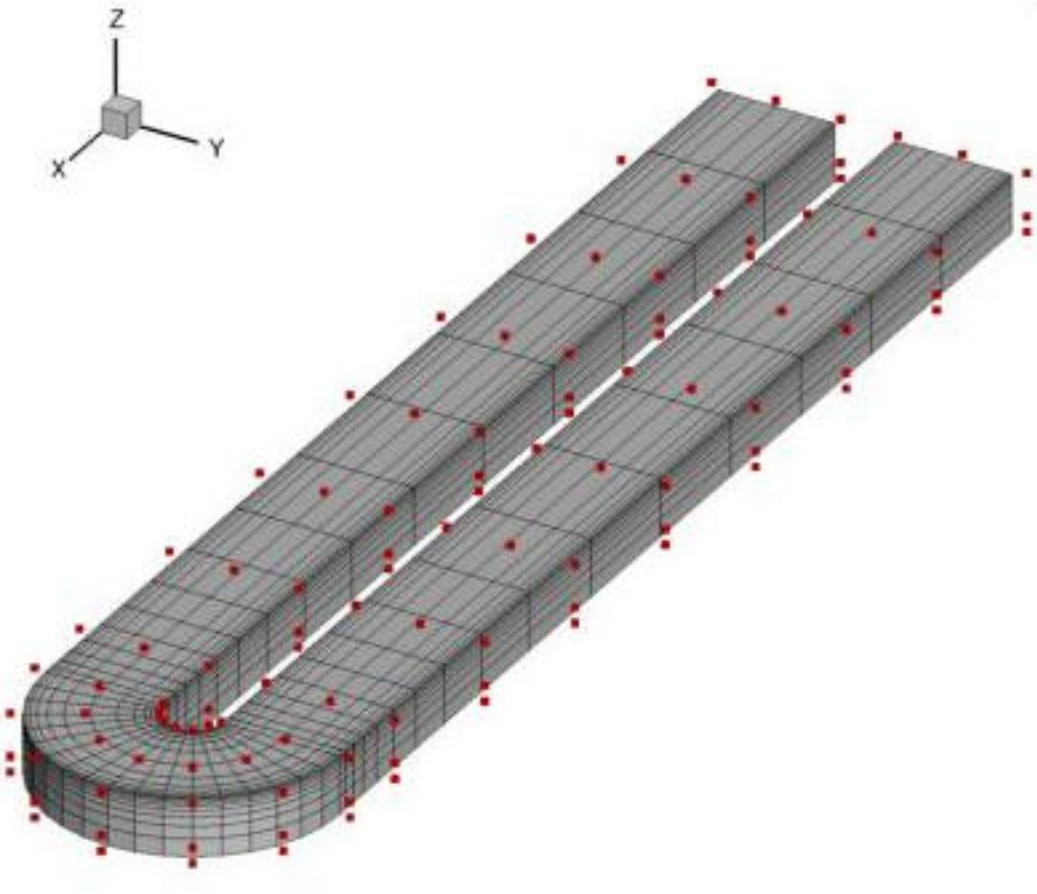
```
./Allrun.sh 1
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 6 steps, see the following figure. The baseline design has Nu=25.14 and the optimized design has Nu=26.89.



We use ICEM to generate the FFD points.



We use simpleTDAFoam, which is based on simpleDAFoam with an extra scalar equation for temperature.

1.3.3 Structure

List of cases:

Axial compressor rotor

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is a structural optimization case for a axial compressor rotor (Rotor 67). The summary of the case is as follows:

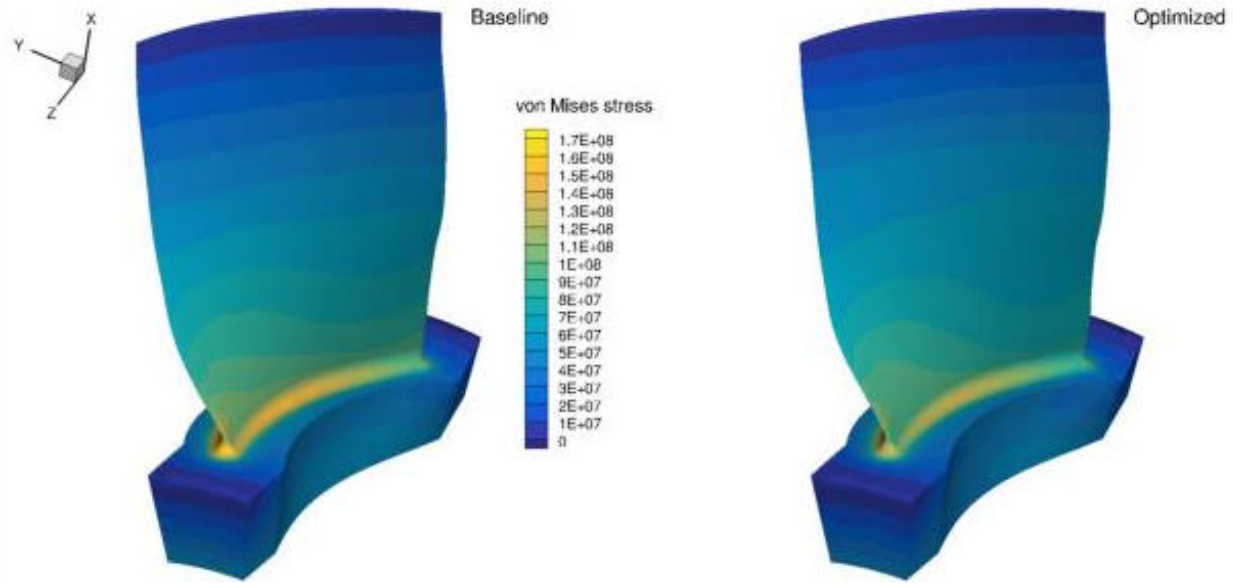
Case: Structural optimization for the engine fan
Geometry: Rotor 67
Objective function: Maximal von Mises stress
Design variables: 120 FFD points moving in the x, y, and z directions
Constraints: None
Mesh cells: 94K
Adjoint solver: solidDisplacementDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

