Cite as: Garcia, L.: Topology optimization of fluids through the continuous adjoint approach in OpenFOAM. In Proceedings of CFD with OpenSource Software, 2019.,http://dx.doi.org/10.17196/OS_CFD#YEAR_2019

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY PROF. HAKAN NILSSON

Topology Optimisation of Fluids Through the Continuous Adjoint Approach in OpenFOAM

Developed for OpenFOAMv1906

Author:
MSc. Ing. Luis Fernando Garcia Rodriguez
Univesidade de São Paulo
Ingarcia1703@usp.br

Peer reviewed by:
Seyed Morteza Mousavi

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying _les. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

December 22, 2019

Learning outcomes

The main requirements of a tutorial is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- How to create the base solver of topology optimization: “myAdjointShapeOptimizationFoam” solver.

The theory of it:

- Topology optimisation of fluids considering the frozen turbulence approach and the turbulent adjoint equations.

How it is implemented:

- Modifications needed to use myAdjointShapeOptimizationFoam at the version 1906 of OpenFOAM.
- The structure of the turbulent adjoint solver “adjointOptimizationFoam”.

How to modify it:

- The material modeling and volume constraint equations are included at “myAdjointShapeOptimizationFoam” solver to create the new solver “frozenTopologyOptimization”, based on the frozen turbulence assumption.

Prerequisites

It is suggested to have a background at the following topics:

- CFD modeling in OpenFOAM.
- Modification and structure of OpenFOAM software [7].
- Develop the tutorial of “myAdjointShapeOptimizationFoam” [8].
- Review in topology optimization of fluids [2].

TABLE OF CONTENTS

1. Topology Optimization

The Topology Optimization Method (TOM) consists in determine the material distribution in a design domain to maximize or minimize an objective function subject to certain constraints (SIGMUND et al., 2003). To maximize/minimize the objective function at flow devices is done by adding the velocity field $\boldsymbol{u}$ multiplied by a scalar field α to the momentum equations, so regions with a high value of α determine a low permeability area (solid portion) and regions with a low value of α determine a high permeability area (fluid portion). In Figure 1, a scheme of how the Topology Optimization Method is applied to flow machine devices can be seen.

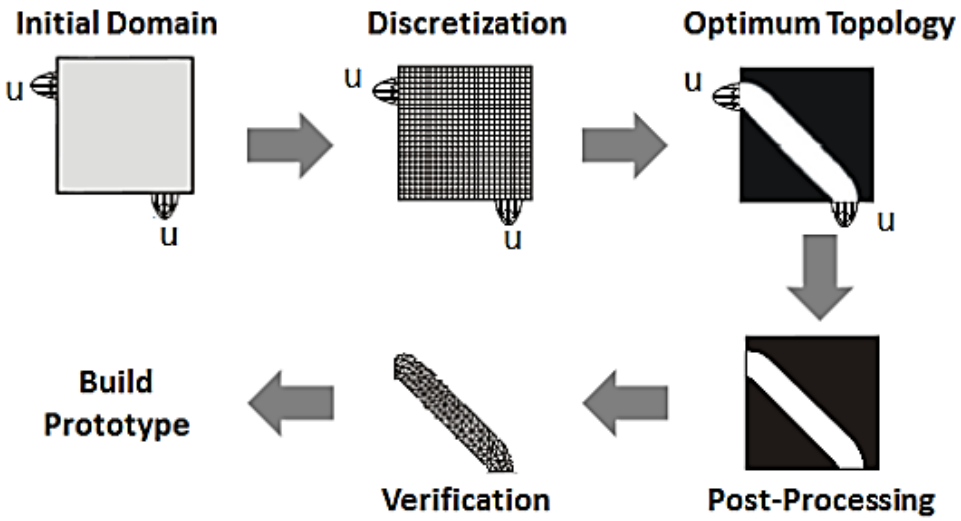


Figure 1 - Topology Optimization of flow machine [1]

Initially, the design domain is chosen (in this case, half of a circular crown). Then, the domain is discretized using some numerical method followed by TOM performing, which results in a geometry that must be post processed.

1.1. Theoretical Formulation

Topology Optimization applied to flow problems using the Finite Volume Method (FVM) and a continuous adjoint formulation for the sensitivities calculation is proposed [2]. The optimization problem is stated as:

$$\begin{array}{ll} \text{Min} & J = J(\alpha, \boldsymbol{u}, p) \\ \text{subject to} & R(\alpha, \boldsymbol{u}, p) = 0 \end{array} \tag{1}$$

Where J is the objective function to be minimized at this case, $\boldsymbol{u}$ is the velocity vector, p is the pressure and α are the design variables. The constraint $R(\alpha, \boldsymbol{u}, p)$

1.2. State Equations

The restrictions (R) of the objective functions (J) are expressed in terms of the state variables u, p and a porosity term α label as the design variable which determines the permeability of a cell to represent a solid or fluid distribution. At this case, the incompressible steady state Navier Stokes equations are considered and modified including the porosity term "$\alpha \boldsymbol{u}$**"** at the end of the momentum equation:

$$R^p = -\frac{\partial \boldsymbol{u}_j}{\partial x_j} = 0$$
$$R_i^u = \boldsymbol{u}_j \frac{\partial \boldsymbol{u}_i}{\partial x_j} - \frac{\partial}{\partial x_j}\left[(v + v_t)\left(\frac{\partial \boldsymbol{u}_i}{\partial x_j} + \frac{\partial \boldsymbol{u}_j}{\partial x_i}\right)\right] + \frac{\partial p}{\partial x_i} + \underbrace{\alpha \boldsymbol{u}_i}_{T_{a,v}} = 0 \quad (1)$$

where R^p and R_i^u are the restrictions of the conservative and momentum equations respectively. The constrained optimization problems can be solved using the augmented Lagrange Multipliers:

$$L = J + \int_\Omega (\boldsymbol{v}, q) R \, d\Omega = J + \int_\Omega q R^p \, d\Omega + + \int_\Omega v R^u \, d\Omega \quad (2)$$

where Ω is the flow domain, $\boldsymbol{v}$ is the adjoint velocity and q the adjoint pressure, both terms known as the Lagrange multipliers. To solve the augmented Lagrange function its differentiation is calculated in terms the state and design variables, i.e. u, p and α. The viscosity is not differentiated, which is a first approximation known as "frozen turbulence". By doing so, the augmented Lagrangian function differentiation is stated as:

$$\delta L = \delta_u L + \delta_p L + \delta_\alpha L \quad (3)$$

and to find the minimum optimal global point of the variable field, the following statement must be accomplished:

$$\delta_\alpha L = 0 \wedge \delta_u L + \delta_p L = 0 \quad (4)$$

By calculating $\delta_\alpha L = 0$ and $\delta_u L + \delta_p L = 0$ the sensitivity and the adjoint equations can be stablished respectively.

1.3. Adjoint Equations

The adjoint continuity equations can be formulated as:

$$\begin{aligned}
&\delta_u L + \delta_p L = 0 \\
&\left(\frac{\delta J}{\delta u} + \int_\Omega q \frac{\delta R^p}{\delta u}\, d\Omega + \int_\Omega \boldsymbol{v} \frac{\delta R^u}{\delta u}\, d\Omega\right) + \left(\frac{\delta J}{\delta p} + \int_\Omega q \frac{\delta R^p}{\delta p}\, d\Omega + \int_\Omega \boldsymbol{v} \frac{\delta R^u}{\delta p}\, d\Omega\right) = 0 \\
&\left(\frac{\delta J}{\delta u} + \int_\Omega q \frac{\delta(\nabla \cdot \boldsymbol{u})}{\delta u}\, d\Omega + \int_\Omega \boldsymbol{v} \frac{\delta\left((\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla\cdot\left(\nu(\nabla\boldsymbol{u} + \nabla\boldsymbol{u}^{\boldsymbol{T}})\right) + \alpha\boldsymbol{u}\right)}{\delta u}\, d\Omega\right) \\
&\qquad + \left(\frac{\delta J}{\delta p} + \int_\Omega \boldsymbol{v} \frac{\delta\left(\frac{1}{\rho}\nabla p\right)}{\delta p}\, d\Omega\right) = 0
\end{aligned} \tag{5}$$

By using Frechet derivatives and considering that in topology optimization the total derivative can be simplified as a local derivative as there is no change variation, the field adjoint state equations can be found:

$$\begin{aligned}
&-2\mathrm{D}(\boldsymbol{v})\boldsymbol{u} - \nabla\cdot\left(2\nu D(\boldsymbol{v})\right) + \alpha\boldsymbol{v} = -\nabla q \\
&\nabla\cdot v = 0
\end{aligned} \tag{6}$$

and its boundaries conditions as well. For the inlet and wall, the equations are stated as:

$$\begin{aligned}
&v_t = 0 \\
&v_n = -\frac{\partial J_\Gamma}{\partial p} \\
&\nabla q \cdot n = 0
\end{aligned} \tag{7}$$

and for the outlet boundary condition it is found that:

$$\begin{aligned}
&\boldsymbol{v}\cdot\boldsymbol{u} + v_n u_n + \nu(\boldsymbol{n}\cdot\nabla)v_n - q + \frac{\partial J_\Gamma}{\partial u_n} = 0 \\
&u_n v_t + \nu(\mathrm{n}\cdot\nabla)v_t + \frac{\partial \mathrm{J}_\Gamma}{\partial u_t} = 0
\end{aligned} \tag{8}$$

1.4. Sensitivity Equation

It can be found by differentiating the augmented Lagrange Function in terms of the design variable α:

$$\begin{aligned} &\delta_\alpha L = 0 \\ &\frac{\delta J}{\delta \alpha} + \int_\Omega q \frac{\delta R^p}{\delta \alpha}\, d\Omega + \int_\Omega \boldsymbol{v} \frac{\delta R^u}{\delta \alpha}\, d\Omega = 0 \\ &\int_\Omega \boldsymbol{v} \frac{\delta(\alpha \boldsymbol{u})}{\delta \alpha}\, d\Omega = 0 \\ &\int_\Omega \boldsymbol{v} u\, d\Omega = 0 \end{aligned} \tag{9}$$

1.5. Topology Optimization in OpenFOAM

The theory explained has been used to formulate a topology optimization solver in OpenFOAM 2.2.x in a previous work. The solver is based in the "adjointShapeOptimizationFoam" solver and the code structure has been written as straight line code composed by the following items:

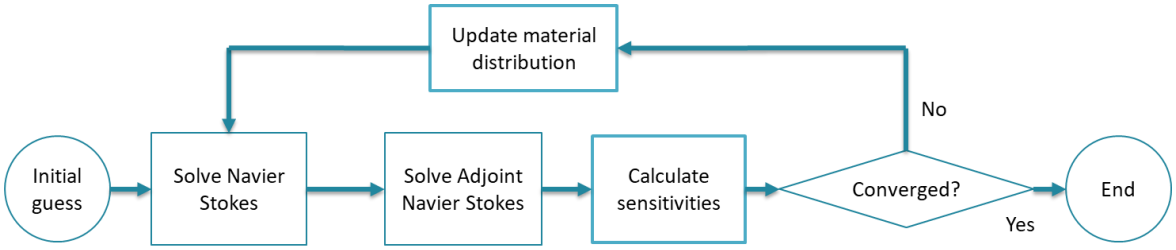


Figure 2. Topology Optimization Flowchart

As the code proposed in [8] is a modification of the original solver "adjointShapeOptimizationFoam", it cannot be found in an original version of OpenFOAM. Therefore, an executable file to generate such modified solver is developed for the readers to establish the base of the new solver. Each line modification can be find at [8].

