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]

June 2019

[1]Prof. K.C. Giannakoglou, Dr. E.M. Papoutsis-Kiachagias, K.T. Gkaragkounis
[2]support: vaggelisp@gmail.com

Contents

1 Introduction **5**

2 optimisationDict **7**
2.1 optimisationManager 7
2.2 primalSolvers 7
2.2.1 Entries within each *primalSolver* sub-dictionary 8
2.2.1.1 solutionControls 9
2.2.1.1.1 averaging 9
2.2.1.2 fvOptions 10
2.3 adjointManagers 10
2.3.1 Entries within each *adjointManager* sub-dictionary 11
2.3.1.1 Entries within each *adjoinSolvers* sub-dictionary 12
2.3.1.1.1 objectives 13
2.3.1.1.2 ATCModel 15
2.3.1.1.3 solutionControls 16
2.4 optimisation 17
2.4.1 sensitivities 17
2.4.1.1 surface 17
2.4.1.2 surfacePoints 20
2.4.1.3 multiple 20

3 fvSolution **23**

4 fvSchemes **25**

5 adjointRASProperties **27**
5.1 adjointSpalartAllmaras 27

6 Adjoint boundary conditions **29**
6.1 *Ua* boundary conditions 29
6.2 *pa* boundary conditions 30
6.3 *nuaTilda* boundary conditions 30

6.4 *ma* boundary conditions 30
6.5 *da* boundary conditions 31

7 Applications **33**
7.1 solvers 33
7.1.1 adjointOptimisationFoam 33
7.1.1.1 Solution of the primal and adjoint equations and computation of sensitivities 33
7.2 utilities 34
7.2.1 computeSensitivities 34

Chapter 1

Introduction

The present User Manual serves as a guide for the setup and usage of the OpenFOAM executable *adjointOptimisationFoam*, included in OpenFOAM-v1906. 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, [1, 2, 3]. 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 3 describes the entries to be added to *fvSolution* while chapter 4 the ones to be added to *fvSchemes*. Chapter 5 describes entries related to the adjoint to turbulence models while chapter 6 provides guidelines for defining the adjoint boundary conditions. Chapter 7 describes the applications (solvers and utilities) used to compute the flow (primal) and adjoint equations and the sensitivity derivatives.

Chapter 2

optimisationDict

optimisationDict is the main dictionary in which almost all information about the solution of the primal and adjoint equations is set up. It is located in *system* and needs to be present for practically all applications presented in section 7 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)
singleRun is used to solve the primal and adjoint equations just once, without performing an optimisation loop. It is the only option in v1906, with additional ones coming up in future releases.

2.2 primalSolvers

```
primalSolvers
{
    p1
    {
        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;
            }
        }
        fvOptions{}
    }
}
```

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. This flag should be set to true 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 ran instead of *adjointOptimisationFoam*.
Note: the equivalent entries in *system/fvSolution/SIMPLE* will be disregarded.

Additional entries include:

nIters
Maximum number of iterations when solving the primal equations.

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.

2.2.1.2 fvOptions

The *fvOptions* dict is optional. Source terms that are generally applied through the *system/fvOptions* dict (e.g. MRFSource, explicitPorosityModel, etc) should be inserted here.

2.3 adjointManagers

```
adjointManagers
{
    am1
    {
        primalSolver              p1;
        operatingPointWeight      1;
        adjointSolvers
        {
            as1
            {
                // choose adjoint solver
                //-------------------------
                active                    true;
                type                      incompressible;
                solver                    adjointSimple;
                useSolverNameForFields    false;
                computeSensitivities      true;
                // 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;
                }
            }
        }
    }
}
}
```

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 Defaults to 1
When having multiple objective functions defined across many operating points, they have to be concatenated into a single one using appropriate weights, i.e. $J = \sum_i w_i^{op} J_i^{op}$, where J is the concatenated objective function summing contributions from all operating points, 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 are going to be solved for this *adjointSolver*.

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

type: (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. Should be set to *true* if more than one *adjointSolvers* are present.

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

2.3.1.1.1 objectives

type: (incompressible)
Type of objective functions to be constructed. Only one option is valid for the moment.

objectiveNames
A list of dictionaries corresponding to the objective functions to be minimized. 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.

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

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, moment, PtLosses)
The type of the objective to be minimized.

weight
Objective function weight (see also eq. 2.1).

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

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 minimized. If the maximization of a force is targeted, use the opposite force direction vector.

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 minimized, $\mathbf{x}$ 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.;
};
```

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 happens before the flow equations are solved, so the flow initialization might affect it.

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

sbend/laminar

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 industrial 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, loosing some 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 this many times, using slightly different discretization 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)
How will the cells next to the *zeroATCPatchTypes* will be chosen for smoothing the ATC term. If *faceCells* is used, every cell having a face in the *zeroATCPatchTypes* boundaries will be chosen whereas if *pointCells* is used, every cell that has a point in the *zeroATCPatchTypes* will be used.

```
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. These can be useful indicators of the simulation stability.