```
./Allrun.sh 1
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 4 steps, see the following figure. The baseline design has sigma=1.828e8 Pa and the optimized design has sigma=1.507e8 Pa.



We use solidDisplacementDAFoam. The mesh and FFD setup are same as *Axial compressor rotor*. The rotor runs at 1860 rad/s.

1.3.4 Hydrodynamics

List of cases:

Bulk carrier hull

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is a hydrodynamic optimization case for a bulk carrier hull (JBC). The summary of the case is as follows:

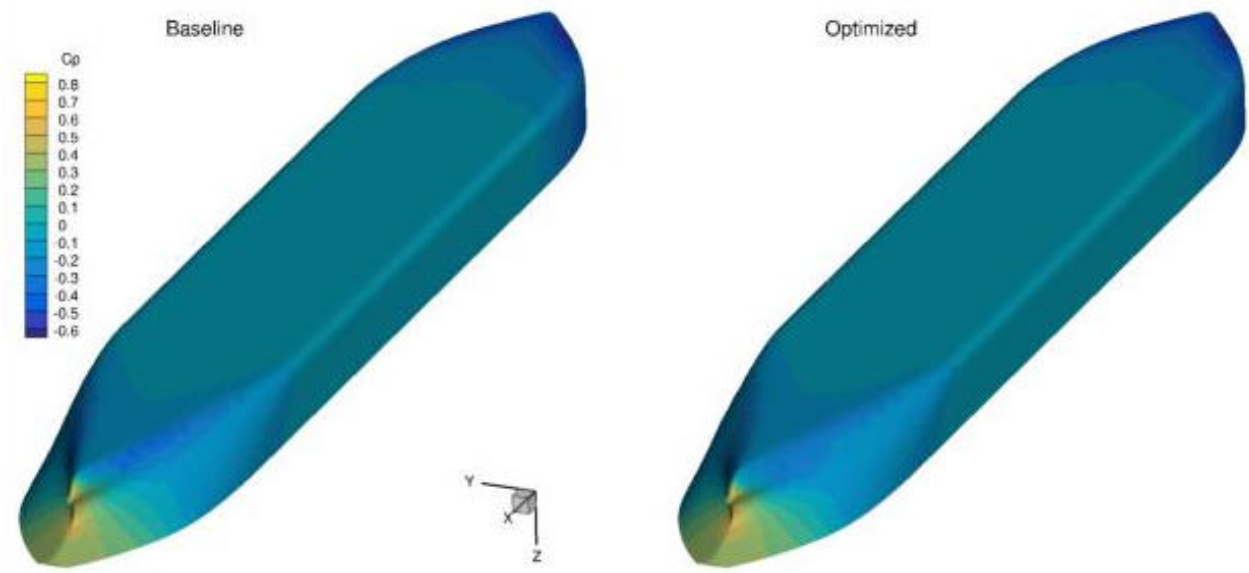
Case: Ship hydrodynamic optimization with self-propulsion
Geometry: Japan Bulk Carrier (JBC) hull
Objective function: Weighted drag and wake distortion
Design variables: 32 FFD points moving in the y direction
Constraints: Volume, thickness, symmetry, and curvature constraints (total number: 83)
Mach number: <0.01
Reynolds number: 7.5 million
Mesh cells: 40K
Adjoint solver: simpleDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

