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 2020

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

Contents

Chapter 1

Introduction

The present User Manual serves as a guide for the setup and usage of the OpenFOAM executable *adjointOptimisationFoam*, included in OpenFOAM-v2006. 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, [2, 5, 9]. 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, chapter 6 provides guidelines for defining the adjoint boundary conditions while chapter 7 explains the setup for deforming the mesh during shape optimisation runs, including the setup of a volumetric B-splines morpher. Chapter 8 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 is set up. It is located in *system* and needs to be present for practically all applications presented in section 8 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 just once, without performing an optimisation loop while *steadyOptimisation* is used when an automated optimisation loop is targeted. Further details about the setup of the code in each of these scenarios are given in sections 8.1.1.1 and 8.1.1.2.

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.

nInitialIters optional, default=nIters
The number of primal iterations to be executed in the first optimisation cycle. Could potentially be higher than *nIters*, since the primal equations will likely require more iterations to converge in the first optimisation cycle than in the subsequent ones.

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

2.2.1.2 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;
                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 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.

isConstraint: (true|false)
Whether the objective function of this solver will act as a constraint. See also section 2.4.3 for the appropriate *updatheMethods* to be used in the presence of constraints.

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, forceTarget, moment, PtLosses, partialVolume)
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 **n** the unit normal vector. Vector **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.

sensitivityMaps/naca0012/laminar/drag

forceTarget

$$J = F - F_{tar} \tag{2.3}$$

where F is the force projected to a certain direction, as defined in the *force* objective, and F_{tar} a target force value.

The *forceTarget* dictionary is the same as that of of the *force* objective, with an additional entry specifying the target force value

```
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.;
    target      0.45;
}
```

Note: The *forceTarget* objective should only be used as a constraint (see also the *isConstraint* entry in section 2.3.1.1).

moment

$$J = \frac{\int_{S_W} \rho r_i^M e_{ijk}(x_j - x_j^C)(-\tau_{kl} n_l + p n_k) dS}{\frac{1}{2}\rho A l U_\infty^2} \tag{2.4}$$

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.;
};
```

sensitivityMaps/naca0012/laminar/moment

PtLosses

$$J = -\int_{S_{I,O}} \left(p + \frac{1}{2}v_k^2\right) v_i n_i dS \tag{2.5}$$

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);
};
```

sensitivityMaps/sbend/laminar

partialVolume

$$J = \frac{V - V_{init}}{V_{init}} \tag{2.6a}$$

$$V = -\frac{1}{3}\int_{S_W} x_k n_k dS \tag{2.6b}$$

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);
};
```

Note: The *partialVolume* objective should only be used as a constraint (see also the *isConstraint* entry in section 2.3.1.1).

shapeOptimisation/naca0012/lift/opt/constraintProjection

nutSqr

$$J = \int_{\Omega'} \nu_t^2 d\Omega \tag{2.7}$$

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 [6].

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

zones are the *cellZones* defining Ω'.

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.

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. These can be useful indicators of the simulation stability.

2.4 optimisation

The *optimisation* dict is optional and should be present only when an automated optimisation loop is to be executed or 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

sensitivityType
(
surface
surfacePoints
volumetricBSplines
volumetricBSplinesFI
Bezier
BezierFI
multiple
)

patches
On which patches to compute sensitivities. Wildcards and group names allowed. Even for sensitivity computations that do not depend on surface integrals (e.g. *sensitivityVolBSplinesFI* or *sensitivityBezierFI*), the *patches* entry should be completed correctly since the so-called direct sensitivities will be computed there.

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 aerodynam-

ics, 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, [2]. 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 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 (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, [5]; 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.

sensitivityMaps/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 volumetricBSplines

This option computes sensitivity derivatives w.r.t. the control points of a volumetric B-splines morpher. The theoretical background for the latter can be found in [6] whereas its OpenFOAM setup is explained in chapter 7. Sensitivities are computed using the chain rule, i.e.

$$\frac{\delta J}{\delta b_n} = \frac{\delta J}{\delta x_i}\frac{\delta x_i}{\delta b_n} \tag{2.8}$$

when $\delta J/\delta x_i$ is the sensitivity map (see section 2.4.1.1) and $\delta x_i/\delta b_n$ is computed analytically on the surface, by differentiating the volumetric B-splines morpher. The default settings provided in section 2.4.1.1 are used to compute the sensitivity map; these can be altered through the optional *surfaceSensitivities* sub-dictionary.

