USER MANUAL

"*adjointOptimisationFoam*, an OpenFOAM-based optimisation tool"

Prepared by

the Parallel CFD & Optimization Unit,
School of Mechanical Engineering,
National Technical University of Athens [1]

December 2023

[1]Prof. K.C. Giannakoglou, Dr. E.M. Papoutsis-Kiachagias, N. Galanos
[2]support: vpapout@mail.ntua.gr

Contents

1 Introduction **7**

2 optimisationDict **9**

2.1 optimisationManager 9

2.2 primalSolvers 9

2.2.1 Entries within each *primalSolver* sub-dictionary 11

2.2.1.1 solutionControls 12

2.2.1.1.1 averaging 12

2.3 adjointManagers 13

2.3.1 Entries within each *adjointManager* sub-dictionary 14

2.3.1.1 Entries within each *adjoinSolvers* sub-dictionary 15

2.3.1.1.1 objectives 16

2.3.1.1.2 ATCModel 24

2.3.1.1.3 solutionControls 25

2.4 optimisation 25

2.4.1 convergence 25

2.4.2 designVariables 26

2.4.2.1 Topology optimisation-related design variables 27

2.4.2.2 Shape optimisation-related design variables 35

2.4.3 sensitityType 36

2.4.3.1 topO 37

2.4.3.2 surface 37

2.4.3.3 surfacePoints 41

2.4.3.4 shapeESI 41

2.4.3.5 shapeFI 42

2.4.3.6 multiple 43

2.4.4 updateMethod 43

2.4.4.1 steepestDescent 44

2.4.4.2 conjugateGradient 44

2.4.4.3 BFGS 45

2.4.4.4 DBFGS 46

2.4.4.5 LBFGS 46
2.4.4.6 SR1 47
2.4.4.7 constraintProjection 47
2.4.4.8 SQP 48
2.4.4.9 nullSpace 48
2.4.4.10 ISQP 50
2.4.4.11 (GC)MMA 52

3 fvOptions 57

4 fvSolution 59

5 fvSchemes 61

6 adjointRASProperties 63
6.1 adjointSpalartAllmaras 63
6.2 adjointkOmegaSST 64

7 Adjoint boundary conditions 65
7.1 *Ua* boundary conditions 65
7.2 *pa* boundary conditions 66
7.3 *nuaTilda* boundary conditions 66
7.4 *ka* boundary conditions 66
7.5 *wa* boundary conditions 67
7.6 *ma* boundary conditions 67
7.7 *da* boundary conditions 67

8 dynamicMeshDict 69
8.1 Volumetric B-splines Morpher 69
8.1.1 Control points and B-splines properties 69
8.1.1.1 Entries in each dict within *volumetricBSplinesMotionSolver-Coeffs* 72
8.1.1.2 Computing parametric coordinates 74
8.1.1.3 Visualizing the control points 75
8.2 Laplace-based Grid Displacement Equation 75

9 Applications 77
9.1 solvers 77
9.1.1 adjointOptimisationFoam 77
9.1.1.1 Solution of the primal and adjoint equations and computation of sensitivities 77
9.1.1.2 Automated shape optimisation loops 78

9.2 utilities 78
9.2.1 preProcessing 78
9.2.1.1 writeMorpherCPs 78
9.2.2 postProcessing 79
9.2.2.1 computeSensitivities 79
9.2.2.2 cumulativeDisplacement 79

A A Short Introduction to Topology Optimisation 81

Chapter 1

Introduction

The present User Manual serves as a guide for the setup and usage of the OpenFOAM executable *adjointOptimisationFoam*, included in OpenFOAM-v2312, for topology (TopO) and shape (shpO) optimisation. Emphasis is given on the dictionaries and entries required to setup the continuous adjoint solvers and their utilities. The manual assumes that the reader is familiar with the OpenFOAM environment. No theoretical background for the adjoint method is provided in this document, unless necessary for the explanation of the code setup. The reader should refer to the relevant publications for details on the adjoint method, [4, 10, 19]. A complete list of bibliographic references to the developed adjoint methods can be found in the relevant publications listed here.

In the contents of this manual, the following conventions are used. Keywords mentioned in *italics* will refer to OpenFOAM dictionaries or dictionary entries. Blue color will be used to identify dictionaries or entries that are optional. Red color will be used to identify default values for variables, if they are not explicitly provided. Green color will be used to indicate the path to certain tutorials. All tutorials pertaining to *adjointOptimisationFoam* can be found under

```
$FOAM_TUTORIALS/incompressible/adjointOptimisationFoam
```

Magenta color will be used to indicate that an option is run time modifiable.

Chapter 2 describes in detail the entries of *optimisationDict*, the basic dictionary driving the adjoint code. Chapter 4 describes the entries to be added to *fvSolution* while chapter 5 the ones to be added to *fvSchemes*. Chapter 6 describes entries related to the adjoint to turbulence models, chapter 7 provides guidelines for defining the adjoint boundary conditions while chapter 8 explains the setup for deforming the mesh during shape optimisation runs, including the setup of a volumetric B-splines morpher. Chapter 9 describes the applications (solvers and utilities) used to solve the flow (primal) and adjoint equations, compute the sensitivity derivatives and perform automated shape optimisation loops.

Chapter 2

optimisationDict

optimisationDict is the main dictionary in which almost all information about the solution of the primal and adjoint equations and the execution of the automated adjoint-based optimisation loop (fig. 2.1) is set up. It is located in *system* and needs to be present in practically all applications presented in section 9 to run. The various sub-dictionaries and entries of *optimisationDict* are presented in detail in the sections that follow.

2.1 optimisationManager

```
    optimisationManager  singleRun;
```

The *optimisationManager* entry defines the mode of operation of the *adjointOptimisationFoam* executable.

optimisationManager: (singleRun, steadyOptimisation)
singleRun is used to solve the primal and adjoint equations corresponding to each primal and adjoint solver just once, without performing an optimisation loop. *steadyOptimisation* is used when an automated optimisation loop is targeted. Further details about the setup of the code in either scenario are given in sections 9.1.1.1 and 9.1.1.2.

2.2 primalSolvers

```
primalSolvers
{
    p1
```

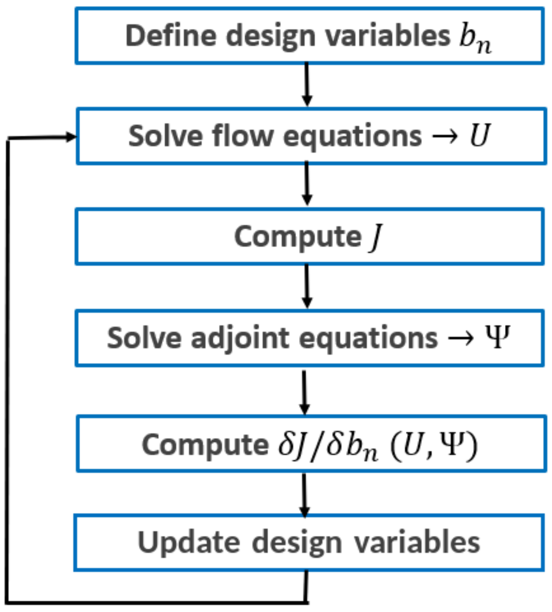
Define design variables b_n
Solve flow equations → U
Compute J
Solve adjoint equations → Ψ
Compute $\delta J/\delta b_n\ (U,\Psi)$
Update design variables

Figure 2.1: The adjoint-based shape/topology optimisation loop executed by *adjointOptimisationFoam* when running in *steadyOptimisation* mode. When performing shape optimisation, the design variables control the aerodynamic shape. In topology optimisation, the design variables are the values of the porosity/impermeability field at each cell, indicating the presence of void (fluid) or solid phase.

```
    {
        active                   true;
        type                     incompressible;
        solver                   simple;
        useSolverNameForFields   false;
        solutionControls
        {
            nIters  3000;
            residualControl
            {
                "p.*"   1.e-8;
                "U.*"   1.e-8;
            }
            averaging
            {
                average    true;
                startIter  1000;
                }
            }
        }
}
```

The *primalSolvers* dictionary is where the solver(s) of the primal equations are defined. One set of primal equations will be solved for each sub-dictionary within *primalSolvers*. A situation in which more than one primal solvers must be used is when tackling multi-point optimisation problems (e.g. minimizing airfoil drag in two different farfield flow angles).

2.2.1 Entries within each *primalSolver* sub-dictionary

active: (true|false)
Whether the primal equations corresponding to this solver are going to be solved or not.

type: (incompressible)
Type of the primal solver. Only one option valid for now.

solver: (simple, RASTurbulenceModel)
Solution algorithm used to solve the primal equations. *simple* will replicate the behaviour of *simpleFoam* while *RASTurbulenceModel* will solve the turbulence model PDEs, as set-up in *constant/turbulenceProperties*, using constant *U* and *phi* fields.

useSolverNameForFields:(true|false)
If set to true, all flow variable names related to this solver will be appended with the solver name (e.g. “U” would become “Up1”). If this is the case, the entries in *fvSolution* (solvers, relaxationFactors, etc) and *fvSchemes* (discretization schemes for grads, divs, etc) have to appropriately be adapted manually (the use of wildcards can produce compact setups in these cases). As of v2312, this flag is set to true automatically for multi-point runs and, for convenience sake, should better be kept to false for single-point runs. If set to true, boundary conditions will be read in the following way:

- If a file exists with the specific field name (e.g. “Up1”), boundary conditions will be read from there.
- If not, the code will attempt to read the base field file (e.g. “U”). If this fails, the code will exit with an appropriate error message.

Note: Boundary conditions that require field names (e.g. inletOutlet requires the “phi” name, which defaults to “phi”) should be set appropriately.

2.2.1.1 solutionControls

solutionControls contains entries used to manage the solution process of the primal equations. For the *simple* solver, among others, its entries include all entries that would be read through *system/fvSolution/SIMPLE* if *simpleFoam* was used instead of *adjointOptimisationFoam*.
Note: if *system/fvSolution/SIMPLE* exists, it will take precedence over *solutionControls* in *optimisationDict*; hence it is recommended to avoid its inclusion.

Additional entries include:

nIters
Maximum number of iterations for the numerical solution of the primal equations.

nInitialIters optional, default=nIters
The number of iterations for solving the primal equations in the first optimisation cycle could potentially be higher than *nIters*. This is because the solution of the primal PDEs at each new optimisation cycle uses the fields computed at the previous cycle as initialisation. Thus, the primal equations at each optimisation cycle other than the first one will likely require less iterations than the first cycle to converge.

2.2.1.1.1 averaging

averaging is optional. It controls averaging of the primal fields during the solution of the primal equations. This is mainly used to feed the adjoint equations with averaged primal fields in cases a limit-cycle oscillation manifests during the primal solution (e.g. solving a, practically, unsteady flow using a steady-state solver like *simpleFoam*).

average (true, false)
Whether to perform averaging or not. If set to true, all primal fields related to the solver will be averaged (e.g. *U*, *p*, *phi*, turbulence model variables, etc). Averaged field names consist of the original field name, appended by 'Mean'.

startIter
Starting iteration of the averaging process.

```
shapeOptimisation/motorBike
```

2.3 adjointManagers

```
adjointManagers
{
    am1
    {
        primalSolver              p1;
        operatingPointWeight      1;
        adjointSolvers
        {
            as1
            {
                // choose adjoint solver
                //---------------------
                active                    true;
                type                      incompressible;
                solver                    adjointSimple;
                useSolverNameForFields    false;
                computeSensitivities      true;
                isConstraint              false;
                // manage objectives
                //-----------------
                objectives
                {
                    type   incompressible;
                    objectiveNames
                    {
                        losses
                        {
                            type     PtLosses;
                            weight   1;
                            patches  (Inlet Outlet);
                        }
                    }
                }
                // ATC treatment
                //-------------
                ATCModel
                {
                    ATCModel              standard;
```

```
                        extraConvection     0;
                        zeroATCPatchTypes   ();
                        nSmooth             0;
                        maskType            faceCells;
                    }
                    // solution control
                    //-----------------
                    solutionControls
                    {
                        nIters              3000;
                        printMaxMags        true;
                        residualControl
                        {
                            "pa.*"          1.e-7;
                            "Ua.*"          1.e-7;
                        }
                        averaging
                        {
                            average    true;
                            startIter  1000;
                        }
                    }
                }
            }
        }
}
```

One *adjointManager* should be defined for each primal solver present in the *primalSolvers* dictionary (section 2.2). Each *adjointManager* is responsible for the adjoint PDEs to be solved at the corresponding operating point.

2.3.1 Entries within each *adjointManager* sub-dictionary

primalSolver

The name of the primal solver dict (section 2.2) corresponding to the current operating point.

operatingPointWeight 1;

When having multiple objective functions defined across many operating points, they have to be concatenated into a single J using user-defined weights, i.e. $J = \sum_i w_i^{op} J_i^{op}$, where J_i^{op} is the objective of i-th operating point (see also section 2.3.1.1.1 and eq. 2.1) and w_i^{op} the corresponding weight; *operatingPointWeight* corresponds to w_i^{op}.

adjointSolvers
A list of dictionaries, setting up the adjoint solvers to be used in this operating point. One set of adjoint PDEs will be solved for each adjoint solver and one corresponding set of sensitivity derivatives will be computed. Use multiple *adjointSolvers* only if sensitivities of multiple objectives must be computed separately from each other. If the weighted sum of different objectives is of interest, a single *adjointSolver* should be used and the weights of each objective should be defined in the *objectives* dictionary, section 2.3.1.1.1.
Note: The names of the sub-dictionaries within *adjointSolvers* should be unique across all operating points (i.e. across all *adjointManagers*).

2.3.1.1 Entries within each *adjoinSolvers* sub-dictionary

active: (true|false)
Whether the adjoint equations should be solved for this *adjointSolver*.

type: (incompressible null)
Type of the adjoint solver; *incompressible* corresponds to adjoint solvers related to incompressible flows whereas the *null* type should be used to compute sensitivity derivatives for geometric objective functions (i.e. objective functions not including flow variables; see also section 2.3.1.1.1).
solver (adjointSimple)
Solution algorithm used to solve the adjoint equations. Only the *adjointSimple* option is available for now.

useSolverNameForFields: (true|false)
The equivalent of the *useSolverNameForFields* in the *primalSolver* setup, section 2.2.1. Is automatically set to *true* if more than one incompressible *adjointSolvers* are present (i.e., wont be affected by the presence of *null* adjoint solvers)).

computeSensitivities: (true|false)
Whether to compute sensitivity derivatives or not, after solving the adjoint equations.

isConstraint: (true|false)
Whether the objective function of this adjoint solver will act as a constraint. See also section 2.4.4 for the appropriate *updateMethods* to be used in the presence of constraints.

isDoubleSidedConstraint: (true|false)
Whether the objective function of this adjoint solver will act as a double-sided inequality constraint of the form $l_t < J < u_t$, where l_t is the lower constraint value provided by the *targetLeft* entry in each *objective* dictionary of this adjoint solver and u_t is the upper target value provided by *target* in the same dictionary. See also section 2.4.4 for the appropriate *updateMethods* to be used in the presence of inequality constraints.

```
shapeOptimisation/naca0012/laminar/multipleConstraints
```

2.3.1.1.1 objectives

type: (incompressible geometric)
Type of objective functions to be constructed; *incompressible* pertains to objective functions depending on (incompressible) flow fields whereas *geometric* ones rely only on geometric quantities and are usually employed as constraints.

objectiveNames
A list of dictionaries corresponding to the objective functions to be minimised. Each objective function value is written in a file located in the *optimisation* folder, under *objective/TimeName/objectiveName+AdjointSolverName*. One set of adjoint equations is solved for each *adjointSolver*, minimizing the weighted sum of the objectives declared in *objectiveNames*, i.e.

$$J^{op} = \sum_i w_i J_i \tag{2.1}$$

where superscript op refers to the current operating point; one should make a distinction between w_i, which weight the objective functions of the current operating point and w_j^{op}, which are used to concatenate various J_j^{op}, given by eq. (2.1), into a single objective function.

