Improving Efficiency of a Discrete Adjoint CFD Code for Design Optimization Problems

Zahrasadat Dastouri and Uwe Naumann

Abstract Sensitivity analysis with the aim of design optimization is a growing area of interest in Computational Fluid Dynamics (CFD) simulations. However, one of the major challenges is to deal with a large number of design variables for large-scale industrial applications. One of the effective solution approaches is to compute adjoint-based sensitivities in the differentiated CFD code. In this paper, we develop a discrete adjoint code for an unstructured pressure-based steady Navier-Stokes solver using Algorithmic Differentiation (AD) by operator overloading (O-O) tool. To reduce the huge memory requirement of the adjoint code we apply effective techniques by implementation of checkpointing schemes and by symbolic differentiation of the iterative linear solver. We combine the flexibility of an operator overloading tool with the efficiency of an adjoint code generated by source transformation through coupling these approaches. Moreover, we improve the performance of the adjoint computation by exploiting the mathematical aspects of the involved fixed-point iteration through reverse accumulation. We compare the effectiveness of these methods in terms of reduction of the numerical cost and accuracy of the sensitivities for the optimization of a vehicle climate duct industrial test case.

1 Introduction

Design optimization for fluid flow plays a significant role in a wide range of engineering applications including aeronautics [9, 4], turbo-machinery [11] and automotive design [13, 14]. The shape of objects in the flow domain, or on the boundary

Zahrasadat Dastouri
Software and Tools for Computational Engineering (STCE),RWTH Aachen, LuFG Informatik 12, D-52062 Aachen, Germany e-mail: dastouri@stce.rwth-aachen.de

Uwe Naumann
Software and Tools for Computational Engineering (STCE),RWTH Aachen, LuFG Informatik 12, D-52062 Aachen, Germany e-mail: naumann@stce.rwth-aachen.de

of the flow domain, is one means of controlling the flow. Finding the shape for such an object that delivers the best possible performance of the fluid flow system in a quantitatively measurable sense leads to a shape optimization problem. In realistic configurations, the optimization problem scales with the large number of design parameters and constraints. The effect of independent design variables on the system performance can be calculated in terms of sensitivities that are the derivatives of one or more quantities (outputs) with respect to one or several independent variables (inputs). By default, these derivatives can be obtained by divided (finite) differences from perturbed solutions. This method is both costly and subject to inaccuracies.

Algorithmic Differentiation (AD) [12, 7] is a well known technique to evaluate derivatives based on the application of the chain rule of differentiation to each operation in the program flow. For a given implementation of the flow model $F : \mathbb{R}^n \to \mathbb{R}^m$ over a computational grid, the computer program is developed to simulate the functional dependence of one or more objectives $\mathbf{y} \in \mathbb{R}^m$ on a potentially large number of the input variables $\mathbf{x} \in \mathbb{R}^n$ by simulation of:

$$F : \mathbb{R}^n \to \mathbb{R}^m, \mathbf{y} = F(\mathbf{x}) \tag{1}$$

AD enables us to compute the corresponding $\frac{\partial y}{\partial x}$ derivatives in forward (forward mode) or backward (reverse mode). There are two main methods for implementing AD: by source code transformation (S-T) or by used of derived data types and operator overloading (O-O). In O-O AD the code segments and arguments of the primal code are stored inside a memory structure called tape during the forward run of the primal. In reverse mode the stored values on the tape are interpreted to get the resultant adjoints derivatives, while in the S-T approach the code is parsed at compile time and the actual derivative code is generated.

Prior to this paper [2], we have described a design framework for application of the AD tool `dco/fortran`[1] *(Derivative Code by Overloading in Fortran)* to the CFD solver *GPDE (General Partial Differential Equation) solver* [10]. GPDE is an unstructured pressure-based steady Navier-Stokes solver with finite volume spatial discretization for incompressible viscous flow computation [3]. In [2] , we discussed the implementation of `dco/fortran` in the original CFD solver from scratch to get the tangent and adjoint version of the primal CFD code. Moreover, we addressed proper solution algorithms adapted to the code including an equidistant checkpointing scheme for the iterative solver and development of symbolically differentiated of linear solver.