```
type                volumetricBSplines;
patches             (lower upper);
surfaceSensitivities
{
    // the options listed in section 2.4.1.1
```

```
    // can be overridden here
}
```

It should be noted that since the E-SI approach for computing sensitivity maps assumes a grid displacement model of a set of Laplace-based PDEs but the volumetric B-splines morpher actually moves the mesh using an analytic/algebraic formula, slight inconsistencies might emerge in the computed sensitivities. For optimisation runs in which the maximum sensitivity accuracy is required, the *volumetricBSplinesFI* option (section 2.4.1.4) is proposed as a more expensive but more accurate alternative. Upon computation, the sensitivity derivatives are written in a file named *volumetricBSplines*, appended by the *adjointSolver* name and the time-step value and located in the *optimisation/volumetricBSplinesDerivatives* folder.

shapeOptimisation/sbend/laminar/primalAdjoint/

2.4.1.4 volumetricBSplinesFI

This options computes sensitivities of the objective function w.r.t. the control points of a volumetric B-splines morpher (see chapter 7) using the Field Integrals (FI), [2], approach. No additional entry is required apart from the mandatory ones, i.e.

```
type     volumetricBSplinesFI;
patches  (pressure suction);
```

Note that depending on the case (large number of CFD grid points inside the parameterized domain, large number of control points and high basis degree), computing sensitivities with this approach could be time consuming. Upon computation, the sensitivity derivatives are written in a file named *volumetricBSplinesFI*, appended by the *adjointSolver* name and the time-step value and located in the *optimisation/volumetricBSplinesFIDerivatives* folder.

shapeOptimisation/sbend/laminar/opt/unconstrained/SD

2.4.1.5 Bezier

Sensitivities computed w.r.t. the control points of a Bézier–Bernstein curve, using the chain rule and either the SI or the E-SI approach, depending on the setup of the *surfaceSensitivities* (see also sections 2.4.1.1 and 2.4.1.3). A series of *pointTensorFields* named $dxidXj\_0, dxidXj\_1, \ldots dxidXj\_n$ should be present in the time-step folder to obtain the parameterization information (practically, the Bézier–Bernstein basis functions for each parameterized surface point and each control point).

```
type                Bezier;
patches             (lower upper);
surfaceSensitivities
{
    // the options listed in section 2.4.1.1
    // can be overridden here
}
```

An additional dictionary is required, placed as a direct subDict of *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
    );
}
```

Upon computation, the sensitivity derivatives are written in a file named *Bezier*, appended by the *adjointSolver* name and the time-step value and located in the *BezierDerivatives* folder. Sensitivities w.r.t. the x coordinates of all control points are written first, followed by those w.r.t the y and z coordinates.

sbend/laminar/primalAdjoint

2.4.1.6 BezierFI

Sensitivities computed w.r.t. the control points of a Bézier–Bernstein curve, utilizing the FI approach. A vectorial Laplace equation needs to be solved for each design variable, to propagate the parameterization information given in the $dxidXj\_i$ files (see also section 2.4.1.5) 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     BezierFI;
patches (pressure suction);
dxdbSolver
{
    iters 100;
    tolerance 1.e-11;
}
```

A *volVectorField* named *mTilda* is used for this task with zero *fixedValue* boundary conditions for all patches The *mTilda* 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 discretization 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. Upon computation, the sensitivity derivatives are written in a file named *BezierFI*, appended by the *adjointSolver* name and the time-step value and located in the *optimisation/BezierFIDerivatives* folder. Sensitivities w.r.t. the x coordinates of all control points are written first, followed by those w.r.t the y and z coordinates.

shapeOptimisation/naca0012/drag/primalAdjoint

2.4.1.7 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*.

shapeOptimisation/sbend/laminar/primalAdjoint

2.4.2 optimisationType

