Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 429 – 438

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

A Discrete Adjoint Model for OpenFOAM

M. Towara[a,*], U. Naumann[a]

[a]Software and Tools for Computational Engineering, RWTH Aachen University, 52056 Aachen, Germany

Abstract

We present a discrete adjoint version of the open-source fluid-dynamics package OpenFOAM obtained by operator overloading which yields, in comparison to continuous adjoint versions, a greater flexibility and robustness. We discuss our implementation and how the discrete adjoint version of OpenFOAM differs from existing continuous implementations. To reduce the inherent memory requirement of discrete adjoint code we introduce a checkpointing scheme to trade computation time for memory. Moreover we show results from a relevant reference case.

Keywords: CFD, OpenFOAM, discrete adjoints, shape optimization

1. Introduction

In this paper we show how to apply Algorithmic Differentiation (AD) [1] to an industrial size CFD code. We use the popular computational fluid dynamics solver package OpenFOAM[2]. OpenFOAM is open-source under the GPL and is implemented in C++.

In section 2 we show the background of the studied CFD shape optimization problem. We introduce and compare continuous and discrete mode for obtaining derivatives of the problem. In section 3 we present the implementation of our discrete adjoint solver `discreteAdjointSimpleFoam` which calculates derivatives needed for shape optimization. Furthermore we implement techniques to reduce the memory footprint of this solver. In section 4 we present results for a optimization test case obtained with our discrete adjoint solver.

2. CFD and AD Background

2.1. Primal Navier-Stokes Equations

The Navier-Stokes equations for incompressible, isothermal, Newtoinian fluid consist of the equations of momentum and mass conservation:

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} = \nu \nabla^2 \mathbf{v} - \frac{1}{\rho} \nabla p + \mathbf{b} \tag{1}$$

$$\nabla \cdot \mathbf{v} = 0 \tag{2}$$

*Markus Towara. Tel.: +49-241-80-20127.
E-mail address: towara@stce.rwth-aachen.de.

doi:10.1016/j.procs.2013.05.206

where $\mathbf{v}$ denotes the velocity vector, p the pressure, ν the kinematic viscosity, ρ the density and $\mathbf{b}$ a vector of external body forces (e.g. gravity).

For three dimensions this leads to four partial differential equations with four unknowns. Typically one wants to solve the equations of momentum and mass conversation independently to be able to apply the discretizations for generic conservation eqauations [3]. The velocities can be uniquely derived from the momentum equations, if one treats the pressure as a source term. But the equation of mass conservation does not contain the pressure p, so an additional hypothesis is needed. This leads to iteration methods such as the SIMPLE-Algorithm [4]. The latter algorithm is implemented in the OpenFOAM solver binary `simpleFoam` which calculates stationary incompressible flows with a pseudo-time stepping scheme.

For the use case of shape optimization we add an additional resistance term $-\alpha\mathbf{v}$ to the right hand side of the momentum equation (1) and neglect external body forces. This resistance term will allow us to penalize the velocity in individual cells in the discretized equations. For this one aims to find cells in which a reduction of velocity (by increasing α) will lead to an improved flow w.r.t. to a given cost function.

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} = \nu\nabla^2\mathbf{v} - \frac{1}{\rho}\nabla p - \alpha\mathbf{v}$$

This approach or slight variations thereof are widely used in shape optimization [5, 6].

2.2. Continuous Adjoint Formulation

To be able to understand the fundamental differences between the continuous and discrete approach to adjoining the Navier-Stokes equations we first briefly introduce the derivation of the continuous equations. From the primal equations of mass and momentum conservation the adjoint counterparts can be derived analytically by variation techniques [5]. These adjoint equations can be solved along the primal equations to obtain the sensitivities of the chosen cost functional J w.r.t. the parameters α:

$$\begin{aligned} -2\mathbf{D}(\mathbf{u})\mathbf{v} &= -\nabla q + \nabla\cdot(2\nu\mathbf{D}(\mathbf{u})) - \alpha\mathbf{u} - \frac{\partial J_\Omega}{\partial \mathbf{v}} \\ \nabla\cdot\mathbf{u} &= \frac{\partial J_\Omega}{\partial p}. \end{aligned}$$