In this paper our emphasis is to extend the AD techniques for the improvement of efficiency of the adjoint code. We replace the equidistant checkpointing scheme by binomial checkpointing scheme using `REVOLVE` [6]. Also we combine the flexibility and robustness of operator overloading with the efficiency of source transformation by coupling `dco/fortran` and `TAPENADE` [8] for the most expensive part of the adjoint code. In addition, we get the benefit of the reverse accumulation

[1] developed at the institute *Software and Tools for Computational Engineering* at RWTH Aachen University implementing AD by overloading in Fortran [12]

technique [1] for the fixed point iterative construction in the primal code that carries out the nonlinear iterations for solving the momentum and mass conservation equations. This implementation shows that the derivatives converge with a lower number of iterations compared to the primal code, yielding a significant improvement in performance. We verify the sensitivity results by finite differences for a medium size vehicle climate duct test case that is presented in [13]. The performance comparison results for different sizes of test cases prove the efficiency and robustness of `dco/fortran` for calculating derivatives. It provides an easy to use implementation generating accurate and detailed sensitivities for shape optimization problems in industrial CFD applications.

1.1 CFD Simulation

GPDE solver is an incompressible, steady-state flow solver with cell-centered storage, face-based residual assembly; it works on unstructured meshes with collocated variables and uses the SIMPLE [15] pressure correction algorithm in a pseudo time stepping scheme towards a steady solution. It is written in Fortran 90-95 (7,000 lines) as a test-bed for developing adjoint Navier-Stokes fields and is specifically designed to facilitate interfacing with optimization libraries. The case study that is used for flow model simulation and sensitivity studies is the S-bend channel flow case which is a simplified vehicle climatisation duct. Test case is carried out at Reynold number 500 on the hexahedral mesh for three different sizes of 47000, 130,000, and 500,000 elements. The boundary condition is defined as uniform flow at inlet. The simulation in GPDE is performed for 397 outer iterations of the primal code until we attain convergence with the tolerance of $1.0E-08$ for velocity reduction. The geometry and velocity vector fields along the channel are shown in Figure 1.

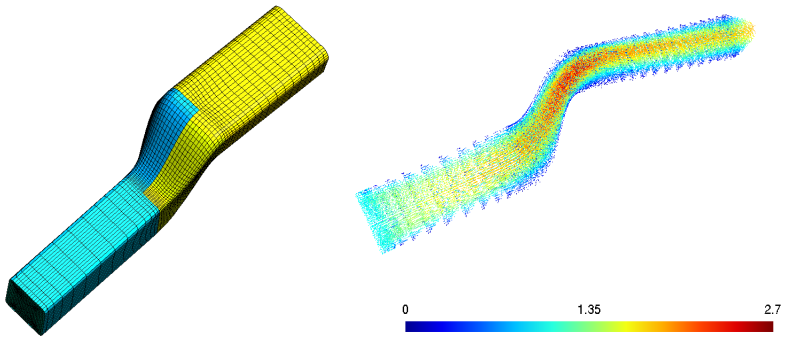

Fig. 1 Geometry and velocity vector field of the S-bend channel flow

2 Development Of Incompressible Adjoint Solver Using `dco/fortran`

The AD tool `dco/fortran` uses the operator overloading technique of the programming language written in Fortran to calculate the derivatives of a software. For given implementation of the primal function in Equation 1, the function $F^{(1)}$: $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m \times \mathbb{R}^m$, defined as:

$$(\mathbf{y}, \mathbf{y}^{(1)}) = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}), \tag{2}$$

where $\mathbf{y}^{(1)} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$, is referred to as the *tangent* model of F. The directional derivatives $\mathbf{y}^{(1)}$ are computed in direction of $\mathbf{x}^{(1)} \in \mathbb{R}^n$ at the current point $\mathbf{x} \in \mathbb{R}^n$. The function $F_{(1)} : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n \times \mathbb{R}^m$, defined as

$$(\mathbf{y}, \mathbf{x}_{(1)}) = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}), \tag{3}$$

is referred to as the *adjoint* model of F. The adjoint model implementation yields the objective $\mathbf{y}$ and the product $\mathbf{x}_{(1)} \equiv \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$ of its gradient at the current point $\mathbf{x} \in \mathbb{R}^n$. For the purpose of our CFD optimization, we applied both models by using `dco/fortran` [2], however the adjoint model is preferred since a large number of inputs (n) are mapped to a rather small amount of outputs $(m = 1)$.

2.1 Black-box Adjoint Approach

For applying `dco/fortran` to the primal code, we need to have access to its source. Moreover, we need to distinguish the design variables and objective function in the CFD routine of the primal code. The primal run time with passive variables is 3.32 min. This run time is obtained for convergence tolerance of $1.E-8$ yielding 397 iterations of our iterative solver for the S-Bend test case. By overloading data types and function (*activation*) from passive type (real) to active type (tangent or adjoint), the run time and memory usage increase. The primal run time with active data type using `dco/fortran` is 5.92 min (without tape generation) which is factor 1.78 compared to passive type. It becomes more costly during taping process for the iterative solver. By black-box approach we mean we get the sensitivities without further exploiting the structure of the code or implementing the techniques to improve the performance of the differentiated code. The objective function in GPDE flow solver is defined as pressure loss to get the *surface mesh sensitivities* with respect to corresponding independent design variables such as surface node coordinates. These sensitivities are verified with finite differences in next section. However the resulting memory usage of the black-box adjoint approach is not acceptable for real world problems. Measures need to be taken to reduce the computational cost.