Start by locating in your own applications/solvers library. If there is non location, it can be created by using:

```
OFv1906
```

```
cd $FOAM_RUN
mkdir -p ../applications/solvers/
```

and create the executable file by writing in the terminal:

```
touch frozenTurbulenceTO
chmod +x frozenTurbulenceTO
```

After that, copy and paste the following lines at the created “frozenTurbulenceTO” file. Copy at first the code of this page, and then press enter to separate the code written at the next page:

```
#!/bin/sh
mkdir topologyOptimization
cd $WM_PROJECT_DIR
cp -r applications/solvers/incompressible/adjointShapeOptimizationFoam $WM_PROJECT_USER_DIR/applications/solvers/topologyOptimization
cd $WM_PROJECT_USER_DIR/applications/solvers/topologyOptimization
mv adjointShapeOptimizationFoam myAdjointShapeOptimizationFoam
cd myAdjointShapeOptimizationFoam
mv adjointShapeOptimizationFoam.C myAdjointShapeOptimizationFoam.C
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
sed -i s/adjointShapeOptimizationFoam/myAdjointShapeOptimizationFoam/g Make/files

#modification of the steepest descent algorithm
sed -i s/"alpha + lambda"/"alpha - lambda"/g myAdjointShapeOptimizationFoam.C

#creating the power dissipation BC
cp -r adjointOutletVelocity/ adjointOutletVelocityPower
cp -r adjointOutletPressure/ adjointOutletPressurePower
cd adjointOutletPressurePower
sed s/adjointOutletPressure/adjointOutletPressurePower/g adjointOutletPressureFvPatchScalarField.C > adjointOutletPressurePowerFvPatchScalarField.C
sed s/adjointOutletPressure/adjointOutletPressurePower/g adjointOutletPressureFvPatchScalarField.H > adjointOutletPressurePowerFvPatchScalarField.H
cd ../adjointOutletVelocityPower
sed s/adjointOutletVelocity/adjointOutletVelocityPower/g adjointOutletVelocityFvPatchVectorField.C > adjointOutletVelocityPowerFvPatchVectorField.C
sed s/adjointOutletVelocity/adjointOutletVelocityPower/g adjointOutletVelocityFvPatchVectorField.H > adjointOutletVelocityPowerFvPatchVectorField.H
cd ..
rm adjointOutletPressurePower/adjointOutletPressureFvPatchScalarField.*
rm adjointOutletVelocityPower/adjointOutletVelocityFvPatchVectorField.*

#adding the BC
sed -i '2 a adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C' Make/files
sed -i '32 a #include "RASModel.H"' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '104 a scalarField Up_n = phip / patch().magSf(); // Primal' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '105 a scalarField Uap_n = phiap / patch().magSf(); // Adjoint' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '106 a const incompressible::RASModel& rasModel =db().lookupObject<incompressible::RASModel>("TurbulenceProperties");' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '107 a scalarField nueff = rasModel.nuEff()().boundaryField()[patch().index()];' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '108 a const scalarField& deltainv = patch ().deltaCoeffs(); //distance inverse' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '109 a scalarField Uaneigh_n = (Uap.patchInternalField() & patch().nf());' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '111 a /*' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '113 a */' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i "114 a operator+= ((Up&Uap) + (Up_n*Uap_n) + nueff*deltainv*(Uap_n-Uaneigh_n) - 0.5*mag(Up)*mag(Up) - (Up & patch().Sf()/patch().magSf())*(Up & patch().Sf()/patch().magSf()));" adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
```

```
#adjoint velocity power dissipation modification
sed -i '3 a adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C' Make/files
sed -i '32 a #include "RASModel.H"' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '93 a const fvsPatchField<scalar>& phip =patch().lookupPatchField<surfaceScalarField, scalar>("phi");' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '97 a const fvPatchField<vector>& Uap = patch().lookupPatchField<volVectorField, vector>("Ua");' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '101 a const incompressible::RASModel& rasModel = db().lookupObject<incompressible::RASModel>("TurbulenceProperties");' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '102 a scalarField nueff = rasModel.nuEff()().boundaryField()[patch().index()];' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '103 a const scalarField& deltainv = patch().deltaCoeffs();' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '104 a scalarField Up_ns = phip/patch().magSf();' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '105 a vectorField Up_t = Up - (phip*patch().Sf())/(patch().magSf()*patch ().magSf());' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '106 a //tangential component of adjoint velocity in neighbouring node' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '107 a vectorField Uaneigh = Uap.patchInternalField();' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '108 a vectorField Uaneigh_n = (Uaneigh & patch().nf())*patch().nf();' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '109 a vectorField Uaneigh_t = Uaneigh - Uaneigh_n;' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '110 a vectorField Uap_t = ((Up_ns*Up_t) + nueff*deltainv*Uaneigh_t)/(Up_ns+nueff*deltainv);' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '111 a vectorField Uap_n = (phiap*patch().Sf())/(patch().magSf()*patch().magSf());' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '117 a /*' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '120 a */' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '121 a operator==(Uap_t+Uap_n);' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C


#including sensitivity
sed -i '244 a sens=Ua&U;' myAdjointShapeOptimizationFoam.C
sed -i '$ a volScalarField sens(IOobject("sensitivity",runTime.timeName(),mesh,IOobject::READ_IF_PRESENT,IOobject::AUTO_WRITE),Ua&U);' createFields.H


#printing the cost function
sed -i '$ a dictionary optFunc = mesh.solutionDict().subDict("objectiveFunctionDict");' createFields.H
sed -i '$ a int nObjPatch = optFunc.lookupOrDefault<scalar>("numberObjectivePatches", 0);' createFields.H
sed -i '$ a int objFunction = optFunc.lookupOrDefault<scalar>("objectiveFunction", 0);' createFields.H
sed -i '$ a wordList objPatchNames=optFunc.lookup("objectivePatchesNames");' createFields.H
sed -i '$ a Info<< "Initializing objective function calculation" << endl;' createFields.H
sed -i '$ a Info<< "The objective function chosen is" << objFunction<<endl;' createFields.H
sed -i '$ a Info<< "Name of the patches for which the cost function will be calculated" << objPatchNames<<endl;' createFields.H
sed -i '$ a Info<< "Number of patches" << nObjPatch<<endl;' createFields.H
sed -i '$ a label objPatchList [nObjPatch];' createFields.H
sed -i '$ a int iLoop;' createFields.H
sed -i '$ a for (iLoop=0; iLoop<nObjPatch; iLoop++){' createFields.H
sed -i '$ a objPatchList [iLoop] = mesh.boundaryMesh().findPatchID(objPatchNames[iLoop]);}' createFields.H
sed -i '103 a #include "costFunction.H"' myAdjointShapeOptimizationFoam.C
touch costFunction.H
echo "scalar jDissPower(0);">>costFunction.H
sed -i '$ a for (iLoop=0; iLoop<nObjPatch; iLoop++)' costFunction.H
sed -i '$ a {' costFunction.H
sed -i '$ a if (objFunction==1) {' costFunction.H
sed -i '$ a jDissPower = jDissPower - sum(phi.boundaryField()[objPatchList[iLoop]]*(p.boundaryField()[objPatchList [iLoop]] + 0.5*magSqr(U.boundaryField()[objPatchList[iLoop]])));' costFunction.H
sed -i '$ a }' costFunction.H
sed -i '$ a }' costFunction.H
sed -i '$ a if (objFunction==1) {' costFunction.H
sed -i '$ a Info<<"Objective Function (Power Dissipated) J:"<<jDissPower<<endl;}' costFunction.H
```

It will create the executable file to update the “adjointShapeOptimizationFOAM” solver based on [8]. The code lines size had been modified to avoid problems during the implementation, nevertheless the code can be read easily at the created file or at the annexes of this work.

As the solver is not made for OFv1906, some additional commands have to be added at the end of the previous code:

```
#modifications to the code
sed -i '33 a #include "IncompressibleTurbulenceModel.H"' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '34 a #include "turbulentTransportModel.H"' adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '33 a #include "IncompressibleTurbulenceModel.H"' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '34 a #include "turbulentTransportModel.H"' adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
```

End up by compiling the updated topology optimization solver by writing at the terminal

```
cd topologyOptimization/myAdjointShapeOptimizationFoam
wclean
wmake
```

At this point, the solver "myAdjointShapeOptimization" based on [8] has been created. Nevertheless, its results can be improved my modifying the state equations including additional main functions.

1.6. "myAdjointShapeOptimization" Modification

Material Model: At the end of the momentum equation, an interpolation function α is introduced to represent the porous media through the domain for improving the optimization results [1]. It behaves as identifying a high permeability zone ($\alpha(\gamma)\sim 0$) where fluid domain is achieved, and low permeability zones ($\alpha(\gamma) \gg 1$) where solid domain cells can be placed. The goal is for the design variable, γ, to take values of 0 or 1, as $\gamma\sim 0$ returns a near solid domain ($\alpha(\gamma) \approx \alpha_{max}$) and $\gamma\sim 1$ returns a near fluid domain ($\alpha(\gamma) \approx \alpha_{min}$), and choose α in such a way that intermediate values of γ are suppressed. Therefore, the following convex interpolation function is adopted, which is a function of the q parameter that controls the grey level areas:

$$\alpha(\gamma) = \alpha_{max} + (\alpha_{\ min} - \alpha_{\ max})\gamma \frac{1+q}{\gamma+q} \tag{10}$$

where minimum and maximum α value can be given in terms of the Darcy equation, i.e. $\alpha_{max} = v/Dal^2$. At [1] the $\alpha_{max}$and $\alpha_{\ min}$ values are expressed as $2.5\mu/0.01^2$ and $2.5\mu/100^2$ respectively, and by using those correlations the q parameter shows the following behavior:

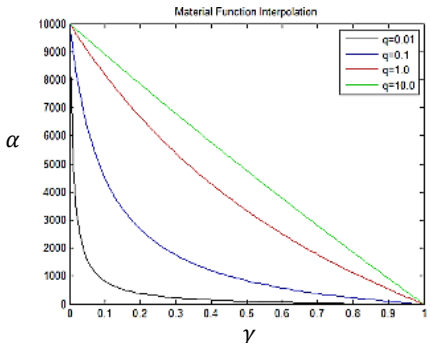


Figure 3. Interpolation scheme of alpha distribution under different q values.

The definition of the interpolation function α, should be included at myAdjointShapeOptimizationFoam.C, between the "alpha" steepest descent equation and the "zeroCells" declaration, as:

```
alpha = alphaMax + (alphaMin - alphaMax)*(vf*(1 + q)/(vf + q));
```

where `vf` is equal to γ.

Volume constraint: when considering the Lagrangian function the addition of the volume constraint term has given also reliable results [4], which is established as:

$$L = J - \lambda_{vol} c_{vol} + w_{vol} c_{vol}^2 \tag{11}$$

where λ_{vol} is the Lagrange multiplier of the volume constraint (c_{vol}) and w_{vol} is a scalar weight factor defined by the user at the first iteration of the solver. The volume constraint c_{vol} is defined as

$$c_{vol} = \left(\int \gamma d\Omega - V_{target} \right)^2 \tag{12}$$

where $\int \gamma d\Omega$ is the distribution of the porosity field along the domain and V_{target} is the minimum volume fraction aimed to be occupied by solid.

Therefore, by considering the material distribution and the volume constraint at the optimization problem, the state equation of the augmented Lagrangian updates to:

$$L_{aug} = J - \lambda_{vol} c_{vol} + w_{vol} c_{vol}^2 + \int_\Omega q R^p \, d\Omega + \int_\Omega \boldsymbol{v} R^{\boldsymbol{u}} \, d\Omega \tag{13}$$

affecting the calculation of the sensitivity differential, $\delta_\alpha L_{aug} = 0$, as it depends on the design variable α:

$$\frac{\delta L}{\delta \alpha} = \frac{\delta J}{\delta \alpha} + \int_\Omega q \frac{\delta R^p}{\delta \alpha}\, d\Omega + \int_\Omega \boldsymbol{v} \frac{\delta R^u}{\delta \alpha}\, d\Omega + \int_\Omega \left(-\lambda_{vol} + 2 w_{vol} c_{vol}\right) \frac{\delta c_{vol}}{\delta \alpha} d\Omega$$

$$\delta_\alpha L = \int_\Omega \boldsymbol{u} \cdot \boldsymbol{v}\, d\Omega\, \frac{\partial \alpha}{\partial \gamma} + \left(-\lambda_{vol} + 2 w_{vol} c_{vol}\right) \frac{\delta c_{vol}}{\delta \alpha} \tag{14}$$

$$\delta_\alpha L = (\boldsymbol{u} \cdot \boldsymbol{v})(\mathrm{V_{domain}}) \left[(\underline{\alpha} - \overline{\alpha}) q \frac{(1+q)}{(\gamma+q)^2} \right] + \left(-\lambda_{vol} + 2 w_{vol} \left(\int \gamma d\Omega - V_{target} \right)^2 \right) 2 \left(\int \gamma d\Omega - V_{target} \right) V_{cell}$$

which differs from the sensitivity equation defined in "myAdjointShapeOptimization", $\delta_\alpha L = \boldsymbol{v} \cdot u$.