In addition to the already introduced variables $\mathbf{v}, p, \alpha, \nu$ there are now also variables $\mathbf{u}$ for the adjoint velocity, q for adjoint pressure and J for the used cost function. The partial derivative $\frac{\partial J_\Omega}{\partial \mathbf{v}}$ denotes the dependance of the cost functional on the inner domain. For the use case of ducted flows this term often vanishes, because the cost function only depends on the boundary of the domain (e.g. total pressure loss).

The boundary conditions for the adjoint equations have to be derived by hand for each individual cost function. For the derivation of the boundary conditions for ducted flow using the total dissipation of the system as cost function see [5]. The solver `adjointShapeOptimizationFoam` which is available in OpenFOAM 2.0 and following versions implements the pressure loss minimization cost function

$$J_p = \int_\Gamma p \; \mathrm{d}\Gamma$$

which corresponds to our discrete implementation. In [5] a slightly different total dissipation minimizing formulation is proposed:

$$J = -\int_\Gamma \left(p + \frac{1}{2}v^2\right)\mathbf{v}\cdot\mathbf{n}\; \mathrm{d}\Gamma.$$

For both formulations Γ denotes the inflow and outflow boundaries.

The sensitivity of the cost function J w.r.t. the resistance term for each individual cell α_i can be determined by the inner product of primal and adjoint velocities, scaled by the cell volume V_i:

$$\frac{\partial J}{\partial \alpha_i} = (\mathbf{v}_i\cdot\mathbf{u}_i)\, V_i.$$

2.3. Optimization Algorithm

If the sensitivities of J with respect to each α_i are known (either obtained by continuous or discrete methods), then the resistance field α can be updated by a suitable optimization algorithm to improve the solution w.r.t. the given cost functional.

The simplest method to update α is to apply method of steepest gradient descent with fixed stepsize:

$$\alpha_i^{t+1} = \alpha_i^t - \lambda \frac{\partial J^t}{\partial \alpha_i^t}.$$

To obtain a steady solution after a finite number of optimization steps α is capped below zero and above a certain threshold (otherwise α would rise indefinitely in regions of negative sensitivities and could fall below zero in regions with positive sensitivities, which is physically impossible)

$$0 \leq \alpha \leq \alpha_{max}.$$

2.4. Discrete Adjoint Formulation

We treat our optimization problem as $J : \boldsymbol{R}^n \to \boldsymbol{R}^m$, $J(\mathbf{x}) \to min!$, where each function evaluation $J(\mathbf{x})$ implies a complete solver run on the nonlinear equation systems. For the purpose of shape optimization m will either be one or $O(1)$.

First order AD assumes J to be at least once continuously differentiable at all points of interest. For a given implementation of $\mathbf{y} = J(\mathbf{x})$, a corresponding (first-order) tangent-linear code computes a directional derivative

$$\mathbf{y}^{(1)} = J^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv \nabla J \cdot \mathbf{x}^{(1)},$$

where $\mathbf{x}^{(1)} \in \boldsymbol{R}^n$, $\mathbf{y}^{(1)} \in \boldsymbol{R}^m$, and $\nabla F = \nabla F(\mathbf{x}) \in \boldsymbol{R}^{m \times n}$ denotes the Jacobian matrix of F. A (first-order) adjoint code computes

$$\mathbf{x}_{(1)} = J_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv \nabla J^T \cdot \mathbf{y}_{(1)},$$

where $\mathbf{x}_{(1)} \in \boldsymbol{R}^n$ and $\mathbf{y}_{(1)} \in \boldsymbol{R}^m$. Dense Jacobians can be obtained in tangent-linear mode at $O(n) \cdot Cost(F)$, where $Cost(F)$ denotes the computational cost of a single evaluation of F. The same Jacobian is obtained in adjoint mode at $O(m) \cdot Cost(F)$.

