CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Description of adjointShapeOptimizationFoam and how to implement new objective functions

---

Developed for OpenFOAM-2.2.x

Author:
Ulf Nilsson

Peer reviewed by:
Daniel Lindblad
Olivier Petit

January 20, 2014

Table of contents

Introduction

CFD simulations have gained an increased importance in product design and development applications. The advantages of optimizing shape parameters to increase efficiency have grown in importance in many idustrial branches. The car industry being the perfect example, where a small reduction in drag results in both economical gains for the customer and environmental benefits, increasing the competitiveness of the product.
However the optimization problem is not trivial. Considering the problem of reducing the drag of a geometric shape the decision variables are continuous functions and could result in a very large number of variables when discretized. Hence, finding an efficient optimization algorithm is important.
In this report a solver applying a gradient based method, using the adjoint approach to compute the sensitivities with respect to the design variables, is studied. The design variable, the porosity of each cell, α, is then used to punish the unfavourable cells. The update is done in a "one-shot" approach, using partially converged fields to compute the sensitivity and a steepest descent algorithm to update the porosity. A short summary of the theory will be included, however for more detailed derivations the reader is referred to the articles given below.
The report aims to describe `adjointShapeOptimizationFoam` and how modifications can be made to the code to solve the topology optimization problem for different cost functions. In order to be able to validate (to some extent) the results, descriptions of how to calculate the cost function and sensitivity of each cell will be presented.
A short description of the current implementation will be included, to motivate the necessary modifications. Describing the code of `adjointShapeOptimizationFoam` also gives an opportunity to identify possible problems with the solver.

Othmer, C. & de Villiers, E. & Weller, H.G. (2007). Implementation of a continuous adjoint for topology optimization of ducted flows. *American Institute of Aeronautics and Astronautics*, AIAA-3947.

Othmer, C. (2008). A continuous adjoint formulation for the computation of topological and surface sensitives of ducted flows. *Int. J. Numer. Meth. Fluid*, 58, 861-877.

Chapter 1

Theory

A general optimization problem includes an objective function to be minimized or maximized subject to different constraints. In the case of a general cost function, in our case depending on flow variables $\mathbf{v}$ and p and the design variable α, subject to the constraint of fulfilling the incompressible, steady state Navier-Stokes (NS) equations the problem can be written as

$$\begin{aligned} &\text{minimize} \;\; \mathrm{J} = \mathrm{J}(\mathbf{v}, p, \alpha) \\ &\text{such that} \;\; (\mathbf{v} \cdot \nabla)\mathbf{v} + \nabla p - \nabla \cdot (2\nu \mathrm{D}(\mathbf{v})) + \alpha \mathbf{v} = 0, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \nabla \cdot \mathbf{v} = 0. \end{aligned} \tag{1.1}$$

The above equation introduces the porosity as a sink term using Darcy's law. Cells contributing negatively to the cost function can therefore be punished by increasing the porosity. The kinematic viscousity is defined as the sum of molecular and turbulent viscousity and the strain rate tensor is denoted $D(\mathbf{v}) = \frac{1}{2}\left(\nabla \mathbf{v} + (\nabla \mathbf{v})^T\right)$. I.e.the constraints are the Reynolds averaged NS using the Boussinesq assumption to model the Reynold stresses. Introducing Lagrange multipliers the Lagrangian relaxation of the problem reads

$$\text{minimize} \;\; \mathrm{L} := \mathrm{J} + \int_{\Omega} (\mathbf{u}, q) R \mathrm{d}\Omega. \tag{1.2}$$

In eq. 1.2 Ω refers to the flow domain and R to the incompressible RANS and continuity equation (i.e. the constraints in eq. 1.1), while the adjoint velocity and pressure, $(\mathbf{u}, q)$, are introduced as Lagrange multipliers. Hence the notation $(\mathbf{u}, q)R$ denotes the fact that each of the constraint is multiplied with a Lagrange multiplier and the contribution is added together and summed over the total domain. To arrive at the desired sensitivity for the Lagrange relaxation with respect to design variables the total variation of L, eq. 1.3, is studied and the adjoint velocity and pressure chosen in such a way that the variation with respect to the other variables, $(\mathbf{v}, p)$, vanishes, according to

$$\delta L = \delta_{\alpha} L + \delta_{\mathbf{v}} L + \delta_p L, \tag{1.3}$$

$$\delta_{\mathbf{v}} L + \delta_p L = 0. \tag{1.4}$$

Eq. 1.3 and 1.4 give the expression for the sensitivity of the cost function with respect to the porosity in cell i as

$$\frac{\partial L}{\partial \alpha_i} = \mathbf{u}_i \cdot \mathbf{v}_i V_i, \tag{1.5}$$

where V_i is the volume of cell i. After some derivation, the condition in eq. 1.4 gives the adjoint Navier-Stokes equations eq. 1.6-1.7. It should be noted that even though the terminology implies a physical meaning of the adjoint variables they should not be interpreted as a velocity or a pressure in the physical sense. The names are rather used to emphasize the similarities with the primal

variables, suggesting that a similar solution procedure can be applied.

$$-2\mathrm{D}(\mathbf{u})\mathbf{v} = -\nabla q + \nabla \cdot (2\nu \mathrm{D}(\mathbf{u})) - \alpha \mathbf{u} - \frac{\partial \mathrm{J}_\Omega}{\partial \mathbf{v}}, \tag{1.6}$$

$$\nabla \cdot \mathbf{u} = \frac{\partial \mathrm{J}_\Omega}{\partial p}. \tag{1.7}$$

For cost functions which do not contain any contribution from the flow domain, but only from the boundaries, the term including J_Ω, where $_\Omega$ denote the volume part of the domain excluding the boundaries, is equal to zero and hence adjoint momentum, eq. 1.6, and pressure, eq. 1.7, do not change when varying between cost functions of this type. The variations enter the solver only through the boundary conditions, which can be derived from eq. 1.8 and 1.9, using $_\Gamma$ to denote the boundary of the domain.

$$\int_\Gamma \left(\mathbf{n}(\mathbf{u}\cdot\mathbf{v}) + \mathbf{u}(\mathbf{v}\cdot\mathbf{n}) + 2\nu\mathbf{n}\cdot D(\mathbf{u}) - q\mathbf{n} + \frac{\partial \mathrm{J}_\Gamma}{\partial \mathbf{v}} \right) \delta\mathbf{v} \, \mathrm{d}\Gamma$$

$$- \int_\Gamma 2\nu\mathbf{n}\cdot D(\delta\mathbf{v})\cdot\mathbf{u} \, \mathrm{d}\Gamma = 0, \tag{1.8}$$

$$\int_\Gamma \left(\mathbf{u}\cdot\mathbf{n} + \frac{\partial \mathrm{J}_\Gamma}{\partial p} \right) \delta p \, \mathrm{d}\Gamma = 0. \tag{1.9}$$

The above equation stem from the same conditions as eq. 1.6 and 1.7, hence a decomposition of the expressions has been done (including integration by parts) into a contribution from the domain and a contribution from the boundaries of the domain, resulting in eq. 1.6-1.9.
The primal equations together with the adjoint, eq. 1.6-1.9, is the general form of the optimization problem. The derivation of the adjoint equations from the condition stated in eq. 1.4 includes, apart from a few rows of derivations, also an approximation known as "frozen turbulence". This approximation is used when neglecting the variation of ν when the adjoint equations are derived, which is correct for laminar, but not turbulent regions. Neglactations done in the derivations of the equations together with the frozen turbulence approximation are the main reasons for the inaccuracies when solving the adjoint equations. For more information about the derivations, validity of the implementation and inaccuracies the reader is reffered to [2].

1.1 Boundary conditions

In the following sections boundary conditions for the adjoint quantities for two different cost functions will be presented. The primal boundary conditions can be seen in *Table 1.1*, note that the derivations in the following sections are only valid for those conditions. Since the scope of this project is to cover the implementation of the conditions, the derivation of the boundary conditions will not be included. Only the final expressions of different cost functions will be presented, eq. 1.10-1.12 and 1.13-1.14, calculated from the general expressions, eq. 1.8 and 1.9, with derivations valid for ducted flows.

	Wall	Inlet	Outlet
$\mathbf{v}$	No-slip	Prescribed value	Zero gradient
p	Zero gradient	Zero gradient	$p = 0$

Table 1.1: Boundary conditions for the primal quantities $\mathbf{v}$ and p.

Wall and inlet boundary conditions

$$\mathbf{u}_t = 0, \tag{1.10}$$

$$u_n = -\frac{\partial J_\Omega}{\partial p}, \tag{1.11}$$

$$\mathbf{n} \cdot \nabla q = 0. \tag{1.12}$$

Outlet boundary conditions

$$q = \mathbf{u} \cdot \mathbf{v} + u_n v_n + \nu(\mathbf{n} \cdot \nabla) u_n + \frac{\partial J_\Gamma}{\partial v_n}, \tag{1.13}$$

$$0 = v_n \mathbf{u}_t + \nu(\mathbf{n} \cdot \nabla)\mathbf{u}_t + \frac{\partial J_\Gamma}{\partial \mathbf{v}_t}, \tag{1.14}$$

where the derivatives $\frac{\partial J_\Gamma}{\partial \mathbf{v}_t}$ and $\frac{\partial J_\Gamma}{\partial v_n}$ are the tangential and normal components of $\frac{\partial J_\Gamma}{\partial \mathbf{v}}$ respectively.