The update of the material distribution implemented at "myAdjointShapeOptimizationFoam" solver is expressed in terms of α, the design variable. The current modification of the solver considers α an interpolation function and non the design variable. The design variable is γ, and by using the steepest descent algorithm, expressed as

$$\alpha_{n+1} = \alpha_n - (\boldsymbol{u} \cdot \boldsymbol{v})(\mathrm{V}) * \boldsymbol{lambda} \tag{15}$$

written in myAdjointShapeOptimizationFoam.C at line 109 as

```
alpha += mesh.fieldRelaxationFactor("alpha") *(min(max(alpha - lambda*(Ua & U), zeroAlpha), alphaMax) - alpha);
```

should be updated to:

$$\gamma_{n+1} = \gamma_n - (\boldsymbol{u} \cdot \boldsymbol{v})(\mathrm{V_{domain}}) \left[(\underline{\alpha} - \overline{\alpha}) q \frac{(1+q)}{(\gamma+q)^2} \right] + \left(-\lambda_{vol} + 2 w_{vol} \left(\int \gamma d\Omega - V_{target} \right)^2 \right) 2 \left(\int \gamma d\Omega - V_{target} \right) V_{cell} * lambda \tag{16}$$

written as

```
vf = min(max(vf-lambda.value()*sens*cte3, zeroVf), vfMax);
```

where "lambda" is a time-step relaxation factor related with the cell volume, `sens` is the sensitivity equation (5) and `cte3` is a value 1 factor to control the units between lambda and the sensitivity

equation. The evaluation of the design variable, γ, is established between the two values: zero written as `zeroVf` and 1 written as `vfMax`.

Up to this point the sensitivity calculation has been called, but its evaluation along with the integral calculation along the domain of the design variable ($\int \gamma d\Omega$) has not been defined. To do so, at the end of the adjoint Pressure-velocity SIMPLE corrector, after the "turbulence->correct" command, the integral is defined as:

```
    intvfVol = 0.0;
    forAll(mesh.C(), ID){intvfVol += vf[ID]*mesh.V()[ID];}
```

where `intvfVol` is the definition of the design variable integral along the domain ($\int \gamma d\Omega$). By doing so, the sensitivity calculation can be updated at line 246, as:

```
sens=(Ua&U)*vol*((alphaMin-
alphaMax)*q*(1+q)/((vf+q)*(vf+q)))*cte1+cte2*(-lvol+2*wf*sqr(intvfVol-
vTarget))*2*vCell;
```

Finally, the function to update the Lagrange multiplier λ_{vol} and the weight factor w_{vol} defined at [4] should be included as:

```
      lvol = lvol-2*wf*sqr(intvfVol-vTarget);
      wf = min (gamma*wf, wMax);
```

createFields.H MODIFICATIONS

After that, the createFields.H file must be updated by the new set up sensitivity calculation and include the new variables used. To do so, a zero value dimensioned scalar should be defined for the volumetric Lagrange multiplier λ_{vol}, in order to avoid a negative value through its update.

It can be included after the zeroAlpha definition:

```
dimensionedScalar zeroLvol(dimless, Zero);
```

Before the alpha variable definition (line 109), the new variables to calculate the sensitivity are included:

```
//domain volume
dimensionedScalar vol
(
"vol",
dimless,
laminarTransport
);

//minimum value of interpolation function
dimensionedScalar alphaMin
(
       "alphaMin",
       dimless/dimTime,
       laminarTransport
);

//grey control
dimensionedScalar q
(
       "q",
       dimless,
       laminarTransport
);

//volume of each cell
dimensionedScalar vCell
(
       "vCell",
       dimless,
       laminarTransport
);

//Minimum volume value occupied by solid
dimensionedScalar vTarget
(
       "vTarget",
       dimless,
       laminarTransport
);
```

The previous values have been defined as constants entries that must be selected according to the case of modeling. Now, the design variable γ, label as `vf` in the code, is specified as an entry value along the patches of the domain (as the state and adjoint variables), to have an initial guess for its calculation. As well its minimum and maximum value are defined, of zero and 1 respectively.

```
//Minimum value of design variable
dimensionedScalar zeroVf
(
       "zeroVf",
       dimless,
       laminarTransport
);

//Maximum value of the design variable
dimensionedScalar vfMax
(
       "vfMax",
       dimless,
       laminarTransport
);

volScalarField vf
(
    IOobject
    (
        "vf",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Then the variables to update the volumetric Lagrange multiplier and the weight factor are defined

```
//lagrange multiplier lambda vol (lvol0)
dimensionedScalar lvol0
(
       "lvol0",
       dimless,
       laminarTransport
);

//volumetric weight factor
dimensionedScalar wf
(
       "wf",
       dimless,
       laminarTransport
);

//volumetric weight factor Max
dimensionedScalar wMax
(
       "wMax",
       dimless,
       laminarTransport
```

```
);

//time-step factor to update wf
dimensionedScalar gamma
(
       "gamma",
       dimless,
       laminarTransport
);
```

At the sensitivity calculation, some constants with value of 1 should be included to overcome units troubles, because every variable introduced has been defined as dimensionless:

```
//constants for units managing
dimensionedScalar cte1
(
       "cte1",
       dimTime,
       laminarTransport
);

dimensionedScalar cte2
(
"cte2",
sqr(dimLength)/sqr(dimTime),
laminarTransport
);

dimensionedScalar cte3
(
"cte3",
sqr(dimTime)/sqr(dimLength),
1.0
);
```

As the α variable has ben defined as the interpolation function, the equation for the material modeling should be updated:

```
volScalarField alpha
(
    IOobject
    (
        "alpha",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
```

```
    ),
    alphaMax + (alphaMin - alphaMax)*(vf*(1 + q)/(vf + q))
);
```

After the “zeroCells” line, the volumetric Lagrangian value should be defined using the previous constants:

```
volScalarField lvol
(
    IOobject
    (
        "lambdaVol",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    max(lvol0-2*wf*sqr(1-(alpha/alphaMax)-vf), zeroLvol)
);

scalar intvfVol=0.0;
```

Then, the sensitivity definition is updated:

```
volScalarField sens
(
       IOobject
       (
              "sensitivity",
              runTime.timeName(),
              mesh,
              IOobject::READ_IF_PRESENT,
              IOobject::AUTO_WRITE
       ),
       (Ua&U)*vol*((alphaMin-alphaMax)*q*(1+q)/((vf+q)*(vf+q)))*cte1+cte2*(-
lvol+2*wf*sqr(intvfVol-vTarget))*2*vCell
);
```

Finally the name of the new solver should be updated from “myAdjointShapeOptimizationFoam” to “frozenTopologyOptimization”, as it is an algorithm to calculate topology optimization distribution based on the frozen turbulence assumption. To do so, enter the following commands:

```
mv myAdjointShapeOptimizationFoam.C frozenTopologyOptimization.C
sudo       i/myAdjointShapeOptimizationFoam/frozenTopologyOptimization/g
Make/files
cd ..
```

```
mv myAdjointShapeOptimizationFoam frozenTopologyOptimization
```

Now the solver is ready to be used, compile by entering the following:

```
cd frozenTopologyOptimization
wclean
wmake
```

1.7. Optimization Algorithm

The Augmented Lagrange Methods for turbulent flow, constrained optimization comprises the following steps:

a) Initialize the porosity field α and the Lagrangian multipliers λ_k (the latter with zero values). The penalty factor w takes on a user-defined value. A constant γ with which the value of w is multiplied in each optimization cycle along with the maximal allowed w^{max} value are also defined. The initial value of w along with the γ and w^{max} values are quantities linked with the imposition of constraints that the user has to specify.

b) Solve the state equations in which the porosity-dependent terms have been added to the turbulence model and momentum equations.

c) Compute the values of the objective and constraint functions. Terminate the algorithm if L is less than a threshold value.

d) Solve the adjoint equations, including the adjoint to the turbulence model equations.

e) Compute the sensitivities $\delta L/\delta\alpha$.

f) Update the porosity field using the steepest descent formula

$$a^{n+1} = a^n - \eta \frac{\delta L}{\delta \alpha}\Big|^n$$

where n is the optimization cycle index and η an user defined step. The porosity value is allowed to vary within the range $[0, \alpha_{max}]$. The maximum porosity value needs to have a sufficiently large value in order to zero the velocities in the solidified parts of the porous media.

g) Update λ_k and w as follows:

$$\lambda_k^{n+1} = \lambda_k^n - 2wc_k^n$$

$$w_k^{n+1} = \min(\gamma w^n, w^{max})$$

h) Repeat step b).

1.8. Setting up the Case

The structure of a case is similar to a tutorial in OpenFOAM. A good case could be starting from the “pitzdaily” which can be located at:

```
cd $FOAM_TUTORIALS/incompressible/adjointShapeOptimizationFoam
cp –r pitzDaily $FOAM_RUN/tutorials/
```

The folder can be adapted to solve a state case of Topology Optimization based in [1]. At the present document the “diffuser” case is solved, therefore the name of the case should be stablished as such:

```
mv pitzDaily diffuser
```

The domain of the case consists in a square with an inlet and outlet parabolic velocity, with maximum value of 1 and 3 respectively.

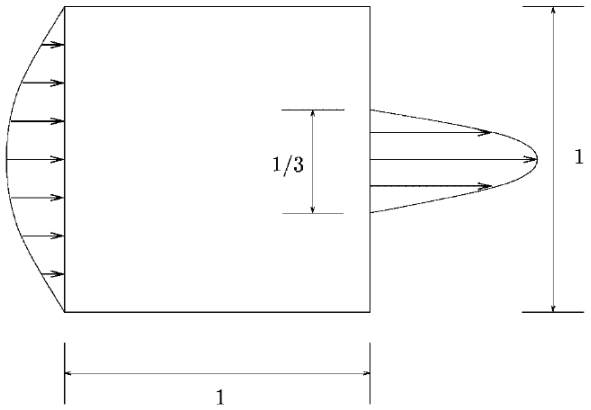


Figure 4. Design domain for the diffuser example. [1]

The folders at the case must be modified at following files:

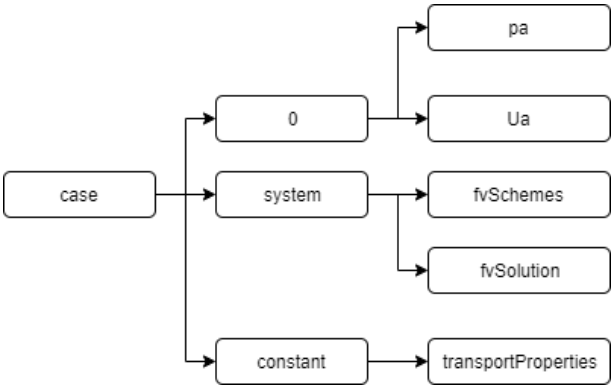


Figure 5. Diffuser case structure folders.

Starting by the 0 folder, where the boundary conditions of the adjoint variables are stated, they depend on the objective function to be evaluated, e.g. total pressure losses or power dissipation. At [8] the power dissipation boundary condition is implemented and will be used. By calculating such objective function, the boundary conditions of the state and adjoint variables are stated as:

Table 1. Boundary conditions of state and adjoint variables.

	Inlet	**Outlet**	**Wall**
(adjoint Velocity) **Ua**	parabolicVelocity max. Value 1	adjointOutletVelocityPower	NoSlip
(state velocity) **U**	ParabolicVelocity Max. Value 1	ParabolicVelocity Max. Value 3	NoSlip
(adjoint pressure) **pa**	zeroGradient	adjointOutletPressurePower	zeroGradient
(state pressure) **p**	ZeroGradient	FixedValue	zeroGradient

At the `system/fvSolution` file, the objective function must be called and the patches where the optimization will be made [8]:

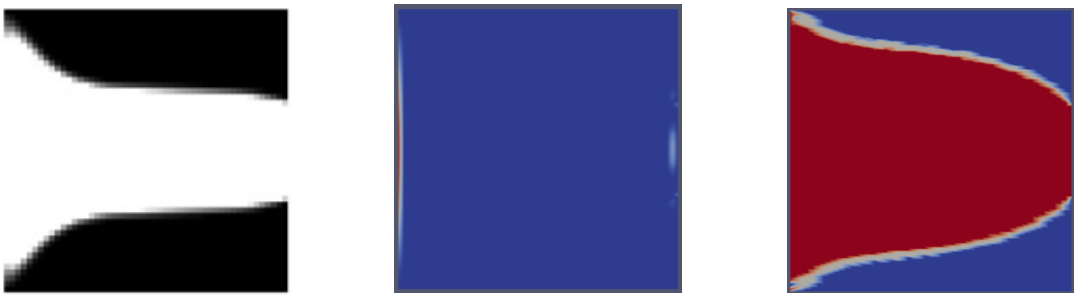
```
objectiveFunctionDict
{
    objectiveFunction         1;
    numberObjectivePatches    2;
    objectivePatchesNames     (inlet outlet);
}
```

Finally, every new variable defined at the sensitivity calculation must be placed at the constant/transportProperties file. It is highlighted that these parameters must be adjusted according to the case:

```
   object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel  Newtonian;

nu              1;

lambda          1;
alphaMax        25000;
vol          0.05;
alphaMin     0.00025;
q            0.1;
lvol0        0;
wf           1;
cte1         1;
cte2         1;
vCell        0.000005;
vTarget      0.5;
vfMax        1;
wMax         1000;
gamma        10;
cte3         1;
```

As the optimization problem is not considering the turbulence phenomena, it is recommended to switch off its parameter at the `turbulenceProperties` file.

At following the solvers “myAdjointShapeOptimizationFoam” and “frozenOptimization” are compared by using a 100x100 mesh on the domain:



(a) (b) (c)

Figure 6. Diffuser results from (a) benchmark case, (b) myAdjointShapeOptimizationFoam and (c) frozenOptimization solver

It can be seen that the developed solver “frozenOptimization” is able to provide an accurate result than “myAdjointShapeOptimizationFoam”. It is encouraged to the reader vary the values at the transportProperties file to obtain better results and check its influence.

2. Turbulence modeling at Topology Optimization

The impact of neglecting the differentiation of the turbulent viscosity (frozen turbulence) had lead that depending on the problem, the computed sensitivities can differ from FD even in computed sign. The approach using the continuous adjoint for high Reynolds recommends the use of the adjoint wall functions as the incomplete differentiation of the turbulence model with wall functions may lead to bad results. Also, the strict mesh requirements associated with low-Re turbulence models (particularly close to the solid walls), latter cannot always be used in industrial applications. In contrast, high Reynolds numbers turbulence models, employing the wall functions technique are usually used. The need to obtain accurate sensitivity derivatives for these applications dictates the necessity for differentiating high-Re turbulence models, and the law of the wall.

In low-re turbulence models the multipliers of flow variables variations are set to zero in the boundary integrals in order to derive the adjoint boundary conditions; similarly, when dealing with High-Re turbulence models, flow variations at the wall boundaries should be expresses as functions of the friction velocity variation, the multiplier of which should be set to zero in order to derive the adjoint high-Re boundary conditions. The differentiation of the high-Re turbulence models using the continuous adjoint method for the $k-\epsilon$ can be found at [5], introducing the adjoint friction velocity and the so-called adjoint wall functions. The theory and implementation were bases on an in-house, vertex- centred finite volume code, using a pseudo-compressibility scheme to solve the primal and adjoint equations for incompressible flows. The implementation of the primal wall functions is based on a slip velocity condition, where the "real" solid is assumed to lie at a distance Δ_{rw} underneath the grid boundary. The differentiation of the incompressible high-Re Spalart-Allmaras model can be found at [4], based at the cell-centered pressure-based implementation, where a no-slip velocity boundary condition is imposed on the wall boundaries along with the law of the wall, expressed by a single formula governing both the linear sublayer and logarithmic part of the boundary layer.

2.1. Spalart Allmaras modified turbulence model

A porosity dependent term is also added to the Spalart-Allmaras model equation, yielding:

$$R^{\tilde{\nu}} = u_j \frac{\partial \tilde{\nu}}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\tilde{\nu}}{\sigma}\right)\frac{\partial \tilde{\nu}}{\partial x_j}\right] - \frac{c_{b2}}{\sigma}\left(\frac{\partial \tilde{\nu}}{\partial x_j}\right)^2 - \tilde{\nu}P(\tilde{\nu}) + \tilde{\nu}D(\tilde{\nu}) + \underbrace{\alpha\tilde{\nu}}_{T_{a,\tilde{\nu}}} = 0 \quad (17)$$

This allows the computation of zero $\tilde{\nu}$ values in the solidified parts, whereas the original Spalart-Allmaras equation is solved in the fluid parts [4].

Calculating the distance nearest wall method

The law of the wall is applied to the fluid-solid interfaces, S_a, after following the steps described:

- Given a porosity distribution, track the fluid-solid interfaces. This is done by employing a simple criterion: let face f belongs to cells C_1 and C_2 (Figure 7).

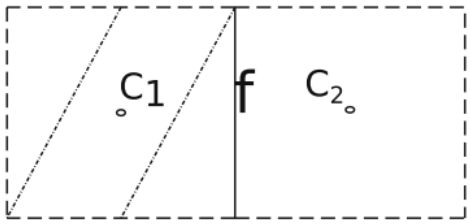


Figure 7. Internal mesh face f belongs to cells $C1$ and $C2$. If $C1$ belongs to the solidified domain where $C2$ to the fluid domains, face f is added to the fluid-solid interface.

If $\alpha(C_1) \cdot \alpha(C_2) = 0$, i.e. are orthogonal vectors, and $\alpha(C_1) + \alpha(C_2) > \epsilon$[1], then add face f to the list with the internal mesh faces comprising the fluid-solid interface.

- Identify which of the C_1, C_2 cells belongs to the fluid domain by checking whether $\alpha(C_1) > 0$ or $\alpha(C_2) > 0$. Based on the velocity magnitude of this cells, compute the friction velocity, v_τ as:

$$f_{WF} = -e^{-\kappa B}[e^{\kappa v} - 1 - \kappa v^+ - \frac{(\kappa v^+)^2}{2} - \frac{(\kappa v +)^3}{6} + y^+ - v^+ = 0$$
$$; B \approx 5.5, \quad y_p^+ = \frac{\Delta^p v_\tau}{v}, \quad v_p^+ = \frac{|v_i|^p}{v_\tau} \text{ and } \quad v_\tau^2 = -\left[(v + v_t)\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)\right]^f n_j t_j^I \tag{18}$$

where n_j and t_j^I are the components of the normal to the wall and parallel to the velocity at the first cell P (considered as parallel velocity component to the wall) unit vectors. The indices f denote quantities defined at the boundary wall face and p the first cell center (Figure 8).

[1] where ϵ is a user defined infinitesimally small positive number and $\alpha = 0 \ \therefore \ \alpha \leq \epsilon$ corresponds to a fluid region and $\alpha \neq 0 \ \therefore \ \alpha > \epsilon$ corresponds to a solidified region.

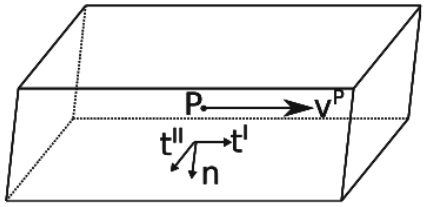


Figure 8. Finite volume adjacen to the wall, where $\boldsymbol{n}$ is the outwards normal unit vector, $\boldsymbol{t}^{I}$ is parallel to the velocity velocity vector at first cell center P and $\boldsymbol{t}^{\boldsymbol{II}} = e_{ijk} n_j t_k^I$.

- Compute the "artificial" $v_t|^f$ value using:

$$v_t^f = -\frac{v_\tau^2}{\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)^f n_j t_i^I} - v \approx |Pf| \frac{v_\tau^2}{v_i^P t_i^I} - v \tag{19}$$

  Here, $v_i|^f = 0$ is not imposed as a "hard" conditions, but practically results from the fact that f is located at the fluid-solid boundary.

- Compute the viscous flux at f through:

$$-\left[(v + v_t)\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v}{\partial x_i}\right)\right]^f n_j \approx -\left(v + v_t^f\right) \frac{v_i^f - v_i^P}{|Pf|} \tag{20}$$

  i.e. the normal velocity gradient at f is computed through a local FD scheme. However, any differentiation normal to the boundary must be avoided on the coarses meshes used with high Re turbulence models. So that error made due the discretization of the normal velocity gradient is corrected by computing and using an artificial v_t^f, so that the wall shear stress and that computed by differentiating the velocity field in space and multiplying by v_t^f be equal.

Regarding the rest of the domain boundaries, typical boundary conditions imposed for internal aerodynamics simulations are used. Then, the Hamilton-Jacobi equation is used to compute the distances from the evolving fluid-solid interface, S_a (See derivation at [4] section 4.4.1.) [4].

2.2. Turbulent State Equations

An optimization problem defined as minimize the objective function F, subject to E equality constraints can be expressed as:

$$\begin{aligned} \text{Min} \quad & F \\ \text{Subject to} \quad & c_k = 0, k = 1, \dots, E \end{aligned}$$

The Augmented Lagrange Multiplier (ALM) method is used to cope with equality constraints. In the ALM, the Lagrangian function L is defined as:

$$L = F - \lambda_k c_k + w c_k^2 \tag{21}$$

where λ_k is the $k-th$ Lagrange multipliers and w a scalar weight factor. Since the restrictions of each variable are set to zero, i.e. $R^p = R_i^v = R^{\tilde{v}} = R^{\Delta} = 0$, an augmented objective function L_{aug}, can be defined and minimized instead, defined as:

$$L_{aug} = L + \int_{\Omega} q R^p \, d\Omega + \int_{\Omega} u_i R_i^v \, d\Omega + \int_{\Omega} \tilde{v}_a R^{\tilde{v}} \, d\Omega + \int_{\Omega} \Delta_a R^{\Delta} \, d\Omega \tag{22}$$

Since in topology optimization there are no changes in the computational domain and mesh, the total derivate is equal to the partial derivative:

$$\frac{\delta \phi}{\delta \alpha} = \frac{\partial \phi}{\partial \alpha}, \qquad \frac{\delta_S \phi}{\delta \alpha} = \frac{\partial \phi}{\partial \alpha} \tag{23}$$

therefore, the variation in the augmented function with respect to the porosity variable is expressed as:

$$\begin{aligned} \frac{\delta L_{aug}}{\delta \alpha} = \frac{\delta F}{\delta \alpha} + (-\lambda_k + 2 w c_k) \frac{\delta c_k}{\delta \alpha} + \int_{\Omega} q \frac{\partial R^p}{\partial \alpha} d\Omega + \int_{\Omega} u_i \frac{\partial R_i^v}{\partial \alpha} d\Omega + \int_{\Omega} \tilde{v}_a \frac{\partial R^{\tilde{v}}}{\partial \alpha} d\Omega \\ + \int_{\Omega} \Delta_a \frac{\partial R^{\Delta}}{\partial \alpha} d\Omega \end{aligned} \tag{24}$$

The derivative depends of the objective function to be analyzed $\frac{\delta F}{\delta \alpha}$, and its constraints imposed, $\frac{\delta c_k}{\delta \alpha}$, further detail on a derivation for a general objective function or constrained can be found at [4], section 3.3.2. Focusing on the Navier Stokes equations, $\frac{\partial R^p}{\partial \alpha}$ and $\frac{\partial R^\Delta}{\partial \alpha}$ are equal to zero as are not expressed in terms of the design variable. Then, the next two integrals are defined as:

$$\int_\Omega u_i \frac{\partial(\alpha v_i)}{\partial \alpha} d\Omega = \int_\Omega u_i v_i \, d\Omega + \int_\Omega \alpha u_i \frac{\partial v_i}{\partial \alpha} d\Omega \tag{25}$$