For the purpose of our OpenFOAM optimization we use the adjoint model, because a large number of inputs (n) are mapped to a rather small amount of outputs (m).

Second and higher derivatives can be obtained by re-applying the tangent-linear model on the existing tangent-linear or adjoint code.

Conceptually, AD is based on the fact that each computer program can be decomposed at run time into a *single assignment code*

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\quad v_j = \varphi_j(v_i)_{i \prec j}, \end{aligned}$$

where $i \prec j$ denotes a direct dependence of the variable v_j on v_i. The result of each *elemental function* φ_j is assigned to an unique auxiliary variable v_j. The *n independent inputs* $x_i = v_i$, for $i = 0, \ldots, n-1$, are mapped onto *m dependent outputs* $y_j = v_{n+p+j}$, for $j = 0, \ldots, m-1$. The values of *p intermediate variables* v_k are computed for $k = n, \ldots, n+p-1$.

Assuming differentiability of the elemental functions at the current point, the tangent-linear mode of AD augments each elemental assignment with its tangent-linear model as follows:

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\quad v_j^{(1)} = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_i^{(1)} \\ &\quad v_j = \varphi_j(v_i)_{i \prec j}. \end{aligned} \tag{3}$$

Directional derivatives are propagated in parallel with the original function evaluation.

In adjoint mode, a forward evaluation of the original program is succeeded by the propagation of adjoints for all v_i in reverse order, that is, for $i = n + p - 1, \ldots, 0$:

$$\left.\begin{array}{l} \text{for } j = n, \ldots, n+p+m-1 \\ \quad v_j = \varphi_j(v_i)_{i<j} \end{array}\right\} \text{forward section}$$
$$\left.\begin{array}{l} \text{for } i = n+p-1, \ldots, 0 \\ \quad v_{(1)i} = \sum_{j:i<j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_{(1)j} \end{array}\right\} \text{reverse section} \tag{4}$$

Note that the v_j computed in the forward section are potentially required as arguments of local partial derivatives within the reverse section. They are read in reverse with respect to the original order of their evaluation. The additional (compared with the tangent-linear code) persistent memory requirement of the adjoint code is $O(n+p+m)$. The data flow reversal is the main challenge in adjoint AD. It is responsible for black-box AD typically not being applicable to large-scale numerical simulations. The available persistent memory may simply not be large enough [7].

2.5. Discrete vs. Continuous Adjoint Formulation

The different approaches to the determination of adjoints may lead to differences in the results. In the continuous approach the equations to determine the adjoint quantities are derived analytically from the primal equations. This assumes that the exact primal solution can be obtained by solving the primal equations. Those systems (primal & adjoint) are discretized and solved coupled in variables (the primal velocity **v** appears in the adjoint equations) but independently in terms of numerical methods. So the discretization of equations and the accuracy of the solution may differ between primal and adjoint solutions. The discrete approach on the other hand generates

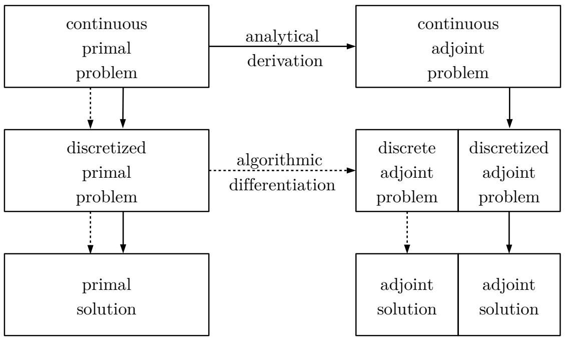
continuous primal problem
analytical derivation
continuous adjoint problem
discretized primal problem
algorithmic differentiation
discrete adjoint problem
discretized adjoint problem
primal solution
adjoint solution
adjoint solution

Fig. 1. Comparison of continuous and discrete adjoint model. Solid edges belong to the continuous model, dashed lines correspond to the discrete model

the adjoints directly from the primal discretization. Thus the adjoints correspond to the results of the solution of the primal equations. It is important to note that these adjoints are exact for the steps taken in the given program. But if the solution of the primal equations has not converged yet, the adjoints, while giving the exact adjoints of the iteration history, may not give the desired adjoints of the primal solution.In Fig. 2.5 we illustrate the different approaches to continuous and discrete adjoint calculation.