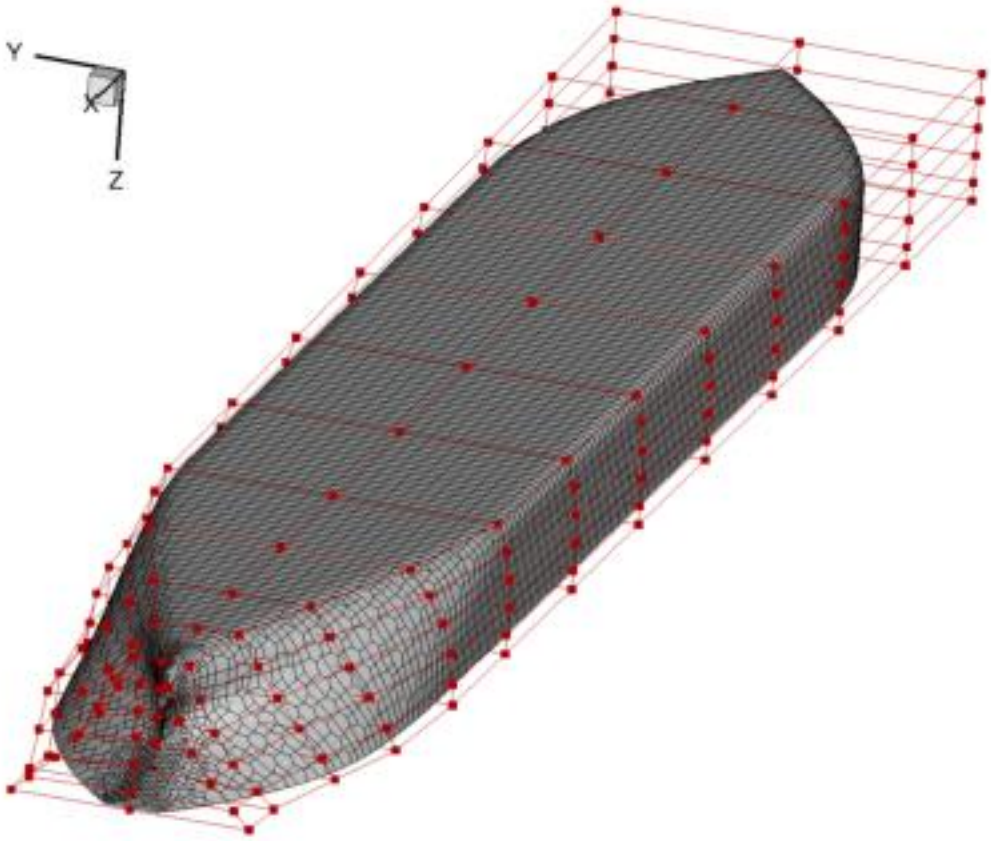
```
./Allrun.sh 4
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 20 steps, see the following figure. The baseline design has C_D=001773 and the optimized design has C_D=0.001692.



We use simpleDAFoam and ignore the free surface. We generate the unstructure mesh using `snappyHexMesh`. The mesh and FFD setup is as follows.



The runScript.py is written for multi objectives configurations where we can combine drag and wake distortion. In this case, the weight for wake distortion is set to be 0. We need to use more refined mesh to obtain an accurate wake distortion value. We also have a curvature constraint for this case.

1.3.5 Aerothermal

List of cases:

U-bend internal cooling channel

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is an aerothermal optimization case for a U-bend internal cooling channel. The summary of the case is as follows:

Case: Aerothermal optimization for U bend cooling channels with radiation and buoyancy
Geometry: von Karman U bend duct
Objective function: Weighted pressure loss and Nusselt number
Design variables: 114 FFD points moving in the x, y, and z directions
Constraints: Symmetry and curvature constraints (total number: 41)
Mach number: 0.02
Reynolds number: 4.2e4

Mesh cells: 4.8K

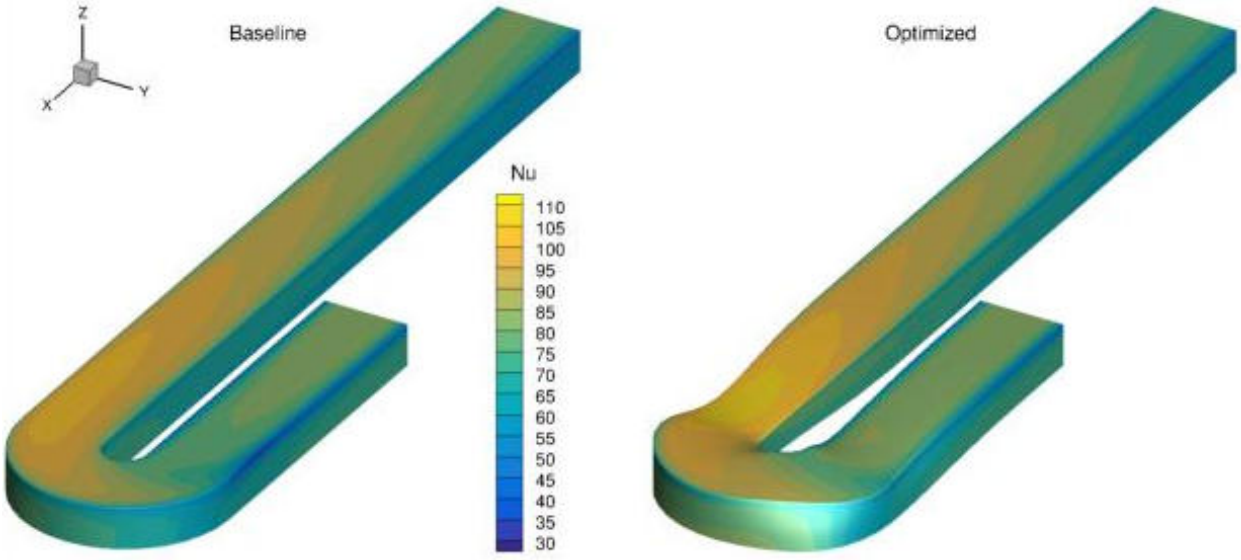
Adjoint solver: buoyantBoussinesqSimpleDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **run** folder and run:

```
./Allrun.sh 1
```

The optimization progress will then be written in the **log.opt** file.

For this case, the optimization converges in 8 steps, see the following figure. The baseline design has CPL=1.152, Nu=76.82 and the optimized design has CPL=0.7764, Nu=78.73.



We use buoyantBoussinesqSimpleDAFoam, which contains heat transfer, buoyancy, and radiation.

1.3.6 Aerostructural

List of cases:

Axial compressor rotor

NOTE: Before running this case, please read the instructions in *NACA0012 airfoil incompressible* to get an overall idea of the DAFoam optimization setup.

This is an aerostructural optimization case for an axial compressor rotor (Rotor 67) at subsonic conditions. The summary of the case is as follows:

Case: Rotor67 Aerostructural optimization

Geometry: Rotor 67: an axial compressor rotor

Objective function: Torque

Design variables: 40 FFD points moving in the x, y, and z directions (120 design variables in total)

Constraints: Constant mass flow rate and total pressure ratio, von Misese stress constraint

Rotation speed: -840 rad/s (50% design speed)

Inlet absolute Mach number: 0.29

Reynolds number: 0.85 million

Mesh cells: 61K for fluid and 94K for solid

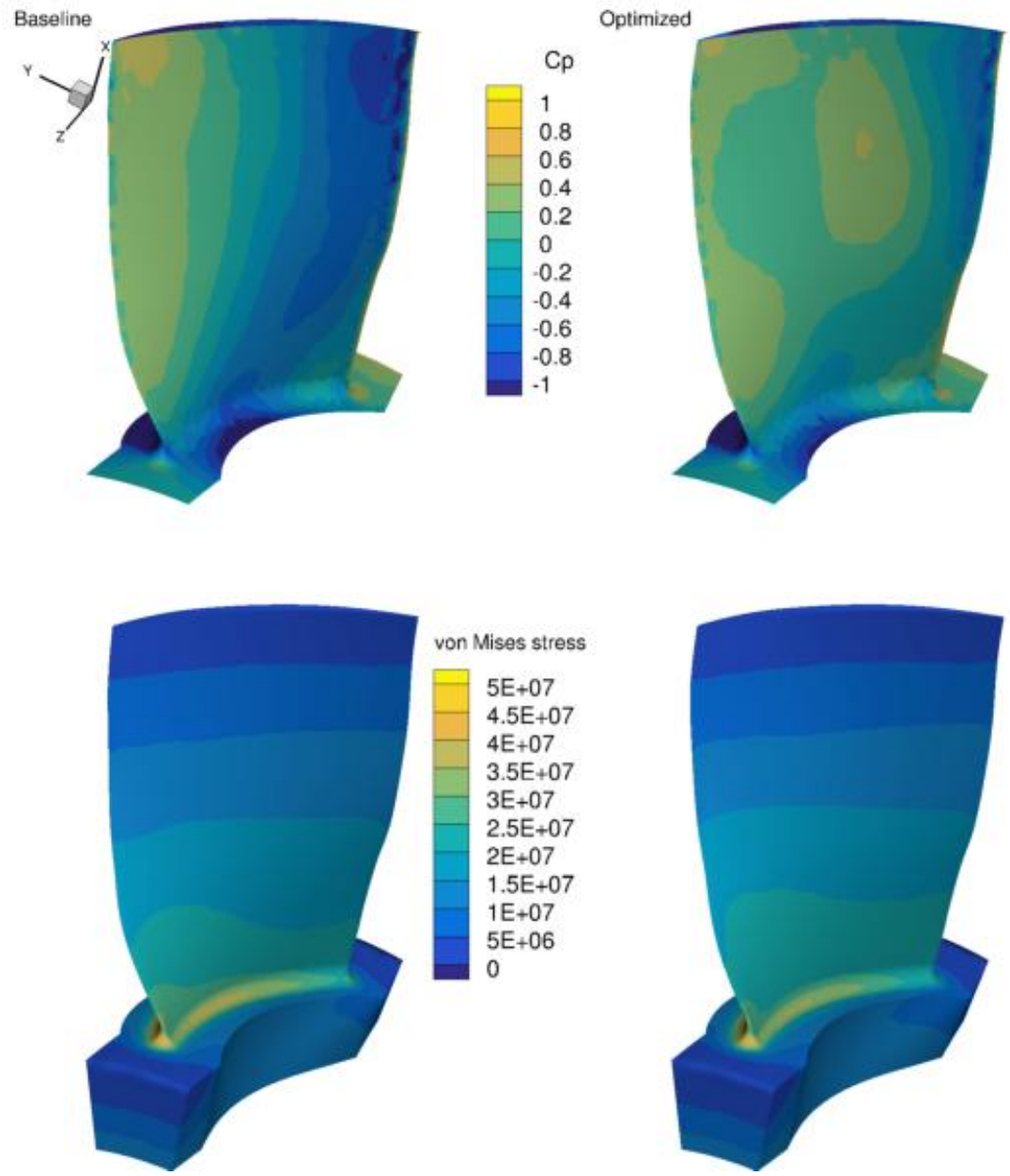
Adjoint solver: rhoSimpleDAFoam and solidDisplacementDAFoam

The configuration files are available at Github. To run this case, first source the DAFoam environment (see *Tutorials*). Then you can go into the **runFluid** folder and run:

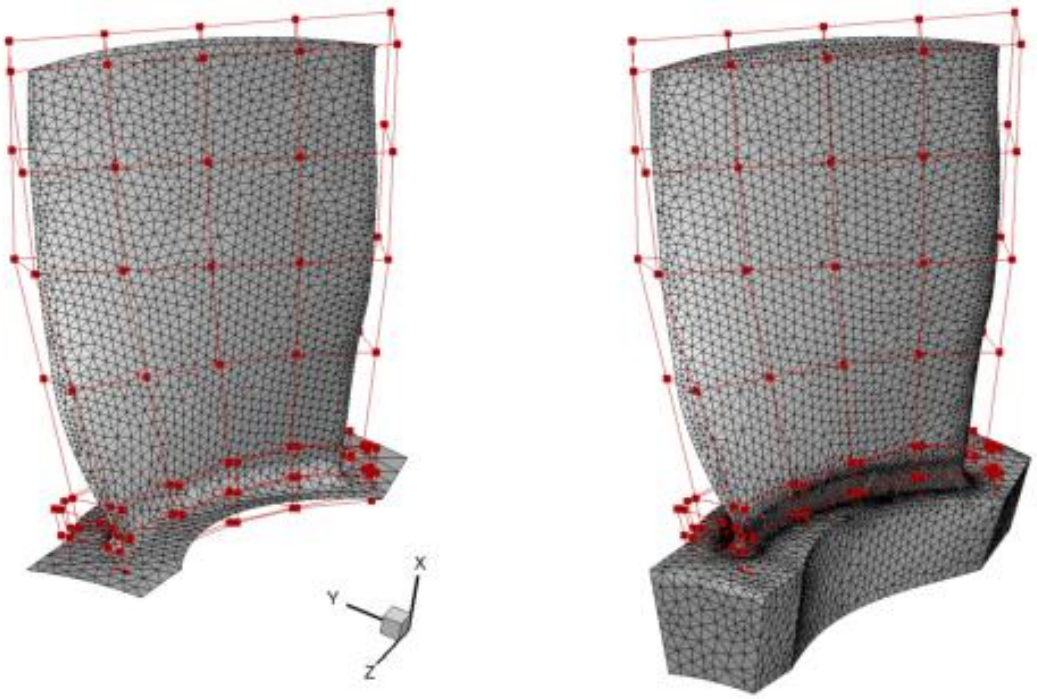
```
./Allrun.sh 4
```

The optimization progress will then be written in the **log.opt** file. The flow simulation results are stored in optOutputFluid and the solid simulation results are stored in optOutputSolid.