```
optimisationType
{
    type            shapeOptimisation;
    writeEachMesh   true;
}
```

optimisationType is a subDict that should be present if *optimisationManager* is set to *steadyOptimisation* (see section 2.1) and is directly placed under *optimisation*. Its only mandatory entry is

optimisationType (shapeOptimisation)
defines the type of adjoint-based optimisation to be conducted. Only *shapeOptimisation* is available at the moment.

For *shapeOptimisation* runs, an additional optional entry can be supplied:

writeEachMesh (true|false)
Whether to write the meshes produced in each optimisation cycle or not, independent of whether the flow fields are going to be written in this optimisation cycle.

2.4.3 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

$$b_i^{new} = b_i^{old} + \eta p_i \tag{2.9}$$

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, constraintProjection, SQP)
Which method to use to update the design variables. Only the *constraintProjection* and *SQP* methods can handle constraints.

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

2.4.3.1 steepestDescent

The simplest and most robust (but also, providing the slowest convergence) method to update the design variables. The update vector is computed as

$$p_i = -\frac{\delta J}{\delta b_i} \tag{2.10}$$

No additional dictionary entries are required.

2.4.3.2 conjugateGradient

The Conjugate Gradient method, [1], 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;
    activeDesignVariables ( 1 2 5 7 ....);
}
```

betaType (FletcherReeves, PolakRibiere, PolakRibiereRestarted)
An optional entry choosing the formula to update the β variable in Conjugate Gradient, [4]. Defaults to *FletcherReeves* which has proved to be the most robust and should be preferred.

activeDesignVariables
A *labelList* informing the method which design variables are allowed to be updated. If no *labelList* is found, the method will update all design variables, except those kept fixed by the parameterization itself (see also section 2.4.1.5 and 7.1.1.1).

shapeOptimisation/motorBike

2.4.3.3 BFGS

```
method BFGS;
BFGS
{
    nSteepestDescent 1;
    etaHessian       1;
    scaleFirstHessian true;
    activeDesignVariables ( 1 2 5 7 ....);
}
```

The quasi-Newton BFGS method [4]. The update is computed through

$$\frac{\delta^2 \tilde{J}}{\delta b_i \delta b_j} p_j = -\eta_H \frac{\delta J}{\delta b_i} \tag{2.11}$$

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 2.4.3.4) 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/SD
shapeOptimisation/sbend/laminar/opt/unconstrained/BFGS
```

nSteepestDescent 1
Number of steepest descent updates conducted before applying the BFGS approach. Should be at least one. Could be more 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 be theoretically 1. Since, however, BFGS is a quasi-Newton method an η_H value should 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 [4]. Usually improves the convergence speed of the method.

activeDesignVariables
Same as the corresponding entry in section 2.4.3.2.

2.4.3.4 LBFGS

```
method LBFGS;
LBFGS
{
    nSteepestDescent 1;
    etaHessian       1;
    nPrevSteps       10;
    activeDesignVariables ( 1 2 5 7 ....);
}
```

The limited memory variant of the BFGS quasi-Newton method [3]. *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. *LBFGS* has the same (optional) entries as *BFGS* (with the exception of

scaleFirstHessian) and includes one more

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

2.4.3.5 SR1

```
method SR1;
SR1
{
    nSteepestDescent 1;
    etaHessian       1;
    activeDesignVariables ( 1 2 5 7 ....);
}
```

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

2.4.3.6 constraintProjection

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

constraintProjection is an *updateMethod* that supports the handling of equality constraints, [8] . In particular, the update direction is that of steepest descent, if the part that is normal to all constraint isolines is subtracted. *constraintProjection* is ideal 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

2.4.3.7 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, [4]. It attempts to iteratively satisfy the Karush-Kuhn-Tucker (KKT) conditions. The necessary Hessian matrix is approximated using BFGS and, hence, its optional entries are the same as the ones described in section 2.4.3.3. *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 2.4.3.3).

shapeOptimisation/sbend/laminar/opt/constrained/SQP

■

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

eta

If an η value (see eq. (2.9)) is manually set here, it will be used for the optimisation loop. Otherwise, an η value will be computed based on the entries defined in *meshMovement*, see section 2.4.4.

lineSearch

Line search methods can be used to adjust the *eta* value from one optimisation cycle to the next. Each type of line search method attempts to find an *eta* value that satisfies a certain kind of conditions (see [4] for more details); if these conditions are not met, the last update is undone and the previously computed **p** direction (see eq. (2.9)) 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, none)
The *type* entry defines what kind of condition should be met for accepting the current *eta* value. *ArmijoConditions* is the only available option at the moment and, according to it, the following inequality should hold in order to accept the current *eta* value

$$\phi(\mathbf{b}+\eta\mathbf{s}) \leq \phi(\mathbf{b}) + c_1\eta D(\phi(\mathbf{b});\mathbf{s}), \tag{2.12}$$

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 [4]

$$\phi = J + \mu \sum_{i=1}^{M} |e_i| \tag{2.13}$$
$$\mu = max(|\lambda_i|), i \in [1, M]$$

In eq. (2.12), the default value of c_1 is 10^{-4} as suggested in [4] for quasi-Newton approaches computing $\mathbf{s}$. The initial step is tested and if eq. (2.12) 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. (2.13), 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

2.4.4 meshMovement

