11th World Congress on Computational Mechanics (WCCM XI)
5th European Conference on Computational Mechanics (ECCM V)
6th European Conference on Computational Fluid Dynamics (ECFD VI)
E. Oñate, J. Oliver and A. Huerta (Eds)

A DISCRETE ADJOINT VERSION OF AN UNSTEADY INCOMPRESSIBLE SOLVER FOR OPENFOAM USING ALGORITHMIC DIFFERENTIATION

A.Sen[1], M.Towara[2], and U.Naumann[3]

[1] Software and Tools for Computational Engineering (STCE),
RWTH Aachen University 52062 Aachen, Germany
e-mail: sen@stce.rwth-aachen.de

[2] STCE, e-mail: towara@stce.rwth-aachen.de

[3] STCE, e-mail: naumann@stce.rwth-aachen.de

Key words: Discrete Adjoints, Algorithmic Differentiation, OpenFOAM, Unsteady Flow

Abstract. The comparatively low computational costs of adjoint based gradient methods for optimization problems with a high number of degrees of freedom have allowed them to become one of the most interesting approaches in CFD shape optimization. At the core of such optimization techniques lies the computation of topology sensitivity maps. The two most common approaches for computing adjoint based sensitivities are continuous and discrete (or a combination of both). The continuous approach involves deriving the adjoint equations analytically from the primal equations and then discretizing and solving them alongside the primal equations. The discrete approach in contrast directly differentiates the discretization and solution of the primal equations (basically the code of the program is differentiated), leading to sensitivities consistent with the discretization. The discrete approach has the distinct advantage of flexibility and robustness for a wide range of optimization problems. Most industrial flows exhibit some degree of unsteadiness which leads to lack of robustness and instability of steadily obtained adjoints. Thus an unsteady adjoint is needed. While the continuous approach gets significantly more complex by going from steady to unsteady (the adjoint equations cannot be just solved in parallel to the primal equations any more, because the adjoint now depends on all time steps, thus requiring to store intermediate values), the discrete approach (which had to store the intermediate values already for the steady case) functionally stays the same.

In this paper, we present a discrete adjoint solver based on *pisoFoam*, an incompressible transient solver of the widely used finite volume based open-source CFD tool *OpenFOAM*. The sensitivity maps are generated by the algorithmic differentiation tool *dco/c++*.

1 Introduction

The two principle methods for applying Algorithmic Differentiation [1] are source transformation and operator overloading. *dco/c++*[2] uses the operator overloading approach. Algorithmic Differentiation was applied to the whole *OpenFOAM* framework [3], an open-source CFD software package, thus allowing it to introduce adjoint solvers by adapting the existing primal ones. In this paper, we present the generic idea about how to obtain a differentiated code from the existing solvers in OpenFOAM from scratch.

In section 2, we briefly discuss the OpenFOAM CFD package and the required background in CFD and AD to generate a differentiated code. In section 3 we talk about a somewhat black-box implementation of AD to an existing unsteady incompressible solver in OpenFOAM, *pisoFoam*. The discrete approach generates considerable overheads in terms of runtime and memory requirements. Intermediate values from the whole iteration history need to be stored in order to obtain the sensitivities. In Section 4, we also look at effective techniques to tackle this by implementation of checkpointing schemes (e.g. *Revolve* Algorithm [4]) and treatment of the inner iterative linear solvers used for solving the underlying partial differential equations[5].

As a case study for our discrete adjoint unsteady solver we present the results of sensitivity calculations of the flow around a cylinder leading to vortex shedding (see Fig [5]). Different optimization objectives are feasible, e.g. topology optimization with respect to pressure loss / flow uniformity or decreasing the vortices by applying active flow control (blowing / suction) on the surface of the cylinder. This is a well studied problem (e.g. [6]) and we intend to use this test case to verify our solver.

2 Background

2.1 The OpenFOAM CFD code

The OpenFOAM CFD code uses a numerical approach with a few core features which aid in understanding the implementation of the code in C++.

