INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2005; **47**:939–945
Published online 27 January 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/fld.885

On the performance of discrete adjoint CFD codes using automatic differentiation

J.-D. Müller and P. Cusdin[*,†]

School of Aeronautical Engineering, Queen's University, Belfast, BT9 5AG, U.K.

SUMMARY

Adjoint methods are a computationally inexpensive way of deriving sensitivity information where there are fewer dependent (cost) variables than there are independent (input) variables. Automatic differentiation (AD) software makes it possible to create discrete adjoint codes with minimal human effort, an issue that had previously restricted acceptance of adjoint CFD codes. In terms of computational performance, automatic code is often assumed to be inferior to hand code. The structure of the underlying code is critical to the performance of the transformed code. This paper reviews the implementation of AD on Fortran CFD codes and gives details of how small rearrangements can be used to produce competitive tangent and adjoint code using source transformation AD. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: discrete adjoint; CFD; automatic differentiation; adifor; TAF; TAMC; Tapenade

1. INTRODUCTION

Sensitivity information in computational fluid dynamics (CFD) is useful for a number of purposes, including design optimization, mesh adaptation and flow control. In these examples, there are usually many more input parameters (e.g. design variables) than there are cost functionals (e.g. lift, drag) and adjoint methods compute the sensitivity information at a much lower computational cost than conventional state-gradient-based methods such as a tangent linearization or a finite difference.

Adjoint methods were introduced to aerodynamic design by Pironneau [1] and extended to whole-aircraft configurations by Jameson [2]. These early implementations made use of the continuous approach [3]. In the alternative discrete approach [4, 5], the adjoint code is derived via the chain-rule directly from the primal code. This simplifies many of the implementation issues (e.g. weak adjoint boundary conditions) and, since the system eigenvalues of the primal, linear and adjoint codes are identical, convergence of the primal solution guarantees

[*]Correspondence to: P. Cusdin, School of Aeronautical Engineering, Queen's University Belfast, BT9 5AG, U.K.
[†]E-mail: p.cusdin@qub.ac.uk

Received 27 April 2004
Revised 24 November 2004
Accepted 25 November 2004

convergence of the linear and adjoint solutions [6]. On the other hand, the task of applying the chain-rule to a long primal code is very tedious and error-prone.

Automatic differentiation (AD) [7] can be used to perform much of the primal code differentiation and has the potential to allow a more widespread employment of adjoint methods. Its use on CFD code is described in, e.g. Reference [8]. Unfortunately, automatic code is often thought of as inferior to hand-written code because it cannot exploit some of the efficiency measures that could be made manually. This paper contributes to the progress of adjoint CFD codes by showing how the latest AD tools can be used to obtain tangent and adjoint CFD code that is equivalent to hand-written code in terms of computational performance. Four major source transformation packages (Adifor [9], TAF/TAMC [10] and Tapenade [11]) are applied to 2D Euler and 3D Navier–Stokes CFD codes and compared to hand-written code. Typical performance impediments of AD are outlined, as well as the steps that can be taken to avoid them.

2. DISCRETE ADJOINTS

AD is used to derive tangent and adjoint CFD code using the primal flow solver as its input. The following analysis is based on an explicit finite volume method for the Euler and Navier–Stokes equations. The primal equations can be written with local timestepping for convergence to the steady state as

$$\frac{\partial W}{\partial t} + R(W) = 0 \tag{1}$$

where W is the vector of flow variables and the residual R incorporates the spatial discretization terms.

The sensitivity of a cost functional $L(W,\alpha)$ with respect to α is

$$\frac{\mathrm{d}L}{\mathrm{d}\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial W}u = \frac{\partial L}{\partial \alpha} + g^{\mathrm{T}}u \tag{2}$$

where α is some design parameter (e.g. angle of attack). The derivatives $\partial L/\partial\alpha$ and $g=(\partial L/\partial W)^{\mathrm{T}}$ are easy to compute [12]. The flow perturbation, $u=\partial W/\partial\alpha$, is derived from the tangent linearization for a steady solution of Equation (1):

$$R(W,\alpha)=0, \quad \frac{\partial R}{\partial W}\frac{\partial W}{\partial \alpha} = -\frac{\partial R}{\partial \alpha}, \quad \mathbf{A}u = f \tag{3}$$

in which $\mathbf{A}$ is the Jacobian of the residual vector, $\mathbf{A}=\partial R/\partial W$, and f is the sensitivity of the residual with respect to the design parameter, $f=-\partial R/\partial\alpha$. The tangent linear system is solved with an iterative loop in a manner similar to that for the primal code. In tangent mode, AD produces code to compute the product $\mathbf{A}u$, given the primal residual routine as its input.