Table 1 Gradient comparison of finite difference and adjoint-tangent codes using *dco/fortran* AD tools

	max gradient dir-1	max gradient dir-2	function value
primal	− − −	− − −	6.8847553685898450
tangent *dco*	0.0002714178530824**02**	−0.0005045825549462**3**	6.8847553685898450
adjoint *dco*	0.0002714178530823**83**	−0.0005045825549461**0**	6.8847553685898450
finite difference	0.000271420868003247	−0.00050458552891541	6.8847553685898450

2.2 Verification of Adjoint and Tangent Codes with Finite Difference

To verify the results of adjoint code, the numerical first order sensitivity results using `dco/fortran` for black-box approach are compared with finite difference method along desired lines of geometry that determined in Figure 1.a. Due to the memory restriction for the black-box adjoint code (100GB), we limit the number of iterations to 10 time steps. In the finite difference method the derivatives of functions are approximated by differences in the values of the solution between a given value of the input variable and a small increment h. The value of h can be determined by smallest relative error between the derivatives of adjoint code and finite difference as presented in Figure 2.

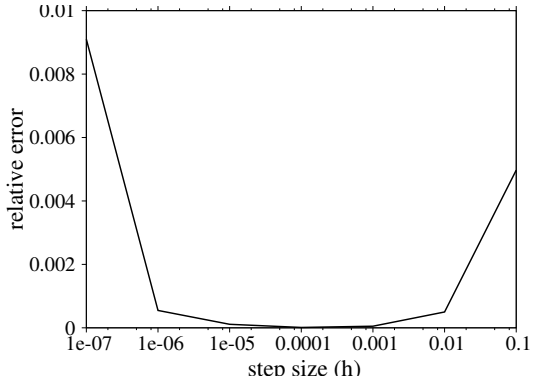

Fig. 2 Selecting step size increment h for finite difference comparison. Relative error is the smallest value between the derivatives of adjoint code and finite difference.

Table 1 shows for overloading approach, results of forward and reverse adjoint sensitivities in each direction match perfectly up to machine precision that is accurate up to tolerance 1.0E-7 for finite difference method. The pressure function values is the same with primal output.

3 Efficient Adjoint CFD Code

Simplicity of implementation of the adjoint method by overloading tool comes with a price in taping process. In a CFD solver, all inner iterations for linear solver and outer iterations for calculating the velocity and pressure have to be taped. The needed size of this tape grows dynamically proportional to the number of operations in the calculation that leads to considerable memory consumption.

Our solution to this problem is to differentiate linear solver symbolically and to employ a checkpointing scheme for the outer pseudo time stepping loop [2]. The symbolic differentiation of linear solver is called from the tape during interpretation process. This will dramatically decreases the memory consumption of the adjoint solver and has a significant effect on execution speed up as it presented in Figure 4.

3.1 Checkpointing

The basic idea of checkpointing is to split the entire program into several sequential blocks of operations whose computational graphs of each fit into the available memory [5]. In case of an iterative solver these blocks consist of a certain number of iterations. These blocks are then taped and interpreted one at a time to produce the resulting adjoint values of the complete computation.

Here we briefly overview the functionality and the performance cost of the equidistant checkpointing scheme using `dco/fortran` and later we replace the checkpointing algorithm by the binomial checkpointing strategy using `REVOLVE`.

3.1.1 Equidistant Checkpointing

Rather than storing and taping every variable and operation during the forward run, checkpoints are stored at strategically chosen intervals q for n_chk (number of checkpoint steps) where $n\_chk = n\_iter/q$ and n_iter is the total number of iterations. The maximum size of interval q is limited to the available memory.
In equidistant checkpointing for the CFD solver, when the number of checkpoints decreases, the memory usage of computer grows dramatically. Figure 3 shows the growing memory requirement of the adjoint code using equidistant checkpointing by increasing the size of the checkpoint interval. The huge memory consumption limits the size of checkpoints interval to 30 iteration for S-bend test case which results in total number of 13 checkpoints for 397 iteration.