1.1.1 Power dissipation

The cost function of dissipated power is defined as

$$J := -\int_\Gamma (p + \frac{1}{2}v^2)\mathbf{v} \cdot \mathbf{n} \, \mathrm{d}\Gamma. \tag{1.15}$$

This is a cost function without any volumetric contribution, $J_\Omega = 0$ and $J_\Gamma = (p + \frac{1}{2}v^2)\mathbf{v} \cdot \mathbf{n}$. The boundary conditions for the wall and inlet reduces to

$$\begin{cases} \mathbf{u}_t & = 0, \\ u_n & = \begin{cases} 0 & \text{at wall,} \\ v_n & \text{at inlet,} \end{cases} \\ \mathbf{n} \cdot \nabla q & = 0. \end{cases} \tag{1.16}$$

The outlet boundary conditions for the adjoint velocity and pressure are defined as

$$\begin{cases} q & = \mathbf{u} \cdot \mathbf{v} + u_n v_n + \nu(\mathbf{n} \cdot \nabla) u_n - \frac{1}{2}v^2 - v_n^2, \\ 0 & = v_n(\mathbf{u}_t - \mathbf{v}_t) + \nu(\mathbf{n} \cdot \nabla)\mathbf{u}_t. \end{cases} \tag{1.17}$$

Where the pressure term has not been forgotten, but simply left out because of the fact that the boundary condition for the primal pressure is 0 on the outlet patch.

1.1.2 Flow uniformity at the outlet

The cost function describing uniformity at the outlet is defined below.

$$J := \int_{\Gamma_o} \frac{c}{2}(\mathbf{v} - \mathbf{v}^d)^2 \, \mathrm{d}\Gamma_o. \tag{1.18}$$

In eq. 1.18, c is a constant to maintain unit consistiency while $\mathbf{v}^d$ is the prescribed, wanted velocity in the outlet plane. As for the case with dissipated power, the volumetric contribution to the cost function is zero, however in this case the inlet boundary does not contribute, only the outlet Γ_o. The boundary conditions reduces to

$$\begin{cases} \mathbf{u} & = 0, \\ \mathbf{n} \cdot \nabla q & = 0, \end{cases} \tag{1.19}$$

at the wall and inlet and

$$\begin{cases} q & = \mathbf{u} \cdot \mathbf{v} + u_n v_n + \nu(\mathbf{n} \cdot \nabla) u_n + c(v_n - v_n^d), \\ 0 & = v_n \mathbf{u}_t + \nu(\mathbf{n} \cdot \nabla)\mathbf{u}_t + c(\mathbf{v}_t - \mathbf{v}_t^d), \end{cases} \tag{1.20}$$

for the outlet.

1.2 Steepest descent algorithm

Steepest descent is a gradient based algorithm, which for a linear system uses a search direction defined by

$$\mathbf{p}_k = -\nabla f(\mathbf{x}_k), \tag{1.21}$$

where $\mathbf{x}_k$ denotes the current location at iteration step k and f is a general function to be minimized, continuous in some neighbourhood around $\mathbf{x}_k \in \mathbb{R}^n$. An arbitary small change of the valiables, $\mathbf{x}_k$, in this direction obviously gives a reduction of the general cost function, f.[4]
In our case with the sensitivity defined in eq. 1.5, a steepest descent algorithm can be applied to update the porosity according to

$$\alpha_{n+1} = \alpha_n - \mathbf{u}_i \cdot \mathbf{v}_i V_i \delta. \tag{1.22}$$

In the above equation α_{n+1} and α_n are the new and old porosity values respectively, δ and V_i is the step-length and the volume of the cell considered respectively. The implementation of the steepest descent in `adjointShapeOptimizationFoam` also uses an underrelaxation factor to improve stability. The final form of the implementation can be written as

$$\alpha_{n+1} = \alpha_n(1-\gamma) + \gamma \min\left(\max\left((\alpha_n - \mathbf{u}_i \cdot \mathbf{v}_i V_i \delta\right), 0\right), \alpha_{max}\right), \tag{1.23}$$

where γ is the relaxation factor and α_{max} is a user defined upper limit to the porosity. Note the use of the min and max functions, limiting the porosity so that it never becomes negative and never greater than α_{max}. However α_{max} also limit each step length, reducing the impact of a large, negative value of the product $\mathbf{u}_i \cdot \mathbf{v}_i V_i$.

Chapter 2

adjointShapeOptimizationFoam

To be able to compute the sensitives by the use of an adjoint method it has been showed in the theory section that one has to solve two system of governing equations, the primal and the adjoint. The two include similar terms which enables the use of similar type of algorithms to solve the system. The solver implements the "one-shot" approach to solve the primal and adjoint systems to compute the sensitivities using the only partially converged quantities. The adjoint system is solved similarily to the primal, with a SIMPLE-type algorithm to couple the pressure and velocity. Below in figure 2.1 the solution procedure is graphically presented.

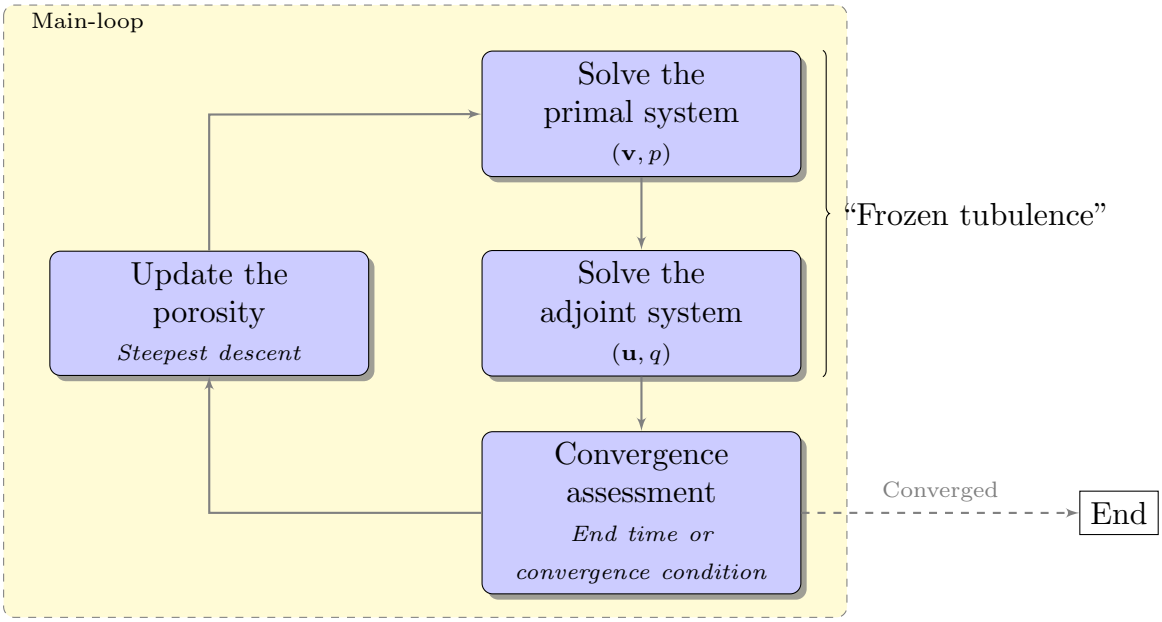


Figure 2.1: Block scheme of the solution procedure used in `adjointShapeOptimizationFoam`.

2.1 The solver

The adjointShapeOptimizationFoam solver is located in `$FOAM_SOLVERS/incompressible/adjointShapeOptimizationFoam`, which has a directory structure which can be seen in figure 2.2. It is based on `simpleFoam` and the differences can easily be identified by studying the uppermost levels of the code in `adjointShapeOptimizationFoam.C`. In the following sections the code will be examined in order to understand the implementation of the

theory explained in previous sections.

```
adjointShapeOptimizationFoam/
├── adjointContinuityErrs.H
├── adjointOutletPressure/
│   ├── adjointOutletPressureFvPatchScalarField.C
│   └── adjointOutletPressureFvPatchScalarField.H
├── adjointOutletVelocity/
│   ├── adjointOutletVelocityFvPatchVectorField.C
│   └── adjointOutletVelocityFvPatchVectorField.H
├── adjointShapeOptimizationFoam.C
├── createFields.H
├── createPhia.H
├── initAdjointContinuityErrs.H
└── Make/
    ├── options
    └── files
```

Figure 2.2: Directory structure for `adjointShapeOptimizationFoam`.

2.1.1 adjointShapeOptimizationFoam.C

The solver is initialized by including a number of files:

```
69 int main(int argc, char *argv[])
70 {
71         #include "setRootCase.H"
72         #include "createTime.H"
73         #include "createMesh.H"
74         #include "createFields.H"
75         #include "initContinuityErrs.H"
76         #include "initAdjointContinuityErrs.H"
```

Listing 2.1: file `adjointShapeOptimizationFoam.C`

Those header files are included from `src/OpenFOAM/lnInclude/` and `src/finiteVolume/lnInclude/` except for `createFields.H` and `initAdjointContinuityErrs.H` which are located in the `adjointShapeOptimizationFoam/` directory.
The first header file constructs an `argList` object, `args`, of the arguments provided to the main function. The case path and root path are checked together with the argument list. Additional information about the case, e.g. number of processors and processor directories, is stored.
`createTime.H` and `createMesh.H` construct an object of the class `Time`, `runTime`, and one of the class `fvMesh`, `mesh`, respectively. The `runTime` object contains time information, all the necessary counters and values needed from the control dictionary, while the object `mesh` contains information about the finite volume mesh, e.g. position of nodes, face areas and cell volumes, but also information about interpolation schemes to be used.

Line $71 - 75$ are included similarily in most solvers. As for `createFields.H`, it initializes the variables used by `adjointShapeOptimizationFoam`. This includes the primal and adjoint variables, as well as viscosity and other dimensioned scalars and the turbulence model. A few of the initializations are studied in the following section. As for `initAdjointContinuityErrs.H` it, similarily to its primal equivalent, initializes the cumulative adjoint continuity error by setting it to zero.