Note: The names in *objectiveNames* should be unique across all *adjointManagers* points and *adjointSolvers*.

Entries in each dictionary under *objectiveNames*

The entries in each dictionary under *objectiveNames* depend on the objective type. The two mandatory entries are

type (force, forceTarget, moment, PtLosses, nutSqr, flowRate, flowRatePartition, uniformityPatch, uniformityCellZone, powerDissipation, partialVolume, topOVolume, topOSolidVolume)

The type of the objective to be minimized.

weight
Objective function weight (see also eq. (2.1). Since the software is developed to compute the minimum of an objective function, in case the user wants to maximise it, a negative weight should be used.

target
Each objective function F can also be used as a constraint. To do so, this entry is used to assign a target value F_{tar} to F, i.e. $J = F - F_{tar}$. To use the defined objective as a constraint, the *isConstraint* entry should also be set to *true*, section 2.3.1.1.

targetLeft
Should be prescribed if *isDoubleSidedConstraint* is set to *true* in the corresponding adjoint solver (see also section 2.3.1.1), and corresponds to l_t in a constraint of the form $l_t < J < u_t$. See also section 2.4.4 for the appropriate *updateMethods* to be used in the presence of inequality constraints.

normalize: (true|false)
It specifies whether the value of J is normalised with a factor, which is equal to either the J value at the first optimisation cycle or a value defined by the user through the entry normFactor.

A typical setup and a short description for each of the available objectives follows

Objective function of type *incompressible*

force

$$J = \frac{\int_{S_W} \rho(-\tau_{ij} n_j + p n_i) r_i dS}{\frac{1}{2}\rho A U_\infty^2} \tag{2.2}$$

where τ_{ij} are the components of the stress tensor, p is the pressure divided by the constant density ρ and $\mathbf{n}$ the unit normal vector. Vector $\mathbf{r}$ defines the direction in which the force vector should be projected (e.g. parallel to the farfield velocity to minimize drag). In what follows, repeated indices imply summation. In addition, S_W are the wall patches on which *force* is defined, A is the frontal area and U_∞ the farfield velocity magnitude.

A typical force dictionary would read

```
drag
```

```
{
    weight       1.;
    type         force;
    patches      ("wall.*" wallGroup); //wild cards, group names, etc
    direction    (0.99939 0.03489 0);
    Aref         2.;
    rhoInf       1.225;
    UInf         1.;
}
```

Note: Recall that the code assumes objectives are going to be minimised. To maximise a force, either provide the opposite force direction vector or use a negative weight.

sensitivityMaps/naca0012/laminar/drag

moment

$$J = \frac{\int_{S_W} \rho r_i^M e_{ijk}(x_j - x_j^C)(-\tau_{kl} n_l + p n_k) dS}{\frac{1}{2}\rho A l U_\infty^2} \tag{2.3}$$

where $\mathbf{r^M}$ is the moment direction to be minimised, $\mathbf{x}$ the position vector of each boundary face, $\mathbf{x^C}$ the position vector of the rotation center, l the reference length and e_{ijk} the permutation symbol. The rest of the symbols coincide with those defined in *force*.

A typical moment dictionary would read

```
moment
{
    weight            1.;
    type              moment;
    patches           ("wall.*" wallGroup);
    direction         (0 0 1);
    rotationCenter    (0 0 0);
    Aref              1.;
    lRef              1.;
    rhoInf            1.225;
    UInf              6.;
};
```

sensitivityMaps/naca0012/laminar/moment

PtLosses

$$J = -\int_{S_{I,O}} \left(p + \frac{1}{2}v_k^2\right) v_i n_i dS \tag{2.4}$$

where S_I and S_O are the inlet and outlet patches, respectively. The inlet and outlet patches can be prescribed in the *patches* entry.

Note: In case the *patches* entry is missing, the code will attempt to identify the inlet/outlet patches automatically, by checking the mass flow from each mesh patch. This identification of the inlet and outlet patches happens before the flow equations are solved, so the flow initialisation might affect it.

```
losses
{
    weight              1.;
    type                PtLosses;
    patches             (Inlet Outlet);
};
```

sensitivityMaps/sbend/laminar

nutSqr

$$J = \int_{\Omega'} \nu_t^2 d\Omega \tag{2.5}$$

where Ω' is the part of the computational domain in which the objective is defined and ν_t is the turbulent viscosity. The objective has been used in the past to qualitatively quantify and minimize noise [11].

```
noise
{
    weight              1.;
    type                nutSqr;
    zones               (zone1 zone2 ...);
};
```

zones are the *cellZones* defining Ω'.

shapeOptimisation/sbend/turbulent/SA/opt/nutSqr

flowRate

$$J = \int_{S_O} v_i n_i dS \tag{2.6}$$

where S_O are the patches used to define the objective function (usually one or more outlets).

```
flowRate
{
    weight          -1; // maximize
    type            flowRate;
    patches         (outlet1);
}
```

Note: You can use a negative weight if you want to maxinize your objective function.

shapeOptimisation/fork-uneven/flowRate

flowRatePartition

$$\begin{aligned} J &= \frac{1}{2}\sum_{l=1}^{L}\left(t_l - \frac{\int_{S_{O_l}} v_i n_i dS}{m_I}\right)^2 \\ m_I &= -\int_{S_I} v_i n_i dS \end{aligned} \tag{2.7}$$

defines a metric that quantifies the distribution of the inlet flow rate (m_I) to specific outlets (S_{O_l}) with target percentages ($t_l, l \in [1, L]$, where L is the number of specified outlet patches).

```
partition
{
    weight          1;
    type            flowRatePartition;
    inletPatches    (inlet); // used to compute m_I
    outletPatches   (outlet1 outlet2);
    // Optional entry. If abscent, inlet flow rate will
    // be partitioned equally between outlets
    // targetFractions (0.5 0.5);
}
```

```
shapeOptimisation/fork-uneven/flowRatePartition
```

uniformityPatch

$$J = \frac{1}{2}\frac{\int_S (v_i - \overline{v_i})^2 dS}{\int_S dS}$$
$$\overline{v_i} = \frac{\int_S v_i dS}{\int_S dS} \tag{2.8}$$

is a uniformity index to be minimized, quantifying the variance of the velocity vector v_i w.r.t. the spatially averaged velocity $\overline{v_i}$, over prescribed surfaces (patches) S.

```
uniformity
{
   weight              1;
   type                uniformityPatch;
   patches             (outlet2);
}
```

Note: In case the *patches* entry is missing, the code will attempt to identify the outlet patches automatically, by checking the volume flow-rate from each mesh patch. This identification happens before the flow equations are solved, so the flow initialization might affect it.

```
shapeOptimisation/fork-uneven/uniformityPatch
```

uniformityCellZone

$$J = \frac{1}{2}\frac{\int_{\Omega'} (v_i - \overline{v_i})^2 d\Omega}{\int_\Omega' d\Omega}$$
$$\overline{v_i} = \frac{\int_\Omega' v_i d\Omega}{\int_\Omega' d\Omega} \tag{2.9}$$

similar to *uniformityPatch*, but this time defined over parts of the interior of the computational domain, Ω', set through *cellZones*.

```
uniformity
{
   weight              1;
```

```
    type                uniformityCellZone;
    zones               (zone);
}
```

shapeOptimisation/sbend/laminar/opt/unconstrained/uniformityCellZone

powerDissipation

$$J = \frac{1}{2}\int_{\Omega'} (\nu + \nu_t) \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right)^2 d\Omega \tag{2.10}$$

is the fluid power that is dissipated within part of the computation domain, Ω', defined through *cellZones*.

```
powerDissipation
{
    weight              1;
    type                powerDissipation;
    zones               (zone);
}
```

Note: In the absence of stresses on the "inlets" and "outlets" of the *cellZones* used to define the objective function, the latter is equivalent to volume flow-rate weighted total pressue losses (see *PtLosses*), [9].

shapeOptimisation/sbend/turbulent/SA/opt/powerDissipation

Objective functions of type *geometric*

partialVolume

$$J = \frac{V - V_{init}}{V_{init}} \tag{2.11a}$$

$$V = -\frac{1}{3}\int_{S_W} x_k n_k dS \tag{2.11b}$$

where V is the volume enclosed by the patches defining S_W and V_{init} is the volume of the initial geometry, defined in the same way.

```
losses
{
    weight          1.;
    type            partialVolume;
    patches         (pressure suction);
};
```

shapeOptimisation/naca0012/lift/opt/constraintProjection

topOVolume

$$J = \left(\frac{\int_\Omega (1-\beta) d\Omega}{\int_\Omega d\Omega} - \pi_{tar} \right) \frac{1}{\pi_{tar}} \tag{2.12}$$

quantifies the difference of the volume occupied by the fluid and a target value π_{tar}, normalised with the latter, in topology optimisation. This constraint is frequently used in topology optimisation since it prevents the creation of islands of fluids inside the solid domain and accelerates the convergence of the algorithm. Since this objective is already divided by the target volume, there is no need to normalize it; π_{tar} is defined through the *percentage* entry in the objective dictionary.

```
vol
{
    weight          1.;
    type            topOVolume;
    percentage      0.462;

}
```

topologyOptimisation/monoFluidAero/laminar/1_Inlet_2_Outlet/porosityBased/R_20x

topOSolidVolume

$$J = \left(\frac{\int_\Omega \beta d\Omega}{\int_\Omega d\Omega} - \pi_{tar} \right) \frac{1}{\pi_{tar}} \tag{2.13}$$

Similar to *topOVolume*, but quantifies the (normalised) percentage of the computational domain occupied by solid, instead of the fluid.

```
vol
{
    weight          1.;
    type            topOSolidVolume;
    percentage      0.462;

}
```

2.3.1.1.2 ATCModel

The *ATCModel* dict provides the available options for the so-called Adjoint Transpose Convection (ATC) term, existing 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 complex cases. Its entries read:

ATCModel (standard, UaGradU, cancel)
Form of the ATC term. The *standard* option computes it as $u_j \frac{\partial v_j}{\partial x_i}$, where **v** and **u** are the primal and adjoint velocity vectors, respectively. It is formulated by differentiating the non-conservative form of the convection term in the primal momentum equations. The *UaGradU* option 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. The *cancel* option excludes the ATC term from the adjoint momentum equations during the solution of the adjoint PDEs (at the same time, of course, with reduced accuracy depending on the case). In order of decreasing robustness, the options can be given as (*cancel, standard, UaGradU*).

extraConvection Defaults to 0.
In order to facilitate convergence, add and subtract the adjoint convection term as many times as specified by this entry, using slightly different discretisation schemes in order to add numerical dissipation.

zeroATCPatchTypes Defaults to an empty *wordList*.
A *wordList*. Zero the ATC term next to patches of the provided types. No zeroing will be conducted if the *wordList* is empty.

zeroATCZones Defaults to an empty *wordList*.
A *wordList*. Similar to *zeroATCPatchTypes* but works on the provided *cellZones*.

nSmooth Defaults to 0.
Propagate the smoothing of the ATC term, applied to the cells collected through *zeroATCPatchTypes* and *zeroATCZones*, by using a Laplacian-like filter *nSmooth* times.

maskType (faceCells, pointCells)
Determines the way of selecting cells next to the *zeroATCPatchTypes* for smoothing the ATC term. If *faceCells* is used, every cell having a face in the *zeroATCPatchTypes* boundaries is chosen whereas if *pointCells* is used, every cell that has a point in the *zeroATCPatchTypes* will be picked.

```
sensitivityMaps/naca0012/turbulent/liftFullSetup
```

2.3.1.1.3 solutionControls

solutionControls has entries used 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, section 2.2.1.1. Averaging can be applied to the adjoint fields, in a similar manner used for the primal ones, section 2.2.1.1.1. In this case, the mean adjoint fields will be used to compute the sensitivity derivatives.

Additional entries read:

printMaxMags: (true|false)
Whether to print the maximum values of the adjoint fields to the log file (useful convergence metrics).

2.4 optimisation

The *optimisation* dict should be present only when an automated optimisation loop is to be executed or sensitivity derivatives should be computed. Its sub-dictionaries follow:

2.4.1 convergence

An optional dictionary including convergence criteria for the optimisation loop. If the dictionary is absent, the optimisation will run for as many cycles as defined in *system/controlDict.endTime*. Its entries read

```
convergence
{
    designVariables     1.e-04;
```

```
        objective       1.e-04;
        feasibilityThreshold    1.e-06;
    }
```

designVariables
The maximum correction of the design variables, normalised with the current values of the latter, must be lower than this threshold to consider the optimisation loop converged.

objective
The ratio $(J^{k+1} - J^k)/J^k$ should be smaller than this value to to consider the optimisation loop converged, where J is the objective function value and the exponent indicates the optimisation cycle counter.

feasibilityThreshold *1.e-06*
In optimisation runs including constraints, apart from satisfying either the *designVariables* or the *objective* criteria described above, all constraints should additionally have a value less than the *feasibilityThreshold* to consider the optimisation loop converged.

If both the *designVariables* and *objective* criteria are defined, satisfying one of them is enough to terminate the optimisation loop.

```
shapeOptimisation/naca0012/laminar/multipleConstraints
```

2.4.2 designVariables

A dictionary defining the design variables of the optimisation problem, how to compute sensitivity derivatives of the objective and constraint function w.r.t. them and the magnitude of their initial (i.e. in the first optimisation cycle) update.

```
    designVariables
    {
        type            shape;
        shapeType       volumetricBSplines;
        sensitivityType shapeFI;
        patches         (motorBikeGroup);
        maxInitChange   2.e-3;
        lowerBound      0.1;
        upperBound      1.1;
    }
```

type (topO, dynamicTopO, levelSet, shape)
The type of the chosen design variables. Types *topO, dynamicTopO* and *levelSet* are all used to perform topology optimisation and are discussed in more detail in section 2.4.2.1. The *shape* option corresponds to shape optimisation which has its own sub-options, described in section 2.4.2.2.

sensitivityType (topO, surfacePoints, surface, shapeESI, shapeFI, multiple)
The type of sensitivity derivatives to be computed. For all design variables options related to topology optimisation (see *type* above), the entry *sensitivityType topO;* suffices. Entries *shapeESI, shapeFI* are used for shape optimisation while entries *surface-Points, surface* are used to compute sensitivity maps and are further discussed in section 2.4.2.2.

maxInitChange
This entry is used to define the max. change of the design variables during the first optimisation cycle. By defining this quantity, the *eta* entry in *updateMethod* (section 2.4.4) is defined implicitly and can thus be excluded from that dictionary. In any case, either *designVariables/maxInitChange* or *updateMethod/eta* has to be present in order to the define the step of the update.

lowerBound(s) and *upperBound(s)*
lowerBounds and *upperBounds* are optional lists corresponding to the user-defined lower and upper values of each design variable. These entries can be used in both shape and topology optimisation, as long as the *updateMethod* used supports bound constraints (see section 2.4.4). In some cases, it is more convenient to specify a common value for the lower and upper bounds of all design variables, through the *lower-Bound* and *upperBound* enties. For the *topO* and *dynamicTopO* design variables, lower and upper bounds of 0 and 1, respectively, are applied automatically.