3.1.2 Binomial Checkpointing-`REVOLVE`

The `REVOLVE` algorithm, introduced by Griewank et al[6], provides an optimal checkpointing schedule for a prescribed number of n_{cp} checkpoint slots. It always

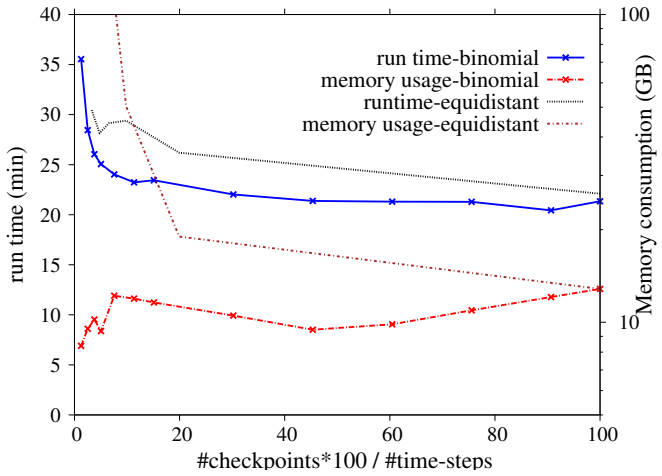

Fig. 3 Memory-run time trade-off for binomial-equidistant checkpointing using `REVOLVE` and `dco/fortran`. Checkpoint are stored and restored from the tape.

tapes only one time step and minimizes the amount of extra forward steps needed by setting checkpoints in a binomial fashion. This method yields logarithmic grows in spatial complexity. It works through the following four stages:

- 0. Initialisation: memory space is reserved for q checkpoints, the first being the initial state;
- 1. Multiple Forward: starting from the last checkpoint assigned, computation is advanced to the penultimate state without taping intermediate values; if any checkpoint slots are free, as many of them as possible are set to intermediate states along the way (snapshots);
- 2. Combined Reverse: a forward sweep is performed with taping of intermediate values from the penultimate state to the currently final state. On the same segment a reverse step is then performed to compute the adjoints. This next-to-last state now becomes the final state, and if it corresponds to the last checkpoint set then it is freed up for subsequent reassignment;
- 3. Termination: stage 1 and 2 are repeated until all checkpoints are freed up, in which case the procedure has reached regular termination.

Figure 3 displays the run time and memory usage for the adjoint calculation as a function of the number of checkpoints for overall fixed number of time steps (here 397). For this comparison, the memory that is required for store and restoring variable values for the adjoint sweep is referred to the tape. The alternative approach is to write and read the checkpoints values to and from the disk.

As it can be seen form the graph, in binomial checkpointing decreasing the number of snapshots (checkpoints) has a significant increase impact on run time. Unfortunately, this procedure is always a trade-off between memory consumption and run time. The memory consumption range between 7 to 15 GB that shows a noticable

reduction compared to equidistant checkpointing. If we checkpoint all the steps of the iteration, binomial checkpointing and equidistant checkpointing are equivalent in terms of performance. However binomial checkpointing is a preferred checkpointing scheme for higher number of iteration since both memory consumption and slowdown factor grow only like log(n_iter).

3.2 Hybrid Overloading- Source Transformation Approaches

GPDE is alternatively differentiated by the source transformation tool, `TAPENADE`[8]. This approach reduces the memory requirement and it is easier for the compiler to perform compile time optimizations. However in terms of ease of implementation, ability to handle arbitrary functions and less changing the original source code operator overloading provides the differentiated code with a greater flexibility and robustness in comparison with AD by source transformation. Therefore coupling these two AD tools remains an efficient approach to decrease on one hand the development time of differentiated code and in the other hand to reduce the memory requirement of the adjoint code. The implementation steps for applying `TAPENADE` AD tool via a `dco/fortran` tape is explained in Figure 5.

The idea is to extract the computationally most expensive part of the adjoint code and differentiate it by `TAPENADE`. The adjoints generated by `TAPENADE` is imported to dco tape. By using gprof for the GPDE adjoint code, the `gradient` function is determined for taking most of the execution time. This function is responsible for calculating `div` term in momentum and pressure equations[2]. Instead of recording the operations of such a function inside the forward run, the differentiated procedure by `TAPENADE` is registered as the external function to the tape and executed in the reverse run of the adjoint code. Figure 4 presents the improvement in performance by coupling two AD approaches additionally to the symbolic differentiating treatment of linear solver.