The start of the main loop of the solver is included below in Listing 2.2.

```
83  Info<< "\nStarting time loop\n" << endl;
84
85  while (simple.loop())
86  {
87          Info<< "Time = " << runTime.timeName() << nl << endl;
88
89          laminarTransport.lookup("lambda") >> lambda;
90
91          //alpha +=
92          //      mesh.relaxationFactor("alpha")
93          //   *(lambda*max(Ua & U, zeroSensitivity) - alpha);
94          alpha +=
95                  mesh.fieldRelaxationFactor("alpha")
96          *(min(max(alpha + lambda*(Ua & U), zeroAlpha), alphaMax) - alpha);
97
98          zeroCells(alpha, inletCells);
99          //zeroCells(alpha, outletCells);
100
101         // Pressure-velocity SIMPLE corrector
102         {
103                 // Momentum predictor
104
105                 tmp<fvVectorMatrix> UEqn
106                 (
107                 fvm::div(phi, U)
108                 + turbulence->divDevReff(U)
109                 + fvm::Sp(alpha, U)
110                 );
111
112                 UEqn().relax();
113
114                 solve(UEqn() == -fvc::grad(p));
```

Listing 2.2: file `adjointShapeOptimizationFoam.C`

At lines 94 – 96 the porosity field is updated using a steepest descent algorithm. The relaxation factor is user defined and found using the `fieldRelaxationFactor()` function, which reads the relaxation factor from the `fvSolution` dictionary. Studying the code and comparing it to eq. 1.23 it can be seen that the scalar "lambda" which is defined in `createFields.H` as

```
90  dimensionedScalar lambda(laminarTransport.lookup("lambda"));
```

Listing 2.3: file `createFields.H`

is the cell volume times the step length. Note that it is a user supplied quantity given in the case directory `constant/transportProperties` and finding the correct step size is not only important for convergence, but also an optimization problem of its own. Also in a non-uniform mesh the step size will be different in the cells because of the varying volume. The max function makes sure α never becomes negative, while the min function puts an upper limit to the porosity. However the sign on line 96, where the new value of alpha is defined, seems to be incorrect. In order to implement the steepest descent method defined in eq. 1.23 it should be changed to a minus sign, resulting in

```
96   *(min(max(alpha - lambda*(Ua & U), zeroAlpha), alphaMax) - alpha);
```

On line 98 the porosity at the inlet cells are put to zero with the function `zeroCells` defined before the main function in `adjointShapeOptimizationFoam.C`. It is a templated functions enabling the use of more than one type of variable, e.g. scalar, vector or tensor, which sets the value of the provided variable to zero in the region specified.

As for the momentum predictor lines 105− in the above code, the different terms of the constraints in eq. 1.1 can easily be identified. As for line 108 the term `turbulence` is a pointer to the RASModel used constructed in `createFields.H`. When using the standard k-epsilon model the `divDevReff()` function is defined in
`OpenFOAM/OpenFOAM-2.2.x/src/turbulenceModels/incompressible/RAS/kEpsilon/kEpsilon.C`
as

```
185 tmp<fvVectorMatrix> kEpsilon::divDevReff(volVectorField& U) const
186 {
187         return
188         (
189         - fvm::laplacian(nuEff(), U)
190         - fvc::div(nuEff()*dev(T(fvc::grad(U))))
191         );
192 }
```

Listing 2.4: file `kEpsilon.C`

This can be identified as the term $2\nu D(\mathbf{v})$ in eq. 1.1, with an additional term

$$\nabla \cdot \frac{1}{3}\nu_{eff}\mathrm{tr}\left(\nabla \mathbf{u}\right)^T,$$

from the `dev()` member function, which creates the deviatoric part of the second order tensor argument according to $\mathrm{dev}(\mathbf{A}) = \mathbf{A} - 1/3(\mathrm{tr}\mathbf{A})\mathbf{I}$. The term should tend to zero for incompressible flow due to continuity.
The sink term due to porosity is treated like other source terms and is implemented implicitly using `fvm::Sp(alpha,U)` on line 109.

The primal system is then solved by using the value of the velocity from the momentum predictor to correct the pressure and then correcting the velocity with the new value of p in a momentum corrector, in a SIMPLE-type algorithm. This concludes the solution to the primal system this iteration. The ajoint system is solved in a manner very similar to the primal, using the same viscosity (frozen turbulence) and calculated values of the primal quantities needed:

```
154         // Adjoint Pressure-velocity SIMPLE corrector
155         {
156                 // Adjoint Momentum predictor
157
158                 volVectorField adjointTransposeConvection((fvc::grad(Ua) & U));
159                 //volVectorField adjointTransposeConvection
160                 //(
161                 //      fvc::reconstruct
162                 //      (
163                 //      mesh.magSf()*(fvc::snGrad(Ua) & fvc::interpolate(U))
164                 //      )
165                 //);
166
167                 zeroCells(adjointTransposeConvection, inletCells);
168
169                 tmp<fvVectorMatrix> UaEqn
```

```
170                 (
171                 fvm::div(-phi, Ua)
172                 - adjointTransposeConvection
173                 + turbulence->divDevReff(Ua)
174                 + fvm::Sp(alpha, Ua)
175                 );
176
177                 UaEqn().relax();
178
179                 solve(UaEqn() == -fvc::grad(pa));
```

Listing 2.5: file `adjointShapeOptimizationFoam.C`

The adjoint momentum predictor can be seen to be very similar to the primal. The only differences are a few signs and an additional term "adjointTransposeConvection", which can be identified by comparing the adjoint and primal incompressible, steady-state Navier-Stokes equations as the first term on the RHS in the below equation

$$-2\mathrm{D}(\mathbf{u})\mathbf{v} = -\nabla\mathbf{u}\cdot\mathbf{v} - (\mathbf{v}\cdot\nabla)\,\mathbf{u} \tag{2.1}$$

The second term in the RHS of eq. 2.1 is calculated on line 171.
On line 167 the adjointTransposeConvection term is set to zero at the inlet cells, which could imply stability concerns of the term. No dependence on the cost function is included which indicates a cost function without volumetric contributions.

2.1.2 createFields.H

As mentioned in the previous section the `createFields.H` file is where the variables used by the solver are initiated. The write and read options for the adjoint pressure and velocity are set up using the `IOobject` class in a similar way to the primal quantities.

```
37  Info<< "Reading field pa\n" << endl;
38  volScalarField pa
39  (
40      IOobject
41          (
42          "pa",
43          runTime.timeName(),
44          mesh,
45          IOobject::MUST_READ,
46          IOobject::AUTO_WRITE
47          ),
48      mesh
49  );
```

Listing 2.6: file `createFields.H`

The code makes the solver write the data to the file "pa" in a directory named after the timestep. The same is done for the adjoint velocity, the only difference being that the field is a `volVectorField Ua`. Important to note is that the initial values of the quantity, including boundary conditions and values of the boundary cells are read at this point.

```
65  #include "createPhia.H"
66
67
```

```
68  label paRefCell = 0;
69  scalar paRefValue = 0.0;
70  setRefCell
71  (
72      pa,
73      mesh.solutionDict().subDict("SIMPLE"),
74      paRefCell,
75      paRefValue
76  );
77
78
79  singlePhaseTransportModel laminarTransport(U, phi);
80
81  autoPtr<incompressible::RASModel> turbulence
82  (
83      incompressible::RASModel::New(U, phi, laminarTransport)
84   );
85
86
87  dimensionedScalar zeroSensitivity("0", dimVelocity*dimVelocity, 0.0);
88  dimensionedScalar zeroAlpha("0", dimless/dimTime, 0.0);
89
90  dimensionedScalar lambda(laminarTransport.lookup("lambda"));
91  dimensionedScalar alphaMax(laminarTransport.lookup("alphaMax"));
92
93  const labelList& inletCells = mesh.boundary()["inlet"].faceCells();
94  //const labelList& outletCells = mesh.boundary()["outlet"].faceCells();
95
96  volScalarField alpha
97  (
98      IOobject
99          (
100                 "alpha",
101                 runTime.timeName(),
102                 mesh,
103                 IOobject::READ_IF_PRESENT,
104                 IOobject::AUTO_WRITE
105         ),
106     lambda*max( Ua & U, zeroSensitivity)
107 );
108 zeroCells(alpha, inletCells);
109 //zeroCells(alpha, outletCells);
```

Listing 2.7: file `createFields.H`

The rest of the file, Listing 2.7, includes definitions of a number of scalars used in the solver as well as the `autoPtr turbulence` to the turbulence model used and reference cells / values for the adjoint and primal pressure, user defined in the `fvSolution` file. Line 65 where the `createPhia.H` is included initiates the `surfaceScalarField phia`. This is the face volume flux defined in `createPhia.H` as

```
49  linearInterpolate(Ua) & mesh.Sf()
```

Listing 2.8: file `createPhia.H`

which interpolates the value of `Ua` at the face from the neighbouring nodes and takes the inner product with the face area vector. The scalars are either defined by the user and looked up in the

case dictionaries such as `alphaMax` on line 91. Other variables are defined in the file itself, such as the reference `inletCells` to the face nodes of the inlet patch according to the code on line 93. The file also includes an initiation of the porosity field, line $96 - 107$. It can be seen to be slightly different than for the primal and adjoint variables. The read option `READ_IF_PRESENT` makes it read stored values if they are present, enabling the use of an initialization of `alpha`. However if an initialization of the adjoint and primal velocities is used when running a case, i.e. fields of values e.g. from another simulation are provided as a better initial guess (which should be considered for a faster, smoother convergence, see section 4), the sign can be seen to be incorrect on line 106.