```
meshMovement
{
    type                        volumetricBSplines;
    maxAllowedDisplacement      1.e-2;
    writeMeshQualityMetrics     false;
}
```

meshMovement is a subDict of the *optimisationDict* and should be present when the *optimisationManager* is set to *steadyOptimisation* and *optimisationType* is set to *shapeOptimisation*. It provides information about how to translate the update of the design variables into a mesh displacement (boundary and interior) in each optimisation cycle of a shape optimisation loop.

type
(
volumetricBSplines
volumetricBSplinesExternalMotionSolver
Bezier
)
Each option is analyzed in the sections that follow.

maxAllowedDisplacement
Provides the maximum boundary displacement to be imposed at the first optimisation cycle. If η is explicitly defined in *optimisationMethod*, section 2.4.3, this entry will be ignored. Otherwise, the η value will be computed based on the *maxAllowedDisplacement* value provided here.

writeMeshQualityMetrics (true|false)
Writes mesh quality metrics, after each mesh update, in fields for visualization purposes.

2.4.4.1 volumetricBSplines

Used to deform the boundary and the interior of the mesh using a volumetric B-splines morpher, the specifics of which are described in section 7.1.

shapeOptimisation/sbend/laminar/opt/unconstrained/BFGS

2.4.4.2 volumetricBSplinesExternalMotionSolver

Similar to *volumetricBSplines*, section 2.4.4.1, but this option will use the volumetric B-splines morpher to displace only the boundary while the mesh displacement will be propagated to the interior using an alternative grid displacement model defined in *dynamicMeshDict*. Since the volumetric B-splines morpher has proved very robust in a number of industrial applications, using *volumetricBSplines* is preferred to *volumetricBSplinesExternalMotionSolver*. This option is provided in the sake of completeness.

2.4.4.3 Bezier

Used to deform the mesh by moving parameterized Bézier–Bernstein surfaces and propagating the movement to the interior mesh by using a grid displacement model defined in *dynamicMeshDict*. The *Bezier* dict mentioned in section 2.4.1.5 needs to be present for this operation as well.

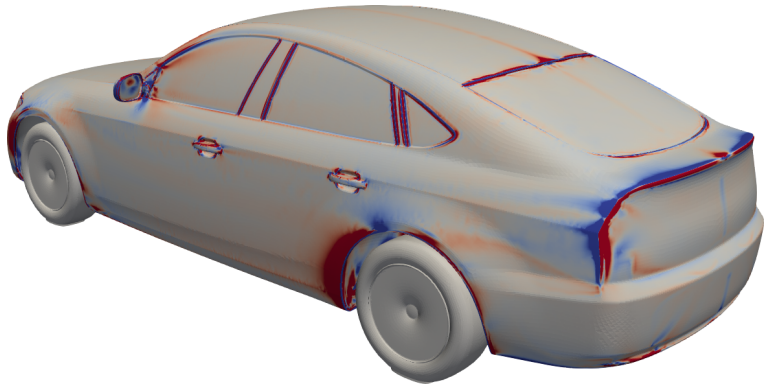

Figure 2.1: Drag sensitivity map computed on the surface of the DrivAer car model. Blue areas should be moved according to the surface normal (“inwards”) to reduce drag while red areas should be moved in the opposite direction.

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 (≈ 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 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.

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

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, [5, 9]. 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.