3. Implementation of the Discrete Adjoint

3.1. Operator Overloading

The operator overloading approach implements the steps required to differentiate a program described in eq. (3) and eq. (4) by overloading the elemental functions φ_j in the code. This allows to propagate the partial derivatives in case of tangent linear mode or to store the required intermediate values which are needed in the reverse section. We store these intermediate values in a data structure called *tape*. The data flow reversal corresponds to the interpretation of the discrete adjoint code. Details on the implementation of operator overloading can be found in [7, 1].

We use the operator overloading tool `dco/c++` [8]. It can be used to obtain first derivatives by tangent linear or adjoint mode. Furthermore second and higher order derivatives are obtained by applying tangent-linear or adjoint modes to the first-order models again [7]. There are other overloading tools available such as ADOL-C [9], FADBAD [10] or CppAD [11]. Refer to www.autodiff.org for a comprehensive list of tools.

3.2. Enabling Operator Overloading in OpenFOAM

In order to be able to perform operator overloading one has to replace all floating point variables relevant to the calculation of the cost functional with an *active* data type. In `dco/c++` this type is called `dco::a1s::type`. As it is hard to identify the relevant variables in such a big project as OpenFOAM, and choosing a too limited set of variables might affect the flexibility of the framework, the most convenient way is to replace them all. `dco/c++` has methods to identify if variables need to be included into the tape. This so called activity analysis [12] keeps the overhead induced by replacing all floating point values with active datatypes low (see Tab. 2).

OpenFOAM features a highly modular and abstracted program structure. It uses global typedefs for its datatypes which simplifies the switching of the used data type from floating point values to our new active data type for the whole framework.

3.3. Black-Box AD Approach

By Black-Box we mean applying the operator overloading technique to a program without further exploiting knowledge about the underlying program structure. Of course to be able to use the overloading tools, complete access to the sources of the program is needed.

With the help of our discrete adjoint OpenFOAM framework the calculation of derivatives is quite forward. Here we will briefly present how to adjoin a complete solver run to obtain derivatives $\frac{\partial J}{\partial \alpha_i}$ of the Navier-Stokes equations with additional resistance term as presented in section 2.1. Note that this only calculates the sensitivities of the cost-function w.r.t. one given state of α and does not imply an optimization. Listing 1 shows the main routine of a solver executable `discreteAdjointSimpleFoam` which is derived from the OpenFOAM standard solver `simpleFoam`. In the code `U` corresponds to $\mathbf{v}$, `p` to p and `alpha` to α. The inclusion of the resistance term $\alpha\mathbf{v}$ is handled in line 16. The only additional overhead comes from:

- memory allocation for the tape (line 4)
- initializing the cost function (line 3)
- registering the individual entries of alpha as inputs (lines 6-7)
- evaluation of the cost function (lines 43-49)
- setting of output adjoints and interpretation (lines 53-54)
- retrieving the calculated sensitivities (lines 56-60)

The calculation of the cost function can easily be replaced by other formulations, as there are no strong limitations to what can be evaluated (for example J can depend on any value of p or $\mathbf{v}$, regardless if it is lying in the domain or on the boundary, in contrast to the continuous version where only values on the boundary can be evaluated without altering the governing equations). Some examples which come to mind are:

- minimization/maximization of the velocity in a certain cell / a cell set
- maximization of curl in a certain region (mixing)
- achieving prescribed outflow fractions over multiple outlets