1) The governing system of equations of our problem creates multiple matrix equations, one for each individual equation as opposed to one big matrix equation for the entire system of equations by decoupling them. They are solved within an iterative sequence.

2) The matrix equations mentioned in 1) are constructed using the Finite Volume method [7].

3) The solution variable for each matrix equation is defined at cell center.

4) The coupling of equations like velocity and pressure are performed using well known algorithms such as *PISO* [8] and *SIMPLE* [9].

The presented research is supported by the project **About Flow**, funded by the European Commission under FP7-PEOPLE-2012-ITN-317006.

2.2 The primal Navier stokes equation

The primal equation that is tackled in the optimization problem is a variation of the Navier Stokes equation with an additional term α, and the continuity equation:

$$\frac{\partial v}{\partial t} + (v \cdot \nabla)v = -\nabla p + \nabla \cdot (2\nu D(v)) - \alpha v \qquad (2.1)$$

$$\nabla \cdot v = 0 \qquad (2.2)$$

Here v denotes the velocity vector, p the pressure, ν the kinematic viscosity, and D(v) is the rate of strain tensor. αv is the additional resistance term. This resistance term comes in handy to penalize the velocity in individual cells in the discretized equations. The basic objective is to locate cells where a reduction in velocity would help to reduce the cost function. This approach has been discussed in considerable detail in [10].

2.3 Continuous vs Discrete Adjoints

In the continuous approach, an additional set of equations is derived from the set of primal equations and their boundary conditions. This set of equations is then discretized alongside the primal equations and solved simultaneously for the adjoint equivalents of the flow variables. The cost function and subsequently the sensitivities are evaluated from the primal and adjoint flow variables.[10]

With the discrete approach however, the cost function and the sensitivities are computed directly from the discretization of the primal equations. Thus in this case, the adjoints correspond to the results of the solution of primal equations only. Computing adjoints via discrete approach gives us the distinct advantage of robustness and flexibility. One can avoid the tedious job of deriving the adjoint equations by hand. This advantage is achieved at the cost of considerable overheads in terms of time and memory requirements since it involves recording and interpreting all the intermediate steps of iteration. In the subsequent sections, we shall look at ways to tackle this challenge.

2.4 The Cost function

Equations 2.1 and 2.2 are solved subject to a constraint that is brought about by the minimization of cost function with respect to the design variables (α in this case) if Topology optimization is desired. α is updated using a suitable optimization algorithm (like steepest descent). In this paper the cost function used is:

$$J = \int_{\tau} (p \ + 0.5\rho v^2) d\tau \qquad (2.3)$$

Here τ denotes the inflow and outflow boundaries. Based on this, the sensitivities $dJ/d\alpha$ are calculated.

3 Implementation of Algorithmic differentiation in *p*isoFoam

3.1 Overloading OpenFOAM CFD code

A basic list of changes that needs to be implemented in OpenFOAM are:

1. All the related *dco* files are appropriately included or linked to.
2. Typically we overload all the *Scalar* and *double* data types with active *dco::a1s::type*.

Theoritically, we just need to recompile OpenFOAM at this point and we are ready. However in practice, it is not so. Capabilities of *dco* have to be taken into account, for example, *dco* does not support *union*. Therefore *unions* have to be replaced in the OpenFOAM CFD code, suitably with the aid of *structures*. Also to avoid ambiguity of data types, it is good practice to explicitly qualify the functions and variables with their relevant namespaces.

3.2 Discrete adjoint version of *p*isoFoam

dco::a1s is a highly flexible and efficient implementation of adjoint Algorithmic differentiation by operator overloading in C++. **dco.hpp** is the interface. The following steps are undertaken to obtain the Discrete adjoint version of pisofoam, *dadpisoFoam.C*.

1. All variables are changed to active data type, *dco::a1s::type*. All operations are overloaded to generate the intermediate representation of the computation in form of a tape
2. Modification of the primal in *pisoFoam.C* to add the porosity term.
3. Memory allocation for the tape.
4. Initializing the cost functions
5. Registering the individual entries of alpha as inputs
6. Evaluation of the cost function
7. Calculating the sensitivities of J with respect to inputs alpha and reverse interpretation of tape.
8. Retrieving the calculated sensitivities.