2.4.2.1 Topology optimisation-related design variables

A short introduction to topology optimisation is given in appendix A. The reader is advised to go through this material before continuing in this section.

type topO;

topologyOptimisation/monoFluidAero/laminar/1_Inlet_2_Outlet/porosityBased/R_20x

These design variables correspond to porosity-based topology optimisation. In this case, the porosity value (stored in field *alpha*) of each cell which does not belong to *fixedZeroPorousZones* and *fixedPorousZones* (see below) is updated in each optimisation cycle.

```
designVariables
{
    type                    topO;
    sensitivityType         topO;
    fixedZeroPorousZones    (fixedZeroZones);
    fixedPorousZones    (fixedZones);
    fixedPorousValues    (1);
    activePorousZone    (activeZones);
    writeAllFields    false;
    regularisation
    {
        regularise    false;
        growFromWalls    false;
        meanRadiusMult    10;
      //radius              0.1;
        iters    500;
        tolerance    1.e-06;
        wallValue    1;
        project    false;
        function            tanh;
        b                   40;
    }
    betaMax      2500;
    maxInitChange  0.2;
}
```

fixedZeroPorousZones, fixedPorousZones, activePorousZones
These cellZone lists are used to tackle a few common scenarios in topology optimisation. Cells in *fixedZeroPorousZones* have a constant zero porosity value throughout the optimisation. Cells in *fixedPorousZones* have a constant porosity value given by the corresponding entry in *fixedPorousValues*; if *fixedPorousValues* is not provided, it is assumed that all its entries are equal to 1. Cells in *activePorousZones* have their porosity value updated during the optimisation. In case *activePorousZones* is empty or missing, all cells not belonging to the cellZones of *fixedZeroPorousZones* and *fixedPorousZones* are updated.

writeAllFields (true|false);
If set to *true*, the field of design variables *alpha* and the sensitivity derivatives of the objective and constraint functions defined in all adjoint solvers (named *topOSens* + *adjointSolverName*, where 'adjointSolverName' is the name of the dictionary defining the adjoint solver, see section 2.3.1) are written to files, according to the *writeInterval* in *system/controlDict*.

regularisation A common practice in topology optimisation is to employ some kind of regularisation to the design variables field, before using it to compute the porosity-dependent terms in the flow equations. Here, a Helmholtz-like filter is used, initially introduced in [5], which reads

$$-\left(\frac{R}{2\sqrt{3}}\right)^2 \frac{\partial^2 \tilde{\alpha}}{\partial x_j^2} + \tilde{\alpha} = \alpha \tag{A.4}$$

where $\tilde{\alpha}$ is the regularised porosity field and R can be seen as a smoothing radius, usually computed as a function of the average grid cell size (see *meanRadiusMult* below). Regularisation, as any other smoothing technique, unavoidably blurs the line between the fluid and solidified domains. To increase the contrast of the $\tilde{\alpha}$ field, a projection step, computing β based on $\tilde{\alpha}$ [5]

$$\beta = \frac{tanh(\eta b) + tanh\left[b\left(\tilde{\alpha} - \eta\right)\right]}{tanh(\eta b) + tanh\left[b\left(1 - \eta\right)\right]} \tag{A.5}$$

(with $\eta = 0.5$ and b being a sharpening parameter) should be additionally implemented. If no regularisation/projection is applied, $\beta = \alpha$. For more details, see also appendix A.

regularise (true|false);
This entry specifies whether regularisation is performed or not.

growFromWalls (true|false);
If set to true, a *fixedValue* boundary condition is applied on the wall boundaries when solving eq. A.4; a *zeroGradient* one is imposed otherwise.

wallValue 1;
If *growFromWalls* has been set to true, *wallValue* defines the Dirichlet boundary condition for $\tilde{\alpha}$, which defaults to 1. If a large regularisation radius is used, this wall value will be propagated to a significant part of the computational domain, blocking it off even in cases with zero values for all design variables. This could happen, for instance,

meanRadiusMultiplier (10)
By default, the smoothing radius R in eq. (A.4) is computed as a multiple of the average cell size, given by $\left(\overline{V}\right)^{1/d}$, where $\overline{V}$ is the average cell volume, and $d = 2,3$ in 2D and 3D cases, respectively. This multiple is given by *meanRadiusMultiplier* which defaults to 10. Alternatively, the radius can be provided directly by the *radius* entry (value in meters).

iters 500;
Max. number of iterations when solving eq. (A.4).

tolerance 1.e-06;
Residual threshold when solving eq. (A.4). In case the solver reaches the max. number of iterations before reaching this threshold, the solution of eq. (A.4) is terminated.

project (true|false, defaults to the value of *regularise*);
This entry specifies whether the projection step of eq. (A.5) is performed or not.

function
This entry corresponds to the projection function. It is usually chosen to be *tanh*. i.e. eq. (A.5).

b
The parameter that controls the steepness of the projection function; the user can define the type of control over b through a sub-dictionary named b, located in *regularisation*. As a rule of the thumb, b should follow the trend of R, i.e. larger R values call for larger b values.
Available types are

type: (constant|scale)
When *constant* is chosen, then the value of b is simply defined as follows

```
    b       7;
```

scale is chosen, when the value of b may change between successive optimisation cycles. This is possible by using any of the available *Function1* objects; a typical set-up follows

```
b
{
    type  scale;
    scale
    {
        type        stepRamp;
        start       0;
        duration    100;
        interval    10;
    }
    value
    {
        type        constant;
        value       constant 100;
    }
}
```

In the above sub-dictionary, *type* expects the name of the function which updates the values of b throughout the optimisation cycles. *stepRamp* is a stepped ramp function which updates b every *interval* optimisation cycles until it reaches the value of 1. Updating b starts from the cycle defined with *start* and goes on for as many optimisation cycles as defined by the *duration* entry; when b reaches the value of 1, it does not change thereafter. In the sub-dictionary above, the value of this function is multiplied by a value produced by another *Function1* object, which is defined in the sub-dictionary *value*. The user can choose any other *Function1* object, implemented in *$FOAM_SRC/OpenFOAM/primitives/functions/Function1*.

betaMax

The value of β is multiplied by β_{max} before being used as a source term in the flow equations. The value of β_{max} should be high enough to block the flow in the solidified part of the computational domain. However, using an excessively high value usually makes the optimisation numerically unstable. This value can either be given directly through the *betaMax* entry (corresponding to *betaMaxType value;* which is the default one) or can be computed based on the following approaches:

```
designVariables
{
    betaMaxType     (value|Darcy|ReynoldsDarcy);
    betaMax          2500;
}
```

When the *Darcy* option is used, β_{max} is computed based on the Darcy number which expresses the ratio between viscous and porous forces and is computed as $Da = \frac{\nu}{L^2 \beta_{max}}$.
In such a case, the following entries should additionally be defined

```
betaMaxType     Darcy;
DarcyCoeffs
{
      Da      1.e-5;
      inletPatches     (inlet1 inlet2 ...);
   //length      1;
}
```

The characteristic *length* L can either be explicitly provided or can be computed using the hydraulic diameter computed based on the *inletPatches*. The Darcy number can be given through the *Da* entry and defaults to 10^{-5}.

```
topologyOptimisation/monoFluidAero/laminar/3DBox/losses
```

Alternatively, β_{max} can be computed based on the product of the Reynolds and Darcy numbers, expressing the ratio between momentum and porous forces and computed as $ReDa = \frac{U}{L\beta_{max}}$. In such a case, the following entries should additionally be defined

```
betaMaxType     ReynoldsDarcy;
ReynoldsDarcyCoeffs
{
      ReDa      1.e-5;
      inletPatches     (inlet1 inlet2 ...);
   //length      1;
      Uref      10;
}
```

where the characteristic length quantities are defined in the same way as in the *Darcy* option and a reference velocity, *Uref*, should be provided too.

type dynamicTopO;

In contrast to *topO*, the porosity field is not updated simultaneously at all cells, but only in the "active" ones. Initially, a set of "active" cells has to be selected. To do so, the user can define the patches (*seedPatches*), or the faceZones (*seedFaceZones*), to which the "active" cells are attached. The user might also provide directly the cellZones (*seedCellZones*) which include the initial "active" cells. In each subsequent optimisation cycle, the set of “active” cells is enriched with cells neighboring to the current ones, before updating the porosity field.

To facilitate this functionality, the *marchingCoeffs* sub-dictionary should be present inside *designVariables*:

```
type (dynamicTopO);
marchingCoeffs
{
    seedPatches (designWall);
    seedCellZones (dynamicCellZone);
    seedFaceZones (dynamicFaceZone);
    marchingStep 2;
}
```

By choosing *marchingStep* to be equal to 1, the list of the “active” cells is enriched by the direct neighbors of the current “active cells” at a subsequent optimisation cycle; a value of 2 indicates that second neighbors are also considered etc.

The rest of the entries mentioned in the *type topO;* section are used here too.

type levelSet;

```
designVariables
{
    type levelSet;
    sensitivityType topO;
    fixedZeroPorousZones    (fixedZones);
    regularisation
    {
        regularise      true;
        meanRadiusMult  10;
        //radius  0.05;
    }
    interpolation
    {
        function      sigmoidalHeaviside;
        meanRadiusMult   1;
        //d              0.01;
    }
    initialisation
    {
        method          meshWave;
    }
    betaMax      2500;
    maxInitChange  0.2;
}
```

The rational of the *levelSet* approach for topology optimisation is similar to that employed when using the *topO* design variables, in the sense of computing and applying a source term that blocks-off counter-productive areas of the computational domain, but differs in the approach used to compute them. Here, the design variables consist of a signed field of the size of the mesh cells (called *alpha*). This field is then regularised using eq. (A.4), to compute the *alphaTilda* field. Then, signed distances are computed for each cell, using the zero iso-surface of the *alphaTilda* field as the seed and maintaining its sign; this result is stored in the *signedDistances* field. Finally, the indicator field, β, is computed a smooth Heaviside function.

Entries *fixedZeroPorousZones, fixedPorousZones, fixedPorousValues, activePorousZones, betaMax* as well as the *regularisation* dictionary perform the same way as in *type topO;* design variables. A couple of points of interest are that *growFromWalls* and *projection* should be set to *false* in the *regularisation* dictionary. Transition from the signed distance field to the indicator field β is performed through the following dictionary

```
    interpolation
    {
        function      sigmoidalHeaviside;
        meanRadiusMult   1;
        //d              0.01;

      // function  smoothHeaviside;
      // b  2;
    }
```

where *function* can be either *sigmoidalHeaviside* or *smoothHeaviside*. In the former, a near-band distance from the zero level-set surface is defined either through *d* (in [m]) or *meanRadiusMultiplier* (as a multiplier of the mean cell size), in which β smoothly transitions from zero to one (corresponding to the fluid and solid domains, respectively). In the latter, a sharpness parameter *b* is defined, where larger values lead to narrower transition areas. The mathematical formulas for the two interpolation functions are

$$\beta^{sigmoid} = 0.5\left(1 + \frac{\Delta}{d_{NB}} + sin\left(\frac{\pi\Delta}{d_{NB}}\right)\frac{1}{\pi}\right)$$
$$\beta^{smooth} = 0.5\,(1 + tanh(b\Delta))$$

where Δ is the signed distance field, d_{NB} the narrow-band distance and b a sharpness parameter.

```
topologyOptimisation/monoFluidAero/laminar/1_Inlet_2_Outlet/levelSet/
R_10x_NB_01x
```

The *levelSet* design variables are in a development stage and could be adjusted soon.

2.4.2.2 Shape optimisation-related design variables

```
designVariables
{
    type            shape;
    shapeType       volumetricBSplines;
    sensitivityType shapeFI;
    patches         (lower upper);
    maxInitChange   2.e-3;
}
```

shapeType (Bezier, volumetricBSplines)
The available parameterisations for shape optimisation. When using the control points of *Bezier* curves as the design variables, an additional dictionary is required, placed as a direct subDict of the *optimisationDict*, defining the number of control points and which of these have a confined movement in each direction.

```
Bezier
{
    nBezier 16;
    confineXmovement
    (
        true false false false false false false true
        true false false false false false false true
    );
    confineYmovement
    (
        true false false false false false false true
        true false false false false false false true
    );
    confineZmovement
    (
        true true true true true true true true
        true true true true true true true true
    );
}
```

shapeOptimisation/sbend/laminar/primalAdjoint

When using the control points of *volumetricBSplines* morphing boxes as the design variables, the latter are defined through *dynamicMeshDict*, as outlined in detail in section 8.1.

shapeOptimisation/sbend/laminar/opt/unconstrained/losses/BFGS

patches
a list of patches to be updated/morphed during shape optimisation.

2.4.3 sensitityType

```
designVariables
{
    ....
    sensitivityType    surfacePoints;
    patches (pressure  suction);
    options  ...
}
```

The *sensitivityType* entry is where the setup for the computation of sensitivity derivatives is provided. Sensitivities will be computed at the end of each adjoint solver, for the adjoint solvers for which *computeSensitivities* is set to *true*, section 2.3.1.1. The available options are *sensitivityType* (topO, surface, surfacePoints, shapeESI, shapeFI, multiple)

and a description for each of them follows. Options *surface* and *surfacePoints* are used to compute sensitivity maps and cannot be used throughout an optimisation loop.

2.4.3.1 topO

The sensitivity type associated with all variants of design variables related to topology optimisation. No additional options can be applied here.