2.4 optimisation

The *optimisation* dict is optional and should be present only when sensitivity derivatives should be computed. Its subDicts follow:

2.4.1 sensitivities

```
sensitivities
{
    type     surfacePoints;
    patches (pressure suction);
    options ...
}
```

The *sensitivities* dict is where the setup for the computation of sensitivity derivatives is provided. Sensitivities will be computed after all adjoint PDEs are solved, for the adjoint solvers for which *computeSensitivities* is set to *true*, section 2.3.1.1. Only two entries are mandatory and based on them, usually additional entries or dictionaries are required. The mandatory entries are

type: (surface, surfacePoints, sensitivityMultiple)

patches
On which patches to compute sensitivities. Wildcards and group names allowed.

2.4.1.1 surface

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. 2.1

A typical setup reads

```
type                                surface;
patches                             ( "wall.*");
```

```
includeSurfaceArea              true;
includeObjectiveContribution    true;
includeMeshMovement             true;
adjointMeshMovementSolver
{
    iters          300;
    tolerance      1.e-7;
}
```

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".

includeObjectiveContribution (true|false)
Certain objectives give the so-called direct contributions to the sensitivities (for instance, changes in the normal surface vector in drag optimisation). This flag determines whether these contributions will be computed or not.

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, [1]. 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 discretization 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 pro-

vided 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 (chapter 5), additional entries may be supplied to the *sensitivities* dict. These read

includeDistance (true|false)
Whether to solve the adjoint to the eikonal equation or not, [2]; only for cases including the adjoint to the Spalart–Allmaras turbulence model, chapter 5. 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 discretization 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 2.4.1.1. The *epsilon* entry (default value 0.1) should be the same as the one used in *fvSchemes*, in the *wallDist/advectionDiffusionCoeffs* dictionary, if *method advectionDiffusion* is chosen. 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.

naca0012/turbulent/liftFullSetup

2.4.1.2 surfacePoints

Same as *surface*, section 2.4.1.1, 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 some of the terms included in the computations (e.g. variation in the normal vector) are better posed when differentiating w.r.t. points. 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 2.4.1.1 are valid here as well.

2.4.1.3 multiple

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

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

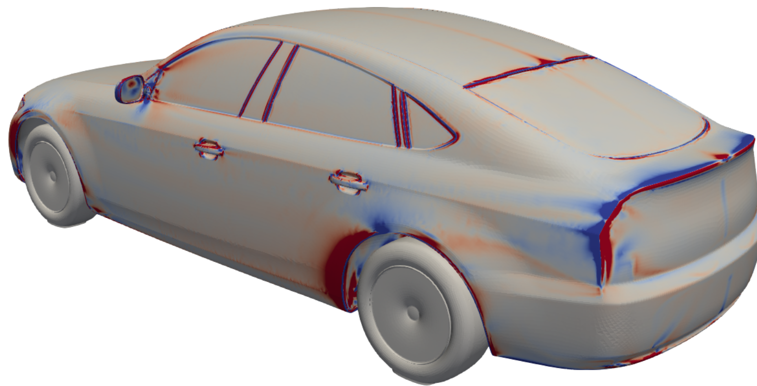

Figure 2.1: Drag sensitivity map computed on the surface of the DrivAer car model. Blue areas should be moved according to the surface normal ("inwards") to reduce drag while red areas should be moved in the opposite direction.

Chapter 3

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 ($\approx$ 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*.

naca0012/turbulent/liftFullSetup

Chapter 4

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(-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.

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*.

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)*.

motorBike

Chapter 5

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)
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* option solves the PDEs of the adjoint to the Spalart Allmaras turbulence model, [2, 3]. Boundary conditions, solvers, relaxation factors and discretization schemes should be set for *nuaTilda* (adjoint to *nuTilda*). Details for each of the above are given in chapters 3, 4 and 6.

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

naca0012/turbulent/liftFullSetup

5.1 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;
}
```

motorBike

Chapter 6

Adjoint boundary conditions

Files defining the adjoint boundary conditions (BCs) should be provided at the *startTime* 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.

6.1 *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, [2]. Otherwise, a typical low-Re boundary condition will be applied, [2].

- *adjointWallVelocityLowRe*: Same as *adjointWallVelocity* but only for low-Re or laminar flows.

- *adjointFarFieldVelocity*: Far-field boundaries where an *inletOutlet* or *freestream* BC is imposed on U.

6.2 *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.

6.3 *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*.

6.4 *ma* boundary conditions

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

6.5 *da* boundary conditions

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

Chapter 7

Applications

7.1 solvers

7.1.1 adjointOptimisationFoam

adjointOptimisationFoam is the main executable. For the moment, only the *singleRun* mode of operation is available, in which the primal and adjoint equations are solved, followed by the computation of sensitivity derivatives. A second mode of operation supporting automated shape optimisation will soon be made available.

7.1.1.1 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 *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*.

naca0012/turbulent/liftMinimumSetup

7.2 utilities

7.2.1 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 2.4.1. Remember to also set the *computeSensitivities* to *true* in the adjoint solver dicts, section 2.3.1.

Bibliography

[1] 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.

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

[3] 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.