Listing 1. Discrete adjoint solver derived from simpleFoam, additional statements are marked with //dco:

```
1 int main(int argc, char *argv[]){
2     // [Initializations, additional includes omitted]
3     scalarDouble J = 0; //dco: result variable for cost func.
```

```
 4      dco::a1s::global_tape = dco::a1s::tape::create();    //dco: create tape
 5
 6      for(int i=0; i<alpha.size(); i++)
 7          global_tape->register_variable(alpha[i]);
 8
 9      for (runTime++; !runTime.end(); runTime++){
10          # include "readSIMPLEControls.H"
11          p.storePrevIter();
12          tmp<fvVectorMatrix> UEqn
13          (
14              fvm::div(phi, U)
15            - fvm::laplacian(nu, U)
16            + fvm::Sp(alpha, U)
17          );
18          UEqn().relax();
19          solve(UEqn() == -fvc::grad(p));
20
21          p.boundaryField().updateCoeffs();
22          volScalarField rAU = 1.0/UEqn().A();
23          U = rAU*UEqn().H();
24          UEqn.clear();
25          phi = fvc::interpolate(U) & mesh.Sf();
26          adjustPhi(phi, U, p);
27
28          // Non-orthogonal pressure corrector loop
29          for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++){
30              fvScalarMatrix pEqn( fvm::laplacian(rAU, p) == fvc::div(phi) );
31              pEqn.setReference(pRefCell, pRefValue);
32              pEqn.solve();
33
34              if (nonOrth == nNonOrthCorr)
35                  phi -= pEqn.flux();
36          }
37
38          # include "continuityErrs.H"
39          p.relax(); // Explicitly relax pressure for momentum corrector
40          U -= rAU*fvc::grad(p);   // Momentum corrector
41          U.correctBoundaryConditions();
42
43          // CALCULATE COST FUNCTION J = Sum over p at inlet
44          J = 0;
45          forAll(pressurePatches(), prpI){
46              label patchI = mesh.boundaryMesh().findPatchID( pressurePatches()[prpI] );
47              const fvPatch& patch = mesh.boundary()[patchI];
48              J += gSum(p.boundaryField()[patchI]*patch.magSf());
49          }
50          runTime.write();
51      }
52
53      dco::a1s::set(J, 1.0, -1); //dco: calc. sensitivities of J w.r.t. the inputs alpha
54      global_tape->interpret_adjoint(); //dco: reverse interpretation of the tape
55
56      double dJda=0;
57      for(int i = 0 ; i<alpha.size(); ; i++){
58          get(alpha[i],dJda,-1); //dco: retrieve the sensitivities from the tape
59          sens[i] = dJda; //dco: save sensitivities in object
60      }
61      runTime.write();
62      return(0);
63  }
```

Unfortunately the simplicity of this approach comes with a price. The tape has to store the whole iteration process. In each outer (pseudo-) time iteration (lines 9-51) sparse linear equation systems for the solution of $\mathbf{v}$ (lines 12-18) and p (lines 30-32) are constructed. These systems are solved with iterative solvers. This iterative solver is called

d times for the solution of $\mathbf{v}$ with $d = 2$ for 2D and $d = 3$ for 3D cases. The equation for p needs to be solved n times where n equals the number of non-orthogonal correction steps for p. If there are additional variables for turbulence the number of solver calls increases even further. All those inner solver calls and iterations have to be taped. This leads to a considerable memory consumption which will exceed the available memory of most machines for all but simple problems. Table 1 lists the memory consumption of the tape holding 10 outer time steps for a coarse testproblem with 1300 cells using different inner solvers. The total tape memory needed is 1167 MB for the left case and 1358 MB for the right case, leading to an additional overhead not related to the solvers of about 19 MB for both cases (for U we measure the difference in tape size from line 12 to line 19, for p we measure the difference from line 21 to line 39.)