4 Checkpointing

The spatial complexity of a reverse mode calculation is dependent on the temporal complexity, i.e the memory requirement is proportional to the time complexity of the evaluation of a function since all the intermediate steps need to be recorded. By the method of checkpointing, we intend to manage the memory requirement of an adjoint evaluation by paying the penalty in terns of runtime. The size of the tape which is used to record the forward execution is kept in check by recording only parts of the forward execution. By virtue of this only those parts recorded during the forward execution may be interpreted during the revese mode execution. The missing information is generated using recomputation. To effectively reduce the amount of function evaluations that are required to be taped at one go, chekpoints are created. Checkpoints are snapshots at particular time steps where a program state is stored, in OpenFOAM this essentially means storing the velocity and pressure fields alongwith the mass fluxes. At this point it is useful to mention that in explicit time-dependent problems which *pisoFoam* deals with, each transformation with a counter that corresponds to discrete time can be referred to as a time step. When there is no explicit time dependency (as is the case with the implementation of *simpleFoam*), each pseudo-time step, that corresponds to propagation from one state to another with a counter, then is considered a time step. Two methods of checkpointing are discussed in principle, equidistant checkpointing and binomial offline checkpointing, *Revolve*, as proposed by [4].

4.1 Equidistant Checkpointing

One of the methods of checkpointing employed here is equidistant, where checkpoints are created at equal intervals between the start time and the last time step. This scheme otherwise works quite well but it is not optimal in that it may require more repeated forward time steps than actually needed for the reversal of the given number of time steps for a fixed number of checkpoints. Also, it is noticed that by fixing the number of checkpoints in this scheme (which essentially implies increasing the number of time steps between the creation of two checkpoints), the time complexity tends to grow exponentially. This drawback is overcome by implementation of the binomial checkpointing algorithm using *Revolve*.

4.2 Binomial Offline Checkpointing, *R*evolve

To optimize the spatial and temporal complexity of the reverse mode, in effect ensuring an efficient trade off between the two, it is important to dwell on the task of optimal positioning of the checkpoints.

The operational difference between equidistant checkpointing and this mode of checkpointing is illustrated in Fig 1 and 2 respectively. In Fig 1, during the reverse mode, the checkpoints are not reused, whereas in Fig 2, as soon as a checkpoint becomes free, a new checkpoint is created between the current time step and the last checkpointed step.

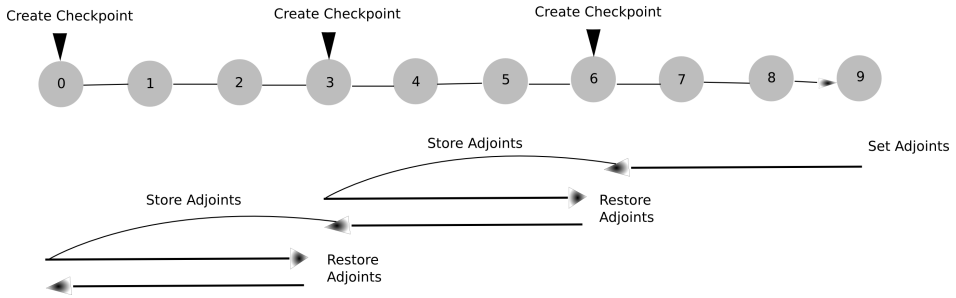

Figure 1: Evaluation process using Equidistant checkpointing

The managing of the information at every checkpoint, i.e saving the system state at the checkpoint and restoring it in order to repeat the subsequent computational steps when required is essentially the same in the two schemes.