The adjoint problem is defined as

$$\left(\frac{\partial \mathrm{R}}{\partial W}\right)^{\mathrm{T}} v = \left(\frac{\partial L}{\partial W}\right)^{\mathrm{T}}, \quad \mathbf{A}^{\mathrm{T}}v = g \tag{4}$$

in which $v = \partial L / \partial R$ and can be interpreted as a Lagrange multiplier [13]. The adjoint system is again solved with an algorithm similar to the primal, where the timestep reverses to take account of the adjoint direction [14].

The adjoint sensitivity is shown to be identical to the tangent method by combining Equations (2) and (3)

$$\frac{\mathrm{d}L}{\mathrm{d}\alpha} = \frac{\partial L}{\partial \alpha} - g^{\mathrm{T}} \mathbf{A}^{-1} f \tag{5}$$

where the associativity property of matrix multiplication is exploited such that one can choose to compute $g^{\mathrm{T}}\mathbf{A}^{-1}f$ either by evaluating $\mathbf{A}^{-1}f$ and then multiplying by g^{T} (tangent method, Equation (3)) or evaluating $g^{\mathrm{T}}\mathbf{A}^{-1}$ and then multiplying by f (adjoint method, Equation (4)). AD is used in reverse mode to derive the code for $\mathbf{A}^{\mathrm{T}}v$.

As the number of design variables usually exceeds the number of cost functionals by a significant margin, each new design variable (which implies a new f) requires another tangent linear solve, whereas the expense of the adjoint method remains almost constant for a fixed number of functionals.

Implementation issues. AD can be used on codes of any length [15]. Some preparation is usually required, e.g. to remove non-standard constructs. Weak boundary conditions can be treated automatically by the AD software. Treatment of strong boundary conditions and validation are discussed in Reference [3]. The source terms f and g are also derived using AD.

3. PERFORMANCE RESULTS

AD transformations (i.e. the tangent or adjoint code that is generated by source-to-source AD) would ideally deliver a computational performance that is equal to or better than that of hand-written sensitivity code. Unfortunately, one is not always able to exploit efficiency measures of adjoint and tangent codes when using AD in the way that is possible when writing the code by hand.

There are already several algorithms used by AD to improve the quality of the data dependency analysis for recomputation [16] and storage [17] of required variables in reverse-mode transformations. However, it is possible to identify particular constructs in the primal that result in transformations where the runtime and memory demand of the sensitivity code is significantly increased compared to hand coding, despite the fact that the specific constructs have minimal impact upon the runtime of the primal itself. The unfavourable primal constructs can usually be re-written in a way that avoids the poor transformation. In other cases, it is possible to edit the transformed code to alter or remove parts that are not a direct consequence of the construction of the primal, an ‘inadmissible’ rearrangement (because the derived code is no longer truly automatic). This inadmissible modification could be made with a post-processing script or included as an option in the AD software. It is possible for the AD tools to be modified by the developers and, by quantifying the performance improvement from each rearrangement, it is possible to prioritize the most profitable modifications.

Code performance is measured on three platforms. The first is a 1.5 GHz Intel Itanium 2 machine (3 MB cache), where the code is compiled using the HP f90 compiler (V 2.7.2). The second is a 2.4 GHz Intel Xeon (512 KB cache) and third a 1.2 GHz AMD Athlon machine

Table I. The absolute runtime (seconds) for the primal and the relative runtimes for the sensitivity code. Light figures refer to the untuned times and bold figures to the tuned times.