Table 1. Memory requirements for ten outer time steps for two different choices of the inner solvers. The convergence criterium is $\epsilon = 10^{-6}$ for all solvers. nNonOrthCorr=1. The used geometry consists of 1300 quad-cells (2D).

variable	solver	total iterations	tape memory
U	PBiCG	93	348 MB
p	GAMG	160	800 MB

variable	solver	total iterations	tape memory
U	smooth	260	280 MB
p	DICPCG	387	1060 MB

In the following we present typical run times for the execution of the OpenFOAM solver. We inspect the following cases:

- unaltered passive version of simpleFoam, i.e., with double datatype still in place
- discrete adjoint version of simpleFoam, i.e. double datatype replaced with `dco::a1s::type`, to study the overhead introduced by the active datatype
- discreteAdjointSimpleFoam with taping but without interpretation
- full discreteAdjointSimpleFoam with taping and interpretation

Table 2. Execution times for the different solver runs. The whole program execution is measured, ten time steps are calculated.

solver	execution time [s]	factor
passive simpleFoam	1.07	1.00
active simpleFoam	2.27	2.12
tape discreteAdjointSimpleFoam	11.9	11.12
full discreteAdjointSimpleFoam	15.9	14.85

We observe that introducing the active datatype roughly doubles the run time if no further actions are undertaken. This shows that the replacement of all floating-point values by active datatypes does not have a dramatic effect on the run time. After we registered some dependent variables in the tape, the run time roughly increases by a factor of five. This is a result that lies in the range one expects from solutions obtained with AD by overloading. If one includes the tape interpretation into the timing the factor increases again by a factor of roughly 1.3. This shows that the tape interpretation is quite efficient and a major part of the effort is already done while creating the tape (a process often called preaccumulation [13]).

One should note that discreteAdjointSimpleFoam includes the additional resistance term in the momentum equations, while simpleFoam does not. For this example the resistance term α was chosen as $\alpha \equiv 0$, so the term should not have any influence on the number of solver iterations and consequently should not have a major influence on run time.

3.4. Checkpointing

In order to reduce the amount of memory occupied by the tape created in the forward run one can introduce a technique called checkpointing [14]. Only parts of the forward execution are taped, the rest is executed in passive mode (i.e. with taping switched off). As we taped only a part of the forward execution we can interpret only this part in the reverse interpretation sweep. After the reverse sweep the partially calculated adjoints are stored in memory and the tape can be erased. Then the program is executed in forward mode again to tape the next part of the program. Afterwards the stored adjoints can be restored and the reverse interpretation can be resumed with the new tape. This alternation between forward execution and reverse interpretation is repeated until the whole program has been taped and interpreted.

To decrease the amount of operations needed to return from the start of the recalculation to the point where the tape is activated again, we can save the program state (create a checkpoint) at given intervals (in the case of OpenFOAM the fields U, p, phi need to be saved). If such checkpoints exist one can resume the forward calculation from the nearest checkpoint instead of needing to start from scratch again.

One has to find a reasonable trade-off between computation time, memory needed for checkpoint storage and memory needed for taping the forward run. There are numerous ways to do this efficiently, for example one could use the revolve algorithm [15].Currently we use an equidistant checkpointing scheme, i.e., checkpoints are written after a given constant number of iterations. See Fig. 3 for a visualization of the needed steps to implement this checkpointing scheme.

Table 3. Example of the memory needed for one typical taping step or checkpoint on a unit-cube blockmesh. The spacial resolution in each direction is doubled from row to row. The tape sizes depend on the amount of solver iterations to convergence and are thus not constant.

Problem size n	Checkpoint Memory [MB/Checkpoint]	max. Tape Memory [MB/Step]
1000	0.174	46.53
8000	1.180	488.3
64000	8.620	5934

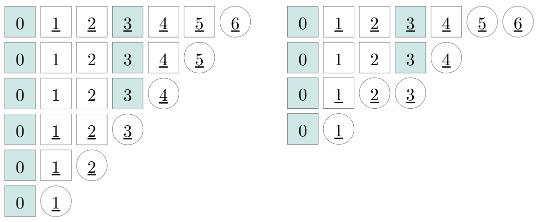

Fig. 2. Needed calculation steps for primal run and complete interpretation of six time steps with one intermediate checkpoint (at t=3) and one (left) resp. two (right) taped time steps. Shaded Time steps are checkpointed, circled time steps are taped for interpretation and underlined steps need to be recomputed from the last checkpoint.