2.1.3 Boundary conditions

The current implementation of the adjoint sovler optimizes for total pressure loss rather than power dissipated or flow uniformity at the outlet which the paper it is based on describes. Hence the boundary conditions will not be equal to those given in the theory section. The below sections try to describe how the boundary conditions are implemented to easier modify those in forthcoming sections, chapter 3. The descriptions will be focused on how the member function `updateCoeffs()` updates the value rather than how the constructor works. Both implementations use an explicit implementation to set the value of the adjoint variables on the outlet patch. It should be noted that the current implementation only has additional implemented boundary condition for the outlet and not the inlet. This is because of the fact that the boundary conditions for the inlet patch can often be set using existing boundary conditions, not unique for the adjoint method.

adjointOutletPressureFvPatchScalarField.C

The `updateCoeffs()` member function for the adjoint pressure can be seen below:

```
86  // * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
87
88  void Foam::adjointOutletPressureFvPatchScalarField::updateCoeffs()
89  {
90          if (updated())
91          {
92          return;
93          }
94
95          const fvsPatchField<scalar>& phip =
96          patch().lookupPatchField<surfaceScalarField, scalar>("phi");
97
98          const fvsPatchField<scalar>& phiap =
99          patch().lookupPatchField<surfaceScalarField, scalar>("phia");
100
101         const fvPatchField<vector>& Up =
102         patch().lookupPatchField<volVectorField, vector>("U");
103
104         const fvPatchField<vector>& Uap =
105         patch().lookupPatchField<volVectorField, vector>("Ua");
106
107         operator==((phiap/patch().magSf() - 1.0)*phip/patch().magSf() + (Up & Uap));
108
109         fixedValueFvPatchScalarField::updateCoeffs();
110 }
```

Listing 2.9: file `adjointOutletPressureFvPatchScalarField.C`

Line 95-105 define the needed `fvsPatchFields` and `fvPatchFields` of variables by looking them up with the `lookupPatchField` function. The operator defined on line 107 then assigns a value to the adjoint pressure, `pa` (q), using an equation which can be identified as

$$q = (u_n - 1)v_n + \mathbf{u} \cdot \mathbf{v}, \tag{2.2}$$

using the definition of `patch().magSf` as the magnitude of the face area vector of the patch in question, i.e. the area of the face. According to the literature the equation which should be solved to assign the adjoint pressure a value at the outlet is given by eq. 1.13. It is clear from eq. 2.2 that the current implementation uses expressions for boundary conditions different than those given in eq. 1.17 or 1.20. However, there is not much information about the actual implementation in the article so there might be approximations done that are not mentioned in the literature, complicating the task of identifying for which cost function the current implementation is suitable.

adjointOutletVelocityFvPatchVectorField.C

The `updateCoeffs()` member function for the adjoint velocity can be seen below:

```
83  // * * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * * * //
84
85  // Update the coefficients associated with the patch field
86  void Foam::adjointOutletVelocityFvPatchVectorField::updateCoeffs()
87  {
88          if (updated())
89          {
90          return;
91          }
92
93          const fvsPatchField<scalar>& phiap =
94          patch().lookupPatchField<surfaceScalarField, scalar>("phia");
95
96          const fvPatchField<vector>& Up =
97          patch().lookupPatchField<volVectorField, vector>("U");
98
99          scalarField Un(mag(patch().nf() & Up));
100         vectorField UtHat((Up - patch().nf()*Un)/(Un + SMALL));
101
102         vectorField Uan(patch().nf()*(patch().nf() & patchInternalField()));
103
104         vectorField::operator=(phiap*patch().Sf()/sqr(patch().magSf()) + UtHat);
105         //vectorField::operator=(Uan + UtHat);
106
107         fixedValueFvPatchVectorField::updateCoeffs();
108 }
```

Listing 2.10: file `adjointOutletVelocityFvPatchVectorField.C`

The implementation for the adjoint velocity boundary condition is similar to the one for pressure. According to the litterature eq. 1.14 should be used to find the tangential component of the velocity, then the total adjoint velocity can be evaluated as $\mathbf{u} = \mathbf{u}_t + \mathbf{u}_n$. The code seen in Listing 2.10 does this on line 104. The mathematical expression for `UtHat` can be seen to be

$$\mathbf{u}_{p,t} = \frac{\mathbf{v}_p - \mathbf{v}_{p,n}}{u_{p,n} + \mathrm{SMALL}}, \tag{2.3}$$

where the indices $_p$, $_n$ and $_t$ denote the current, center node and its normal or tangential component respectively and `SMALL` is added in the denominator to ensure it is always non-zero. As already

mentioned the cost function for which the current boundary conditions are suitable is not easily identified by the expressions given by eq. 2.2 and 2.3. Furthermore the unused variable `Uan`, defined on line 102, serves as another confusion factor. `patchInternalField()` can be used to access quantities from the neighbouring nodes of the patch, generating a `tmp<Field<Type>>`. The `tmp` class enables the fields to be returned from the function without being copied, hence it is suitable when handling demanding variables, memory wise. Hence the function will be used to determine the adjoint velocity in the neighbouring node when modifying the solver. The neighbouring node's velocity is needed if the evaluation of the term

$$\nu(\mathbf{n} \cdot \nabla)\mathbf{u}_t,$$

is approximated as the difference between the velocities divided by the distance between the patch and its neighbouring node. This is the approximation which will be used when modifying the boundary conditions in section 3.3.

Chapter 3

Modify the solver

In this chapter detailed instructions on how to implement your own modifications to the solver are included. Only cost functions without volumetric contributions are considered, which reduces the problem of changing cost function to designing new boundary conditions, according to the theory explained in chapter 1. The same solver will therefore be applied and which cost function that is to be considered is defined when choosing boundary conditions in the case settings.

3.1 Copy and rename the original solver

Copy the original solver to a directory of choice, tentatively:

```
cd $WM_PROJECT_DIR
cp -r --parents applications/solvers/incompressible/adjointShapeOptimizationFoam \
$WM_PROJECT_USER_DIR
```

Rename the solver to name of choice, e.g. `myAdjointShapeOptimizationFoam`

```
cd $WM_PROJECT_USER_DIR/applications/solvers/incompressible/
mv adjointShapeOptimizationFoam myAdjointShapeOptimizationFoam
cd myAdjointShapeOptimizationFoam
mv adjointShapeOptimizationFoam.C myAdjointShapeOptimizationFoam.C
```

Rename the executable and change the location in `Make/files`:

```
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
sed -i s/adjointShapeOptimizationFoam/myAdjointShapeOptimizationFoam/g Make/files
```

Clean the previous compilation and recompile with the command `wclean` and `wmake` respecitvely. If everything worked as intended the command `which myAdjointShapeOptimizationFoam` should return the path to the copied solver.

3.2 Changing the sign in the implementation of the steepest descent algorithm

The sign in the steepest descent algorithm seems to be incorrect, see section 2.1.1. In order for the coming implementations to work correctly it should be changed by replacing the plus sign with a minus sign in `adjointShapeOptimizationFoam.C` according to Listing 2.2 and 2.1.1, in section 2.1.1. Note that the original boundary conditions do not work with this modification, but only the new ones introduced below, section 3.3.

3.3 Changing the cost function

As mentioned in Section 2.1.3 the current boundary conditions are for the cost function describing total pressure loss rather than for power dissipation or flow uniformity at the outlet described in [1]. Below descriptions of how to include additional boundary conditions suited for power dissipation and flow uniformity will be shown.

3.3.1 Flow uniformity at the outlet

Start by moving into the copied directory of the copied solver and copy the original boundary conditions.

```
cd $WM_PROJECT_USER_DIR/solvers/incompressible/myAdjointOptimizationFoam
cp -r adjointOutletVelocity/ adjointOutletVelocityUni/
cp -r adjointOutletPressure/ adjointOutletPressureUni/
```

The next step is to rename the boundary conditions. The easiest way to do this is to use the `sed` command as

```
cd adjointOutletPressureUni
sed s/adjointOutletPressure/adjointOutletPressureUni/g \
adjointOutletPressureFvPatchScalarField.C > adjointOutletPressureUniFvPatchScalarField.C
sed s/adjointOutletPressure/adjointOutletPressureUni/g \
adjointOutletPressureFvPatchScalarField.H > adjointOutletPressureUniFvPatchScalarField.H
cd ../adjointOutletVelocityUni
sed s/adjointOutletVelocity/adjointOutletVelocityUni/g \
adjointOutletVelocityFvPatchVectorField.C > adjointOutletVelocityUniFvPatchVectorField.C
sed s/adjointOutletVelocity/adjointOutletVelocityUni/g \
adjointOutletVelocityFvPatchVectorField.H > adjointOutletVelocityUniFvPatchVectorField.H
cd ..
```

Remove the old files and move back to the folder where the `Make` directory is situated:

```
rm adjointOutletPressureUni/adjointOutletPressureFvPatchScalarField.*
rm adjointOutletVelocityUni/adjointOutletVelocityFvPatchVectorField.*
```

Now the new boundary conditions need be included when compiling, by editing `Make/files`. Or simply open `Make/files` by a text editor of choice and add the line
`adjointOutletVelocityUni/adjointOutletVelocityUniFvPatchVectorField.C` and
`adjointOutletPressureUni/adjointOutletPressureUniFvPatchScalarField.C` by hand, somewhere before the `EXE =` -line.

```
sed -i "s/myAdjointShapeOptimizationFoam.C/myAdjointShapeOptimizationFoam.C\n \
adjointOutletVelocityUni\/adjointOutletVelocityUniFvPatchVectorField.C\n \
adjointOutletPressureUni\/adjointOutletPressureUniFvPatchScalarField.C/g" Make/files
```

Before including descriptions of how the boundary conditions are implemented the needed equations are presented. Starting from eq. 1.20 the boundary calculations needed can be seen below

$$q = \mathbf{u}_p \cdot \mathbf{v}_p + u_{p,n} v_{p,n} + \frac{\nu}{\Delta}\left(u_{p,n} - u_{neigh,n}\right) + c(v_{p,n} - v^d_{n,p}), \tag{3.1}$$

$$\mathbf{u}_t = \frac{\frac{\nu}{\Delta}\mathbf{u}_{neigh,t} - c(\mathbf{v}_{p,t} - \mathbf{v}^d_{p,t})}{v_{p,n} + \frac{\nu}{\Delta}}. \tag{3.2}$$

Note that the term $\nu(\mathbf{n}\cdot\nabla)u_n$ has been approximated as the difference between the velocity in the node at the patch in question and its neighbouring node, in the domain, divided by the distance between them, i.e.

$$\nu(\mathbf{n}\cdot\nabla)\mathbf{u}_t \approx \nu\frac{\mathbf{u}_{p,t}-\mathbf{u}_{neigh,t}}{\Delta}, \tag{3.3}$$

where Δ is the distance between the patch and its neighbouring node one arrive at the following expressions for the pressure and velocity boundary conditions. In eq. 3.1-3.2; $_p$ and $_{neigh}$ denote the node at the patch and its neighbouring node respectively and $_t$ and $_n$ denote the tangential and normal component respectively.