For this case, the optimization converges in 8 steps, see the following figure. The baseline design has C_M=0.01862, m=1.003 m^3/s, p1/p0=1.112, sigma=4.828e7 Pa, and the optimized design has C_M=0.01822, m=1.003 m^3/s, p1/p0=1.112, sigma=4.828e7 Pa.



The mesh and FFD points are as follows. We use ICEM to generate the triangular meshes and FFD points.



In this case, we couple two solvers: rhoSimpleDAFoam (flow simulation) and solidDisplacementDAFoam (structural analysis). Again, they interact through file IO, so the foamRun.sh script is a bit different from that used in *NACA0012 airfoil incompressible*. We also define new functions: `evalConFuncs`, `aeroFuncs`, and `aeroFuncsSens` in the runScript.py

In each optimization case, the **run** folder contains all the optimization setup. The **optOutput** folder stores all the optimization results and logs. We recommend you first read the instructions in *NACA0012 airfoil incompressible* before running other cases. All these tutorials use very coarse meshes, you need to refine the mesh for more realistic runs.

The optimization configurations are defined in **runScript.py**. There are seven sections:

- **Imports**. Import all external modules. No need to change.
- **Input Parameters**. Define the flow, adjoint, and optimization parameters. The explanation of these input parameters is in Python layer. Refer to classes-python-pyDAFoam-PYDAFOAM-aCompleteInputParameterSet()
- **DVGeo**. Import FFD files in plot3d format and define design variables.
- **DAFoam**. Adjoint misc setup. No need to change.
- **DVCon**. Define geometric constraints such as volume, thickness, and curvature constraints.
- **optFuncs**. Link optimization functions. No need to change.
- **Task**. Define optimization tasks (objective function, physical constraints, etc).

Before running the tutorials, you need to load the DAFoam environment.

- If you use the pre-compiled package, run this command to start a container:

```
docker run -it --rm -u dafoamuser -v $HOME:/home/dafoamuser/mount -w /home/
↪dafoamuser/mount dafoam/opt-packages:v1.1 bash
```

This will mount your local computer's home directory to the container's *~/mount* directory and login there. Then, copy the tutorials from *~/repos/dafoam* to *~/mount*:

```
cp -r /home/dafoamuser/repos/dafoam/tutorials .
```

Finally, you can go into the **run** folder of a tutorial and run the optimization. For example, for the aerodynamic optimization of NACA0012 airfoil, run:

```
cd tutorials/Aerodynamics/NACA0012_Airfoil_Incompressible/run && ./Allrun.sh␣
↪1
```

The last parameter **1** means running the optimization using 1 CPU core. After this, check the log.opt for the optimization progress. All the intermediate shapes and logs (flow, adjoint, mesh quality, design variables, etc.) are stored in the optOutput directory. Once the optimization is finished, you can run `exit` to quit the container and use Paraview to post-process the optimization results on your local computer. Remember to choose **Case Type-Decomposed Case** to view the decomposed (parallel) cases in Paraview.

A few notes:

- Treat the Docker container as disposable, i.e., start one container for one optimization run. If the optimization is running and you want to kill it, just run `exit` to quit the container.
- Do not store simulation results in the container because they will be deleted after you exit. Run simulations on the mounted space *~/mount* instead.
- dafoamuser has the sudo privilege and its password is: dafoamuser.
- Always run `./Allclean.sh` before running `./Allrun.sh`.

- If you have compiled DAFoam from the source code following *Installation*, load the OpenFOAM environment:

```
. $HOME/OpenFOAM/OpenFOAM-v1812/etc/bashrc
```

Then, copy the tutorials to your local folder:

```
cp -r $HOME/repos/dafoam/tutorials .
```

Finally, you can go into the **run** folder of a tutorial and run:

```
./Allrun.sh 1
```

A few notes:

- Before running the optimization, source the OpenFOAM environment: ". $HOME/OpenFOAM/OpenFOAM-v1812/etc/bashrc"
- Because the OpenFOAM and Python layers interact through IO, job cleaning needs special attention. We assume you compile DAFoam from source and run it on an HPC system. In this case, the running executives will be automatically cleaned when you kill the job. However, if you compile DAFoam and run it on your local computer (not recommended, use the pre-compiled docker version instead!), you need to manually kill the job and clean the running stuff (e.g., the foamRun.sh script and other running executives).
- Always run Allclean.sh before running Allrun.sh.

1.4 Development

DAFoam contains two main layers: OpenFOAM and Python, and they interact through file input and output.

The OpenFOAM layer is written in C++ and contains libraries and solvers for the discrete adjoint, the documentation of the classes and functions in the OpenFOAM layer is as follows:

OpenFOAM Layer Doxygen

The Python layer contains wrapper class to control the adjoint solvers and also calls other external modules to perform optimization. The input parameters and the APIs of the Python layer are as follows:

Python Layer Doxygen

Refer to classes-python-pyDAFoam-PYDAFOAM-aCompleteInputParameterSet() for detailed explanation of the optimization input parameters.

1.5 Publications

2020

- Ping He, Charles A. Mader, Joaquim R.R.A. Martins, Kevin J. Maki. DAFoam: An open-source adjoint framework for multidisciplinary design optimization with OpenFOAM. AIAA Journal, 2020. https://doi.org/10.2514/1.J058853
- Ping He, Alton J. Luder, Charles A. Mader, Joaquim R.R.A. Martins, Kevin J. Maki. A time-spectral adjoint approach for aerodynamic shape optimization under periodic wakes. In: AIAA Scitech Forum, 2020. AIAA-2020-2114. https://doi.org/10.2514/6.2020-2114

2019

- Ping He, Grzegorz Filip, Kevin J. Maki, Joaquim R. R. A. Martins. Design optimization for self-propulsion of a bulk carrier hull using a discrete adjoint method. Computers & Fluids, 192, pp. 104259, 2019. http://dx.doi.org/10.1016/j.compfluid.2019.104259
- Gaetan K. W. Kenway, Charles A. Mader, Ping He, Joaquim R. R. A. Martins. Effective adjoint approaches for computational fluid dynamics. Progress in Aerospace Sciences, 110, pp. 100542, 2019. http://dx.doi.org/10.1016/j.paerosci.2019.05.002
- Ping He, Charles A. Mader, Joaquim R. R. A. Martins, Kevin J. Maki. Aerothermal optimization of a ribbed U-bend cooling channel using the adjoint method. International Journal of Heat and Mass Transfer, 140, 152-172, 2019. http://dx.doi.org/10.1016/j.ijheatmasstransfer.2019.05.075
- Ping He, Charles A. Mader, Joaquim R. R. A. Martins, Kevin J. Maki. An object-oriented framework for rapid discrete adjoint development using OpenFOAM. In: AIAA Scitech Forum, 2019. AIAA-2019-1210. http://dx.doi.org/10.2514/6.2019-1210

2018

- Ping He, Charles A. Mader, Joaquim R. R. A. Martins, Kevin J. Maki. Aerothermal optimization of internal cooling passages using the adjoint method, In: 2018 Joint Thermophysics and Heat Transfer Conference, 2018. AIAA Aviation Forum, AIAA-2018-4080. http://dx.doi.org/10.2514/6.2018-4080
- Ping He, Grzegorz Filip, Joaquim R. R. A. Martins, Kevin J. Maki. Hull form hydrodynamic design using a discrete adjoint optimization method, In: 13th International Marine Design Conference, 2018
- Ping He, Charles A. Mader, Joaquim R. R. A. Martins, Kevin J. Maki. An aerodynamic design optimization framework using a discrete adjoint approach with OpenFOAM. Computers & Fluids, 168, pp. 285-303, 2018. http://dx.doi.org/10.1016/j.compfluid.2018.04.012

1.6 Contact

If you have questions, please contact: Ping He (drpinghe@umich.edu)