3.5. Implementation of the Optimization Algorithm

We use the steepest gradient descent algorithm as introduced in section 2.3. To obtain the needed derivatives the solver is run with fixed α for a fixed number of iterations. If necessary checkpointing is applied. Those iterations are then adjoined to obtain the sensitivities of the cost functional w.r.t. α. After this the field α can be updated with a steepest gradient descent step. Afterwards the derivatives w.r.t. this new state need to be obtained, so new solver iterations need to be taped. This process is repeated until a set number of optimization steps are completed or α has converged.

4. Results

4.1. Maximization of Velocity in a Specified Single Cell

For this example we examine a rectangular duct of length $10\,l \times 1\,l$. At the inlet $x = 0\,l$ we prescribe a constant velocity magnitude of v_{in} and a zero-gradient condition for the pressure. At the outlet ($x = 10\,l$) we fix p and apply a zero-gradient condition to the velocity. We set the boundaries $y = 0\,l$ and $y = 1\,l$ as no-slip walls. We chose v_{in} such that the Reynolds number $Re = \frac{v_{in} l}{\nu}$ equals $Re = 10$. We assume laminar incompressible steady flow.

The optimization goal is to maximize the velocity component in x-direction in cell i_p which lies nearest to the midpoint $p_m = (5\,l, 0.5\,l)$. To obtain a solution which still retains a reasonable pressure loss we also introduce

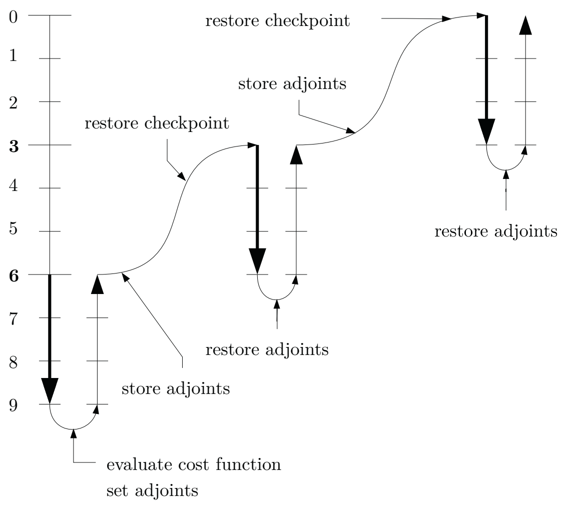

Fig. 3. Visualization of the steps needed to connect the chunks of reverse interpretation to retrieve the correct adjoints. For this example we consider nine time steps, place checkpoints every three time steps and assume we can tape three time steps at a time.

a dependence of the cost functional on pressure loss (otherwise the optimizer would place material everywhere but in the desired cell). As the optimizer is configured to minimize a given cost functional we transform the maximization to a minimization by switching the sign:

$$J = c_1 \cdot J_v + c_2 \cdot J_p = min!.$$

where J_v and J_p are defined as follows:

$$J_v = - \left. v_{x_i} \right|_{i=i_p}$$
$$J_p = \int_\Gamma p \; \mathrm{d}\Gamma.$$

Γ denotes the inflow and outflow boundaries.

For this example we set $c_1 = 1$ and $c_2 = 0.001$. We examine two meshes, one $401\times41\times1$ cubic cell block mesh and one finer $801\times81\times1$ cubic cell block mesh. For both meshes the solution converges to a hourglass shape (see Fig. 4). This shape features a channel through cell i_p with a height of just one cell where $\alpha = 0$. The geometries are symmetric to the x- and y- axis when the origin is placed in the point p_m. Due to leakage in the punished cells the analytic optimal solution for $v_{x_{i_p}}$ can not be reached, but if one thinks of these cells as completely blocked ($\alpha \to \infty$) it is clear that these solutions would be optimal solutions, as the whole flow has to pass through cell i_p. The convergence history (Fig. 5) shows that the cost function J does indeed decrease monotonically.