2.4.3.2 surface

This is used to compute the so-called sensitivity maps, i.e. the derivative of the objective function w.r.t. the normal displacement of the boundary wall faces. Upon computation, a *volScalarField* named *faceSensNormal*, appended with the name of the *adjointSolver*, will be written at the current time-step folder for each *adjointSolver* declared, section 2.3.1. Keeping in mind the convention for the surface normal unit vector, facing from the fluid to the solid boundaries, positive sensitivities indicate a movement opposite to the geometry normal (“outwards” or “inwards”, for external or internal aerodynamics, respectively); negative sensitivities indicate a movement aligned to the geometry normal (“inwards” or “outwards”, for external or internal aerodynamics, respectively) to minimize the given objective function, fig. 2.2

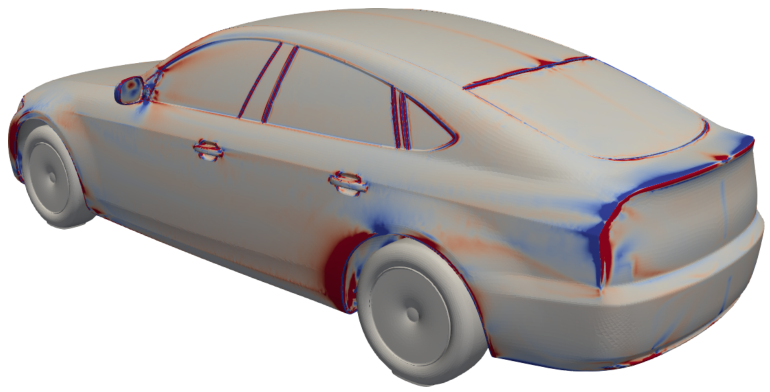

Figure 2.2: Drag sensitivity map computed on the surface of the DrivAer car model. Blue areas should be moved according to the surface normal (“inwards”) to reduce drag while red areas should be moved in the opposite direction.

A typical setup reads

```
sensitivityType                 surface;
```

```
patches                          ( "wall.*");
includeSurfaceArea               true;
includeMeshMovement              true;
adjointMeshMovementSolver
{
    iters          300;
    tolerance      1.e-7;
}
// Entries related to sensitivity smoothing
smoothSensitivities              true;
meanRadiusMultiplier             10;
//radius                         1;
```

includeSurfaceArea (true|false)
Whether to include the local face area in the sensitivity values or not. Should be set to true if the actual impact of a face movement is required and the mesh resolution impact should be taken into consideration (i.e. a unit movement of a face with a large area will cause a relatively big shape change and, hence, will have a large sensitivity value). On the contrary, if a normalized sensitivity distribution is required to get an overview of the surface areas with high optimisation potential, this option should be set to false. In this case, the sensitivity value should be interpreted as “what will be the change in the objective, if a node is moved in such a way that the change in the local face area is unitary”.

includeMeshMovement (true|false)
Whether to take into consideration the sensitivity contribution arising by the adjoint to the grid displacement scheme or not. If set to false, the so-called Surface Integrals (SI) formulation will be used, whereas if set to true, the so-called Enhanced Surface Integrals (E-SI) approach will be employed, [4]. The latter assumes that, after updating the geometry, the grid will be displaced using a set of Laplace-based PDEs and solves the adjoint to that problem. In order to do so, boundary conditions for the adjoint to the grid displacement variable (a *volVectorField* named *ma*) should be set. These should be of zero *fixedValue* type for all boundaries, expect the constrained (i.e. cyclic, processor, symmetry, etc) ones. The *ma* field is generated automatically by the code, unless read from the current time-step folder. In addition, a solver for *ma* should be added to *fvSolution* and a discretisation scheme for *laplacian(ma)* should be added in *fvSchemes/laplacianSchemes*, unless a default one is present. No relaxation is required for the solution of this equation. It is highly recommended to switch the *includeMeshMovement* to true in order to increase the accuracy of the computed sensitivities.

An additional, optional dictionary named *adjointMeshMovementSolver* can be provided to control the convergence of the adjoint grid displacement PDEs. If not provided, the following default values will be used. Its entries read

adjointMeshMovementSolver

iters 1000
Maximum number of iterations for the adjoint grid displacement solver.

tolerance 1.e-06
Residual to be reached before considering the adjoint grid displacement PDEs as converged.

For cases in which the Spalart–Allmaras turbulence model is differentiated (section 6), additional entries may be supplied under the *designVariables* dict. These read

includeDistance (true|false)
Whether to solve the adjoint to the eikonal equation or not, [10]; only for cases including the adjoint to the Spalart–Allmaras turbulence model, section 6. If set to true, boundary conditions for the adjoint distance field (a *volScalarField* named *da*) should be set. These should be of zero *fixedValue* type for inlet and outlet boundaries and *zeroGradient* ones for walls. The *da* field 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 relaxation factor for the *da* equation. A discretisation scheme for *div(-yPhi,da)* should be added in *fvSchemes/divSchemes*. If *includeDistance* is set to true, an additional optional dictionary named *adjointEikonalSolver* can be provided to control the convergence of the adjoint eikonal PDE. A typical example reads

```
includeDistance                true;
adjointEikonalSolver
{
    iters           300;
    tolerance       1.e-7;
    epsilon         0.1;
}
```

The *iters* and *tolerance* entries are identical to the ones in *adjointMeshMovementSolver*, section 2.4.3.2. The *epsilon* entry (default value to the equivalent one in *fvSchemes .wallDict.advectionDiffusionCoeffs* if present, or set to 0.1 otherwise). For cases where stability issues emerge, a higher value can be used.

Note: it is important NOT to use *bounded* divergence schemes for the convection term of the adjoint eikonal equation, since *yPhi* is not conservative.

```
shapeOptimisation/sensitivityMaps/naca0012/turbulent/liftFullSetup
```

smoothSensitivities (true|false)
Whether to smooth the computed sensitivity map. When computing sensitivity maps on surface meshes generated from industrial geometries, the outcome might appear noisy, especially if a volume-to-surface approach is used for meshing, e.g. as used by snappyHexMesh. Even though the sensitivity map is technically correct, the noisy patterns that appear might make the extraction of useful information challenging. Smoothing can be used to facilitate the interpretation of the sensitivity map in such cases. The sensitivity map is smoothed through a Laplace-Beltrami filter of the form

$$-R^2\frac{\partial^2 \widetilde{m}}{\partial x_j^2} + \widetilde{m} = m \tag{2.14}$$

where m is the original sensitivity map, $\widetilde{m}$ the smoothed one and R the smoothing radius. Eq. 2.14 is solved on the part of the surface mesh defined by the patches on which the sensitivity map is computed, using the *finiteArea* infrastructure. If a finite area mesh is provided under *constant*, it will be used; otherwise it is created on-the-fly based on either an *faMeshDefinition* dictionary in the *system* directory, or constructed internally based on the sensitivity patches. An indicative example is given in fig. 2.3.

meanRadiusMultiplier (10)
By default, the smoothing radius R in eq. (2.14) is computed as a multiple of the average size of the boundary mesh elements; this multiple is given by *meanRadiusMultiplier* which defaults to 10. Alternatively, the radius can be provided directly by the *radius* entry (value in meters).
Note: From an optimisation point of view, the smoothing of eq. (2.14) can alternatively be seen as computing the sensitivity derivatives $\delta J/\delta b_i$ of the objective function J w.r.t. a different set of design variables $b_i, i \in [1,N]$, defined as

$$x_i = x_i^{init} + \widetilde{b}_i$$
$$\widetilde{b}_i - R^2\frac{\partial^2 \widetilde{b}_i}{\partial x_j^2} = b_i$$

where x_i are the coordinates of the updated geometry, x_i^{init} the ones of the initial geometry and $\widetilde{b}_i$ a smooth displacement field. In other words, no loss of accuracy is incurred by the smoothing; instead, sensitivities are computed w.r.t. a different set of design variables.

sensitivityMaps/motorBike

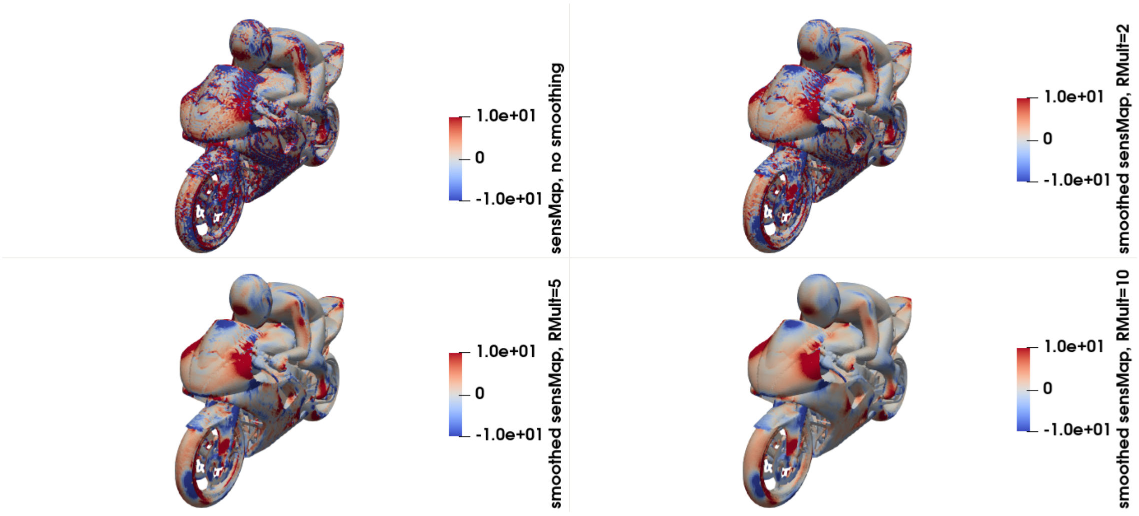
1.0e+01
0
-1.0e+01
sensMap, no smoothing
smoothed sensMap, RMult=2
smoothed sensMap, RMult=5
smoothed sensMap, RMult=10

Figure 2.3: Drag sensitivity map computed on the surface of the motorbike (tutorial under sensitivityMaps/motorBike). Blue areas should be moved according to the surface normal ("inwards") to reduce drag while red areas should be moved in the opposite direction. The four subfigures give the smoothed sensitivity map, computed with a progressively larger smoothing radius.

2.4.3.3 surfacePoints

Same as *surface*, section 2.4.3.2, but sensitivities are computed w.r.t. the normal displacement of boundary points, not faces. When sensitivity maps are of interest, this option should be preferred to *surface* since, in a hypothetical mesh movement, the boundary points would be moved, causing the change of the boundary faces. Upon computation, a *pointScalarField* named *pointSensNormal*, appended with the name of the *adjointSolver*, will be written at the current time-step folder for each *adjointSolver* declared, section 2.3.1. Entries discussed in section 2.4.3.2 are valid here as well, with the exception of the ones corresponding to sensitivity smoothing.

2.4.3.4 shapeESI

This option computes sensitivity derivatives for shape optimisation using the so-called E-SI approach, [4]. Sensitivities are computed using the chain rule, i.e.

$$\frac{\delta J}{\delta b_n} = \frac{\delta J}{\delta x_i}\frac{\delta x_i}{\delta b_n} \tag{2.15}$$

when $\delta J/\delta x_i$ is the sensitivity map (see section 2.4.3.2) and $\delta x_i/\delta b_n$ is computed analytically on the surface, based on the chosen design variables. The default settings provided in section 2.4.3.2 are used to compute the sensitivity map and can be altered in the *designVariables* dictionary if needed.

Upon computation, the sensitivity derivatives are written in a file named after the design variables' type, appended by the *adjointSolver* name and the time-step value and located in the *optimisation/derivatives* folder.

```
shapeOptimisation/sbend/laminar/primalAdjoint
```

2.4.3.5 shapeFI

This option computes sensitivity derivatives for shape optimisation using the so-called FI approach, [4]. This method requires the computation of the grid sensitivities $\frac{\delta x_k}{\delta b_n}$ for the entire computational domain and for each design variable. Hence, computing sensitivities with this approach can be time consuming for large numbers of design variables. Grid sensitivities can be computed analytically if the design variables correspond to *volumetricBSplines*. For *Besier* design variables, a vectorial Laplace equation needs to be solved for each design variable, to propagate the parameterisation information given in the $dxidXj\_i$ files (see also section 2.4.2.2) to the interior mesh. The maximum number of iterations and convergence criterion for these PDEs are read from the optional *dxdbSolver* subDict; if not provided, the default values given below will be used.

```
type                shape;
shapeType           Bezier;
sensitivityType     shapeFI;
patches (pressure  suction);
dxdbSolver
{
    iters 100;
    tolerance 1.e-11;
}
```

A *volVectorField* named *m* is used for this task with zero *fixedValue* boundary conditions for all patches. The *m* field is generated automatically by the code, unless read from the current time-step folder. In addition, a solver for *m* should be added in *fvSolution* and a discretisation scheme for *laplacian(m)* should be added in *fvSchemes/laplacianSchemes*, unless a default one is present. No relaxation is required for the solution of these equations. Sensitivity derivatives are stored in the same location mentioned in section 2.4.3.4.

shapeOptimisation/sbend/laminar/opt/unconstrained/losses/SD

2.4.3.6 multiple

```
designVariables
{
    type              shape;
    shapeType         Bezier;
    sensitivityType   multiple;
    sensitivityTypes  (FI ESI SI);
    patches           (lower upper);
    FI
    {
        sensitivityType     shapeFI;
        patches             (lower upper);
    }
    ESI
    {
        sensitivityType     shapeESI;
        patches             (lower upper);
    }
    SI
    {
        sensitivityType     shapeESI;
        patches             (lower upper);
        includeMeshMovement false;
    }
}
```

Provides a framework for computing multiple types of sensitivity derivatives, mainly for the purpose of comparison (i.e. not optimisation). Sensitivities will be computed for all sub-dictionaries present in the *sensitivityTypes* wordList.

shapeOptimisation/sbend/laminar/primalAdjoint

2.4.4 updateMethod

```
updateMethod
{
    method steepestDescent;
    //eta 1; //optional
    lineSearch
    {
        type ArmijoConditions;
    }
}
```

The method used to update the design variables is defined in this dictionary. All methods update the design variables using a scheme of the form

$$b_i^{new} = b_i^{old} + \eta p_i \tag{2.16}$$

where **b** are the design variables, **p** the update direction and η a user-defined (either explicitly or implicitly, see next entries) step. The dictionary entries read

method (steepestDescent, conjugateGradient, BFGS, DBFGS, LBFGS, SR1, constraint-Projection, SQP, nullSpace, ISQP, MMA)
Defines the method for updating the design variables. Only the *constraintProjection*, *SQP*, *ISQP*, *nullSpace* and *MMA* methods can handle constraints, with only the latter three handling inequality and bound constraints.

A short description of the entries required by each update method follows.

Update methods for unconstrained optimisation

2.4.4.1 steepestDescent

This is the simplest (but least efficient) method to update the design variables. The update vector is computed as

$$p_i = -\frac{\delta J}{\delta b_i} \tag{2.17}$$

No additional dictionary entries are required.