Adjoint pressure

Starting with the boundary condition for the adjoint pressure; open `adjointOutletPressureUni/adjointOutletPressureUniFvPatchScalarField.C` with a text editor of choice. Add `#include "RASModel.H"` in the header, then scroll down to the "member function" section and follow the instructions below to modify the `updateCoeffs()` function.
The flux fields will not be used, hence delete the unused fields `phip` and `phiap` by removing the following lines.

```
const fvsPatchField<scalar>& phip =
        patch().lookupPatchField<surfaceScalarField, scalar>("phi");

const fvsPatchField<scalar>& phiap =
        patch().lookupPatchField<surfaceScalarField, scalar>("phia");
```

Listing 3.1: Remove

To calculate the magnitude of the normal components of the primal and adjoint velocities, use the `patch().nf()` member function to find the face normal vector. The magnitude of the normal component is then calculated taking the inner product of the normal with the velocities and define the scalar fields by adding the definitions seen below.

```
scalarField Up_n = Up & patch().nf(); //Primal

scalarField Uap_n = Uap & patch().nf();//Adjoint
```

The variables needed to compute the first two terms on the RHS of eq. 3.1 are now defined. As for the turbulent viscosity the turbulence model is a `IOobject` and can be found using the `lookupObject()` function. The effective viscosity can then be found using a member function of the turbulence model's class according to

```
const incompressible::RASModel& rasModel =
        db().lookupObject<incompressible::RASModel>("RASProperties");

scalarField nueff = rasModel.nuEff()().boundaryField()[patch().index()];
```

As for the distance between the nodes, Δ, the easiest way to define it is using `deltaCoeffs`, which returns the inverse of the distance between nodes when used as

```
const scalarField& deltainv = patch().deltaCoeffs(); // distance^(-1)
```

The magnitude of the normal component of the adjoint velocity in the neighbouring can be found using `patchInternalField()` which is mentioned in the implementation of the adjoint velocity outlet boundary condition, even though never actually used (see Listing 2.10).

```
scalarField Uaneigh_n = (Uap.patchInternalField() & patch().nf());
```

The remaining variable is the user defined velocity in the outlet plane. In many cases it would be preferrable to define it in a dictionary when defining settings for the case. One way to do this is to include another `volVectorField` in `createFields.H` similarily to the adjoint and primal velocities. To do this a few additions to `createFields.H` is needed. Open `createFields.H` using a text editor of choice and add the following lines at the end of the file.

```
dictionary optFunc =
        mesh.solutionDict().subDict("objectiveFunctionDict");

vector Udlookup =
        optFunc.lookupOrDefault<vector>("Uduserdefnodim", vector(1,0,0));

volVectorField Ud
        (
        IOobject
                (
                    "Ud",
                    runTime.timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
        mesh,
        Udlookup
        );
```

Descriptions of how this is added in the `fvSolution` dictionary is included in section 4.1.3. But reading the code one can see that it should be in a sub-dictionary called "objectiveFunctionDict" and the name should be "Uduserdefnodim".
Save the changes before returning to `adjointOutletPressureUni/adjointOutletPressureUniFvPatchScalarField.C`. The magnitude of the normal component of the user defined vector can now be found by adding the following.

```
const fvPatchField<vector>& Udp =
        patch().lookupPatchField<volVectorField, vector>("Ud");

scalarField Udp_n = (Udp & patch().nf());
```

Every variable needed to define the operator and calculate the value of the adjoint pressure is now present and the operator is readily defined as

```
operator == ( (Uap&Up) + (Up_n*Uap_n) +
        nueff*deltainv*(Uap_n - Uaneigh_n) + (Up_n-Udp_n) );
```

Note that the constant c is equal to 1 and the units seem to coincide without including an additional `dimensioned scalar`, hence it has been left out.

Adjoint velocity

The boundary condition for the adjoint velocity is done similarily; open `adjointOutletVelocityUni/adjointOutletVelocityUniFvPatchVectorField.C` with a text editor of choice. Add `#include "RASModel.H"` in the header, then scroll down to the "member function" section and follow the instructions below to modify the `updateCoeffs()` function.
Many of the steps are similar to steps done in the pressure condition and hence the instructions for those will only be "Include" or "Remove" without any arguments or description of the implementation.
Remove the adjoint flux.

```
const fvsPatchField<scalar>& phiap =
        patch().lookupPatchField<surfaceScalarField, scalar>("phia");
```

Listing 3.2: Remove

Remove also the scalar and vector fields defined in the default code, namely `Un`, `Uthat` and `Uan`. They will be defined in a way similar to what was done for the pressure condition.

```
scalarField Un(mag(patch().nf() & Up));
vectorField UtHat((Up - patch().nf()*Un)/(Un + SMALL));

vectorField Uan(patch().nf()*(patch().nf() & patchInternalField()));
```

Listing 3.3: Remove

Include the adjoint velocity.

```
const fvPatchField<vector>& Uap =
        patch().lookupPatchField<volVectorField,vector>("Ua");
```

Include the turbulence model, define the turbulent viscosity and the inverse distance between the nodes.

```
const incompressible::RASModel& rasModel =
        db().lookupObject<incompressible::RASModel>("RASProperties");

scalarField nueff = rasModel.nuEff()().boundaryField()[patch().index()];

const scalarField& deltainv = patch().deltaCoeffs(); // dist^(-1)
```

Include the user defined velocity as before, however the tangential component is now needed rather than the magnitude of the normal component. The normal component can be calculated by multiplying v_n^d by $\mathbf{n}$ and then tangential part by subtraction $\mathbf{v}_t^d = \mathbf{v}^d - \mathbf{v}_n^d$.

```
        const fvPatchField<vector>& Udp =
        patch().lookupPatchField<volVectorField,vector>("Ud");

        vectorField Udp_n = (Udp & patch().nf())*patch().nf();

        vectorField Udp_t = Udp - Udp_n;
```

Next compute the normal and tangential component of the primal velocity, using the same method as for the user defined velocity.

```
//Primal
vectorField Up_n = (Up & patch().nf())*patch().nf(); //Normal

scalarField Up_ns = (Up & patch().nf()); //Mag. of normal

vectorField Up_t = Up - Up_n; //Tangential
```

Include the adjoint velocity in the neighbouring node and its two components, as vector fields.

```
vectorField Uaneigh = Uap.patchInternalField();

vectorField Uaneigh_n = (Uaneigh & patch().nf())*patch().nf(); //Normal

vectorField Uaneigh_t = Uaneigh - Uaneigh_n; //Tangential
```

Now every term needed to calculate the tangential part of the adjoint velocity, see eq. 3.2, have been defined. Since the definition of the `operator ==` should be an expression for the total adjoint velocity one should also include the normal component in some way. The convinient way to do this is to take the old value of the normal component and add it to the updated value of the tangential component. According to

```
vectorField Uap_n = (Uap & patch().nf())*patch().nf(); //Normal comp.

vectorField Uap_t = (-(Up_t-Udp_t) + nueff*deltainv*Uaneigh_t) /
                        (Up_ns+nueff*deltainv);

operator==(Uap_t+Uap_n);
```

Note that the above operator is supposed to replace the existing one.

3.3.2 Power dissipation

Start by moving into the copied directory of the copied solver and copy the original boundary conditions.

```
cd $WM_PROJECT_USER_DIR/solvers/incompressible/myAdjointOptimizationFoam
cp -r adjointOutletVelocity/ adjointOutletVelocityPower/
cp -r adjointOutletPressure/ adjointOutletPressurePower/
```

The next step is to rename the boundary conditions. The easiest way to do this is to use the `sed` command as

```
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 ..
```

Remove the old files:

```
rm adjointOutletPressurePower/adjointOutletPressureFvPatchScalarField.*
rm adjointOutletVelocityPower/adjointOutletVelocityFvPatchVectorField.*
```

Now the new boundary conditions need be included when compiling, by editing `Make/files` (or do it by hand as explained in section 3.3.1).

```
sed -i "s/myAdjointShapeOptimizationFoam.C/myAdjointShapeOptimizationFoam.C\n \
adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C\n \
adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C/g" Make/files
```

Before including descriptions of how the boundary conditions the needed equations are presented. The boundary calculations that are to be implemented can be derived from eq. 1.17 and are pre-

sented below

$$q = \mathbf{u}_p \cdot \mathbf{v}_p + u_{p,n} v_{p,n} + \frac{\nu}{\Delta}\left(u_{p,n} - u_{neigh,n}\right) - \frac{1}{2} v_p^2 - v_{p,n}^2, \tag{3.4}$$

$$\mathbf{u}_t = \frac{\frac{\nu}{\Delta}\mathbf{u}_{neigh,t} + v_{p,n}\mathbf{v}_{p,t}}{v_{p,n} + \frac{\nu}{\Delta}}, \tag{3.5}$$

where the term $\nu(\mathbf{n}\cdot\nabla)u_n$ has been approximated as the difference between the velocity in the node at the patch in question and its neighbouring node, in the domain, divided by the distance between them, as was done in the previous implementation.