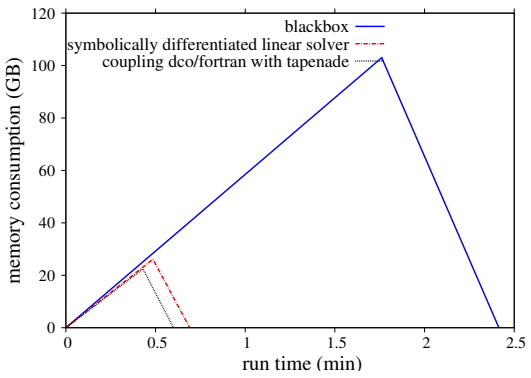

Fig. 4 Performance comparison for 10 iterations; There is significant effect memory usage reduction and execution speed up by coupling two AD approaches additionally to the symbolic differentiating treatment of linear solver.

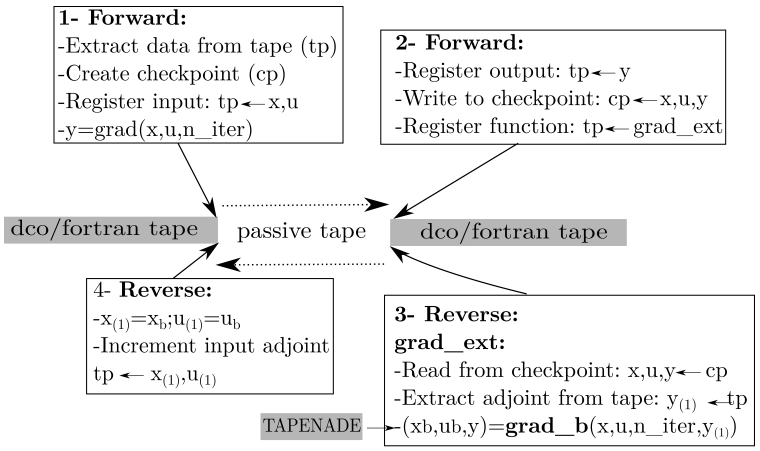

Fig. 5 Calling `TAPENADE` from `dco/fortran` Tape

3.3 Reverse Accumulation for Fixed Point Loop

The fixed point iterative loop in CFD code carries out outer iterations for solving the momentum and mass conservation equations. The memory requirement of the reverse mode grows proportionally with the number of iterations. It is known that, if the fixed point construction converges to the correct value, then the reverse gradient construction converges at the same rate[1]. Therefore, the reverse accumulation technique for fixed point iterative construction in the CFD code is implemented. Instead of taping the whole evaluation sequence, the taping is started only for the last iteration. The reverse accumulation for the fixed point is performed by repeated tape interpretations of the last interval until the convergence of adjoint for state variables is reached.

By applying this technique we reduce memory consumption of the adjoint code independent of the number of iterations. The implementation of the method using `dco/fortran` to compute the gradients of fixed point is illustrated in Algorithm 1 and Figure 6.

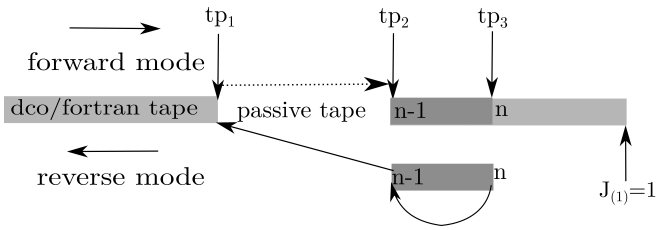

Fig. 6 Reverse accumulation during the taping process

Figure 7 indicates for a converged primal to a fixed point, considerably fewer number of iterations is needed for tape interpretation to converge the adjoints in reverse accumulation. The variation of sensitivity results for different numbers of tape interpretation iterations is compared with finite differences as the reference. It shows the convergence of adjoints is achieved by around 150 iterations compared to 397 iterations in the primal solver.

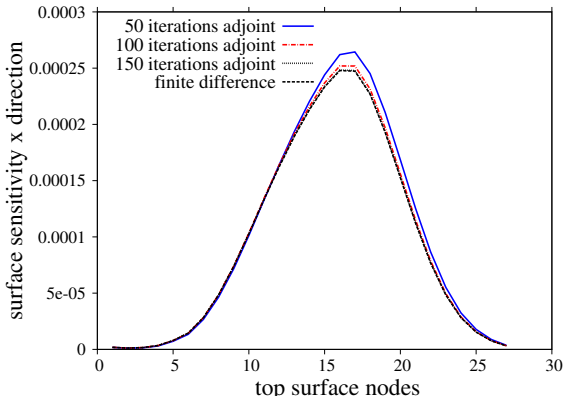

Fig. 7 Adjoint Convergence by increasing Reverse accumulation iterations when primal converges after 397 iterations

4 Numerical Results

Sensitivity analysis results are presented for the pressure loss objective function with respect to surface boundary nodes of the S-bend in three dimensions for 47000 elements test case. The first order sensitivities show the direction of surface change that leads to shape modification in the bend. Table 2 shows for the overloading approach results of surface sensitivity from different techniques including reverse accumulation, checkpointing, symbolically differentiated the linear solver and coupling `dco/fortran` with `TAPENADE` match accurately. The surface sensitivity results are verified with finite differences for the optimum perturbation $h = 10e-4$ which measure reliability of the adjoint code. These results are illustrated and compared along middle lines of the top and bottom surface geometry in Figures 9 and 10 and in a range of surface nodes in Figure 8.