```
sensitivityMaps/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;
}
```

sensitivityMaps/motorBike

Chapter 6

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.

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, [5]. Otherwise, a typical low-Re boundary condition will be applied, [5].

- *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.

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 *nuTilda* 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

dynamicMeshDict

The setup of the volumetric B-splines morpher is explained in detailed in section 7.1. Setting up a Laplace-based grid displacement PDE for use in conjunction with $2D$ Bézier–Bernstein parameterizations in presented in section 7.2.

7.1 Volumetric B-splines Morpher

7.1.1 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 [6] 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. 7.1. 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

$$x_m(u,v,w) = \sum_{i=0}^{I} \sum_{j=0}^{J} \sum_{k=0}^{K} U_{i,pu}(u) V_{j,pv}(v) W_{k,pw}(w) b_m^{ijk} \tag{7.1}$$

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 [7]. 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

$$\mathbf{R}(u,v,w) = \begin{bmatrix} x(u,v,w) - x_r = 0 \\ y(u,v,w) - y_r = 0 \\ z(u,v,w) - z_r = 0 \end{bmatrix} \tag{7.2}$$

where $x_m(u,v,w)$ are computed through eq. (7.1), based on the known $\mathbf{b}$ values. The 3×3 system of eq. (7.2) 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 $\mathbf{b}$, the Cartesian coordinates of each (internal or boundary) mesh point residing within the control grid can be computed through eq. (7.1) 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);
```

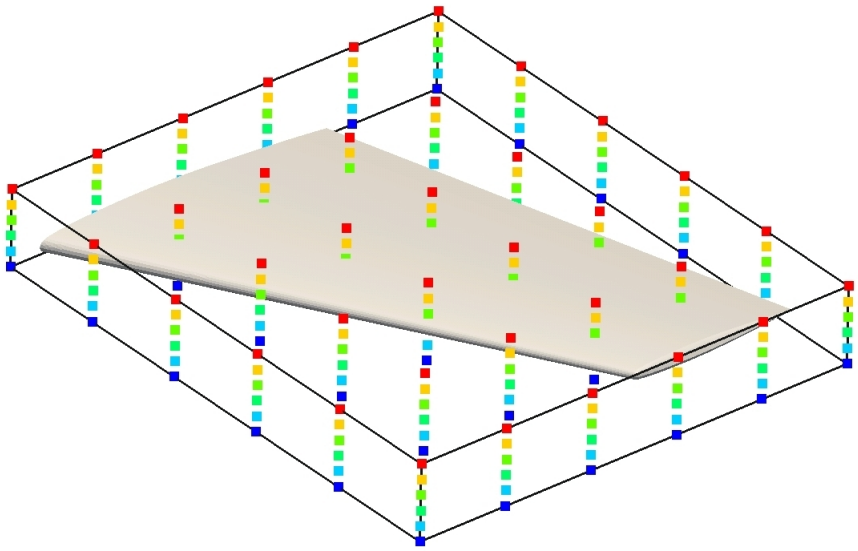

Figure 7.1: Control grid consisting of 6×6×6 control points, around a $3D$ wing. Control points are coloured based their j index value. Surface and volume mesh points, over and around the wing, residing within the boundaries of the control grid (black box) will be displaced following a possible displacement 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. 7.1, will be created for each sub-Dict within *volumetricBSplinesMotionSolverCoeffs*. More than one control boxes are supported, as long as they are not overlapping.

7.1.1.1 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. 7.1.

degreeU, degreeV, degreeW
Basis function degrees in the three parametric coordinates (i.e. pu, pv and pw in eq. 7.1). 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/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 7.1.1.2.

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.

7.1.1.2 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. (7.2). 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.