	2D code						3D code					
	Itanium 2		Xeon		Athlon		Itanium 2		Xeon		Athlon	
Primal (runtime)	37.5	**33.9**	49.9	**47.2**	65.2	**64.8**	104.4	**108.9**	315.0	**310.4**	521.8	**516.4**
Tangent (ratio)												
Hand	2.43		2.33		2.24		1.88		1.20		1.13	
Adifor 2	2.12	**2.12**	2.62	**2.52**	2.43	**2.39**	4.59	**3.84**	1.71	**1.62**	1.68	**1.66**
TAF	4.54	**2.27**	2.99	**2.23**	2.86	**2.10**	4.35	**3.83**	1.25	**1.21**	1.18	**1.17**
Tapenade	1.87	**1.96**	2.58	**2.32**	2.22	**2.17**	4.14	**3.97**	2.14	**2.10**	1.89	**1.87**
Adjoint (ratio)												
Hand	2.24		2.65		2.35		1.73		1.19		1.16	
TAF	8.14	**2.46**	4.09	**2.92**	3.77	**2.65**	3.97	**3.62**	1.25	**1.21**	1.26	**1.22**
Tapenade	11.53	**8.28**	12.02	**3.79**	8.82	**3.14**	28.03	**3.41**	11.64	**1.99**	10.16	**1.62**

(256 KB cache). The code for the second and third platforms is compiled using the Intel Fortran Compiler (V 7.1). The highest level of compiler optimization is used for both the primal and transformed code unless otherwise stated. Version numbers of the AD tools are as follows: Adifor 2.0D, TAF 1.4.20, TAMC 5.3.2 and Tapenade 2.0.4. Where appropriate, the AD tools are configured to produce the most efficient code possible.

Runtime performance is measured by looping over the primal, tangent and adjoint fluxes of 2D and 3D CFD codes. The 2D flux comprises an inviscid flux calculation with Roe dissipation [18]. The 3D code comprises an inviscid flux with a Spalart–Allmaras turbulence calculation [19]. The subroutine inputs are filled with random data and verified to machine-level precision. The time was recorded using the Fortran utility outlined in Reference [20] and averaged over six independent executions.

The results in Table I show that the performance of AD generated code is highly sensitive to small changes in the primal. Without any rearrangements, tangent mode transformations are usually competitive with the hand code on the Xeon and Athlon processors. Runtimes are more varied on the Itanium 2, where Tapenade generally outperforms Adifor and TAF. On the 2D code, Tapenade code even outperforms hand-written code with a ratio of 1.87 compared to 2.43 for hand code.

Further investigation into the poor performance of the automatic 3D code on the Itanium 2 processor suggests a cache-related issue. When the prefetch option on the HP compiler is switched off (it is on by default with the highest optimization setting), runtimes of the hand-written codes are increased significantly. The primal code, for example, increased from 109 to 253 s. Runtimes of the automatic code, however, are unaffected. Ratios of the AD code therefore decrease correspondingly and are similar to those from the Xeon and Athlon processors. The ratio of tangent mode Tapenade, for example, decreases from 3.97 to 1.70.

On the other hand, reverse mode transformations are disappointing, particularly on the 2D code (all platforms/all AD) and 3D code produced by Tapenade. (Note that Adifor 2 does not support reverse mode.) The deficits compared to hand adjoint code can be reduced with code tuning. For TAMC/TAF, the most important rearrangement is to replace all non-continuous intrinsic functions (e.g. `min`, `max` and `abs`) with an appropriate `if..then..else` construct (Table II). Without the rearrangement, TAF and TAMC use the `sign` intrinsic in order to

Table II. Replacing a `max` with an `if..then..else` construct for TAF/TAMC.

Original	Modified
`x = max( a, b )`	`if ( a > b ) then`
	`    x = a`
	`else`
	`    x = b`
	`end if`

Table III. Replacing an `if` statement with a full `if..then..else` construct to reduce the number of store/recall calls in Tapenade code.

Original	Modified
`x = a`	`if ( b > c ) then`
`if ( b > c ) x = b`	`    x = b`
	`else`
	`    x = a`
	`end if`

Table IV. Reducing overwritten variables in a `do` loop for reverse mode Tapenade.