2.4.4.2 conjugateGradient

The Conjugate Gradient method, [2], for updating design variables. Significantly faster than *steepestDescent* but can still tolerate discrepancies in the sensitivities, in cases where some balance should be struck between accuracy and stability. An additional dictionary might be provided

```
conjugateGradient
{
    betaType  FletcherReeves;
}
```

betaType (FletcherReeves, PolakRibiere, PolakRibiereRestarted)
An optional entry choosing the formula to update the β variable in Conjugate Gradient, [7]. Defaults to *FletcherReeves* which has proved to be the most robust and should be preferred.

```
shapeOptimisation/motorBike
```

2.4.4.3 BFGS

```
method  BFGS;
BFGS
{
    nSteepestDescent  1;
    etaHessian        1;
    scaleFirstHessian  true;
}
```

The quasi-Newton BFGS method [7]. The update is computed through

$$\frac{\delta^2 \tilde{J}}{\delta b_i \delta b_j} p_j = -\eta_H \frac{\delta J}{\delta b_i} \tag{2.18}$$

where $\frac{\delta^2 \tilde{J}}{\delta b_i \delta b_j}$ is an approximation of the objective function Hessian and η_H is a user-defined constant. *BFGS* and its limited memory variant (see section 2.4.4.5) are probably the most widely used methods to update the design variables in general optimisation problems. Their convergence is significantly faster than *conjugateGradient* or *steepestDescent*, however they require highly accurate sensitivity derivatives. In cases that the primal and adjoint equations are converging without difficulties, *(L)-BFGS* should be the preferred methods; otherwise, *conjugateGradient* should be employed. The convergence rates of *BFGS* and *steepestDescent* can be compared by running the cases under

```
shapeOptimisation/sbend/laminar/opt/unconstrained/losses/SD
shapeOptimisation/sbend/laminar/opt/unconstrained/losses/BFGS
```

nSteepestDescent *1*;
Number of steepest descent updates to be conducted before applying the BFGS approach. Should be at least one, with higher values potentially needed in stiff problems. The η value defined (explicitly or implicitly) in *updateMethod* will be used for these updates.

etaHessian *1*;
In Hessian-based methods, the η value should theoretically be equal to 1. Since, however, BFGS is a quasi-Newton method an η_H value can also be provided. It is usually in the range of [0.5-1].

scaleFirstHessian (true|false);
Whether to scale the first Hessian matrix computed using a correction proposed in [7]. Usually improves the convergence rate of the method.

2.4.4.4 DBFGS

Similar to *BFGS*, but the vectors included in the rank-2 update of the Hessian matrix can be "damped" to maintain a positive-definite Hessian matrix throughout the optimisation, [7] (section 18.4). The damped formula is applied if $s_i y_i < \gamma s_i B_{ij} s_j$, where s_i is the last update of the design variables, y_i is the difference between the sensitivity derivatives of the last two optimisation cycles, B_{ij} is the approximate Hessian and γ defines the threshold for applying the damping, tunable through *gamma* in the DBFBS dictionary, with a default value of 2. All other inputs described in section 2.4.4.3 can be applied here too.

2.4.4.5 LBFGS

```
method LBFGS;
LBFGS
{
    nSteepestDescent 1;
    etaHessian       1;
    nPrevSteps       10;
}
```

The limited memory variant of the BFGS quasi-Newton method [6]. *LBFGS* is closely related to *BFGS*, however, instead of approximating and storing the (inverse) Hessian matrix, only a few vectors are stored that represent it implicitly. Hence, LBFGS is usually employed when the number of design variables is too large to allow storing the

Hessian matrix (for instance, when supporting the *ISQP* method for topology optimisation, see section 2.4.4.10). *LBFGS* has the same (optional) entries as *BFGS* (with the exception of *scaleFirstHessian*), with the addition of the following entries

nPrevSteps *10*;
Number of vectors used to implicitly retrieve the approximation to the Hessian matrix. A relatively small number, usually around 10, is enough in most problems.

useSDamping, useYDamping *false*;
Apply a damping similar to the one mentioned in section 2.4.4.4, on either the s or y vectors, to maintain the positive-definiteness of the Hessian matrix.

2.4.4.6 SR1

```
method SR1;
SR1
{
    nSteepestDescent  1;
    etaHessian        1;
}
```

The Symmetric Rank One (SR1) quasi Newton update method [7]. Similar convergence characteristics as *BFGS* and identical optional entries, with the exception of *scaleFirstHessian*.

Update methods handling equality constraints

2.4.4.7 constraintProjection

```
method constraintProjection;
constraintProjection
{
    useCorrection  true;
}
```

constraintProjection is an *updateMethod* that supports the handling of equality constraints, [13] . In particular, the update direction is that of steepest descent, if the part that is normal to all constraint isolines is subtracted. *constraintProjection* is wall suited for tackling (almost) linear constraint functions throughout the optimisation. One optional entry can be provided

useCorrection (true|false)
Whether or not to use a correction taking into consideration the non-linearity of the constraint function w.r.t. the design variables. It should be noted that if *useCorrection* is set to *false*, *constraintProjection* can be used to only keep the value of the constraint function the same as in the first optimisation cycle and not to obtain a user-defined target value.

```
shapeOptimisation/naca0012/lift/opt/constraintProjection
```

2.4.4.8 SQP

```
method   SQP;
SQP
{
    etaHessian          0.8;
    nSteepestDescent    1;
    scaleFirstHessian   true;
}
```

SQP implements the Sequential Quadratic Programming method for updating the design variables in the presence of equality constraints, [7], in order to iteratively satisfy the Karush-Kuhn-Tucker (KKT) conditions. The necessary Hessian matrix is approximated using BFGS and, hence, its optional entries are those described in section 2.4.4.3. *SQP*, like *BFGS* for unconstrained optimisation problems, exhibits a very fast converge but requires a high accuracy of the sensitivity derivatives (see also comments in 2.4.4.3).

```
shapeOptimisation/sbend/laminar/opt/constrained/SQP
```

Update methods handling inequality constraints

2.4.4.9 nullSpace

The *nullSpace* update method, [1], can handle inequality and bound constraints. The update of the design variables is split into a null-space approach, which reduces the objective function while keeping the constraint values constant in a first-order/linearised sense, and a range-space approach that reduces the constraint values by a desired amount. The concept is somehow similar with the one of *constraintProjection*, extended however so support inequality constraints and a mechanism for excluding constraint gradients from the null-space that “align” with the objective gradient, through the solution of a cheap dual problem of the size of the flow-related (i.e.non bound-related) constraints. The method has proved to perform well in topology optimisation problems, has a cheaper cost per update computation than *ISQP* (see section 2.4.4.10) but usually converges slower than the latter. The entries related to it follow

```
method    nullSpace;
nullSpace
{
      maxIters      6000;
      maxLineSearchIters      10;
      dualTolerance      1.e-12;
      violatedConstraintsThreshold      1.e-3;
      perturbation      1.e-2;
      adaptiveStep      true;
      lastAcceleratedCycle      20;
      strictMaxDVChange      false;
      maxDVChange      maxInitChange;
      targetConstraintReduction      0.5;
}
```

maxIters: 1000
Maximum number of Newton iterations for solving the dual sub-problem (includes iterations from all tolerance values used for the inequality constraints).

maxLineSearchIters: 10
Maximum number of line search iterations for determining the step of the update during the solution of the sub-problem.

dualTolerance: 1.e-12
The dual sub-problem is solved with an interior point method, using a sequence of stricter tolerances on the inequality constraints. This value represents the final/stricter tolerance.

violatedConstraintsThreshold: 1.e-3
The dual sub-problem includes only the violated and saturated constraints. Since, however, these could change from one optimisation cycle to the next, especially if the optimal solution lays in the vicinity of the target constraint values, a continuous change in this set of constraints could cause the optimisation to oscillate. To avoid this, non-violated constraints, up to a value of *violatedConstraintsThreshold* are also included in the set of constraints included in the dual sub-problem.

perturbation: 1.e-2
If all design variables lay on their bounds and there is at least one violated flow-related constraint, the system of equations of the dual sub-problem becomes singular. To avoid this, *perturbation* is added/subtracted from the design variables on the lower/upper bounds, at the beginning of the optimisation loop.

adaptiveStep: true
The null- and range-space updates are multiplied by scalars a_J and a_C, respectively, to put emphasis on the reduction of either the objective or the constraints of the optimisation and to accelerate the convergence. These values can be adjusted throughout the optimisation cycles (and up to the *lastAcceleratedCycle*) if *adaptiveStep* is set to *true*. In specific, a_J is computed such that the maximum change of the design variables in each optimisation cycle, up to the *lastAcceleratedCycle*, is equal to *maxDVChange*; the latter can be either provided directly or defaults to *maxInitChange*. If *strictMaxDVChange* is set to *false*, the a_J value computed at the *lastAcceleratedCycle* will be kept constant throughout the rest of the optimisation loop. Otherwise, a_J might be reduced to maintain the maximum update of the design variables per optimisation cycle smaller than the *maxDVChange*. In any case, a_J will not increase after the *lastAcceleratedCycle*.

targetConstraintReduction: 0.5
This entry is used to indirectly define the a_C value multiplying the range-space part of the update, related to the reduction of the constraints. In each optimisation cycle, constraints of the form $c^{k+1} < 0$ are relaxed as, $c^{k+1} - (1-t)c^k < 0$ where c is the constraint value, the exponent indicates the optimisation cycle counter and t is the target reduction of the constraint in this optimisation cycle, expressed as a fraction of its current value that can be set through *targetConstraintReduction*. A value of 1 means that no modification is applied to the constraint whereas a value of 0 means than even an infinitesimal reduction of the constraint is acceptable.

2.4.4.10 ISQP

```
method    ISQP;
ISQP
{
    etaHessian               0.8;
    epsMin      0.7;
    maxIters      1000;
    maxLineSearchIters      10;
    maxDxIters      1000;
    relTol      0.01;
    preconditioner     diag;
    targetConstraintReduction     1;
}
```

Similar to *SQP* but has also the ability to tackle inequality constraints as well as bounds for the design variables. The Hessian of the Lagrangian function is approximated using *LBFGS* and the Quadratic Programming (QP) problem, which might include inequality constraints, is solved using an interior point method [7]. Each iteration of the QP problem requires the solution of a potentially dense system of equations with the size of the active design variables and can, hence, have a non-negligible cost depending on their number (e.g. topology optimisation). This system is solved using a matrix-free Conjugate Gradient algorithm using preconditioners taking advantage of its structure. The optional entries of *ISQP* are those described in section 2.4.4.5, with the additions of

epsMin: 1.e-07
The QP problem is solved with an interior point method, using a sequence of stricter tolerances on the inequality constraints. This value represents the final/stricter tolerance.

maxIters: 1000
Maximum number of Newton iterations for solving the QP problem (includes iterations from all tolerance values used for the inequality constraints).

maxLineSearchIters: 10
Maximum number of line search iterations for determining the step of the update during the solution of the QP problem.

maxDxIters: 1000
Maximum number of steps for the matrix-free Conjugate Gradient solver of the QP problem.

relTol: 0.1
Similar notion to *relTol* as found in *fvSolution/solvers* but this time for the solution of the equations of the QP problem.

preconditoner: (diag, invHessian, ShermanMorrison)
The preconditioner of the matrix-free Conjugate Gradient solver of the QP problem. For cases with many design variables, it can make a significant difference in the CPU cost of solving the QP problem. Defaults to a *diagonal* preconditioner including the inverse of the diagonal part of the approximate Hessian, enhanced with quantities re-

lated to bound constraints, if present. The *invHessian* preconditioner uses the inverse of the approximate Hessian matrix, computed through *LBFGS*, as the preconditoner while the *ShermanMorrison* one augments the latter using the Sherman-Morrison formula for computing the inverse of a matrix appended by a rank-one update [14]. *ShermanMorrison* has the largest cost per preconditioner-vector product which depends on the number of (non-bound) constraints but usually leads to the fastest convergence in terms of overall QP problem iterations and the lowest CPU cost.

targetConstraintReduction: 1
The linearisation of the optimisation problem might lead to QP problems with no feasible solutions, depending on the initialisation. This can lead to very expensive of even divergent solutions for the QP problem. To relax them, a similar approach to that mentioned in the *targetConstraintReduction* entry of section 2.4.4.9 is used, using $max\left(c^{k+1} - (1-t)c^k, c^{k+1}\right) < 0$ as the relaxed form of the constraint. A typical value used in topology optimisation is 0.1.

Examples of the use of *ISQP* for shape and topology optimisation can be found in

```
shapeOptimisation/naca0012/laminar/multipleConstraints
topologyOptimisation/monoFluidAero/laminar/3DBox/
losses-mass-uniformity-SQP
```

whereas a comparison of *nullSpace* and *ISQP* for a topology optimisation problem can be found in

```
topologyOptimisation/monoFluidAero/laminar/3DBox/losses-mass-uniformity
topologyOptimisation/monoFluidAero/laminar/3DBox/
losses-mass-uniformity-SQP
```

2.4.4.11 (GC)MMA

The Method of Moving Asymptotes (MMA) [16] handles inequality constraints as well as bounds for the design variables. The Globally Convergent variant of MMA (GCMMA) [18] is also implemented, in the form of a line search method. The mathematical optimisation problem solved in each optimisation cycle *MMA* is solved using an interior point method and the *maxIters, maxLineSearchIters* have the same meaning as in section 2.4.4.10. A number of optional entries exists too, with names and effects similar to the ones presented in [17]. In contrast to other methods, *MMA* does not use a step size and hence *eta* should always be set to 1.

The corresponding dictionary is set up as

```
updateMethod
{
    method                MMA;
    eta                   1;
    c                     100;
    d                     1;
    z                     1;
    alpha0                1;
    zeta                  1;
    maxIters              1000;
    maxLineSearchIters    10;
    asymptoteDecrease     0.7;
    asymptoteIncrease     1.2;
    move                  0.5;
    raa0                  1.e-05;
    variableRho           false;
}
```

```
lineSearch
{
    method      GCMMA;
    maxIters    10;
}
```

topologyOptimisation/monoFluidAero/laminar/1_Inlet_2_Outlet/levelSet/R_10x_NB_01x

■

The rest of the (optional) entries in the *updateMethod* dictionary read

eta 1;

The η value, multiplying the update of the design variables, (see eq. (2.16)), is set here. If *maxInitChange* is set in the *designVariables* dictionary, this entry can be omitted.

lineSearch none;

Line search methods can be used to adjust the *eta* value between successive optimisation cycles. Each type of line search method attempts to find an *eta* value that satisfies a certain kind of conditions (see [7] for more details); if these conditions are not met, the last update is undone and the previously computed **p** direction (see eq. (2.16)) is multiplied with a new *eta* value. Hence, line search methods can be seen as executing an inner "optimisation loop" identifying an appropriate step value within each optimisation cycle. Their use is optional; if none is provided (i.e. the *lineSearch* dictionary is missing), the initial *eta* value will be maintained for all optimisation cycles. Additional entries in the *lineSearch* dictionary read

```
lineSearch
{
    type        ArmijoConditions;
    minStep     0.3;
    maxIters    4;
    c1          1.e-04;
    ratio       0.7;
}
```

type (ArmijoConditions, GCMMA, none)
The *type* entry defines what kind of condition should be met for accepting the current *eta* value. According to the *ArmijoConditions*, the following inequality should hold in order to accept the current *eta* value

$$\phi(\mathbf{b}+\eta\mathbf{s}) \le \phi(\mathbf{b}) + c_1\eta D(\phi(\mathbf{b});\mathbf{s}), \tag{2.19}$$

where $D(\phi(\mathbf{b});\mathbf{s})$ is the directional derivative of ϕ w.r.t. $\mathbf{b}$ in the direction of $\mathbf{s}$ and ϕ is the l_1 merit function defined as [7]

$$\phi = J + \mu\sum_{i=1}^{M}|e_i| \tag{2.20}$$
$$\mu = max(|\lambda_i|), i \in [1, M]$$

In eq. (2.19), the default value of c_1 is 10^{-4} as suggested in [7] for quasi-Newton approaches computing $\mathbf{s}$. The initial step is tested and if eq. (2.19) is not satisfied, it is successively reduced by a factor of *ratio* for a maximum of *maxIters* times and the primal equations are solved anew. In eq. (2.20), λ_i are the Lagrange multipliers in case *SQP* is used as the *updateMethod* and $e_i, i \in [1, M]$ are the M constraint values; if no constraint is present, $\mu = 0$. *minStep* prevents the *eta* value from being reduced below a certain threshold. Assigning *type* to *none* is the equivalent of excluding the *lineSearch* dictionary altogether.

shapeOptimisation/motorBike

On the other hand, *GCMMA*, as described in [18], is a method to adapt the update of the design variables when using *MMA* as the *updateMethod*. In contrast to other line search methods which maintain the direction and change the step size, *GCMMA* may change the update direction to satisfy criteria related to sufficient reduction of the objective and constraint functions. Nevertheless, the notion of rolling back an update, modifying it and checking for some sufficient reduction criteria fits with the conception of line search methods, justifying the implementation of GCMMA as such. No entry apart from *maxIters* is utilised by *GCMMA*.