4.1 Numerical Cost Comparison

The numerical sensitivity results for different performance improvement strategies are compared in Table 2. These results are obtained for the whole iteration process

Table 2 Gradient comparison of performance improvement strategies using `dco/fortran` and `TAPENADE`

strategy[a]	max sensitivity top surface	max sensitivity bottom surface
finite difference	0.0007771**60737447696**	0.0010119**8156135652**
binomial checkpointing 120 checkpoints	0.0007771**70733573492**	0.0010119**5422914152**
equidistant checkpointing[b] 397 checkpoints	0.0007771**70733573492**	0.0010119**5422914152**
reverse accumulation	0.0007771**71052073637**	0.0010119**5433254612**
reverse accumulation + hybrid dco/fortran-tapenade	0.0007771**71052073173**	0.0010119**5433254530**

[a] In all the methods linear solvers are symbolically differentiated
[b] Each step is checkpointed; q=1

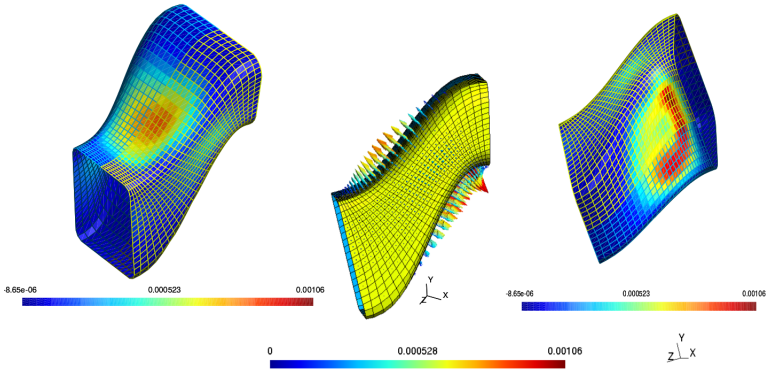

Fig. 8 Sensitivity results map for pressure loss objective function with respect to surface nodes

for the steady solver up to the convergence point of the flow solver. The memory usage of `REVOLVE` and equidistant checkpointing is significant compared to other AD strategies due to the growing memory usage of storing checkpoints for high number of iterations. To reduce storage requirement of taping, for this comparison the checkpoints values are written to and read from disk that yield around 20% decrease in memory consumption compared to storing checkpoints on the tape (Figure 3). There is a significant memory usage reduction and execution speed up by applying reverse accumulation. This performance slightly is improved by coupling two AD approaches.

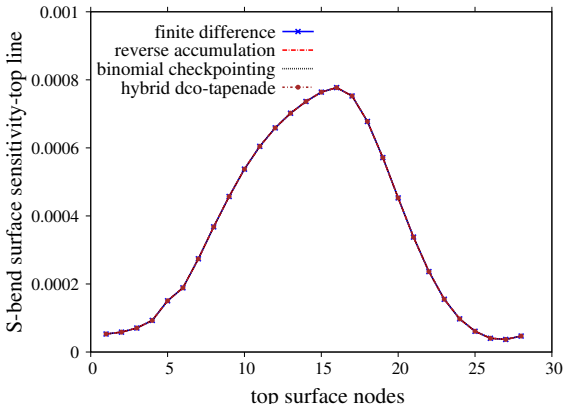

Fig. 9 Sensitivity analysis results comparison for pressure loss objective function with respect to top surface nodes

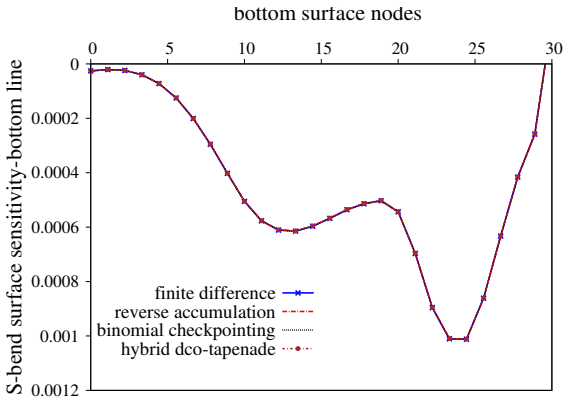

Fig. 10 Sensitivity analysis results comparison for pressure loss objective function with respect to bottom surface nodes