Adjoint pressure

Starting with the boundary condition for the adjoint pressure open `adjointOutletPressurePower/adjointOutletPressurePowerFvPatchScalarField.C` with a text editor of choice. Add `#include "RASModel.H"` in the header, then scroll down to the "member function" section and follow the instructions below to modify the `updateCoeffs()` function.
Create the scalar fields of the magnitude of the adjoint and primal velocities, using the already defined scalar fields `phip` and `phiap` divided by the face area.

```
scalarField Up_n = phip / patch().magSf();//Primal

scalarField Uap_n = phiap / patch().magSf();//Adjoint
```

The variables needed to compute the first two terms on the RHS of eq. 3.4 are now defined. The effective viscosity and the inverse of the distance are defined similarily to what was done for the flow uniformity conditions, include:

```
const incompressible::RASModel& rasModel =
        db().lookupObject<incompressible::RASModel>("RASProperties");

scalarField nueff = rasModel.nuEff()().boundaryField()[patch().index()];

const scalarField& deltainv = patch().deltaCoeffs(); // distance^(-1)
```

The magnitude of the normal component of the adjoint velocity is the remaining term needed before the operator can be redefined according to eq. 3.4.

```
scalarField Uaneigh_n = (Uap.patchInternalField() & patch().nf());

operator == ( (Up_n * Uap_n) + (Uap&Up) +
        nueff*deltainv*(Uap_n-Uaneigh_n) -
        (0.5*mag(Up)*mag(Up)) - (Up & patch().Sf()/patch().magSf())
        * (Up & patch().Sf()/patch().magSf()) );
```

Adjoint velocity

Considering the boundary condition for the adjoint velocity, modify it by opening `adjointOutletVelocityPower/adjointOutletVelocityPowerFvPatchVectorField.C` with a text editor of choice and change it according to the descriptions below.
Add `#include "RASModel.H"` in the header, then modify the member function `updateCoeffs()` so that eq. 3.5 can be calculated. The primal velocity and the adjoint volume flux are already included, however the adjoint velocity and the primal scalar field `phi` are also needed.

```
const fvPatchField<vector>& Uap =
        patch().lookupPatchField<volVectorField, vector>("Ua");

const fvsPatchField<scalar>& phip =
        patch().lookupPatchField<surfaceScalarField, scalar>("phi");
```

Include the effective viscosity and the distance between the node of the patch and its neighbouring node, as was done in the pressure condition.

```
const incompressible::RASModel& rasModel =
        db().lookupObject<incompressible::RASModel>("RASProperties");

scalarField nueff = rasModel.nuEff()().boundaryField()[patch().index()];

const scalarField& deltainv = patch().deltaCoeffs(); // dist^(-1)
```

Considering eq. 3.5 the remaining qunatities needed to compute the tangential component of the adjoint velocity are the magnitude of the normal component of the primal velocity, the tangential component of the primal velocity and the tangential component of the adjoint velocity in the neighbouring node. Include them by adding the following to the code.

```
//Primal velocity, mag of normal component and tangential component
scalarField Up_ns = phip/patch().magSf();

vectorField Up_t =
        Up - (phip * patch().Sf())/(patch().magSf()*patch().magSf());

//Tangential component of adjoint velocity in neighbouring node
vectorField Uaneigh = Uap.patchInternalField();
vectorField Uaneigh_n = (Uaneigh & patch().nf())*patch().nf();
vectorField Uaneigh_t = Uaneigh - Uaneigh_n;
```

The tangential component of the adjoint velocity can now be calculated using the definition given in eq. 3.5.

```
vectorField Uap_t =
       ((Up_ns*Up_t) + nueff*deltainv*Uaneigh_t) / (Up_ns+nueff*deltainv) ;
```

Set the total adjoint velocity by addition of the calculated tangential component and the old value of the normal component (which has to be defined).

```
vectorField Uap_n = (phiap * patch().Sf())/(patch().magSf()*patch().magSf());

operator==(Uap_t+Uap_n);
```

3.4 Including the sensitivity of each cell

The sensitivity of each cell with respect to the cost function can sometimes be useful to have as a result. For example when one considers the cost function of flow uniformity at the outlet a reasonable result would be that the porosity of cells with too high velocity at the outlet patch would be punished by an increased porosity. This might however be unwanted for design purposes, in which case one might not want to update the porosity, but rather have the sensitivity as a result. How to make this addition to the code is explained below.

In order to be able to see the sensitivity of each cell, an addition to `createFields.H` has to be

done and the value has to be updated in every loop in `myAdjointShapeOptimizationFoam.C`. Introduce a scalar field, "sens", by adding the following lines at the end of `createFields.H`

```
volScalarField sens
        (
        IOobject
                (
                    "sensitivity",
                    runTime.timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
        Ua&U
        );
```

Listing 3.4: Addition to `createFields.H`

Update the scalar field in every iteration step by adding the follwing line in the main loop of `myAdjointShapeOptimizationFoam.C`, for example above `runTime.write();`.

```
sens=Ua&U;
```

Listing 3.5: Addition to the main loop of `myAdjointShapeOptimizationFoam.C`

The resuting scalar field will be stored in the file sensitivity in the directory named after the `timeName` in the case directory.

3.5 Print the cost function

In order to see how much the cost function is reduced one has to calculate eq. 1.15 or 1.18 depending on which cost function that is studied. This can be done by including code in the SIMPLE loop of `myAdjointShapeOptimizationFoam.C`. However objects that only need to be initiated once should be initialized in the `createFields.H` file. Also the patches over which the cost function needs to be integrated have to be defined. Since the case might have more than one outlet, with different names, it should be defined in the case settings which patches should be used when calculating the cost function. Also the number of patches for which the calculations are to be done should be defined in the case dictionary.

Start to define the variables that should be looked up, in the same dictionary (`fvSolution`) in which the user defined velocity is defined when the boundary conditions for flow uniformity is used. Add the following lines to the end of `createFields.H`. Note that if flow uniformity at the outlet boundary condition has been implemented according to the description in section 3.3.1 the dictionary `optFunc` does not need to be defined again.

```
dictionary optFunc =
        mesh.solutionDict().subDict("objectiveFunctionDict");

int nObjPatch =
        optFunc.lookupOrDefault<scalar>("numberObjectivePatches",0);

int objFunction =
        optFunc.lookupOrDefault<scalar>("objectiveFunction", 0);

wordList objPatchNames =
        optFunc.lookup("objectivePatchesNames");
```

It should be noted that this code uses an integer `objFunction` to choose which objective function will be calculated. 0 means that no objective function has been defined, 1 and 2 stands for power dissipation and flow uniformity at the outlet respectively, which can be seen when the actual calculation of the cost function is carried out in a later stage. To clarify what is done a few `Info` statements can be included, in order to be able to identify what cost function and settings have been chosen. Print it to the log file using.

```
Info<<"Initializing objective function calculation:"<<endl;
Info<<"The objective function chosen is:"<<objFunction<<endl;
Info<<"Name of the patches for which the cost
        function will be calculated:"<<objPatchNames<<endl;
Info<<"Number of patches:"<<nObjPatch<<endl;
```

The name of the patches can be used to find their actual identities, add the following lines.

```
label objPatchList[nObjPatch];
int iLoop;
for (iLoop=0; iLoop<nObjPatch; iLoop++)
{
        objPatchList[iLoop] =
                mesh.boundaryMesh().findPatchID(objPatchNames[iLoop]);
}
```

Now the calculations of the cost function are readily implemented in the main loop of `myAdjointShapeOptimizationFoam.C`. A convenient way to do this is to include a file where the calculations are done. Add the following to loop in the sovler, e.g. on the line after the `Info<<` statement returning the time step in the very beginning of the while loop:

```
#include "costFunction.H"
```

Create `costFunction.H` and place it in the same directory as the solver definition file. The code needed for the calculations is the implementation of definition of the cost functions, eq. 1.15 and 1.18 for power dissipation and flow uniformity at the outlet respectively. The integrals reduce to summation over values stored in the nodes of the patch in question. The `sum` function can be used for this summation. Start by defining the scalars which will be the values of the objective function.

```
scalar jDissPower(0);
scalar jFlowUni(0);
```

Next construct a for loop running over the number of patches, using `nObjPatch` already defined to formulate the end criteria and if statements to determine which objective function is to be calculated.

```
for (iLoop=0; iLoop<nObjPatch; iLoop++)
{
        if (objFunction==1) {
                jDissPower =
                        jDissPower - sum(
                        phi.boundaryField()[objPatchList[iLoop]]*
                        (p.boundaryField()[objPatchList[iLoop]]
                        + 0.5*
                        magSqr(U.boundaryField()[objPatchList[iLoop]]))
                        );
        }
        else if (objFunction==2) {
                jFlowUni =
                        jFlowUni + sum(0.5*
                        magSqr(U.boundaryField()[objPatchList[iLoop]]
```

```
                        -Ud.boundaryField()[objPatchList[iLoop]])
                        );
        }
}
```

Print the value using `Info` statements and the integer describing which cost function is used.

```
if (objFunction==1) {
        Info<<"Objective function
                (Power dissipated) J: "<<jDissPower<<endl;
}
else if (objFunction==2) {
        Info<<"Objective function
                (Flow uni at outlet) J: "<<jFlowUni<<endl;
}
```

The value of the objective function is printed in the log file when running the solver and must therefore be searched for and stored in some way, an example of how this is done can be found in the next chapter, section 4.2.1.

Chapter 4

Setting up a case

Approximations done in the derivations of the model require an incompressible, steady state case. There are also requirements on the boundary conditions of the primal velocities. Since the aim of this report is not do detailed studies on a specific case the tutorial case `pitzDaily` will be copied from the tutorial folder in the openFoam installation. Note that this case is a simple 2D case and convergence should not prove to be a problem. For more complicated 3D cases with poorer meshes a more detailed choice of schemes and relaxation factors might prove necessary. Also initializing the primal and adjoint fields by running the adjoint solver without updating the porosity and running the simulations from `latestTime` can reduce the stability concerns and decrease the time needed to obtain converged results.
The changes needed to be done to dictionaries in order to adjust the case to the new cost functions will then be discussed and presented. However it should be noted that more effort is needed to establish which settings should and can be used to arrive at as accurate results possible. One example is the interpolation schemes used for the adjoint quantities.
Finally one example of how post processing will be done in the next chapter, to validate (to some extent) the modification done in the previous section, will be included. Note that the changes presented in this section will be for the flow uniformity at the outlet, however a brief discussion of how the case settings should be defined for the power dissipation objective function will be included when they differ.