Chapter 3

fvOptions

In topology optimisation, source terms of the form $\beta_{max} I^{\Phi}\left(\beta\right)\Phi$, where β_{max} is a dimensioned scalar quantity discussed in section 2.4.2.1, Φ is the unknown field of the PDE and β is the fluid-solid indicator field, are introduced in the primal and adjoint equations, to drive the flow/adjoint solution to zero in the solidified areas of the computational domain. These source terms are introduced through the *fvOptions* dictionary, residing in the *system* directory. Such source terms should be added to the primal momentum equations, the turbulence model PDEs and the eikonal equation for computing distances, as well as their adjoint counterparts. An example of the corresponding dictionary follows:

```
sources
{
    type                topOSource;
    names               (U nuTilda yWall Ua nuaTilda da);
    function            BorrvallPetersson;
    b                   10;
    interpolationField  beta;
}
```

type: topOSource
is used to introduce this kind of source terms in topology optimisation problems.

names
expects a list with the names of the fields corresponding to the equations in which the porosity-based source terms will be added. It should be reminded that if more that one primal or non-null adjoint solvers exist, the name of their corresponding fields will be appended with the solver name (see also sections 2.2.1 and 2.3.1.1), requiring an adjustment of the *names* list. Such an example is given in

```
topologyOptimisation/monoFluidAero/turbulent/1_Inlet_2_Outlet/porosityBased/
BP/losses-massConstr
```

The *function* entry is responsible for choosing function I^{Φ} mentioned in the first line of this chapter. The available options are:

BorrvallPetersson

$$I^{\Phi}(\beta) = \frac{\beta}{1 + b\left(1 - \beta\right)} \tag{3.1}$$

with b controlling the steepness of the function w.r.t. β; this is defined by the entry *b*. A similar entry exists in most of the following functions as well. This *function* is usually employed when we want to prevent small β values from introducing considerable source terms to the flow equations (for instance, when choosing a high β_{max} value).

exp

$$I^{\Phi}\left(\beta\right) = e^{-b(1-\beta)} - \left(1 - \beta\right) e^{-b} \tag{3.2}$$

linear

$$I^{\Phi}\left(\beta\right) = \beta \tag{3.3}$$

SIMP

$$I^{\Phi}\left(\beta\right) = \beta^{b} \tag{3.4}$$

sinh

$$I^{\Phi}\left(\beta\right) = 1 - \frac{sinh\left[b\left(1 - \beta\right)\right]}{sinh(b)} \tag{3.5}$$

tanh The expression of this function is given by eq. (A.5). The *eta* entry corresponds to η in eq. (A.5) and is defined in the *tanh* dictionary as

```
eta     0.5;
```

```
    interpolationField      (beta);
```

is used to define the argument of the penalisation function I^{Φ}. For the moment, “beta” is the only available option.

Chapter 4

fvSolution

Additional entries in the *solvers* and *relaxationFactors* subDicts of *fvSolution* need to be provided for each adjoint-related quantity that is computed through the solution of a PDE. In general, the same linear solver used to solve the discretized primal PDE is also used for its adjoint counterpart. In case multi-point runs are conducted, wildcards can be used to avoid repetition. Regarding the *relaxationFactors*, in industrial cases, the typical setup of the primal mean flow quantities (*p 0.3*; *U 0.7*;) is reversed for the adjoint problem (*pa 0.7*; *Ua 0.3*;). In addition, 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*. No relaxation is required for solving the adjoint to the grid displacement PDE for *ma*.

sensitivityMaps/naca0012/turbulent/liftFullSetup

Chapter 5

fvSchemes

Additional entries need to be provided in all subDicts of *fvSchemes* in order to solve the adjoint PDEs. Indicative entries with some comments follow

```
gradSchemes
{
    gradUATC        cellLimited Gauss linear 1;
    gradUaATC       cellLimited Gauss linear 1;
}
```

The *gradSchemes* entries above are set to define the discretization of the grad terms involved in the computation of the ATC term, section 2.3.1.1.2. A *cellLimited* scheme is usually applied in industrial cases whereas a non-limited scheme can be applied in simpler cases.

```
divSchemes
{
    div(-phi,Ua)        bounded Gauss linearUpwind gradUaConv;
    div(-phi,nuaTilda)  bounded Gauss linearUpwind gradNuaTildaConv;
    div(-phi,ka)        bounded Gauss linearUpwind gradKaConv;
    div(-phi,wa)        bounded Gauss linearUpwind gradWaConv;
    div(-yPhi,da)               Gauss linearUpwind gradDaConv;
}
```

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; *div(-yPhi,da)* should be used for the adjoint distance convection term. A first-order scheme (i.e. *Gauss upwind*) might be needed to ensure convergence in challenging industrial cases.

```
laplacianSchemes
{
    default      Gauss linear limited 0.333;
}
```

The default discretization scheme usually suffices for the discretization of the adjoint diffusion term as well as various auxiliary PDEs including a Laplace operator, such as the adjoint to the grid displacement PDE.

sensitivityMaps/naca0012/turbulent/liftFullSetup

Note: In case the *useSolverNameForFields* switch is set to true in either the primal, section 2.2.1, or adjoint, section 2.3.1.1, setup, the field names in the entries of *fvSchemes* should be adapted accordingly in order to use the desired discretization schemes. Special attention should be paid to the *divSchemes*.

sensitivityMaps/sbend/turbulent/lowRe/multiPoint

In addition, if *average* is set to *true* in the *primalSolver* dict (section 2.2.1.1.1) and averaging iterations have been performed for the primal, the adjoint equations that follow will be solved using the mean primal fields. This should be taken into consideration when defining the discretization schemes for the adjoint equations. For instance, *div(-phiMean,Ua)* should be used instead of *div(-phi,Ua)*.

sensitivityMaps/motorBike

Wildcards can be used to cover both above-mentioned cases, as follows

```
divSchemes
{
    "div\(-phi.*,Ua.*\)"  bounded Gauss linearUpwind gradUaConv;
}
```

Chapter 6

adjointRASProperties

```
    adjointRASModel     adjointSpalartAllmaras;
    adjointTurbulence  on;
```

The *adjointRASProperties* dictionary is located in *constant* and is used to define the adjoint turbulence model to be used. Its entries are

adjointRASModel (adjointLaminar, adjointSpalartAllmaras, adjointkOmegaSST)
Type of the adjoint turbulence model. *adjointLaminar* is used either when solving the adjoint to laminar flows or when the "frozen turbulence" assumption is made. No extra PDEs are solved when using this option. The *adjointSpalartAllmaras* and *adjointkOmegaSST* options solve the PDEs of the adjoint to the Spalart Allmaras, [10, 19], and $k-\omega$ SST, [3], turbulence models. Boundary conditions, solvers, relaxation factors and discretization schemes should be set for *nuaTilda* and *ka, wa*, respectively. Details for each of the above are given in chapters 4, 5 and 7.

adjointTurbulence (on|off)
Whether or not to solve the adjoint to the turbulence model PDEs.

sensitivityMaps/naca0012/turbulent/liftFullSetup

6.1 adjointSpalartAllmaras

An optional dictionary can be provided for the *adjointSpalartAllmaras* model. Its entries follow a similar pattern to the ones in *ATCModel*, section 2.3.1.1.2, for smoothing out numerically challenging terms.

```
adjointSpalartAllmarasCoeffs
```

```
{
    nSmooth           0;
    zeroATCPatchTypes (wall patch);
    maskType          pointCells;
}
```

sensitivityMaps/motorBike

6.2 adjointkOmegaSST

No additional options are available for this model.

shapeOptimisation/sbend/turbulent/kOmegaSST/opt

Chapter 7

Adjoint boundary conditions

Files defining the adjoint boundary conditions (BCs) should be provided at the *start-Time* folder. The type of adjoint BCs to be applied in each patch depends on the type of primal BCs used there. In the sections that follow, general guidelines are provided for the definitions of BCs for the adjoint velocity (Ua), adjoint pressure (pa), adjoint turbulence model (*nuaTilda*), adjoint distance (da) and adjoint grid displacement (ma) fields. Boundary conditions for the latter two are set by the solver automatically and are mentioned here in the sake of completeness.

For constrained patches (i.e. slip, symmetry, symmetryPlane, cyclic, etc), the same BC types imposed on the primal fields should also be applied to their adjoint counterparts.

7.1 *Ua* boundary conditions

- *adjointInletVelocity*: Inlet boundaries where a *fixedValue* BC is imposed on U and a *zeroGradient* BC is used for p.

- *adjointOutletVelocity*: Outlet boundaries where a *zeroGradient* BC is imposed on U and a *fixedValue* BC is used for p.

- *adjointOutletVelocityFlux*: Same as *adjointOutletVelocity* but for cases in which back-flow is observed for U at the outlet.

- *adjointWallVelocity*: Wall boundaries where a *fixedValue* BC is imposed on U and a *zeroGradient* BC is used for p. If *nutUSpaldingWallFunction* is imposed on nut (high-Re turbulence models), the boundary condition will automatically apply the adjoint wall function technique, [10]. Otherwise, a typical low-Re boundary condition will be applied, [10].

- *adjointWallVelocityLowRe*: Same as *adjointWallVelocity* but only for low-Re or laminar flows.
- *adjointRotatingWallVelocity*: Same as *adjointWallVelocity* but also provides the contributions to the sensitivity derivatives due to the change in the boundary face positions, in case *rotatingWallVelocity* is used for the primal run.
- *adjointFarFieldVelocity*: Far-field boundaries where an *inletOutlet* or *freestream* BC is imposed on U.

7.2 *pa* boundary conditions

- *zeroGradient*: Inlet and wall boundaries where a *fixedValue* BC has been imposed on U and a *zeroGradient* BC has been used for p.
- *adjointOutletPressure*: Outlet boundaries where a *zeroGradient* BC has been imposed on U and a *fixedValue* BC has been used for p.
- *adjointFarFieldPressure*: Far-field boundaries where an *outletinlet* BC is imposed on p.

7.3 *nuaTilda* boundary conditions

- *adjointInletNuaTilda*: Inlet boundaries where a *fixedValue* BC is imposed on *nuTilda*.
- *adjointOutletNuaTilda*: Outlet boundaries where a *zeroGradient* BC is imposed on *nuTilda*.
- *adjointOutletNuaTildaFlux*: Same as *adjointOutletNuaTilda*, but for cases in which back-flow is observed for U at the outlet.
- *fixedValue*: Wall boundaries, with or without wall functions.
- *adjointFarFieldNuaTilda*: Far-field boundaries where an *inletOutlet* or *freestream* BC is imposed on *nuTilda*.

7.4 *ka* boundary conditions

- *adjointZeroInlet*: Inlet boundaries where a *fixedValue* BC is imposed on *k*.
- *adjointOutletKa*: Outlet boundaries where a *zeroGradient* BC is imposed on *k*.

- *adjointOutletFlux*: Same as *adjointOutletKa*, but for cases in which back-flow is observed for *U* at the outlet.
- *kaqRWallFunction*: Wall boundaries, where a *kqRWallFunction* BC is imposed on *k*.
- *adjointFarFieldTMVar1*: Far-field boundaries where an *inletOutlet* or *freestream* BC is imposed on *k*.

7.5 *wa* boundary conditions

- *adjointZeroInlet*: Inlet boundaries where a *fixedValue* BC is imposed on *omega*.
- *adjointOutletWa*: Outlet boundaries where a *zeroGradient* BC is imposed on *omega*.
- *adjointOutletFlux*: Same as *adjointOutletWa*, but for cases in which back-flow is observed for *U* at the outlet.
- *waWallFunction*: Wall boundaries, where an *omegaWallFunction* BC is imposed on *omega*.
- *adjointFarFieldTMVar2*: Far-field boundaries where an *inletOutlet* or *freestream* BC is imposed on *omega*.

7.6 *ma* boundary conditions

- *fixedValue uniform 0*: All boundaries with a non-constrained type.

7.7 *da* boundary conditions

- *fixedValue uniform 0*: All inlet and outlet boundaries.
- *zeroGradient* All wall boundaries.

Chapter 8

dynamicMeshDict

The setup of the volumetric B-splines morpher is explained in detailed in section 8.1. Setting up a Laplace-based grid displacement PDE for use in conjunction with $2D$ Bézier–Bernstein parameterizations in presented in section 8.2.

8.1 Volumetric B-splines Morpher

8.1.1 Control points and B-splines properties

Before executing the actual optimisation loop, some pre-processing steps have to be undertaken in order to define the control points, the coordinates of which will act as the design variables of the optimisation problem. The mathematical background of the volumetric B-splines morpher is presented in detail in [11] and some basic definitions are repeated herein in the sake of completeness.

Let $b_m^{ijk}, m \in [1,3], i \in [0,I], j \in [0,J], k \in [0,K]$ be the Cartesian coordinates of the ijk-th control point of the $3D$ structured control grid, fig. 8.1. I, J and K are the number of control points (minus 1) per control grid direction. The Cartesian coordinates $\mathbf{x} = [x_1, x_2, x_3]^T = [x, y, z]^T$ of a CFD mesh point residing within the boundaries defined by the control grid are given by

$$x_m(u,v,w) = \sum_{i=0}^{I}\sum_{j=0}^{J}\sum_{k=0}^{K} U_{i,pu}(u)V_{j,pv}(v)W_{k,pw}(w)b_m^{ijk} \tag{8.1}$$

Here, $\mathbf{u} = [u_1, u_2, u_3]^T = [u, v, w]^T$ are the mesh point parametric coordinates, U, V, W are the B-splines basis functions and pu, pv, pw their respective degrees, which may be different per control grid direction.

Details about B-splines basis definitions and properties can be found in [12]. Computing the Cartesian coordinates of any parameterized mesh point is straightforward, at a negligible computational cost, as long as its parametric coordinates $\mathbf{u}$ are known.