$$\int_\Omega \widetilde{\nu_a} \frac{\partial(\alpha \check{\nu})}{\partial \alpha} d\Omega = \int_\Omega \check{\nu} \widetilde{\nu_a} \, d\Omega + \int_\Omega \alpha \widetilde{\nu_a} \frac{\partial \check{\nu}}{\partial \alpha} d\Omega \tag{26}$$

2.3. Field Adjoint Equations

Therefore, the turbulent field adjoint equations can be read as:

$$\begin{aligned}
R^q &= -\frac{\partial u_j}{\partial x_j} = 0 \\
R_i^u &= u_j \frac{\partial v_j}{\partial x_i} - \frac{\partial (v_j u_i)}{\partial x_j} - \frac{\partial}{\partial x_j}\left[(\upsilon + \upsilon_t)\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\right] + \frac{\partial q}{\partial x_i} + \widetilde{\nu_a}\frac{\partial \check{\nu}}{\partial x_i} \\
&\quad - \frac{\partial}{\partial x_i}\left(\widetilde{\nu_a}\check{\nu}\frac{C_Y}{Y} e_{mjk}\frac{\partial v_k}{\partial x_j} e_{mli}\right) + \underbrace{\alpha u_i}_{T_{a,u}} = 0 \\
R^{\tilde{\nu}_a} &= -\frac{\partial (v_j \tilde{\nu}_a)}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\upsilon + \frac{\tilde{\nu}}{\sigma}\right)\frac{\partial \tilde{\nu}_a}{\partial x_j}\right] + \frac{1}{\sigma}\frac{\partial \tilde{\nu}_a}{\partial x_j}\frac{\partial \tilde{\nu}}{\partial x_j} + \frac{2c_{b2}}{\sigma}\frac{\partial}{\partial x_j}\left(\tilde{\nu}_a \frac{\partial \tilde{\nu}}{\partial x_j}\right) + \tilde{\nu}_a \tilde{\nu} C_{\tilde{\nu}} \\
&\quad + \frac{\partial v_t}{\partial \tilde{\nu}}\frac{\partial u_i}{\partial x_j} \times \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) + (-P + D)\tilde{\nu}_a + \underbrace{\alpha \tilde{\nu}_a}_{T_{a,\tilde{\nu}_a}} = 0 \\
R^{\Delta_a} &= -2\frac{\partial}{\partial x_j}\left(\Delta_a \frac{\partial \Delta}{\partial x_j}\right) + \tilde{\nu}\tilde{\nu}_a C_\Delta = 0
\end{aligned} \tag{27}$$

The terms marked as $T_{a,u}$ and $T_{a,\tilde{\nu}_a}$ results from the differentiation of the porosity dependent terms, indicating that u_i and $\tilde{\nu}_a$ will have practically zero values in the solidified parts of the porous media domain.

2.4. Turbulent Adjoint Boundary conditions

As the terms $T_{a,v}$ and $T_{a,\tilde{v}}$ of the primal equations do not contain a differential operator, they do not contribute any additional term to the adjoint boundary conditions. Furthermore, in topology optimization the total derivative is equal to the partial derivative, leading to conclude boundary condition of topology optimization problem based on the high-Re Spalart-Allmaras model is identical to the presented at shape optimization which is explained at following.

After satisfying the field adjoint equations, the remaining terms of the augmented objective function gradient for topology optimization, where non parametrized wall boundary is used (S_{W_p}), reads as:

$$\frac{\delta F_{aug}}{\delta a_n} = \int_S BC_i^u \frac{\partial v_i}{\partial a_n} dS + \int_S \left(u_j n_j + \frac{\partial F_{S_i}}{\partial p} n_i \right) \frac{\partial p}{\partial a_n} dS + \int_S \left(-u_i n_j + \frac{\partial F_{S_k}}{\partial \tau_{ij}} n_k \right) \frac{\partial \tau_{ij}}{\partial a_n} dS \tag{28}$$

where the sub-index S at the integrals stands for the boundaries to be analysed, either inlet, outlet or at the walls.

2.4.1. Inlet Boundaries, S_I

At the inlet boundaries, $\delta v_i/\delta a_n = \partial v_i/\partial a_n$=0, then the first integral vanishes. The second and third integrals must take the following values to accomplish the equal zero condition:

$$\begin{aligned} u_{\langle n \rangle} &= u_j n_j = -\frac{\partial F_{S_{I,i}}}{\partial p} n_i \\ u^I_{\langle t \rangle} &= \frac{\partial F_{S_{I,k}}}{\partial \tau_{ij}} n_k t^I_i n_j + \frac{\partial F_{S_{I,k}}}{\partial \tau_{ij}} n_k t^I_j n_i \\ u^{II}_{\langle t \rangle} &= \frac{\partial F_{S_{I,k}}}{\partial \tau_{ij}} n_k t^{II}_i n_j + \frac{\partial F_{S_{I,k}}}{\partial \tau_{ij}} n_k t^{II}_j n_i \end{aligned} \tag{29}$$

where t^I_i, t^{II}_i are the components of the tangent surface unit vector and $u^I_{\langle t \rangle}$, $u^{II}_{\langle t \rangle}$ are the corresponding components of the adjoint velocity.

2.4.2. Outlet Boundaries, S_o

Along the outlet boundaries $\delta p/\delta a_n = \partial p/\partial a_n$=0, which makes vanish the second integral of the equation $\delta F_{aug}/\delta a_n$. Also, by assuming an almost uniform velocity profile along S_o, the third integral can be neglected. This leaves that the first integrand to be zeroed

2.4.3. Unparameterized/fixed wall boundaries, S_W

For high Reynolds values using wall functions, the equation for the Spalart Allmaras turbulence model varies as

$$\frac{\delta F_{aug}}{\delta a_n} = \int_S BC_i^u \frac{\partial v_i}{\partial a_n} dS + \int_S \left(u_j n_j + \frac{\partial F_{S_i}}{\partial p} n_i\right) \frac{\partial p}{\partial a_n} dS + \int_S \left(-u_i n_j + \frac{\partial F_{S_k}}{\partial \tau_{ij}} n_k\right) \frac{\partial \tau_{ij}}{\partial a_n} dS + \int_S BC^{\widetilde{v_a}} \frac{\partial \tilde{v}}{\partial a_n} dS - \int_S \widetilde{v_a} \left(v + \frac{\tilde{v}}{\sigma}\right) \frac{\partial}{\partial a_n} \left(\frac{\partial \tilde{v}}{\partial x_j}\right) n_j \, dS + \int_S 2\Delta_a \frac{\partial \Delta}{\partial x_j} n_j \frac{\partial \Delta}{\partial b_n} dS \quad (30)$$

Since S_W is fixed, the partial and total derivatives of any flow quantity are identical and the total derivatives of the normal and tangent unit vectors are equal to zero. Due to the Dirichlet condition imposed on $\tilde{v}$, the fourth integral vanishes. To make the equation independent of $\frac{\partial}{\partial a_n}\left(\frac{\partial \tilde{v}}{\partial x_j}\right) n_j$, a zero Dirichlet condition is imposed on $\widetilde{v_a}$. To eliminate the dependency on $\frac{\partial p}{\partial a_n}$, the normal adjoint velocity must be equal to

$$u_{\{n\}} = -\frac{\partial F_{S_{W,i}}}{\partial p} n_i \quad (31)$$

By further developing the first and third integrals a Dirichlet condition for $u_{\langle t \rangle}^{II}$ results along with the following expression

$$u_\tau^2 = (\nu + \nu_t)\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) n_j t_i^I = 0 \quad (32)$$

which can be characterized as the square adjoint friction velocity and its role is similar to that of the primal friction velocity, i.e. is used to compute the adjoint viscous flux in order to complete the adjoint momentum equilibrium at the first cell adjacent to S_W (Figure 8). The adjoint friction velocity is an indispensable part of the adjoint system of equations to the high-Re Spalart-Allmaras model, because the long distance between f(face) and P(center), differentiating normal to the wall is prone to important errors, established at [4] section 5.3.

2.5. Turbulent Sensitivity Derivatives

After satisfying the field adjoint eqautions and their boundary conditions, the rmaining terms giving the gradient expressions reads as

$$\frac{\delta L_{aug}}{\delta \alpha} = \underbrace{\int_{\Omega} v_i u_i \, d\Omega}_{SD_1} + \underbrace{\int_{\Omega} \widetilde{\nu} \widetilde{\nu_a} \, d\Omega}_{SD_2} + \underbrace{\int_{\Omega} \left[\acute{F}_{\Omega}^{\alpha} + (-\lambda_k + 2wc_k) \acute{c_k}_{\Omega}^{\alpha} \right] d\Omega}_{SD_3} \tag{33}$$

The terms $\acute{F}_{\Omega}^{\alpha}$ and $\acute{c_k}_{\Omega}^{\alpha}$ denote the direct dependency of the objective and constraint functions on the porosity variable. In areas where v_i and u_i form an obtuse angle, the local sensitivity is negative leding to an increase in the local porosity value (solid), and reversely in areas with acute angle.

3. Turbulent Adjoint Solver: AdjointOptimizationFoam

The solver is label as “adjointOptimizationFOAM” and its first version has been released at the OpenFOAM version 1906 since June of 2019. It is a functionality targeting automated gradient-based optimization loop assisted by the continuous adjoint method that supports multiple types of optimization (shape, topology etc). The solver is being constructed and up to his point works solves the state, adjoint and sensitivities equations of incompressible steady-state turbulent flow, which forms the highlighted parts of the following optimization loop:

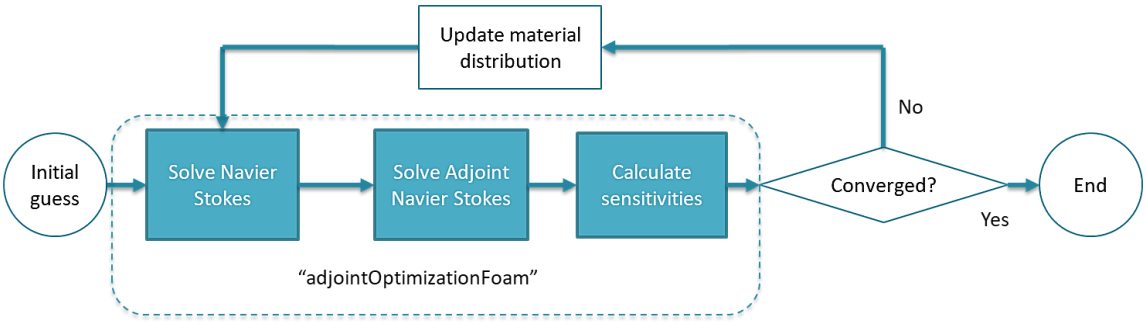


Figure 5.7.1. Turbulent adjoint solver functions in OpenFOAM.

The advantage of considering this solver is the inclusion of turbulent effects. The solver structure is explained at the adjointOptimizationFoam.C file, starting by the classes that solve the fundamental tools of finite volume calculation (fvCFD.H), the class uncharged of making the optimization which has not been released yet just constructed (optimizationManager.H), the one of solving the state equations if necessary (primalSolver.H) and finally the one to calculate adjoint turbulent equations manager (adjointSolverManager.H):

```
#include "fvCFD.H"
#include "optimisationManager.H"
#include "primalSolver.H"
#include "adjointSolverManager.H"
```

Then, each class parameter is called to solve the optimisation problem, as stated in Figure , starting with the |time loop solve the primal equations, then the adjoint equations an finally the sensitivity calculation. It can be seen that the optimisation library is called as well to close the loop of the optimisation, but it has not been created yet, there is only the structure of the software:

```
    Info<< "\nStarting time loop\n" << endl;

    for (om++; !om.end(); om++)
    {
        Info<< "* * * * * * * * * * * * * * * * * * * *" << endl;
        Info<< "Time = " << runTime.timeName()          << endl;
        Info<< "* * * * * * * * * * * * * * * * * * * *" << endl;

        if (om.update())
        {
            om.updateDesignVariables();
        }
        else
        {
            om.solvePrimalEquations();
        }

        om.updatePrimalBasedQuantities();

        om.solveAdjointEquations();

        om.computeSensitivities();
    }
    Info<< "End\n" << endl;

    return(0);
}
```

To control every function of the adjoint software, the createFields.H uses the optimisation manager declared as:

createFields.H:

```
    autoPtr<optimisationManager> optManagerPtr
    (
        optimisationManager::New(mesh)
    );
    optimisationManager& om = optManagerPtr();
```

and composed by the following directories in OpenFOAM:

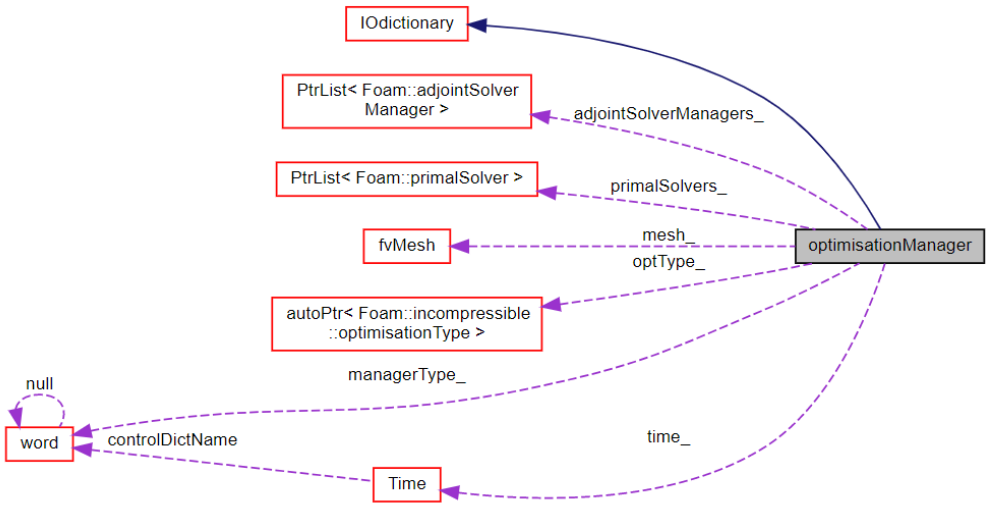


Figure 5.7.2. Optimisation Manager structure

each one pointing to the function controller solver, i.e. the controller of the primal solver resolution, the adjoint solver resolution and the optimisation type to be used, at this case incompressible optimisation only available. Inside each function controller solver can be found the was as the coupling of the state variables are made, e.g. SIMPLE algorithm for the primal solver resolution. At following the adjoint turbulent solver function is explained.

Make

The folder is composed by two directories label as “files” and “options”.

- Files

At first the name of the solver is specified, followed by the commund that creates the execution script of the developed solver, at this case “adjointOptimisationFoam”.

```
adjointOptimisationFoam.C

EXE = $(FOAM_APPBIN)/adjointOptimisationFoam
```

- Options

The different libraries created in the OpenFOAM source are called such as finite volume calculations, turbulence models, turbulence models used at incompressible flow for this case, transport models, and for this case, the calculation of the adjoint, which will be analyzed.

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -
I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/optimisation/adjointOptimisation/adjoint/lnInclude


EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -lincompressibleTransportModels \
    -ladjointOptimisation
```

3.1. adjointOptimisation solver

To compute the adjoint equations and the sensitivities, the adjointSolverManager is called in the code as following:

```
void Foam::adjointSolverManager::solveAdjointEquations()
{    for (adjointSolver& solver : adjointSolvers_)
    {
    objectiveManager& objManager = solver.getObjectiveManager();
    objManager.updateAndWrite();
    solver.solve();    }}
```

being linked at different structures inside the code as shown in Figure .

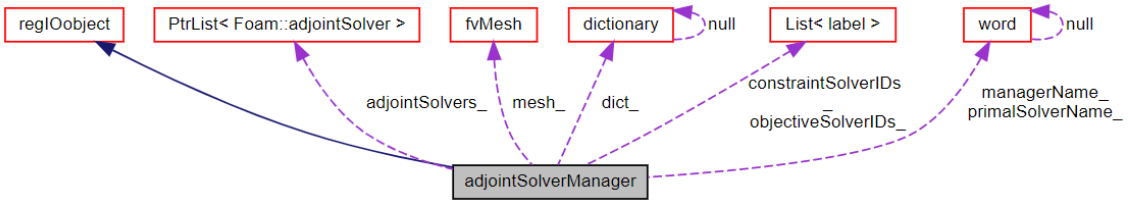


Figure 5.7.3. adjointSolverManager pointers structure.

which ended up being composed by the adjointSimple solver, found at:

```
$FOAM_SRC/optimization/solvers/adjointSolvers/incompressible/adjointSimple
```

The main function of the adjointSimple solver starts by linking the time step iteration with the faces fluxes of the state equations results:

```
void Foam::adjointSimple::solveIter()
{
    const Time& time = mesh_.time();
    Info<< "Time = " << time.timeName() << "\n" << endl;

    const surfaceScalarField& phi = primalVars_.phi();
```

And define the adjoint references: get the adjoint variables (`getAdjointVars`), the adjoint pressure defined as a volumetric scalar field (`pa = adjointVars.paInst`), the adjoint velocity as a volumetric vector field (`Ua = adjointVars.UaInst`), the adjoint flux as a surface scalar field (`phia = adjointVars.phiaInst`), the adjoint turbulence called as an object of the adjointRASModel of the incompressibleAdjoint class (`adjointTurbulence=adjointVars.adjointTurbulence`) and assigning the reference cell and reference value the adjoint pressure:

```
// Grab adjoint references
    incompressibleAdjointVars& adjointVars = getAdjointVars();
    volScalarField& pa = adjointVars.paInst();
    volVectorField& Ua = adjointVars.UaInst();
    surfaceScalarField& phia = adjointVars.phiaInst();
    autoPtr<incompressibleAdjoint::adjointRASModel>& adjointTurbulence
=
        adjointVars.adjointTurbulence();
    const label&  paRefCell  = solverControl_().pRefCell();
    const scalar& paRefValue = solverControl_().pRefValue();
```

Then the momentum equation is defined

```
  // Momentum predictor
    //~~~~~~~~~~~~~~~~~~~

    tmp<fvVectorMatrix> tUaEqn
    (
        fvm::div(-phi, Ua)
      + adjointTurbulence-
>divDevReff(Ua)
      + adjointTurbulence-
>adjointMeanFlowSource()
      ==
        fvOptionsAdjoint_(Ua)
    );

fvVectorMatrix& UaEqn =
tUaEqn.ref();
```

$$\nabla \cdot (-\phi U_a) + DivDevReff(U_a) + adjointMeanFlowSource(\,) = U_a$$

$$DivDevReff(U_a) = -\nabla\left(\nabla v_{eff} U\right) - \nabla \cdot \left(v_{eff} \nabla (UT)\right)$$

$$adjointMeanFlowSource = average\ if\ chosen$$

Therefore the initial equation is:

$$-\nabla \cdot (U_a) - \nabla\left(\nabla v_{eff} U_a\right) - \nabla \cdot \left(v_{eff} \nabla (U_a T)\right) = U_a$$

Then, the boundary condition, the objective function, the addition of ATC terms and the source of the Optimisation type are set, to calculate the adjoint velocity equation (`UaEqn`):

```
    UaEqn.boundaryManipulate(Ua.boundaryFieldRef());

    objectiveManagerPtr_().addUaEqnSource(UaEqn);

    ATCModel_->addATC(UaEqn);

    addOptimisationTypeSource(UaEqn);
```

Then the adjoint velocity equation is relaxed, and the adjoint constraint is applied to the same:

```
    UaEqn.relax();

    fvOptionsAdjoint_.constrain(UaEqn);
```

Finally, the adjoint velocity equation is equaled to the adjoint pressure, request to be solved and correct the adjoint velocity:

```
    if
(solverControl_().momentumPredictor())
    {
        Foam::solve(UaEqn == -
fvc::grad(pa));
```

$$U_a = -\nabla p_a$$

```
        fvOptionsAdjoint_.correct(Ua);
    }
```

After that, a Simple adjoint momentum corrector solver is implemented to couple the adjoint variables. The way is implemented obeys the same order as its defined in the simpleFoam solver.

3.2. Sensitivity calculation

As presented in section 2.5, the sensitivity calculation is obtained after the augmented Lagrangian derivation with respect to its design variable:

$$\frac{\delta L_{aug}}{\delta \alpha} = \underbrace{\int_{\Omega} v_i u_i \, d\Omega}_{SD_1} + \underbrace{\int_{\Omega} \widetilde{v}\,\widetilde{v_a} \, d\Omega}_{SD_2} + \underbrace{\int_{\Omega} \left[\acute{F}_{\Omega}^{\alpha} + (-\lambda_k + 2wc_k)\acute{c}_{k\,\Omega}^{\alpha} \right] d\Omega}_{SD_3}$$

The implementation of the sensitivity calculation can be found at:

```
$FOAM_SRC/optimisation/adjointOptimisation/adjoint/optimisation/adjointSensitiv
ity/incompressible/adjointSensitivity
```

At the end of the adjoint sensitivity member function, the sensitivity calculation can be found as:

```
// Compute dxdb multiplier
flowTerm =
ATCModel->getFISensitivityTerm()
- fvc::grad(p) * Ua
- nuEff*(gradU & (gradUa + T(gradUa)))
+ (- nuEff*(gradUa & (gradU + T(gradU)))
       + fvc::grad(nuEff * Ua & (gradU + T(gradU)))
+ (pa * gradU)
// from the adjoint turbulence model
       + turbulenceTerm.T()
// Term 7, term from objective functions
       + objectiveContributions;
```

which in vectorial form can be written as:

$$ATC + \nabla p * U_a - v_{eff} * \left(\nabla u \cdot (\nabla u_a + \nabla u^T)\right) + (-v_{eff} * \left(\nabla u_a \cdot (\nabla u_a + \nabla u^T)\right) + \nabla(v_{eff} * U_a \cdot (\nabla u_a + \nabla u^T) + p_a * \nabla u + turbulenceTerm + objectiveContributions$$

3.3. AdjointOptimisationFoam user manual

The present section describes the available options for using the adjointOptimisationFoam solver focusing in topology optimisation entries for turbulent flow mainly. To check the other entries available and shape optimisation, the reader is recommended to check [6].

3.3.1. optimisationDict

Located at "system/optimisationDict", is composed by the following commands:

`optimisationManager singleRun;`	➢ The "singleRun" entry is the only available at this point of the solver, which indicates that the primal and adjoint equations are solved without performing an optimisation loop.

`primalSolvers`
`{`
 	
`op1`
`{`
`active true;`
 	
`type incompressible;`
`solver simple;`
 	
`useSolverNameForFields false;`
	➢ The primalSolvers dictionary is where the solver of the primal equations are defined.
	➢ The "active" entry defines if the primal equations corresponding to the solver are going to be solved or not.
	➢ Only the incompressible option is available.
	➢ Solution algorithm to solve the primal equations. Entrances available:
	"simple": replace the behavior of simpleFoam.
	"RASTurbulenceModel": solve the turbulence models PDEs.
	➢ If not specified it means that is in default as "false". When specified as "true", all flow variable names related to this solver will pe appended with the solver name (e.g. "U" would become "Uop1"). Then, the entries in fvSolution and fvSchemes have to

	appropriately be adapted manually. If set to true, boundary conditions will be read in the following way: If a file exists with the specific name (e.g. "Uop1"), boundary conditions will be read from there, otherwise, the code attempt to read the base file (e.g. "U"). If it fails, the code will exit with an appropriate error message.

`solutionControls`
`{`
`consistent yes;`
`nIters 1000;`
`residualControl`
`{`
`"p.*" 1.e-5;`
`"U.*" 1.e-5;`
`}`
	➢ Here are the entries to manage the solution process of the primal equations, e.g. when using the "simple" solver, the entries would be read through system/fvSolution/SIMPLE, which are not going to be explained here.