Table 3 Performance comparison of performance improvement strategies using `dco/fortran` and `TAPENADE`

strategy	run time adjoint(min)/primal(min)	memory (MB)
finite difference	$47k \times 3.32/3.32 = 47k$	143
binomial checkpointing 120 checkpoints	22.10/3.32=6.65	5917
equidistant checkpointing 397 checkpoints	19.25/3.32=5.79	9372
reverse accumulation	8.09/3.32=2.43	3607
reverse accumulation + hybrid dco/fortran-tapenade	6.76/3.32=2.03	3160

4.2 Performance Results for Different Size Test Cases

We present the results of comparison for test cases of different sizes in Figure 11. The number of elements are increased to 130,000 and 500,000 for case 2 and case 3 respectively. Table 4 indicates the constant run time ratio of adjoint to primal

code, however, the memory consumption increases. The memory cost can be reduced by distributing memory usage in a parallel adjoint code for big size industrial test cases [16].

	run time adjoint/primal	memory (MB)
case 1, 47K elements	$8.09/3.32 = 2.43$	3607
case 2, 130K elements	$24.04/10.19 = 2.36$	10443
case 3, 500k elements	$122.24/61.18 = 1.99$	40930

Table 4 Performance comparison of adjoint code for different sizes of S-Bend test case for 400 primal iterations using reverse accumulation

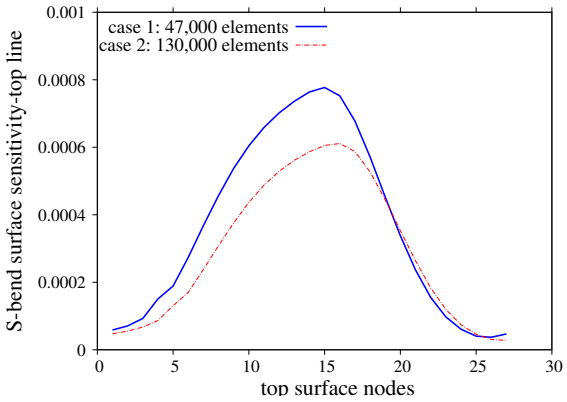

Fig. 11 Variation of top surface sensitivities for different size of test cases

5 Conclusion

Our approach in this paper is adjoint based shape optimization problem based on application of the AD tool to the CFD solver to get accurate sensitivity results. We apply efficient strategies to reduce numerical cost of the adjoint CFD code by implementation of checkpointing schemes and by symbolic differentiation of the iterative linear solver. For the fixed-point iterative construction in the CFD code in steady solver, we get the benefit of reverse accumulation techniques. Moreover, we improve the performance by coupling O-O with S-T tools in a hybrid approach. This technique can be more effective by a proper extract and import of expensive part of the CFD code via S-T tool. We demonstrate significant improvement in terms of memory consumption and run time of the resulting code for a sample CFD problem using the `dco/fortran` AD tool. The numerical derivatives and performance analysis show that `dco/fortran` is a reliable and efficient tool for calculating the sensi-

tivities for CFD code accurately. It provides the easy to use environment for shape optimization in industrial applications. We extend application of `dco/fortran` to generate the sensitivity algorithm for the commercial CFD solver ACE+ developed by ESI group and relevant industrial test cases.