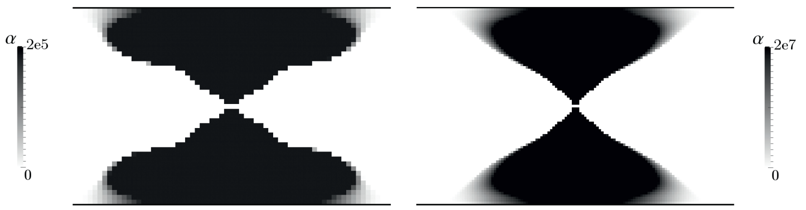

Fig. 4. Optimized shapes for the maximization of velocity. Coarse mesh on the left, fine mesh on the right.

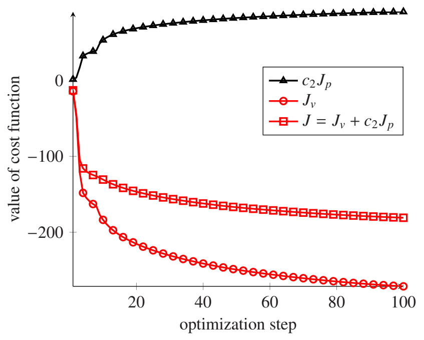

Fig. 5. convergence history of the cost functions over 100 optimization steps for the coarse optimization case

5. Conclusion & Outlook

We presented a new approach to adjoining OpenFOAM and its application to shape optimization. Subsequent work will be done to improve the optimization algorithm by replacing the steepest descent algorithm by a pseudo-Newton algorithm like BFGS [16] or even a Newton algorithm. The latter requires second derivatives, which also can be obtained by the AD tool `dco/c++`. With a more sophisticated optimizer we can also introduce complex constraints like volume constraints. Another area for improvement is to replace the used equidistant checkpointing scheme by revolve [15]. The techniques shown can be potentially applied to all solvers in OpenFOAM, so the optimization can be extended to, for example, turbulent compressible flows.

References

[1] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, 2008, Ch. 6.
[2] OpenCFD Ltd., URL: http://openfoam.com/.
[3] J. H. Ferziger, M. Peric, Computational Methods for Fluid Dynamics, Springer, 1999.
[4] S. Patankar, Numerical heat transfer and fluid flow, Hemisphere Pub. Corp., 1980.
[5] C. Othmer, A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows, International Journal for Numerical Methods in Fluids 58 (8) (2008) 861–877.
[6] L. H. Olesen, F. Okkels, H. Bruus, A high-level programming-language implementation of topology optimization applied to steady-state Navier-Stokes flow, International Journal for Numerical Methods in Engineering 65 (7) (2006) 975–1001.
[7] U. Naumann, The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation., SIAM, 2012, Ch. 1&2.
[8] J. Lotz, K. Leppkes, U. Naumann, dco/c++ - derivative code by overloading in C++, Aachener Informatik Berichte (AIB-2011-06).
[9] A. Walther, A. Griewank, Getting started with Adol-C, U. Naumann and O. Schenk, Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science (2012) 181–202.
[10] C. Bendtsen, O. Stauning, FADBAD, a flexible C++ package for automatic differentiation (IMM–REP–1996–17).
[11] B. M. Bell, CppAD: a package for C++ algorithmic differentiation, Computational Infrastructure for Operations Research, COIN-OR (http://www.coin-or.org/CppAD).
[12] B. Kreaseck, L. Ramos, S. Easterday, M. Strout, P. Hovland, Hybrid static/dynamic activity analysis, Lecture Notes in Computer Science – ICCS 2006 3994 (2006) 582–590.
[13] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, C. Wunsch, OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes, ACM Transactions on Mathematical Software 34 (4) (2008) 18:1–18:36.
[14] J. Grimm, L. Pottier, N. Rostaing-Schmidt, Optimal time and minimum space-time product for reversing a certain class of programs, Computational Differentiation: Techniques, Applications, and Tools (1996) 95–106.
[15] A. Griewank, A. Walther, Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation, ACM Transactions on Mathematical Software 26 (1) (2000) 19–45.
[16] J. Nocedal, S. Wright, Numerical Optimization, Springer, 1999, Ch. 8.