7.1.1.3 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 8.2.1.2. *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 7.1.1.1) 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/BFGS

7.1.1.4 Continuing an optimisation loop using volumetric B-splines

This section covers the scenario of wishing to continue a job which has already ran for a number of optimisation cycles, using the volumetric B-splines parameterization and assuming that the initial setup used the *axisAligned* option for the *controlPointsDefinition* (section 7.1.1). If the second job is submitted using *axisAligned* as well,

then the code will disregard the already updated control point positions, as computed by the previous optimisation cycles, and wrongly re-use the ones suggested by the *{lower,upper}CpBounds*. Instead, for the second job, the *fromFile* option should be used for the *controlPointsDefinition*, continuing the optimisation from the point it terminated during the first job. An example of the steps to be followed can be found in the *Allrun* script of the following tutorial

shapeOptimisation/sbend/turbulent/opt/BFGS-continuation

7.2 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 8

Applications

8.1 solvers

8.1.1 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.

8.1.1.1 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

8.1.1.2 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 geometry and mesh for *n* optimisation cycles), fig. 8.1. 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/BFGS

8.2 utilities

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

8.2.1 preProcessing

8.2.1.1 writeActiveDesignVariables

This utility gathers the IDs of the active design variables, for instance, as defined by the *confine** and *bound*CPs* options in case of a volumetric B-splines morpher (section 7.1), and writes them in the appropriate place within the subDict pertaining to the *updateMethod* defined in *optimisationDict* (section 2.4.3). In *v1912*, it was mandatory to run this utility before *adjointOptimisationFoam* in combination with most *updateMethods*. However, since *v2006*, this is no longer necessary.

8.2.1.2 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 7.1.1.3. 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.

8.2.2 postProcessing

8.2.2.1 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 2.4.1. Remember to also set the *computeSensitivities* to *true* in the adjoint solver dicts, section 2.3.1.

8.2.2.2 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/BFGS

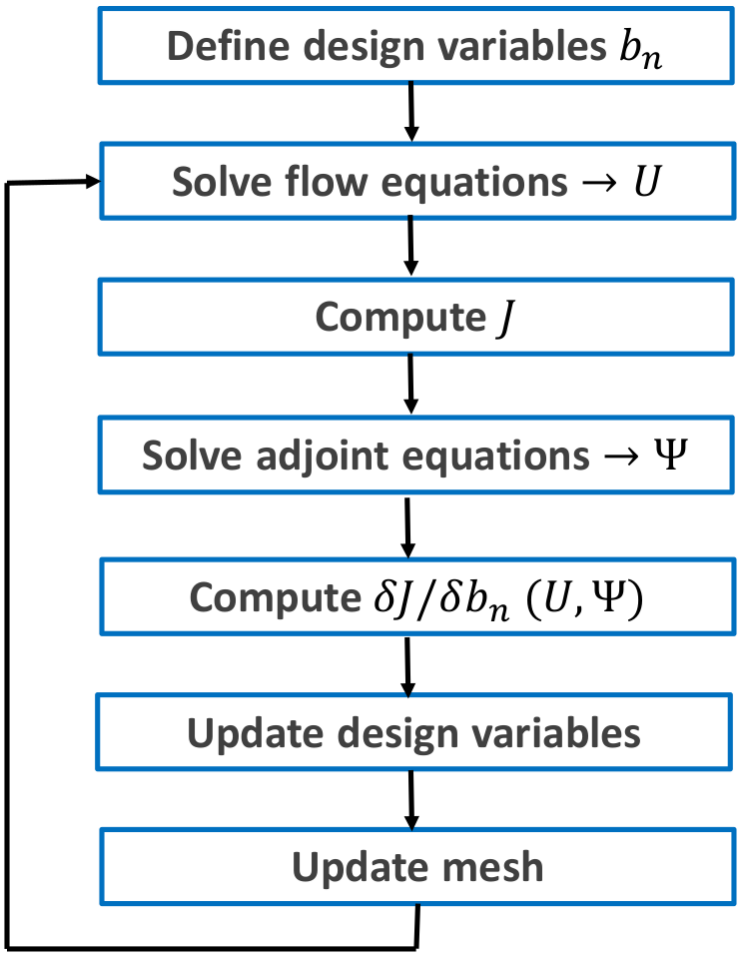

Figure 8.1: The adjoint-based shape optimisation loop executed by *adjointOptimisationFoam* when run in *steadyOptimisation* mode.

Bibliography

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

[2] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.