Original	Modified
`subroutine TOP (x, y)`	`subroutine NEW_OUTER (x, y)`
`do nNode = 1, mNode`	`do nNode = 1, mNode`
`    :`	`    call NEW_TOP (x(nNode), y(nNode))`
`    y(nNode) = a * x(nNode)`	`end do`
`end do`	
	`subroutine NEW_TOP (x, y)`
	`:`
	`y = a * x`

evaluate the derivative, exacerbating the runtime penalty of using a non-continuous intrinsic function. In one case, a runtime improvement of 76% is achieved. TAF (from version 1.4.28) now inserts this replacement automatically, a manual alteration is still necessary for TAMC. The 2D code is particularly sensitive to this rearrangement because of the high concentration of `min`, `max` and `abs` functions (7 intrinsics within 140 lines of code) compared to the 3D code.

Reverse mode Tapenade code is improved by avoiding as many of the store/recall functions (used by its storage approach) as possible. Several different rearrangements of the primal can be used to achieve this, all of them based upon a strategy of eliminating overwritten variables in the code that is submitted to Tapenade. In some cases, the overwritten variables are simple to spot, e.g. if a variable `tmp` is re-used, replace each use with `tmp1`, `tmp2`.... There are other short examples where an overwritten variable is not so obvious (e.g. an `if` statement, see Table III). Another more general rearrangement is to move a `do..end do` loop outside of the code that is submitted to Tapenade (see Table IV). This results in a major improvement in runtime and memory performance for the 3D code. In this example, memory demand is

Table V. Performance improvements by not recomputing the dependent variables. The left columns refer to code that recomputes the dependent variables and the figures on the right are from the pure code. Figures in bold are differentiated automatically and figures in brackets are inadmissible.

	3D code					
	Itanium 2		Xeon		Athlon	
Adifor tangent (modified)	**4.59**	(4.49)	**1.71**	(1.19)	**1.68**	(1.16)
TAF tangent (`-pure`)	4.39	**4.35**	1.75	**1.25**	1.72	**1.18**
Tapenade tangent (modified)	**4.14**	(1.99)	**2.14**	(1.62)	**1.89**	(1.51)

Table VI. Runtime performance of a full 2D CFD code using original and tuned TAF tangent and adjoint fluxes on the Xeon processor.

	Full code		Fluxes	
Primal (runtime)	298.3		49.9	
Tangent (ratio)	2.54	**2.16**	2.99	**2.23**
Adjoint (ratio)	2.98	**2.64**	4.09	**2.92**

reduced by about 15%. The rearrangement in Table IV is possible because the `do` loop is self-adjoint: the order in which the primal, tangent or adjoint codes loop over the mesh is not important. With the original routine, Tapenade is obliged to store all overwritten variables in a forward sweep of the loop (i.e. 1→`mNode`) before reversing the loop (`mNode`→1). With the modified code, however, there is only one assignment for `y` and hence Tapenade does not invoke a store/recall.

A further reduction in runtime is achieved by replacing the generic store/recall function (written in c++) with a customized routine written in Fortran 77, enabling greater compiler optimization.

Pure differentiation. If the primal and adjoint codes are to be converged simultaneously, then the transformed routines must include the code to compute the dependent (output) primal variables. However, there are many cases where this is inappropriate. By default, all of the AD tools tested here include code for the recomputation of dependent variables. Only TAF and TAMC include an option (`-pure`) to exclude the dependent evaluations. Although the runtime penalty for this might at first seem small, it has a knock-on effect for the recomputation or storage of required variables. Table V shows the runtime difference for tangent mode transformations on the 3D code.

Full CFD code. In order to confirm that runtime improvements in the tangent and adjoint fluxes contribute significantly to an improvement in a full CFD code, the original and tuned TAF transformations were applied to the 2D code (Table VI). The results demonstrate the fact that the majority of runtime in a CFD code is spent in the flux calculation and the remainder of a tangent or adjoint CFD code (e.g. update routine) is very similar to the primal code.