Mesh parametric coordinates can be computed with accuracy, since a mapping from $\Re^3(x,y,z) \rightarrow \Re^3(u,v,w)$ is required. This means that volumetric B-splines can reproduce any geometry to machine accuracy.

Given the control points position, the knot vectors and the basis functions degrees, the parametric coordinates (u,v,w) of a point with Cartesian coordinates $\mathbf{r} = [x_r, y_r, z_r]^T$ can be computed by solving the system of equations

$$\mathbf{R}(u,v,w) = \begin{bmatrix} x(u,v,w) - x_r = 0 \\ y(u,v,w) - y_r = 0 \\ z(u,v,w) - z_r = 0 \end{bmatrix} \tag{8.2}$$

where $x_m(u,v,w)$ are computed through eq. (8.1), based on the known **b** values. The 3×3 system of eq. (8.2) can be solved independently for each parameterized mesh point using the Newton-Raphson method, after computing and inverting the Jacobian $\partial x_m / \partial u_j, m, j \in [1,3]$. Since the evaluation of the parametric coordinates of each point is independent from any other mesh point, these computations may run efficiently in parallel.

The aforementioned process has to be done only once and can be seen as the "training phase" of the method. Then, after moving the control points **b**, the Cartesian coordinates of each (internal or boundary) mesh point residing within the control grid can be computed through eq. (8.1) at a very low cost, making volumetric B-splines a powerful surface parameterization and mesh displacement tool.

The parameters for the volumetric B-splines morpher are defined in the *constant/dynamicMeshDict* dictionary. A sample of the latter (excluding the header) is given below, with some comments on its entries

```
solver volumetricBSplinesMotionSolver;
volumetricBSplinesMotionSolverCoeffs
{
    duct
    {
        type      cartesian;
        nCPsU     9;
        nCPsV     5;
        nCPsW     3;
        degreeU   3;
        degreeV   3;
        degreeW   2;

        controlPointsDefinition  axisAligned;
        lowerCpBounds            (-1.1  -0.21  -0.05);
        upperCpBounds            ( 1.1   0.39   0.15);
```

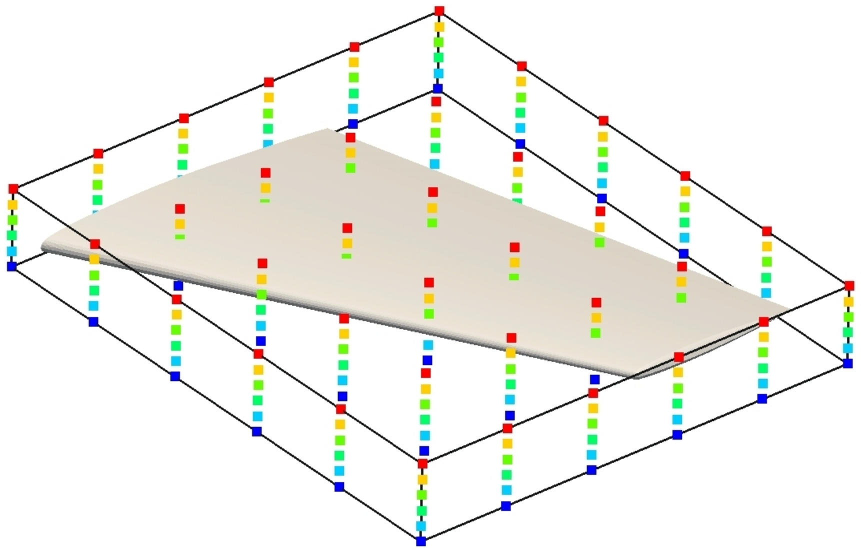

Figure 8.1: Control grid consisting of 6×6×6 control points, around a 3D wing. Control points are coloured based their j index value. Surface and volume mesh points, over and around the wing, residing within the boundaries of the control grid (black box) will be displaced following a possible displacement of the control points.

```
        confineUMovement  false;
        confineVMovement  false;
        confineWMovement  true;
        confineBoundaryControlPoints  false;

        confineUMinCPs ( (true true true) (true true true) );
        confineUMaxCPs ( (true true true) (true true true) );
        confineVMinCPs ( (true true true) );
        confineVMaxCPs ( (true true true) );
        confineWMinCPs ( (true true true) );
        confineWMaxCPs ( (true true true) );
    }
}
```

One morphing box, similar to that presented in fig. 8.1, will be created for each subDict within *volumetricBSplinesMotionSolverCoeffs*. More than one control boxes are supported, as long as they are not overlapping.

8.1.1.1 Entries in each dict within *volumetricBSplinesMotionSolverCoeffs*

type (cartesian, cylindrical)
Coordinate system in which the control points are defined.

controlPointsDefinition (axisAligned, fromFile)
Control points can be defined in two different ways. If the *axisAligned* option is chosen, a control grid aligned with the coordinates system defined by *type* will be constructed, by giving the coordinates of the two far points of the box in *lowerCpBounds* and *upperCpBounds*, similar, for instance, to the way the *boxToCell* option is used in *topoSetDict*. If the *fromFile* option is chosen, control points are read from the *constant/controlPoints/"name""timeName"* file, where *name* is the name of the morphing box (i.e. the name of the current subDict within *volumetricBSplinesMotionSolverCoeffs*) and *timeName* is the current time index (0 if the optimisation is starting from scratch). The above-mentioned file, apart from the typical OpenFOAM header, should include a *vectorList* of the following format

```
controlPoints    27
(
( 0.133 -0.255 0.699 )
( 0.2015 -0.255 0.699 )
( 0.27 -0.255 0.699 )
( 0.133 0 0.699 )
( 0.2015 0 0.699 )
( 0.27 0 0.699 )
( 0.133 0.255 0.699 )
( 0.2015 0.255 0.699 )
( 0.27 0.255 0.699 )
( 0.133 -0.255 0.789 )
( 0.2015 -0.255 0.789 )
( 0.27 -0.255 0.789 )
( 0.133 0 0.789 )
( 0.2015 0 0.789 )
( 0.27 0 0.789 )
( 0.133 0.255 0.789 )
( 0.2015 0.255 0.789 )
( 0.27 0.255 0.789 )
( 0.133 -0.255 0.879 )
( 0.2015 -0.255 0.879 )
( 0.27 -0.255 0.879 )
( 0.133 0 0.879 )
( 0.2015 0 0.879 )
( 0.27 0 0.879 )
( 0.133 0.255 0.879 )
( 0.2015 0.255 0.879 )
( 0.27 0.255 0.879 )
);
```

i.e., a *vectorList* named *controlPoints*, followed by the control points number and the actual list of points. It should be noted that the order in which the points are written is important and should be given by an iso-z, iso-y, iso-x loop (or an iso-W, iso-V, iso-U loop, in the most general case).

nCPsU, nCPsV, nCPsW
Number of control points in the parametric directions, i.e. $I+1$, $J+1$ and $K+1$ in eq. 8.1.

degreeU, degreeV, degreeW
Basis function degrees in the three parametric coordinates (i.e. pu, pv and pw in eq. 8.1). Regarding the choice of the basis functions degrees, a smaller polynomial degree will lead to more localized (and less smooth) geometry changes. The maximum degree per direction is $nCPs-1$, which will lead to a parameterization in which all control points affect all CFD grid points inside the parameterized domain. The suggested basis function degree is 3 since it gives the highly desirable property of local support while at the same time maintains smoothness.

confineUMovement, confineVMovement, confineWMovement (true|false)
Whether to confine or not the movement of all control points in each of the directions of the coordinate system defined by *type*. The corresponding entries in v1912 were named *confineX1movement, confineX2movement, confineX3movement* and are still supported.

confineBoundaryControlPoints (true|false)
When the control box separates the mesh in parameterized and non-parameterized regions, the boundary control points of the control grid have to be fixed in order to ensure C0 continuity at the interface of the two regions. This will ensure that mesh elements will not overlap in the boundaries of the control grid, however, gradient and curvature continuity might not be guaranteed. If these are of importance, the following lists should be set accordingly:

confineUMinCPs, confineUMaxCPs, confineVMinCPs, confineVMaxCPs, confineWMinCPs, confineWMaxCPs (empty lists)
More layers of control points can be kept fixed during the optimisation. The number of control points to be kept fixed in each of the (U, V, W) control grid directions (at the be-

ginning -Min- and end -Max- of the control grid) can be controlled by the *confine*CPs* variables. Each of these entries is a *boolListList*, i.e. a list of lists, containing three *bools* each. A typical example of such an entry reads

```
    confineUMinCPs ( (true true true) (true true true) );
```

In this example, the movement of the first two columns of control points (corresponding to the two triplets of booleans) in the beginning (confineU**Min**CPs) of the U direction (confine**U**MinCPs) of the control grid is constrained in all three spatial directions (each one corresponding to one of the three booleans in each triplet). All *confine*CPs* entries are initialized to empty lists, meaning that no control points will be kept fixed if the entries are not provided. The corresponding entries in v1912 were named *bound*CPs* and are still supported.

```
shapeOptimisation/sbend/laminar/opt/unconstrained/losses/BFGS
```

readStoredData (true|false)
If *readStoredData* is set to true, the code will attempt to read the parametric coordinates from a file named *parametricCoordinate+name*, where *name* is the name of the current subDict within *volumetricBSplinesMotionSolverCoeffs*, if the file exists. Otherwise, the parametric coordinates will be computed anew. For more information, see section 8.1.1.2.

maxIterations (default=10)
Maximum number of Newton-Raphson iterations to be executed for each CFD grid point inside the control box in order to compute its parametric coordinates.

tolerance (default=1.e-10)
Convergence criterion for the Newton-Raphson procedure executed to compute CFD grid point parametric coordinates.

8.1.1.2 Computing parametric coordinates

The parametric coordinates for the points residing within the control boxes are computed by solving a 3×3 system for each point, eq. (8.2). This has to be done only once, as a pre-processing step of the optimisation loop. This will be done automatically by the executable driving the optimisation or the one that computes sensitivity derivatives. Since this step can be potentially expensive for large CFD meshes and a high number of control points, the parametric coordinates are stored as a *pointVectorField* in the 0 folder, named *parametricCoordinates"name"*, where *name* is the control box name (i.e. name of the corresponding subDict within *volumetricBSplinesMotionSolverCoeffs*). If the entry *readStoredData* is set to *true*, the code will attempt to read a stored parametric coordinates file. If the file is not present, parametric coordinates will be computed from scratch.

Since the parametric coordinates depend on the control points positions and the degrees of the basis functions, the *parametricCoordinates** files have to either be removed manually from the starting time-step folder, or the *readStoredData* entry has to be set to *false* each time the control points setup is altered.

8.1.1.3 Visualizing the control points

It is often useful to visualize the control points comprising the control grid before running the optimisation loop. This can be done by running the *writeMorpherCPs* application, see section 9.2.1.1. *writeMorpherCPs* reads *dynamicMeshDict* and produces the *optimisation/controlPoints/"name""TimeName".csv* file, where *name* is the name of control box and *TimeName* the current time index. The file contains 9 columns, including the (x, y, z) coordinates of the control points, their (i, j, k) values in the structured control grid and 3 *active* flags indicating whether each control point is allowed to move in each of the 3 directions defined by *type* (see section 8.1.1.1) during the optimisation. The *csv* file can be visualized in Paraview following the steps listed below:

- Open the file from the File/Open menu and click Apply
- From the Filters/Alphabetical menu, choose TableToPoints
- Select *Points:0, Points:1, Points:2* for the *X,Y and Z* Column, respectively, and click Apply.
- Adjust the size and colouring of the points according to your preference.

Apart from generating a *csv* file before running the optimisation loop, similar files are generated and stored in the same folder for visualization purposes each time a new geometry is created during the optimisation process.

shapeOptimisation/sbend/laminar/opt/unconstrained/losses/BFGS

8.2 Laplace-based Grid Displacement Equation

In case an automatic optimisation loop is targeted for $(2D)$ geometries that have been parameterized using Bézier–Bernstein curves, the boundary movement has to be propagated to the interior mesh. This can be done using a Laplace-based PDE and the mesh movement solvers already existing in OpenFOAM. In particular, the *velocityLaplacian* mesh motion solver is proposed, set up as

```
dynamicFvMesh dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver velocityLaplacian;

velocityLaplacianCoeffs
{
    diffusivity uniform;
}
```

It should be noted that the *velocityLaplacian* motion solver can cope with mesh movement in relatively simple geometries but usually faces difficulties in complicated boundary movements or fine meshes used for low-Re simulations. Using a *diffusivity* option that is based on the inverse distance from the moved boundaries usually improves the results. A *pointVectorField* named *pointMotionU* should be supplied, with zero *fixedValue* conditions for all patches expect the coupled ones. The appropriate boundary conditions for the movement in each optimisation cycle will be set automatically by the code. A *cellMotionU* entry should also be set in *fvSolution.solvers* while the default entry in *system.fvSchemes.laplacianSchemes* should suffice for the discretization of the PDE.

Chapter 9

Applications

9.1 solvers

9.1.1 adjointOptimisationFoam

adjointOptimisationFoam is the main executable. It can be used either to just solve the primal and adjoint equations and compute sensitivities or to execute an automated shape optimisation loop. Its behaviour largely depends on the setup of the *optimisationDict*, as described in chapter 2. In the sections that follow, the differences between the two above-mentioned modes of operation are briefly discussed.

9.1.1.1 Solution of the primal and adjoint equations and computation of sensitivities

In this mode, *adjointOptimisationFoam* functions in a way similar to *simpleFoam*, with the ability to also solve the adjoint equations. It should be noted that the *endTime* entry in *controlDict* will be ignored and the *nIters* entry in *optimisationDict* (sections 2.2.1.1 and 2.3.1.1.3), for each primal and adjoint solver, will be used to define the number of iterations to be executed and the *endTime*. All primal solvers for which the *active* keyword is set to *true* will be executed, followed by the adjoint ones and, finally, the computation of sensitivity derivatives for all adjoint solvers for which the *computeSensitivities* flag is *true* (section 2.3.1.1). Equations for all active primal and adjoint solvers will be iterated either until the residual values declared in *residualControl* (sections 2.2.1.1 and 2.3.1.1.3)) have been achieved or the *nIters* value has been reached. Upon stopping, each solver will write results to the hard drive, with writing also performed based on the *writeInterval* defined in *controlDict*. During the solution of the primal equations, if the *active* keyword of the *adjointSolvers* is set to true, the objective values defined in those *adjointSolvers* will be evaluated during each iteration of the primal solver and their values will be written in *optimisation/objective/timeName/objectiveName+Instant+adjointSolverName*.

```
sensitivityMaps/naca0012/turbulent/liftMinimumSetup
```

9.1.1.2 Automated shape optimisation loops

In this mode, *adjointOptimisationFoam* undertakes the execution of an automated shape optimisation loop (i.e. solve the primal and adjoint equations, compute sensitivity derivatives, update the design variables for *n* optimisation cycles), fig. 2.1. In order to do so, the *optimisationManager* entry in *optimisationDict* should be set to *steadyOptimisation*. The *endTime* in *controlDict* now stands for the number of optimisation cycles to be conducted, while the *writeInterval* entry defines the optimisation cycles interval in which (primal and adjoint) flow results will be stored to the hard drive. It is recommended to set *purgeWrite 0;* and *writeInterval 1;* in *controlDict* in order to store results from all the geometries analyzed during the optimisation loop. The objective functions convergence is written in the *optimisation/objective/timeName/objectiveName+adjointSolverName* whereas the convergence of objective function values within the iterations of the primal solver for all optimisation cycles is stored in *optimisation/objective/timeName/objectiveName+Instant+adjointSolverName*.

```
shapeOptimisation/sbend/laminar/opt/unconstrained/losses/BFGS
```

9.2 utilities

A number of pre- and post-processing utilities related to adjoint-based optimisation exist. These are briefly analyzed in what follows.