Algorithm 1 Implementation of reverse accumulation using `dco/fortran`

In:
-implementation of the adjoint flow function $flow_{(1)}$ for computing the objective and the input adjoints $(\mathbf{x}_{(1)}, J) \equiv flow_{(1)}(flow\_iter(\mathbf{u},\mathbf{p}),\mathbf{x},J_{(1)})$.
-current (surface) geometry: $\mathbf{x}$
-initial flow state variables (velocity, pressure): $\mathbf{u}_{(init)}, \mathbf{p}_{(init)}$
Out:
-pressure loss objective: J
-surface sensitivity: $\mathbf{x}_{(1)}$
Algorithm:
create tape: tp
geometry connectivity for flow field:
$(\mathbf{u}_0, \mathbf{p}_0) = g(\mathbf{x}, \mathbf{u}_{(init)}, \mathbf{p}_{(init)})$
$tp_1 \leftarrow$ get_tape_position
switch_tape_to_passive()
do $i = 1, n$
if $i = n-1$ **then**
  switch_tape_to_active()
  $tp_2 \leftarrow$ get_tape_position
else
  $(\mathbf{u}_i, \mathbf{p}_i) = flow\_iter(\mathbf{u}_{i-1}, \mathbf{p}_{i-1}, n)$
end
end do
$tp_3 \leftarrow$ get_tape_position
$J \equiv obj(\mathbf{u}_n, \mathbf{p}_n)$
$J_{(1)} \leftarrow 1$
interpret_adjoint_to(tp_3)
$(\mathbf{u}_{n(1)}, \mathbf{p}_{n(1)}, J) \equiv obj_{(1)}(\mathbf{u}_n, \mathbf{p}_n, J_{(1)})$
$(\mathbf{u}^0_{n-1(1)}, \mathbf{p}^0_{n-1(1)}) \leftarrow (\mathbf{u}_{n(1)}, \mathbf{p}_{n(1)}, J)$
while $i < n.and.\xi < \varepsilon$ **do**
  $(\mathbf{u}^i_{n(1)}, \mathbf{p}^i_{n(1)}) \leftarrow (\mathbf{u}^{i-1}_{n-1(1)}, \mathbf{p}^{i-1}_{n-1(1)})$
  $\mathbf{u}^*_{(1)}, \mathbf{p}^*_{(1)} \leftarrow \mathbf{u}^i_{n(1)}, \mathbf{p}^i_{n(1)}$
  ! interpret_adjoint_from_to(tp_3, tp_2)
  $(\mathbf{u}_n, \mathbf{p}_n, \mathbf{u}^i_{n-1(1)}, \mathbf{p}^i_{n-1(1)})$
  $\equiv flow\_iter_{(1)}(\mathbf{u}_{n-1}, \mathbf{p}_{n-1}, \mathbf{u}^i_{n(1)}, \mathbf{p}^i_{n(1)})$
  $\xi = max(\mathbf{u}^i_{n-1(1)} - \mathbf{u}^*, \mathbf{p}^i_{n-1(1)} - \mathbf{p}^*)$
end
$\mathbf{u}_{0(1)}, \mathbf{p}_{0(1)} \leftarrow \mathbf{u}^*_{(1)}, \mathbf{p}^*_{(1)}$
! interpret_adjoint_from($tp_{(1)}$)
$\mathbf{x}_{(1)} \equiv g_{(1)}(\mathbf{x}, \mathbf{u}_{0(1)}, \mathbf{p}_{0(1)})$

Acknowledgements The presented research is supported by the project **aboutFlow**, funded by the European Commission under grant no.FP7-PEOPLE-2012-ITN-317006.

References

1. B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.
2. Z. Dastouri, J. Lotz, and U. Naumann. Development of a discrete adjoint cfd code using algorithmic differentiation by operator overloading. In PM. Papadrakakis, M.G. Karlaftis, and N.D. Lagaros, editors, *OPT-i: An International Conference on Engineering and Applied Sciences Optimization*, Athens, 2014. National Technical University of Athens.
3. J.H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, 2002.
4. M.B. Giles, M.C. Duta, J.D. Muller, and N.A. Pierce. Algorithm developments for discrete adjoint methods. *AIAA Journal*, 41(2):198–205, Feb 2003.
5. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:3554, 1992.
6. 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*, 26(1):19–45, March 2000.
7. A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, Jan 2000.
8. L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.
9. A. Jameson, L. Martinelli, and N.A. Pierce. Optimum aerodynamic design using the navier-stokes equations. *Theor. Comp. Fluid. Dyn.*, 10:213–237, 1998.
10. D. Jones, F. Christakopoulos, and J-D. Mller. Preparation and assembly of adjoint cfd codes. *Computers and Fluids*, 46(1):282286, July 2011.
11. S. Kammerer, M. Paffrath, U. Wever, and A.R. Jung. Three-dimensional optimization of turbomachinery bladings using sensitivity analysis. In *Proceedings of ASME Turbo Expo 2003 Power for Land, Sea, and Air*, Georgia, USA, 2003. ASME.
12. U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. SIAM, Philadelphia, 2012.
13. C. Othmer and T. Grahs. Approaches to fluid dynamic optimization in the car development process. In R. Schilling et. al., editor, *Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, Munich, 2005. FLM.
14. C. Othmer, T. Kaminski, and R. Giering. Computation of topological sensitivities in fluid dynamics: Cost function versatility. In P. Wesseling, E. Onate, and J. Periaux, editors, *Eccomas CFD*, Delft, 2006. TU Delft.
15. S.V. Patankar and D.B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal for Heat Mass Transfer*, 15(10):1787–1806, 1972.
16. M. Schanen, U. Naumann, L. Hascoët, and J. Utke. Interpretative adjoints for numerical simulation codes using mpi. In *Proceedings of the 10th International Conference on Computational Science, ICCS'2010*, Amsterdam, 2010.