To apply this checkpointing scheme, the routine *Revolve* is used which sets the checkpoints in a binomial fashion. To further read about the co-relation between the maximum number of time steps, checkpoints and forward steps please refer [4]. For realization of the checkpointing schedule with *Revolve* in the OpenFOAM solver, the following do-loop is implemented in the reverse mode of evaluation:

0: Initialization: Reserve space for the given number of checkpoints and set the first checkpoint to the initial state.

do end = final, 2, -1

1: Forward: Starting from the last checkpoint assigned advance to the last step but one by performing forward time steps without recording the intermediates as in equidistant checkpointing. Checkpoints are set at choosen intermediate steps.

2: Combined reverse: Perform a forward time step with recording of intermediates to the last step and perform a reverse sweep to the last but one step to calculate the adjoints. If the penultimate step is a checkpoint, it is freed for subsequent use. (This is where this implementation differs from that of equidistant checkpointing, in the later, the collapse of the tape takes place by freeing of checkpoints without reuse).

end do

In Fig. 3, growth in clocktime is plotted against the number of iterations for both equidistant checkpointing and binomial checkpointing using *Revolve*. These results are obtained for a fixed number of checkpoints in both the cases. The performance using *Revolve* is noticed to be significantly better than that of equidistant checkpointing. However this drastic improvement in performance is only indicative, as by increasing the number of checkpoints, the performance using the equidistant checkpointing scheme approaches that using *Revolve*. If we checkpoint all the steps of the iteration, *Revolve* and equidistant checkpointing are equivalent in terms of performance.