Copy the tutorial

Change directory to where you want to place your case, e.g. in the run directory in the `$WM_PROJECT_USER_DIR` by typing

```
run
```

Copy the tutorial case using the `cp` command below

```
cp -r $FOAM_TUTORIALS/incompressible/adjointShapeOptimizationFoam/pitzDaily .
```

Moving into the case folder and using a `tree` command one can see a directory struction as presented in Figure 4.1.

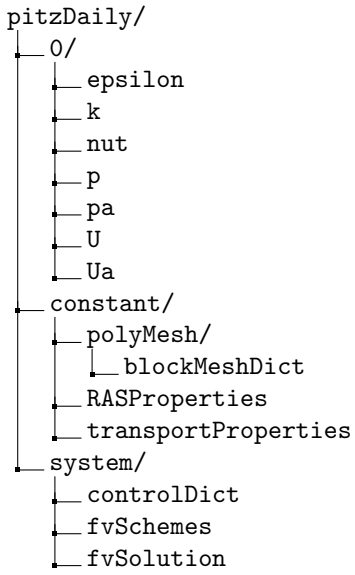


Figure 4.1: Directory structre for the tutorial case `pitzDaily`.

4.1 Pre processing

4.1.1 Initial and boundary conditions

The boundary conditions of the primal and adjoint variables are discussed and summarized in section 1.1. Starting with the primal velocity and pressure the files `0/U` and `0/p` need to be modified so that they coincide with the ones presented in Table 1.1. Studying the given conditions in the copied case one can see that the only one in need of modification is the outlet BC for the velocity, hence change it to

```
outlet
{
    type                zeroGradient;
}
```

Listing 4.1: File `0/U`

The boundary conditions of the adjoint quantities depend on which objective function that is to be applied, as described in section 1.1. Table 4.1 summarizes the derivations done in previous sections. The resulting boundary fields for the adjoint pressure and velocity can be seen in Listing 4.2 and 4.3 respectively. Note that they are for the flow uniformity cost function.

```
boundaryField
{
        inlet
        {
        type            zeroGradient;
        }

        outlet
        {
        type            adjointOutletPressure;
        value           uniform 0;
```

	Wall	Inlet	Outlet
u	Prescribed value, 0	Prescribed value, 0	`adjointOutletVelocityUni`
q	Zero gradient	Zero gradient	`adjointOutletPressureUni`

(a)

	Wall	Inlet	Outlet
u	Prescribed value, 0	Prescribed value, $u_n = v_n$	`adjointOutletVelocityPower`
q	Zero gradient	Zero gradient	`adjointOutletPressurePower`

(b)

Table 4.1: Boundary conditions for the adjoint variables **u** and q, for a) the flow uniformity at the outlet and b) the power dissipation cost function.

```
        }

        upperWall
        {
        type            zeroGradient;
        }

        lowerWall
        {
        type            zeroGradient;
        }

        frontAndBack
        {
        type            empty;
        }
}
```

Listing 4.2: File `0/pa`

```
boundaryField
{
        inlet
        {
        type            fixedValue;
        value           uniform (0 0 0);
        }

        outlet
        {
        type            adjointOutletVelocity;
        value           uniform (0 0 0);
        }

        upperWall
        {
        type            fixedValue;
        value           uniform (0 0 0);
        }

        lowerWall
```

```
        {
        type            fixedValue;
        value           uniform (0 0 0);
        }

        frontAndBack
        {
        type            empty;
        }
}
```

Listing 4.3: File `0/Ua`

When it comes to the evaluation of the variables in the turbulence model used, i.e. `0/epsilon`, `0/k` and `0/nut`, the conditions and wall functions defined in the tutorial will be used.

4.1.2 User defined parameters

`constant/transportProperties`

`transportPropoerties` is a dictionary defining data of the fluid, e.g. viscosity and density. This is also where `adjointShapeOptimizationFoam` expects the factor, describing the step length times the cell volume, for the porosity update in the steepest descent algorithm and the maximum of alpha, as described in section 2.1.2. The value of `alphaMax` is of course case dependent, for cases when the sensitivity has a chance of becoming large, i.e. large velocities (see eq. 1.5), it should be increased. `lambda` is a value of the step size used in the optimization algorithm. Finding an optimal value of it is, as mentioned before, an optimization problem of its own. Too small might result in very slowly converging results, while too large might "miss" the optimal value of alpha resulting in convergence problems. A more detailed analysis should be carried out before deciding a value for larger industrial cases. For the tutorial case the file can be left unedited, however in some of the cases, for which results are shown in the next chapter, it has been decreased to `1e3`.

```
transportModel  Newtonian;

nu              nu [0 2 -1 0 0 0 0] 1e-5;

lambda          lambda    [0 -2 1 0 0 0 0] 1e5;
alphaMax        alphaMax  [0 0 -1 0 0 0 0] 200.0;

// ************************************************************************* //
```

Listing 4.4: File `transportProperties`

`constant/RASProperties`

The dictionary selecting the turbulence model to be used. In the tutorial `kEpsilon` is used, leave the file unedited.

4.1.3 System settings

`system/fvSolution`

The solvers defined by default in the tutorial case will be used. As for the under relaxation factors for the different variables I am sure everyone has their own preferred values. Worth mentioning is that the relaxation factor for the porosity update of course influences the step size mentioned

above. For the case at hand the default values are fine, however the user defined sub-dictionary `objectiveFunctionDict` is located in the solution dictionary and hence has to be added. For the flow uniformity at the outlet the value of the cost function only contain an integral over the outlet and the velocity $\mathbf{v}^d$ has to be defined. Listing 4.5 presents the additional dictionary to be added in `fvSolution`. Note that if power dissipation cost function is to be evaluated `objectiveFunction` should be changed from 2 to 1 and the inlet should be added to both `numberObjectivePatches` and `objectivePatchesNames`.

```
objectiveFunctionDict
{
        objectiveFunction               2;
        numberObjectivePatches          1;
        objectivePatchesNames           (outlet);
        Uduserdefnodim                  (10 0 0);
}


// ************************************************************************* //
```

Listing 4.5: File `system/fvSolution`

`system/fvSchemes`

As seen in Listing 4.5 first order upwind scheme is applied to convection terms. Although inaccurate the article mentions stability concerns if anything else is used, stressing the need of a fine mesh to reduce the error.

`system/controlDict`

One might want to increase the number of iterations in order to let the variables have a little more time to converge. Apart from that use the default contols defined in the tutorial.

4.1.4 Mesh

For demonstrational purposes the mesh will not be changed. However the fact that first order upwind scheme is used result in a necessity of a finer mesh. Since one does not know where the porosity field will be placed beforehand one can argue that a uniform mesh should be used. The porosity field might result in a flow field with gradients (and small scales) offset from the wall regions. The implemented handling, e.g. wall functions or a refined mesh, of those regions does not contribute and the resolution of the primal flow becomes one of the major concerns of the solver.

4.2 Post processing

4.2.1 Using gnuPlot to plot the value of the cost function

The current implementation writes the value of the cost function in the log file when running the solver, if the cost functions is evaluated as showed in section 3.5. Hence the data needs to be extracted from this file. One way is to use the grep command to extract the data and then `gnuPlot` to visualize it. An example using this procedure can be found below.

```
grep 'Objective function (Power dissipated) J:' log > valueObj.xy
sed -i s/'Objective function (Power dissipated) J: '/''/g valueObj.xy
```

Now to use gnuPlot to plot the data stored in the `valueObj.xy` file by typing

```
gnuplot
set terminal epslatex
set output "valueObj.tex"
set format xy "$%g$"
set xlabel "Iteration step"
set ylabel "$J$"
plot [0:nI] [0:yMax] "valueObj.xy" with points
```

Or simply store the `gnuplot` commands in a file, e.g. `gnuCommands.gp`, and type

```
gnuplot gnuCommands.gp
```

Note that `[0:nI]` and `[0:ymax]` set the limit of the x- and y-axis respectively.

Chapter 5

Results

5.1 Flow uniformity at the outlet

5.1.1 pitzDaily

The results from following the instructions in the previous chapter is presented here. Important to note that is that the presented figures originate from an implementation slightly different from the one presented in chapter 3. Some last minute modifications were carried out to correct some sign errors. Since the difference between the results was small the figures were not updated, the reader is instead encouraged to try the supplied solver himself to see the differences. For the interested the below figures is achieved by changing the sign of the velocity outlet boundary condition, eq. 3.2 and 3.5 for flow uniformity and power dissipation respectively, in the supplied solver.
The number of iterations has been increased in order to let the results fully converge. Figure 5.1 shows the velocity field and Figure 5.2 the corresponding sensitivity field. Comparing the two the sensitivity can be seen to be negative where the primal velocity is greater than average and hence punishing those cells by increasing the sensitivity would be favourable w.r.t. reducing the objective function value. In regions with below average velocity the sensitivity is greater than zero, indicating that punishment via increased porosity would increase the objective function value.

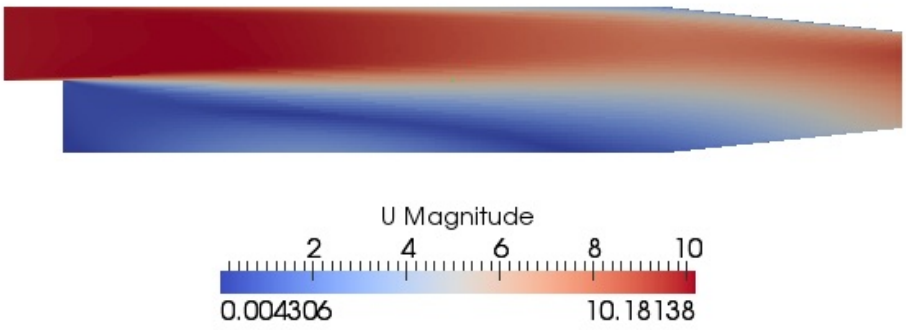


Figure 5.1: Final porosity and primal velocity field.