9.2.1 preProcessing

9.2.1.1 writeMorpherCPs

The *writeMorpherCPs* utility is used to output the volumetric B-splines morpher control points, as defined by the current setup in *dynamicMeshDict* in a form that is convenient for visualization, section 8.1.1.3. It should be noted that only the control point positions are written, without computing the parametric coordinates of CFD grid points residing within the control points box.

9.2.2 postProcessing

9.2.2.1 computeSensitivities

The *computeSensitivities* utility is used to compute sensitivity derivatives at a post-processing step, for a simulation in which the primal and adjoint fields have already been computed but, for instance, *computeSensitivities* was set to *false*. The appropriate dictionary entries must be defined, as discussed in section 2.4.3. Remember to also set the *computeSensitivities* to *true* in the adjoint solver dicts, section 2.3.1.

9.2.2.2 cumulativeDisplacement

This utility is used to compute and write the displacement of all mesh points for each geometry generated by an optimisation loop, from the initial geometry. The vectorial difference of all mesh points ($x_i^{new} - x_i^{old}$) is written in a *pointVectorField* named *displacement* whereas the projection of this difference to the normal vector of the boundary mesh points in the initial geometry ($(x_i^{new}-x_i^{old})n_i^{old}$) is written in a *pointScalarField* named *normalDisplacement*. Keeping in mind the convention for the surface normal unit vector, facing from the fluid to the solid boundaries, positive normal displacements indicate a movement aligned to the geometry normal ("inwards" or "outwards", for external or internal aerodynamics, respectively); negative normal displacements indicate a movement opposite to the geometry normal ("outwards" or "inwards" for external or internal aerodynamics, respectively).

shapeOptimisation/sbend/laminar/opt/unconstrained/losses/BFGS

Appendix A

A Short Introduction to Topology Optimisation

This appendix briefly explains the concept of topology optimisation using the density (porosity)-based approach and the source terms that are introduced in the flow equations. It is not intended as an exhaustive description of the method but rather as a quick guide, to facilitate matching of the *optimisationDict* and *fvOptions* entries with their counterparts in the design variables, the flow and adjoint equations.

In topology optimisation, the value-field of design variables α ("porosity" field) is used to compute the so-called Brinkman penalisation terms that solidify (i.e. drive the solution of the flow equation to zero) the part of the design domain that is counterproductive with respect to (w.r.t.) the objective function J to be minimised. The α field could be directly used to compute the Brinkman penalisation terms, however, this could lead to noisy geometries and a dependency of the optimised solution on the mesh resolution. Instead, a fluid-solid identification field, β, is used to compute the penalisation terms; details on computing the β field based on α are given towards the end of this appendix.

Parts of the computational domain with an (almost) unitary β value indicate the solid area whereas (almost) zero β values correspond to the fluid part of the domain. The interface between the two regions stands for the solid walls of the sought duct system. To simulate the solidification of parts of the domain, flow equations are augmented with β-dependent source terms, whose role is to deactivate the flow equations over the solid. The so-modified flow equations, coupled with the Spalart-Allmaras [15] one-equation turbulence model PDE, the Eikonal equation which computes distances

Δ from the solid walls for steady flows of incompressible fluids read:

$$R^p = -\frac{\partial v_j}{\partial x_j} = 0 \tag{A.1a}$$

$$R_i^v = v_j \frac{\partial v_i}{\partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j} + \frac{\partial p}{\partial x_i} + \beta_{max} I^v(\beta) v_i = 0\,, \quad i = 1,2(,3) \tag{A.1b}$$

$$R^{\widetilde{\nu}} = v_j \frac{\partial \widetilde{\nu}}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\widetilde{\nu}}{\sigma}\right)\frac{\partial \widetilde{\nu}}{\partial x_j}\right] - \frac{c_{b2}}{\sigma}\left(\frac{\partial \widetilde{\nu}}{\partial x_j}\right)^2 - \widetilde{\nu} P(\widetilde{\nu}) + \widetilde{\nu} D(\widetilde{\nu}) + \beta_{max} I^{\widetilde{\nu}}(\beta)\widetilde{\nu} = 0 \tag{A.1c}$$

$$R^{\Delta} = \frac{\partial}{\partial x_j}\left(\frac{\partial \Delta}{\partial x_j}\Delta\right) - \Delta \frac{\partial^2 \Delta}{\partial x_j^2} - 1 + \beta_{max} I^{\Delta}(\beta)\Delta = 0 \tag{A.1d}$$

where v_i are the velocity components, p is the pressure divided by the fluid density, $\tau_{ij} = (\nu + \nu_t)\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)$ is the stress tensor, ν and ν_t are the bulk and eddy viscosities respectively and $P(\widetilde{\nu})$ and $D(\widetilde{\nu})$ are the production and dissipation terms of the turbulence model, respectively, [15]. Furthermore, the $I^v, I^{\widetilde{\nu}}, I^{\Delta}$ functions are used to drive the flow solution towards values corresponding to solid walls (see chapter 3 for the available options of the I functions and how to define them). The β_{max} value is used to ensure that the $v_i, \widetilde{\nu}$ and Δ values are practically zero in the solidified domain. Its value can be computed based on either the Darcy number, quantifying the ratio of viscous and porous forces, [8],

$$Da = \frac{\nu}{\beta_{max} L^2} \Rightarrow \beta_{max} = \frac{\nu}{Da L^2} \tag{A.2}$$

or the product of the Reynolds and Darcy numbers, quantifying the ratio of momentum and viscous forces

$$ReDa = \frac{U}{\beta_{max} L} \Rightarrow \beta_{max} = \frac{U}{ReDa L} \tag{A.3}$$

where L and U are a characteristic length and velocity magnitude of the case under consideration, respectively. More on the computation of β_{max} can be found in section 2.4.2.1. The porosity-dependent terms added to eqs. (A.1b) to A.1d are implemented as *fvOptions*, to allow for code modularity and adaptability, see chapter 3.

Often, in topology optimisation problems, checkerboard artifacts may appear in the α field, in the course of the optimisation. To avoid these artifacts and mitigate the effects of local grid size to the optimised solution, the regularisation of the porosity field [5] is implemented. This is based on a Helmholtz-type filter [5] applied to the porosity field α, namely

$$-\left(\frac{R}{2\sqrt{3}}\right)^2 \frac{\partial^2 \widetilde{\alpha}}{\partial x_j^2} + \widetilde{\alpha} = \alpha \tag{A.4}$$

where $\widetilde{\alpha}$ is the regularised porosity field and R can be seen as a smoothing/regularisation radius, usually computed as a function of the average grid cell size. Regularisation, as any other smoothing technique, unavoidably blurs the line between the fluid and solidified domains. To increase the contrast of the $\widetilde{\alpha}$ field, a projection step, computing β based on $\widetilde{\alpha}$ is introduced, [5]

$$\beta = \frac{tanh(\eta b) + tanh\left[b(\widetilde{\alpha} - \eta)\right]}{tanh(\eta b) + tanh\left[b(1-\eta)\right]} \tag{A.5}$$

with $\eta = 0.5$ and b being a sharpening parameter (higher values lead to a β field that is close to binary). If no regularisation/projection is applied, $\beta = \alpha$ in eqs. (A.1). Inputs related to regularisation and projection are described in detail in section 2.4.2.1. A comparison of the $\alpha, \widetilde{\alpha}$ and β fields for one of the tutorial cases is given in fig. A.1.

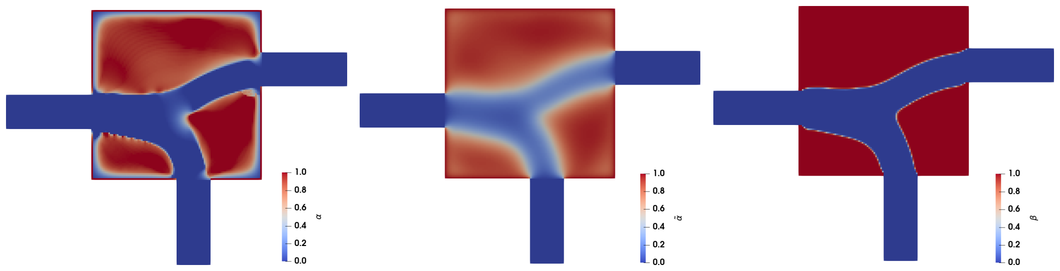

Figure A.1: Example of a topology optimisation problem, with the fluid entering the domain from the left and exiting from bottom and right. From left to right the $\alpha, \widetilde{\alpha}$ and β fields.

Following the mathematical development outlined in [10], the continuous adjoint PDEs to eqs. (A.1) can be derived:

$$R^q = -\frac{\partial u_j}{\partial x_j} = 0 \tag{A.6a}$$

$$R_i^u = \underbrace{u_j \frac{\partial v_j}{\partial x_i}}_{ATC} - \frac{\partial (v_j u_i)}{\partial x_j} - \frac{\partial \tau^{\alpha}_{ij}}{\partial x_j} + \frac{\partial q}{\partial x_i} + \widetilde{\nu_a}\frac{\partial \widetilde{\nu}}{\partial x_i} - \frac{\partial}{\partial x_l}\left(\widetilde{\nu_a}\widetilde{\nu}\frac{C_Y}{Y} e_{mjk}\frac{\partial v_k}{\partial x_j} e_{mli}\right)$$
$$+ \beta_{max} I^u(\beta) u_i = 0 \quad i = 1,2(,3) \tag{A.6b}$$

$$R^{\widetilde{\nu_a}} = -\frac{\partial (v_j \widetilde{\nu_a})}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\widetilde{\nu}}{\sigma}\right)\frac{\partial \widetilde{\nu_a}}{\partial x_j}\right] + \frac{1}{\sigma}\frac{\partial \widetilde{\nu_a}}{\partial x_j}\frac{\partial \widetilde{\nu}}{\partial x_j} + 2\frac{c_{b2}}{\sigma}\frac{\partial}{\partial x_j}\left(\widetilde{\nu_a}\frac{\partial \widetilde{\nu}}{\partial x_j}\right) + \widetilde{\nu_a}\widetilde{\nu}C_{\widetilde{\nu}}$$
$$+ \frac{\partial \nu_t}{\partial \widetilde{\nu}}\frac{\partial u_i}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) + (-P + D)\widetilde{\nu_a} + \beta_{max} I^{\widetilde{\nu}}(\beta)\widetilde{\nu_a} = 0 \tag{A.6c}$$

$$R^{\Delta_\alpha} = -2\frac{\partial}{\partial x_j}\left(\Delta_\alpha \frac{\partial \Delta}{\partial x_j}\right) + \widetilde{\nu}\widetilde{\nu_a}C_\Delta + \beta_{max} I^{\Delta}(\beta)\Delta_\alpha = 0 \tag{A.6d}$$

where u_i are the adjoint velocity components, q the adjoint pressure, $\widetilde{\nu_a}$ the adjoint to the turbulence model variable, $\tau^{\alpha}_{ij} = (\nu + \nu_t)\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$ the adjoint stresses and Δ_α the adjoint distance from the walls. Following the same trend as in eqs. (A.1), the adjoint momentum, turbulence model and eikonal equations include Brinkman penalisation terms too.

Bibliography

[1] F. Feppon, G. Allaire, and C. Dapogny. Null space gradient flows for constrained optimization with applications to shape optimization. *ESAIM: COCV*, 26:90, 2020.

[2] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *Computer Journal*, 7:149–154, 1964.

[3] I.S. Kavvadias, E.M. Papoutsis-Kiachagias, G. Dimitrakopoulos, and K.C. Giannakoglou. The continuous adjoint approach to the k–ω SST turbulence model with applications in shape optimization. *Engineering Optimization*, 47(11):1523–1542, 2015.

[4] I.S. Kavvadias, E.M. Papoutsis-Kiachagias, and K.C. Giannakoglou. On the proper treatment of grid sensitivities in continuous adjoint methods for shape optimization. *Journal of Computational Physics*, 301:1–18, 2015.

[5] B. S. Lazarov and O. Sigmund. Filters in topology optimization based on Helmholtz-type differential equations. *Int. J. Numer. Methods Eng.*, 86:765–781, 2011.

[6] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[7] J. Nocedal and S.J. Wright. *Numerical Optimization.* Springer, New York, 1999.

[8] L. Olesen, F. Okkels, and H. Bruus. A high-level programming-language implementation of topology optimization applied to steady-state Navier-Stokes flow. *International Journal for Numerical Methods in Engineering*, 65:975–1001, 2006.

[9] E.M. Papoutsis-Kiachagias. *Adjoint Methods for Turbulent Flows, Applied to Shape or Topology Optimization and Robust Design.* PhD thesis, National Technical University of Athens, 2014.

[10] E.M. Papoutsis-Kiachagias and K.C. Giannakoglou. Continuous adjoint methods for turbulent flows, applied to shape and topology optimization: Industrial applications. *Archives of Computational Methods in Engineering*, 23(2):255–299, 2016.

[11] E.M. Papoutsis-Kiachagias, N. Magoulas, J. Mueller, C. Othmer, and K.C. Giannakoglou. Noise reduction in car aerodynamics using a surrogate objective function and the continuous adjoint method with wall functions. *Computers & Fluids*, 122:223–232, 2015.

[12] L. Piegl and W. Tiller. *The NURBS book*. Springer, 1997.

[13] J.B. Rosen. The gradient projection method for nonlinear programming. Part I. Linear constraints. *Journal of the Society for Industrial and Applied Mathematics*, 8(1):181–217, 1960.

[14] J. Sherman and W. Morrison. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *The Annals of Mathematical Statistics*, 20:620 – 624, 1949.

[15] P. Spalart and S. Allmaras. A one-equation turbulence model for aerodynamic flows. In *AIAA Paper 1992-0439, 30th Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, 6-9 January 1992.

[16] K. Svanberg. The method of moving asymptotes – a new method for structural optimization. *Int. J. Numer. Methods Eng.*, 24:359–373, 1987.

[17] K. Svanberg. The method of moving asymptotes - modelling ascpects and solution schemes. In *Lecture notes for the DCAMM course*. Advanced Topics in Structural Optimization, Lyngby, Denmark, June 1989.

[18] K. Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM Journal on Optimization*, 12(2):555–573, 2002.

[19] A.S. Zymaris, D.I. Papadimitriou, K.C. Giannakoglou, and C. Othmer. Continuous adjoint approach to the Spalart-Allmaras turbulence model for incompressible flows. *Computers & Fluids*, 38(8):1528–1538, 2009.