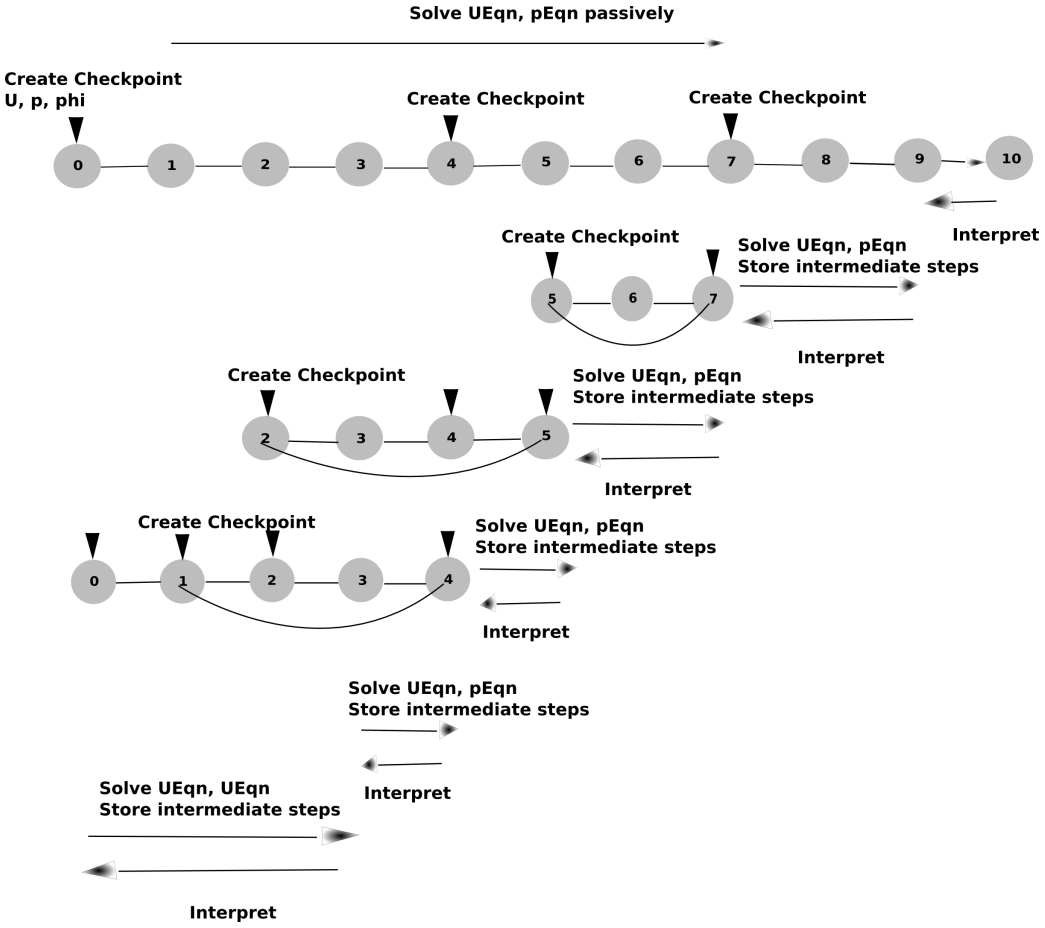

Figure 2: Evaluation process using *Revolve*

4.3 Linear Solver treatment

The traditional discrete approach involves storing data in the forward run and interpreting it in the reverse run. However since most discrete based approaches are severely memory bound, it is essential to look at alternatives. There are two possible methods to tackle this issue. One of them is checkpointing, which is discussed in the previous section. The other is to replace the computationally most expensive part, which often is the linear solver, with a continuous computation of adjoints by solving a different set of linear equations during the reverse run [5]. Of course in large size industrial cases, need may arise to checkpoint even the linear solver by accepting a further hit in terms of computation time.

Fig.4 gives an indicative performance improvement in terms of runtime that is achieved by the linear solver treatment. This performance was achieved when the tolerance of the reverse run was set to the same order as that of the forward run. Therefore it is within the realms of possibility if we vary the tolerance such, for liner solver treatment to yeild

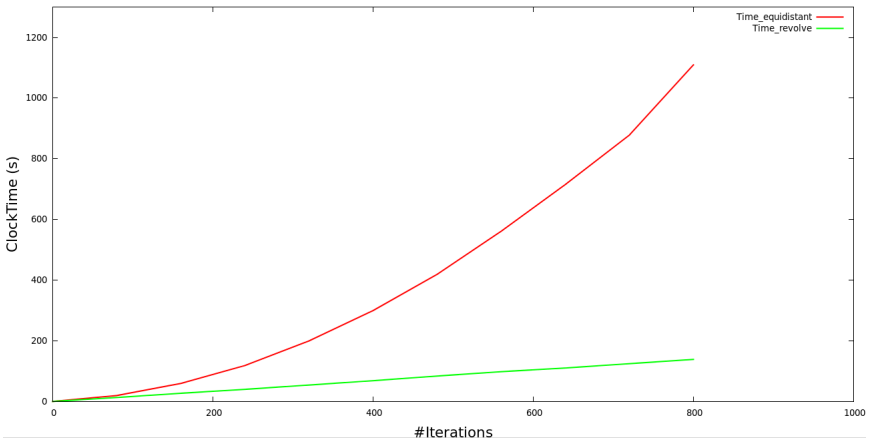

Figure 3: No. of iterations vs Clocktime for *Revolve* and equidistant checkpointing

Table 1: Memory requirements using Linear Solver treatment

Solution mode	UEqn tol.	pEqn tol.	Max. Tape memory
With Linear solver treatment	10^{-7}	10^{-6}	10 MB
Without Linear solver treatment	10^{-7}	10^{-6}	140 MB

to greater computational time than otherwise. However the major advantage is in terms of the memory requirements as shown in Table 1. For the results presented in Fig 4 and Table 1, the following settings are used:
U solver: BICCG, p solver: GAMG, No. of cells: 300.

5 Test Case

2D Flow over a cylinder resulting in vortex shredding is a widely studied case [6]. In this example the calculations were done at $Re = 100$. A relatively coarse mesh of 21730 cells was used to perform the simulation. The Boundary condition at the inlet is Drichlet for velocity and von Newmann for pressure and vice versa at the outlet. The cylinder walls are no slip but we allow slip at the far field. Fig.5 depicts the velocity profile and sensitivity maps which are capped below zero, to potentially show the regions where optimization may place material to build an optimized structure which is the best compromise for all time steps.The cost function. shown in Eq. 2.3 is evaluated and summed up at every time step over a period of 5 seconds, thus capturing the unsteady nature of the problem. The sensitivities are symmetric along the y-axis.