`averaging` `{`	➢ This is an optional entry that controls averaging of the primal fields during the solution of the primal equations. It is mainly used to feed the adjoint equations with averaged primal fields.
`average true;`	➢ It is set as false by default. If it is set as true, all primal field related to the solver will be averaged (e.g. U, p, phi, turbulence models, etc.).

`startIter 500;`
`}`
`}`
`}`
`}`
	➢ Start iteration of the averaging process.

`adjointManagers`
`{`
`adjManager1`
`{`
	➢ One "adjointManager" should be defined for each primal solver present in the "primalSolvers" dictionary. It is responsible for the adjoint PDEs to be solved at the corresponding operating point.

`primalSolver op1;`	➢ Indicate the name of primal solver (e.g. op1)
`operatingPointWeight 1;`	➢ Its default value is 1. Stablishes the weight factor of using multiple objective functions, i.e. $J = \sum_i w_i^{op} J_j^{op}$, where w_i^{op} is the weight factor.

`adjointSolvers`
`{`
`adjS1`
`{`
	➢ Here a list of dictionaries are defined to set up the adjoint solvers to be used in this point. One set of adjoint PDEs will be solved for each adjoint solver and one corresponding set of sensitivity derivatives will be computed. Multiple adjointSolvers can be used if sensitivities of multiple objectives must be computed separately from each other.

`//choose adjointsolver` `active true;`	➢ True is stablished by default, and specifies whether the adjoint equations are going to be solved for this adjointSolver.
`Type incompressible;`	➢ Only incompressible is available.
`Solver adjointSimple;`	➢ Solution algorithm used to solve the adjoint equations. Only adjointSimple is available.
`useSolverNameForFields false;`	➢ "false" is stablished as default. Its equivalent at the defined in the primalSolvers section. Should be set to "true" if more than one adjointSolver is present.
`computeSensitivities true;`	➢ The default values is "true". Specifies if should be computed sensitivity derivatives after solving the adjoint equations.

`// manage objectives`
`objectives`
`{`
`type incompressible;`
	➢ Type of objective functions to be constructed. Ony incompressible is valid for the moment.

```
        objectiveNames
```

- Here the list of dictionaries corresponding to the objective functions to be minimized is specified. Each objective function value is written in a file located in the "optimisation" folder, under "objective/TimeName/objective Name+AdjointSolverName. One set of adjoint equations is solved for each "adjointSolver", minimizing the weighted sum of the objectives declared in "objectiveNames".

```
            {
                lift
                {
              weight   1.;
               type  force;
patches (pressure suction);
direction     (0.3 -0.9 0);
Aref                   2.;
rhoInf              1.225;
UInf                    1;
                }
            }
        }
```

- The entries in each dictionary under objectiveNames depend on the objective type. The two mandatory entries are:
  "type" (force, moment, PtLosses)
  "weight" objective Function weight. (if multiple objective functions are used).

```
// ATC treatment
ATCModel
{
```

- Here the options of the "Adjoint Transpose Convection (ATC) term are provided, which exists in the adjoint momentum equations. The ATC is numerically stiff and can often cause convergence difficulties for the adjoint equations. The ATCModel dict provides some options to smooth it in order to facilitate convergence in industrial cases.

```
        ATCModel standard;
```

- Available entrances: standard, UaGradU, cancel
  The "standard" computes it as $u_j \frac{\partial v_j}{\partial x_i}$, where u is the adjoint velocity vector and v the

	primal velocity vector. It is formulated by differentiating the non-conservative form of the convection term in the primal momentum equations. "UaGradU" computes the ATC term as $-v_j \frac{\partial u_j}{\partial x_i}$ and is formulated by differentiating the conservative form of the convection term in the primal momentum equations. "cancel" excludes the ATC term from the adjoint momentum equations during the solution of the adjoint PDEs.
`extraConvection 0;`	➢ Default value is zero. In order to facilitate convergence, add and subtract the adjoint convection term this many times, using slightly different discretization schemes in order to add numerical dissipation.
`nSmooth 0;`	➢ Default value of zero. Propagate the smoothing of the ATC term applied to the cells collected through zeroATCPatchTypes and zeroATCZones, by using a Laplacian-like filter nSmooth times.
`zeroATCPatchTypes ();`	➢ Defaults to an "empty" wordlist. Zero the ATC term next to patches of the provided types. No zeroing will be conducted if the wordlist is empty.
`ZeroATCZones ();`	➢ Similar to the previous described, but works on the provided cellZones.
`maskType faceCells;`	➢ Default is "faceCells" but can be specified as pointCells. It stablishes how will the cells next to the zeroATCPatchTypes will be chosen for smoothing the ATC term. If
`}`	

	faceCells is used, every cell having a face in the zeroATCPatchTypes boundaries will be chosen whereas if pointCells is used, every cell that has a point in the zeroATCPatchTypes will be used.

`// solution control`
`solutionControls`
`{`
`nIters 3000;`
 	
`ResidualControl`
`{`
`"pa.*" 1.e-6;`
`"Ua.*" 1.e-6;`
`"nuaTilda.*" 1.e-6;`
`}`
`}`
`}`
`}`
	➢ Has entries to manage the solution process of the adjoint equations. Its entries are the same as the ones in the solutionControls dictionary of the primalSolvers dict. Averaging can be applied to the adjoint fields, and the mean adjoint fields will be used to compute the sensitivity derivatives. Additional entries read:

`PrintMaxMags false;`	➢ Established as false by default. Define whether to print the maximum values of the adjoint field to the log file. These can be useful indicators of simulation stability.

If sensitivity derivatives are computed, then the following optimisation part should be included:

`optimisation`
`{`
`sensitivities`
`{`
 	
`type surfacePoints;`
 	
`patches (lower upper);`
`}`
`}`
	➢ At the sensitivities dict, the setup for the computation of sensitivity derivatives is provided. Sensitivities will be computed after all adjoint PDEs are solved, for the adjoint solvers for which “computeSensitivities” is set to “true”. Only two entries are mandatory:
	➢ Avialable options: “surface”, “surfacePoints” and “sensitivityMultiple”.
	➢ The patches to compute sensitivities.

`optimisation/sensitivity/type: surface`

The “surface” entry is used to compute the sensitivity maps, i.e. the derivative of the objective function with respect to the normal displacement of the boundary wall faces. It is related to mesh movement in the domain of shape optimisation, therefore is not described, except for the “includeDistance” entry related to the Spalart-Allmaras turbulence model. The way as the “surface” entry looks is shown at following, the other entries should be set as “false” because “true” is the default value.

Table 2. “surface” entries at sensitivities calculation.

```
type                            surface ;
patches                         ( "wall . * " ) ;
includeSurfaceArea              false;
includeObjectiveContribution    false;
includeMeshMovement             false;
includeDistance                 true;
```

IncludeDistance: Used for cases including the adjoint to the Spalart-Allmaras turbulence model. If is set to “true” the boundary conditions for the adjoint distance field should be set, which is a `volScalarField` named as “`da`”. The BC should be set as “fixedValue” for inlet and oulet boundaries, and “zeroGradient” for walls. Then, the adjoint field “da” is generated automatically by the code, unless read from the current time-step folder. In addition, a solver for “da” should be added to “fvSolution” along with a realaxation factors for the “da” equation. A discretization scheme for $div(-iPhi, da)$ should be added in “$fvSchemes/divSchemes$”. The adition of the “$adjointEikonalSolver$” is not necessary entry as it is working for topology optimisation.

`optimisation/sensitivity/type: surfacePoints`

It accomplishes the same function as “surface”, but sensitivities are computed with respect to the normal displacement of boundary points, not faces. When sensitivity maps are of interest, this option is preferred to “surface” since some of the terms included in the computations (e.g. variation in the normal vector) are better posed when differentiating with respect to points.

`optimisation/sensitivity/type: multiple`

Ir provides a framework for computing multiple types of sensitivity derivatives. Sensitivities will be computed for all sub-dictionaries in sensTypes.

Table 3. "multiple" example entrie at sensitivies definition for optimisation.

```
sensitivities
{
  type          multiple ;
  patches      ( lower upper ) ;
  sensTypes
  {
       faces
       {
         type    surface ;
         patches ( lower upper ) ;
       }
       points
       {
         type     surfacePoints ;
         patches  ( lower upper ) ;
       }
   }
}
```

3.3.2. fvSolution and fvSchemes

- fvSolution

Relaxation factors for the adjoint turbulence variables are generally small (≈ 0.1) for industrial cases. A relaxation of about 0.5 is utilized when solving the adjoint distance PDE for da.

- FvSchemes

Additional entries need to be provided in all subDicts of fvSchemes in order to solve the adjoint PDEs. A $divScheme$ of the form of $div(-phi, adjointField)$ should be used for the convection term of the adjoint mean flow and turbulence model PDEs. Also, a $div(-yPhi, da)$ should be used for the adjoint distance convection term. A first order scheme, like $Gauss\ upwind$ might be needed to ensure convergence in challenging industrial cases. An example of the $divScheme$ is provided at Table 4.

Table 4. divScheme used at tutorial naca0012 turbulent.

```
divSchemes
{
    default            Gauss linear;
    div(phi,U)         bounded Gauss linearUpwind gradUConv;
    div(-phi,Ua)       bounded Gauss linearUpwind gradUaConv;
    div(yPhi,yWall)            Gauss linearUpwind gradDConv;

    div(phi,nuTilda)   bounded Gauss linearUpwind gradNuTildaConv;
    div(-phi,nuaTilda) bounded Gauss linearUpwind gradNuaTildaConv;
    div(-yPhi,da)              Gauss linearUpwind gradDaConv;
}
```

3.3.3. Turbulence modeling

Inside the constant folder, a directory label as "adjointRASProperties" specifies the adjoint turbulence properties as follows:

```
adjointRASModel adjointSpalartAllmaras;
```

- There are two options available: "adjointLaminar" and "adjointSpalartAllmaras". The first one is used when solving laminar flow or using the "frozen turbulence assumption". The second one solves the PDEs of the adjoint to the Spalart Allmaras turbulence model. At using it, BC should be set for `nuaTilda`.

```
adjointSpalartAllmarasCoeffs
{
    nSmooth           0;
    zeroATCPatchTypes ();
    maskType          faceCells;
}

adjointTurbulence on;
```

- This entry is optional and its used for smoothing out numerically challenging terms.

3.3.4. Adjoint boundary conditions

These files are provided at the "0" folder and the type of adjoint BCs to be applied in each patch depends on the type of primal BCs used there. The guidelines provided are for the adjoint velocity (U_a), the adjoint pressure (p_a), adjoint turbulence mode ($nuaTilda$) and adjoint distance (da). For constrained patches like symmetry, cyclic, etc.., the same BC types on the primal fields should also be applied to their adjoint counterparts. Table 5 presents a summary of the available BC.

Table 5. Adjoint boundary conditions.

	Adjoint Velocity (U_a)	**Adjoint Pressure** (p_a)	**Adjoint modified viscosity** "nuaTilda" $(\widetilde{\nu_a})$	**Normal distance** (d_a)
Inlet $u = cte$ $\nabla p = 0$ $v_t = cte$	`adjointInletVelocity`	`ZeroGradient`	`AdjointOnletNuaTilda`	`FixedValue uniform 0`
Outlet $\nabla u = 0$ $p = cte,$ $\nabla v_t = 0$	`adjointOutletVelocity`	`adjointOutletPressure`	`adjointOutletNuaTilda`	`FixedValue uniform 0`
	`AdjointOutletVelocityFlux (back-flow observed)`		`AdjointOutletNuaTilda (back flow observed)`	
Wall $u = cte$ $\nabla p = 0$	`adjointWallVelocity`		`fixedValue`	`zeroGradient`
	`adjointWallVelocityLowRe`			
Freestream InletOutlet	`adjointFarFieldVelocity`	`adjointFarFieldPressure`	`adjointFarFieldNuaTilda`	

STUDY QUESTIONS

1. Different objective functions can be considered during an optimisation as total pressure loses or entropy generation. By considering such, how will be affected the programming on the topology optimisation solvers?
2. During the first step of the solver use, a discretization of the domain is made. How will affect the mesh density in topology optimisation ?
3. After analizing the turbulent adjoint solver “adjointOptimisationFoam”, it can be seen that an optimisation is not feasible yet, why is it the reason?
4. What will be the objective of developing a turbulent adjoint solver with a complex structure as the recent released one?