4. CONCLUSIONS

Discrete tangent linear and adjoint codes can be derived manually and are efficient in terms of computational expense but extremely inefficient in terms of human effort. AD can be used to derive the sensitivity code with minimal human effort, but a computational penalty is usually expected. With careful preparation, following the basic rules outlined in this paper, it is possible to use AD to generate code that is computationally almost as efficient as hand code but with much less human effort. AD can then be used to propagate development of the primal through to the tangent and adjoint codes at almost no additional expense.

ACKNOWLEDGEMENTS

The developers of AD have been actively supportive of this work. In particular, we would like to thank Thomas Kaminski, Ralf Giering (FastOpt), Laurent Hascoët (INRIA) and Mike Fagan (Rice University).

REFERENCES

1. Pironneau O. On optimum design in fluid mechanics. *Journal of Fluid Mechanics* 1974; **64**:97–110.
2. Jameson A. Aerodynamic design via control theory. *Journal of Scientific Computing* 1988; **3**:233–260.
3. Giles MB, Duta MC, Müller J-D, Pierce NA. Algorithm developments for discrete adjoint methods. *AIAA Journal* 2003; **41**(2):198–205.
4. Anderson WK, Bonhaus DL. Airfoil design on unstructured grids for turbulent flows. *AIAA Journal* 1999; **37**(2):185–191.
5. Mohammadi B, Pironneau O. Mesh adaptation and automatic differentiation in a CAD-free framework for optimal shape design. *International Journal for Numerical Methods in Fluids* 1999; **30**(2):127–136.
6. Giles MB. On the use of Runge–Kutta time-marching and multigrid for the solution of steady adjoint equations. *NA Group Report NA-00/10*, Oxford University Computing Laboratory, 2000.
7. Griewank A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM: Philadelphia, PA, 2000.
8. Mohammadi B, Malé J-M, Rostaing-Schmidt N. Automatic differentiation in direct and reverse modes: application to optimum shapes design in fluid mechanics. In *Computational Differentiation: Techniques, Applications, and Tools*, Berz M, Bischof CH, Corliss GF, Griewank A (eds). SIAM: Philadelphia, PA, 1996; 309–318.
9. Bischof CH, Carle A, Khademi P, Mauer A. ADIFOR 2.0: automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering* 1996; **3**(3):18–32.
10. Giering R, Kaminski T. Recipes for adjoint code construction. *ACM Transactions of Mathematical Software* 1998; **24**(4):437–474.
11. The TAPENADE tutorial [http://www-sop.inria.fr/tropics/tapenade/tutorial.html].
12. Cusdin P. Automatic sensitivity code for computational fluid dynamics. *Ph.D. Thesis*, QUB School of Aeronautical Engineering, 2005.
13. Giles MB, Pierce NA. An introduction to the adjoint approach to design. *Technical Report No. 00/04*, Oxford University Computing Laboratory, Oxford, March 2000.
14. Elliott J, Peraire J. Practical 3D aerodynamic design and optimisation using unstructured meshes. *AIAA* 1997; **35**(9):1479–1485.
15. Giering R, Kaminski T. Using TAMC to generate efficient adjoint code: comparison of automatically generated code for evaluation of first and second order derivatives to hand written code from the Minpack-2 collection. In *Automatic Differentiation for Adjoint Code Generation*, Faure C (ed.). INRIA: Sophia Antipolis, France, 1998; 31–37.
16. Giering R, Kaminski T. Generating recomputations in reverse mode AD. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Corliss G, Faure C, Griewank A, Hascoët L, Naumann U (eds). Computer and Information Science, Chapter 34. Springer: New York, NY, 2001; 293–298.
17. Faure C, Naumann U. Minimizing the tape size. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Corliss G, Faure C, Griewank A, Hascoët L, Naumann U (eds). Computer and Information Science, Chapter 34. Springer: New York, NY, 2001; 293–298.
18. Roe PL. Approximate Riemann solvers, parameter vectors and difference schemes. *Journal of Computational Physics* 1981; **135**:250–258.
19. Spalart P, Allmaras S. A one-equation turbulence model for aerodynamic flows. *La Recherche Aerospatiale* 1994; **1**:5–21.
20. Cusdin P. A utility for timing Fortran code. *Technical Memorandum QUB-SAE-03-03*, QUB School of Aeronautical Engineering, Belfast, 2003.