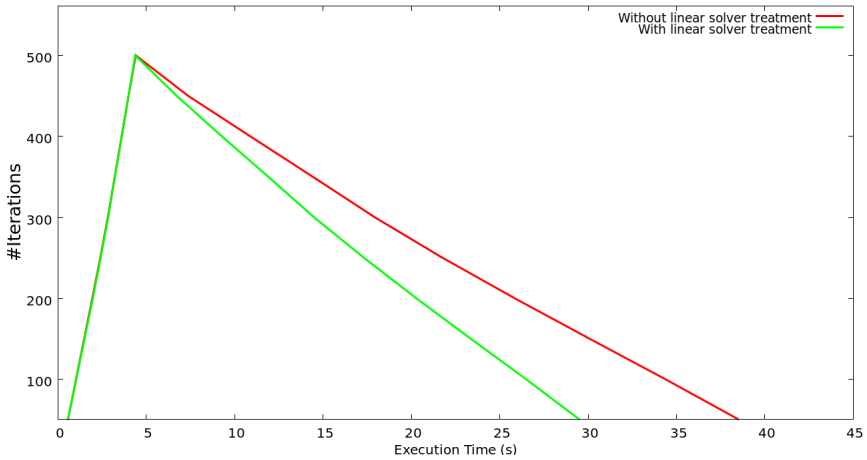

Figure 4: Improvement in terms of computation time achieved via linear solver treatment. During the forward run, the same set of linear equations are solved passively, hence the clocktime is the same, however during reverse run, the effect of the linear solver treatment can be noticed.

Figure 5: The velocity profile and sensitivity plot for the test case

6 Future Work

In this paper a method to obtain an adjoint solver based on an unsteady OpenFOAM solver is outlined. Since discrete based methods are severely memory and time bound, methods to tackle these challenges have been discussed. Further work may involve Topology optimization for turbulent cases. Binomial offline checkpointing scheme may further be developed to employ Multistage checkpointing using *Revolve*, a scheme which uses checkpoints both in memory and disk, when the cost of storing/retrieving checkpoints from disc is not negligible. Also to cover a wider range of CFD problems that often require adaptive mesh generation and employ strategies like remeshing or refinement, problems in which the knowledge of the complete forward solution is not known apriori, online checkpointing scheme yields optimal or near optimal checkpointing distribution.

REFERENCES

1. A. Griewank and A. Walther, Evaluating Derivatives Principles and Techniques of Algorithmic Differentiation, *Second Edition, SIAM*, 2008

2. J. Lotz, K. Leppkes, and U. Naumann, dco/c++ - Derivative Code by Overloading in C++, *Aachener Informatik-Berichte (AIB)*, 2011.

3. M.Towara and U.Naumann, A Discrete Adjoint Model for OpenFOAM, *Procedia Computer Science*, vol. 18, pp. 429-438, 2013.

4. A. Griewank and A. Walther, Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation, *ACM Transactions on Mathematical Software*, vol. 26(1), pp. 1945, 2000.

5. M. Giles, Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation, *Advances in Algorithmic Differentiation*, Springer 2008

6. O. Marquet and D. Sipp, Active steady control of vortex shedding: an adjoint-based sensitivity approach, *Seventh IUTAM Symposium on Laminar-Turbulent Transition, Springer Netherlands* , 2010

7. C.J. Greenshields, H.G. Weller, and A. Ivankovic, The finite volume method for coupled fluid flow and stress analysis, *Computer modelling and simulation in engineering*, 4:213-218, 1999.

8. R. I. Issa, Solution of the Implicitly Discretized Fluid Flow Equation by Operator Splitting, *J. Comput. Phys.*, vol. 62, pp. 40-65, 1986.

9. S.V. Patankar, Numerical heat transfer and fluid flow, Hemisphere Publishing Corp., *1980 - adsabs.harvard.edu*, pp. 210, 1980.

10. C. Othmer, A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows, *International Journal for Numerical Methods in Fluids*, vol. 58 (8), pp. 861877, 2008.