REFERENCES

[1] T. Borrvall and J. Petersson, “Topology optimization of fluids in Stokes flow,” *Int. J. Numer. Methods Fluids*, vol. 41, no. 1, pp. 77–107, 2003.

[2] J. S. Romero and E. C. N. Silva, “A topology optimization approach applied to laminar flow machine rotor design,” *Comput. Methods Appl. Mech. Eng.*, vol. 279, pp. 268–300, 2014.

[3] OpenFOAM® v1906: New and improved numerics. (n.d.). Retrieved October 27, 2019, from https://www.openfoam.com/releases/openfoam-v1906/numerics.php#numerics-adjoint

[4] Papoutsis-Kiachagias, E. M., & Giannakoglou, K. C. (2016). Continuous Adjoint Methods for Turbulent Flows, Applied to Shape and Topology Optimization: Industrial Applications. *Archives of Computational Methods in Engineering*, *23*(2), 255–299. https://doi.org/10.1007/s11831-014-9141-9

[5] Zymaris, A. S., Papadimitriou, D. I., Giannakoglou, K. C., & Othmer, C. (2010). Adjoint wall functions: A new concept for use in aerodynamic shape optimization. *Journal of Computational Physics*, *229*(13), 5228–5245. https://doi.org/10.1016/j.jcp.2010.03.037

[6] goleman, daniel; boyatzis, Richard; Mckee, A. (2019). Manual adjointOptimisationFoam. *Journal of Chemical Information and Modeling*, *53*(9), 1689–1699. https://doi.org/10.1017/CBO9781107415324.004

[7] F. Moukalled, L. Mangani, and M. Darwish, *The finite volume method in computational fluid dynamics : An Advanced Introduction with OpenFOAM and Matlab*, vol. 113. 2016.

[8] U. Nilsson, “Description of AdjointShapeOptimizationFoam and how to implement new cost functions,” pp. 1–40, 2013.

ANNEXES

A1. Code to create “myAdjointShapeOptimization”

```
#!/bin/sh
mkdir topologyOptimization
cd $WM_PROJECT_DIR
cp -r applications/solvers/incompressible/adjointShapeOptimizationFoam
$WM_PROJECT_USER_DIR/applications/solvers/topologyOptimization
cd $WM_PROJECT_USER_DIR/applications/solvers/topologyOptimization
mv adjointShapeOptimizationFoam myAdjointShapeOptimizationFoam
cd myAdjointShapeOptimizationFoam
mv adjointShapeOptimizationFoam.C myAdjointShapeOptimizationFoam.C
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
sed -i s/adjointShapeOptimizationFoam/myAdjointShapeOptimizationFoam/g
Make/files
#modification of the steepest descent algorithm
sed -i s/"alpha + lambda"/"alpha - lambda"/g myAdjointShapeOptimizationFoam.C
#creating the power dissipation BC
cp -r adjointOutletVelocity/ adjointOutletVelocityPower
cp -r adjointOutletPressure/ adjointOutletPressurePower
cd adjointOutletPressurePower
sed s/adjointOutletPressure/adjointOutletPressurePower/g
adjointOutletPressureFvPatchScalarField.C >
adjointOutletPressurePowerFvPatchScalarField.C
sed s/adjointOutletPressure/adjointOutletPressurePower/g
adjointOutletPressureFvPatchScalarField.H >
adjointOutletPressurePowerFvPatchScalarField.H
cd ../adjointOutletVelocityPower
sed s/adjointOutletVelocity/adjointOutletVelocityPower/g
adjointOutletVelocityFvPatchVectorField.C >
adjointOutletVelocityPowerFvPatchVectorField.C
sed s/adjointOutletVelocity/adjointOutletVelocityPower/g
adjointOutletVelocityFvPatchVectorField.H >
adjointOutletVelocityPowerFvPatchVectorField.H
cd ..
rm adjointOutletPressurePower/adjointOutletPressureFvPatchScalarField.*
rm adjointOutletVelocityPower/adjointOutletVelocityFvPatchVectorField.*
#adding the BC
```

```
sed -i '2 a
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C'
Make/files
sed -i '32 a #include "RASModel.H"'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '104 a scalarField Up_n = phip / patch().magSf(); // Primal'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '105 a scalarField Uap_n = phiap / patch().magSf(); // Adjoint'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '106 a const incompressible::RASModel& rasModel
=db().lookupObject<incompressible::RASModel>("TurbulenceProperties");'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '107 a scalarField nueff =
rasModel.nuEff()().boundaryField()[patch().index()];'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '108 a const scalarField& deltainv = patch ().deltaCoeffs(); //distance
inverse'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '109 a scalarField Uaneigh_n = (Uap.patchInternalField() &
patch().nf());'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '111 a /*'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '113 a */'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '114 a operator== ((Up&Uap) + (Up_n*Uap_n) + nueff*deltainv*(Uap_n-
Uaneigh_n) - 0.5*mag(Up)*mag(Up) - (Up & patch().Sf()/patch().magSf())*(Up &
patch().Sf()/patch().magSf()));'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
#adjoint velocity power dissipation modification
sed -i '3 a
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C'
Make/files
sed -i '32 a #include "RASModel.H"'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '93 a const fvsPatchField<scalar>& phip
=patch().lookupPatchField<surfaceScalarField, scalar>("phi");'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '97 a const fvPatchField<vector>& Uap =
patch().lookupPatchField<volVectorField, vector>("Ua");'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
```

```
sed -i '101 a const incompressible::RASModel& rasModel =
db().lookupObject<incompressible::RASModel>("TurbulenceProperties");'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '102 a scalarField nueff =
rasModel.nuEff()().boundaryField()[patch().index()];'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '103 a const scalarField& deltainv = patch().deltaCoeffs();'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '104 a scalarField Up_ns = phip/patch().magSf();'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '105 a vectorField Up_t = Up -
(phip*patch().Sf())/(patch().magSf()*patch ().magSf());'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '106 a //tangential component of adjoint velocity in neighbouring node'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '107 a vectorField Uaneigh = Uap.patchInternalField();'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '108 a vectorField Uaneigh_n = (Uaneigh & patch().nf())*patch().nf();'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '109 a vectorField Uaneigh_t = Uaneigh - Uaneigh_n;'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '110 a vectorField Uap_t = ((Up_ns*Up_t) +
nueff*deltainv*Uaneigh_t)/(Up_ns+nueff*deltainv);'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '111 a vectorField Uap_n =
(phiap*patch().Sf())/(patch().magSf()*patch().magSf());'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '117 a /*'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '120 a */'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '121 a operator==(Uap_t+Uap_n);'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
#including sensitivity
sed -i '244 a sens=Ua&U;' myAdjointShapeOptimizationFoam.C
sed -i '$ a volScalarField
sens(IOobject("sensitivity",runTime.timeName(),mesh,IOobject::READ_IF_PRESENT,I
Oobject::AUTO_WRITE),Ua&U);' createFields.H
#printing the cost function
sed -i '$ a dictionary optFunc =
mesh.solutionDict().subDict("objectiveFunctionDict");' createFields.H
```

```
sed -i '$ a int nObjPatch =
optFunc.lookupOrDefault<scalar>("numberObjectivePatches", 0);' createFields.H
sed -i '$ a int objFunction =
optFunc.lookupOrDefault<scalar>("objectiveFunction", 0);' createFields.H
sed -i '$ a wordList objPatchNames=optFunc.lookup("objectivePatchesNames");'
createFields.H
sed -i '$ a Info<< "Initializing objective function calculation" << endl;'
createFields.H
sed -i '$ a Info<< "The objective function chosen is" << objFunction<<endl;'
createFields.H
sed -i '$ a Info<< "Name of the patches for which the cost function will be
calculated" << objPatchNames<<endl;' createFields.H
sed -i '$ a Info<< "Number of patches" << nObjPatch<<endl;' createFields.H
sed -i '$ a label objPatchList [nObjPatch];' createFields.H
sed -i '$ a int iLoop;' createFields.H
sed -i '$ a for (iLoop=0; iLoop<nObjPatch; iLoop++){' createFields.H
sed -i '$ a objPatchList [iLoop] =
mesh.boundaryMesh().findPatchID(objPatchNames[iLoop]);}' createFields.H
sed -i '103 a #include "costFunction.H"' myAdjointShapeOptimizationFoam.C
touch costFunction.H
echo "scalar jDissPower(0);">>costFunction.H
sed -i '$ a for (iLoop=0; iLoop<nObjPatch; iLoop++)' costFunction.H
sed -i '$ a {' costFunction.H
sed -i '$ a if (objFunction==1) {' costFunction.H
sed -i '$ a jDissPower = jDissPower -
sum(phi.boundaryField()[objPatchList[iLoop]]*(p.boundaryField()[objPatchList
[iLoop]] + 0.5*magSqr(U.boundaryField()[objPatchList[iLoop]])));'
costFunction.H
sed -i '$ a }' costFunction.H
sed -i '$ a }' costFunction.H
sed -i '$ a if (objFunction==1) {' costFunction.H
sed -i '$ a Info<<"Objective Function (Power Dissipated)
J:"<<jDissPower<<endl;}' costFunction.H
#modifications to the code
sed -i '33 a #include "IncompressibleTurbulenceModel.H"'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '34 a #include "turbulentTransportModel.H"'
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C
sed -i '33 a #include "IncompressibleTurbulenceModel.H"'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
sed -i '34 a #include "turbulentTransportModel.H"'
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C
```

A2. OptimisationManager.H

```
#include "runTimeSelectionTables.H"
#include "IOdictionary.H"
#include "optimisationTypeIncompressible.H"
#include "primalSolver.H"
#include "adjointSolverManager.H"


namespace Foam
{

/*---------------------------------------------------------------------*\
                    Class optimisationManager Declaration
\*---------------------------------------------------------------------*/

class optimisationManager
:
    public IOdictionary
{
protected:

    // Protected data

        fvMesh& mesh_;
        Time& time_;
        PtrList<primalSolver> primalSolvers_;
        PtrList<adjointSolverManager> adjointSolverManagers_;
        const word managerType_;
        autoPtr<incompressible::optimisationType> optType_;


private:

    // Private Member Functions

        //- Disallow default bitwise copy construct
        optimisationManager(const optimisationManager&) = delete;

        //- Disallow default bitwise assignment
        void operator=(const optimisationManager&) = delete;


public:

    //- Runtime type information
    TypeName("optimisationManager");


    // Declare run-time constructor selection table
```

```
        declareRunTimeSelectionTable
        (
            autoPtr,
            optimisationManager,
            dictionary,
            (
                fvMesh& mesh
            ),
            (mesh)
        );


    // Constructors

        //- Construct from components
        optimisationManager(fvMesh& mesh);


    // Selectors

        //- Return a reference to the selected turbulence model
        static autoPtr<optimisationManager> New(fvMesh& mesh);


    //- Destructor
    virtual ~optimisationManager() = default;


    // Member Functions

        virtual PtrList<primalSolver>& primalSolvers();

        virtual PtrList<adjointSolverManager>&
adjointSolverManagers();

        virtual bool read();

        //- Prefix increment,
        virtual optimisationManager& operator++() = 0;

        //- Postfix increment, this is identical to the prefix
increment
        virtual optimisationManager& operator++(int) = 0;

        //- Return true if end of optimisation run.
        //  Also, updates the design variables if needed
        virtual bool checkEndOfLoopAndUpdate() = 0;

        //- Return true if end of optimisation run
        virtual bool end() = 0;

        //- Whether to update the design variables
        virtual bool update() = 0;

        //- Update design variables.
        //  Might employ a line search to find a correction
satisfying the step
```

```
        //  convergence criteria
        virtual void updateDesignVariables() = 0;

        //- Solve all primal equations
        virtual void solvePrimalEquations();

        //- Solve all adjoint equations
        virtual void solveAdjointEquations();

        //- Compute all adjoint sensitivities
        virtual void computeSensitivities();

        //- Solve all primal equations
        virtual void updatePrimalBasedQuantities();
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * //

} // End namespace Foam
```