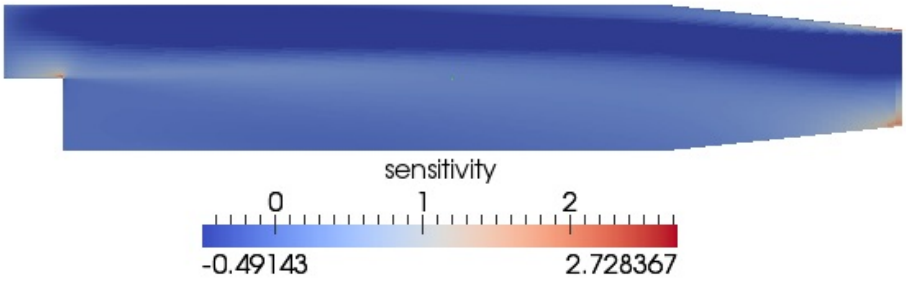
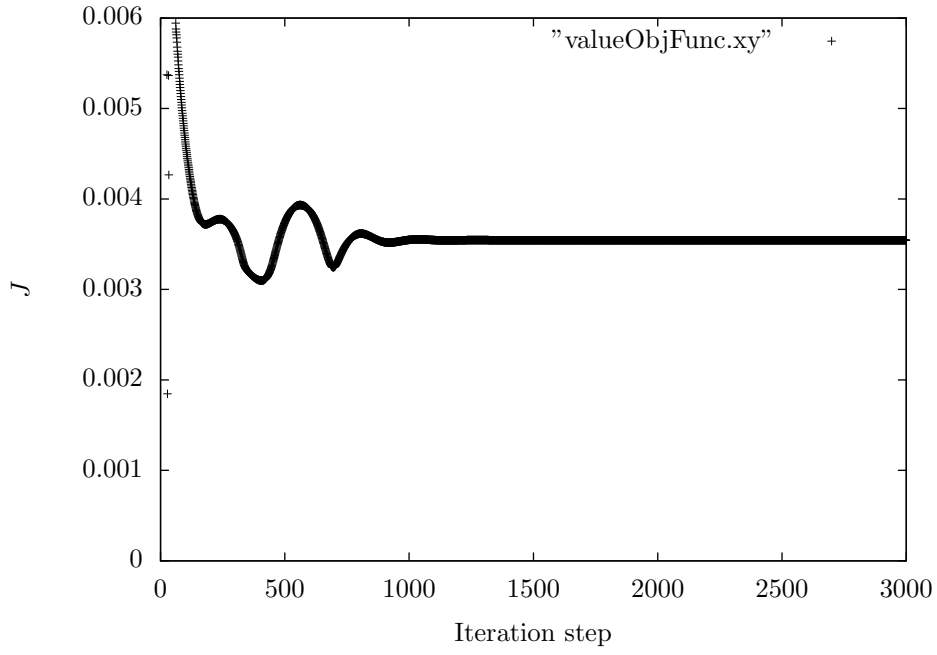


Figure 5.2: Final porosity and primal velocity field.

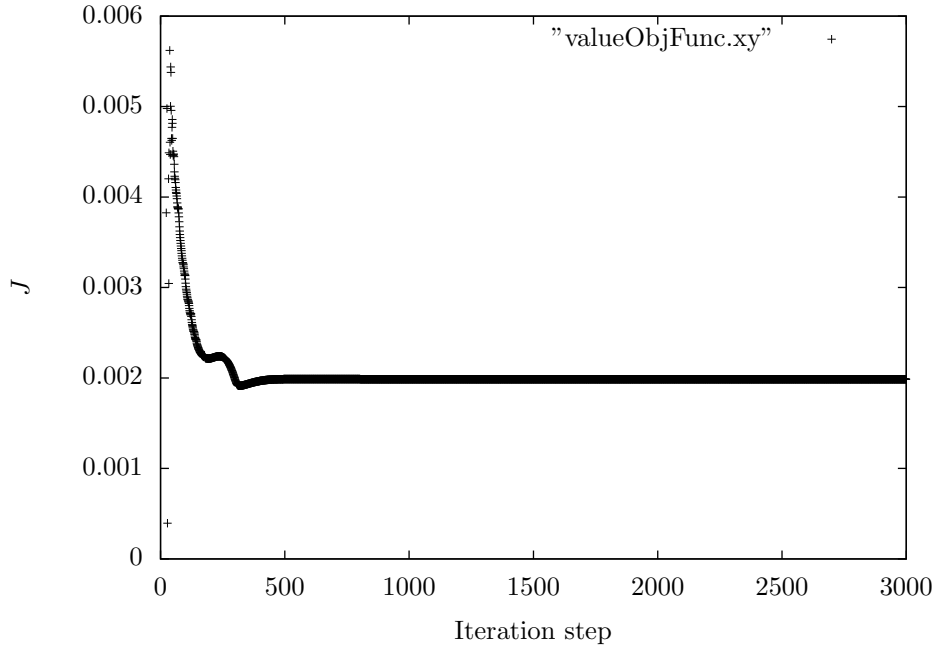
5.2 Power dissipation

5.2.1 pitzDaily

In this section results from the `pitzdDaily` case is presented using the solver for power dissipation cost function. The simulations are run for 3000 iterations steps to ensure converged results, however studying Figure 5.3 the results can be seen to converge earlier than that. Comparing the value of J when the porosity field is used as compared to when it is not, Figure 5.3b) and 5.3a) respectively, the value of the cost function can be seen to be reduced from 0.00354508 to 0.00198719. Hence a reduction of about 43.9% in power dissipated. The final corresponding porosity field can be seen in Figure 5.4.



(a)



(b)

Figure 5.3: Graphs displying the value of the objective function for the `pitzDaily` case; a) Using a relaxation factor for `alpha` of 0, i.e. not updating the porosity; b) Updating the porosity using the sovler for power dissipation.

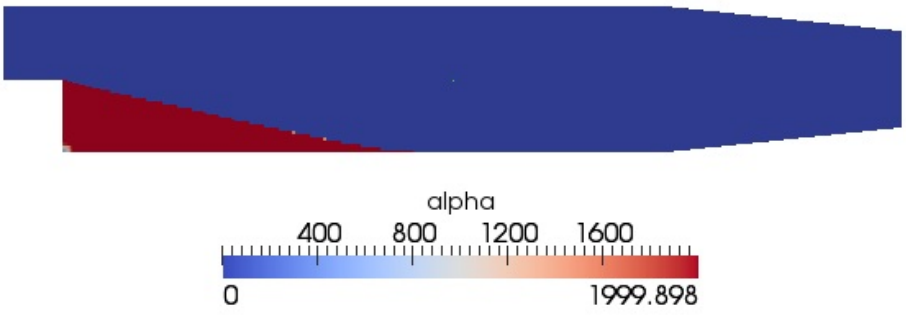


Figure 5.4: Final porosity and primal velocity field.

5.2.2 Pipe bend example

To test the modifications on a different case a `blockMeshDict` for a ducted flow in a box was created. As an industrial application this can be seen as a simplified case in which one wants to desgin the optimal pipe bend in a domain with constraints described by the boundaries. The case folder can be downloaded on the course homepage. Note that the mesh is uniform and fine throughout the entire domain to reduce the errors caused by applying first order upwind scheme and because the porosity update can happen anywhere. Figure 5.5-5.9 visuzalize the porosity field for different iteration steps.

Figure 5.5: Porosity field of the “pipe-bend” test case, at iteration number 200.

Figure 5.6: Porosity field of the “pipe-bend” test case, at iteration number 600.

Figure 5.7: Porosity field of the “pipe-bend” test case, at iteration number 1100.

Figure 5.8: Porosity field of the “pipe-bend” test case, at iteration number 3600.

Figure 5.9: Porosity field of the "pipe-bend" test case, at iteration number 7500. Note that it converged earlier that this iteration step.

Chapter 6

Limitations

Even though the advantages of an adjoint based optimization algorithm are many the project work has also highlighted problems with the method, or rather problems concerning the implementation of the algorithm. One of the biggest of those concern being the bad resolution of the primal flow. The results of the case presented in section 5.2.2 serves as a good example of one of the reasons, namely that the porosity field displaces the near wall region to areas where it is not resolved. However it might serve as a first rough topology change as a first step in a design procedure.

The current implementation is also limited in the sense that it only handles a very specific kind of cases. The first and foremost being that it is only suited for ducted, steady state flows, where the approximations done are reasonable.

6.1 Project suggestions

- Implement a process where the porosity field is used to create a new geometry.

- Modify the present solver to give surface sensitives rather than topology changes.

- Modify the present solver to a cost function with volumetric cost functions. This will result in some contribution to the adjoint Navier-Stokes equations.

- Analyze the optimization algorithm of the present sovler in more detail. Is it the optimal way to implement the steepest descent algorithm, what about the step length? Describe how modifications to the optimization method can be done / change the optimization algorithm from steeepest to something else, e.g. Newtons method.

Bibliography

[1] Othmer, C. & de Villiers, E. & Weller, H.G. (2007). Implementation of a continuous adjoint for topology optimization of ducted flows. *American Institute of Aeronautics and Astronautics*, AIAA-3947.

[2] Othmer, C. (2008). A continuous adjoint formulation for the computation of topological and surface sensitives of ducted flows. *Int. J. Numer. Meth. Fluid*, 58, 861-877.

[3] Versteeg, H. K. & Malalasekera, W. (2007). *An Introduction to Computational Fluid Dynamics The Finite Volume Method.* Harlow: Pearson Education Limited.

[4] Andreasson, N. & Evgrafov, A. & Patriksson, M. (2005). *An Introduction to Continuous Optimization* Lund: Studentlitteratur AB