INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2005; **47**:1315–1321
Published online 27 January 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/fld.927

Elimination AD applied to Jacobian assembly for an implicit compressible CFD solver

Mohamed Tadjouddine[1,*,†], Shaun A. Forth[1] and Ning Qin[2]

[1] *Engineering Systems Department, Cranfield University (Shrivenham Campus), ESD AMOR Group, Swindon SN68LA, U.K.*
[2] *Department of Mechanical Engineering, The University of Sheffield, Mappin Street, S1 3JD, U.K.*

SUMMARY

In CFD, Newton solvers have the attractive property of quadratic convergence but they require derivative information. An efficient way of computing derivatives is by *algorithmic differentiation* (AD) also known as *automatic differentiation* or *computational differentiation*. AD allows us to evaluate derivatives, usually at a cheap cost, without the truncation errors associated with finite-differencing. Recently, efficient and reliable AD tools for evaluating derivatives have been published. In this paper, we use some of the best AD tools currently available to build up the system Jacobian involved in the solution of a finite-volume parabolized Navier–Stokes (PNS) solver. Our aim is to direct scientists and engineers confronted with the calculation of derivatives to the use of AD and to highlight those AD tools that they should try. Moreover, we introduce an AD tool that produces Jacobian code that runs usually twice as fast as that from conventional AD tools. We further show that the use of AD increases the performance of a Newton-like solver for the PNS equations. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: PNS; Newton solver; vertex elimination algorithm; algorithmic differentiation

1. INTRODUCTION

The difficulty lies, not in the new ideas, but in escaping the old ones, which ramify, for those brought up as most of us have been, into every corner of our minds.
Keynes, John Maynard, 1883–1946.

This quotation can be applied to most new technologies. Techniques are invented, investigated, experimented, yet the majority of practitioners are too cautious to adopt them for some

*Correspondence to: Mohamed Tadjouddine, Engineering Systems Department, Cranfield University (Shrivenham Campus), ESD AMOR Group, Swindon SN68LA, U.K.
†E-mail: M.Tadjouddine@cranfield.ac.uk

Contract/grant sponsor: EPSRC; contract/grant number: GR/R85358/01

Received 27 April 2004
Revised 17 September 2004
Accepted 7 December 2004

reasons. Such is the case for using algorithmic differentiation (AD) [1] in computing derivatives of functions represented by computer programs.

Computing derivatives is ubiquitous in scientific computation and is essential to enable Newton solvers. In CFD, the calculation of steady compressible flow solutions reduces to the solution of a nonlinear system of equations of the form

$$\mathbf{R}(\mathbf{q}) = 0 \tag{1}$$

in which **R** represents the residual vectors in a finite-volume or finite-element calculation, and **q** represents the set of flow variables at each point of the mesh.

To solve this system, one can use the following two approaches:

- Explicit time marching based on Runge–Kutta solvers, with acceleration methods such as enthalpy damping, implicit residual smoothing and multigrid.
- Newton-like iteration methods necessarily involving a linear solver with coefficient matrix given by the system Jacobian or an approximation to it.

We focus on the second, which can be written as

$$\mathbf{q}_{n+1} = \mathbf{q}_n - \mathbf{P}(\mathbf{q}_n)\mathbf{R}(\mathbf{q}_n) \tag{2}$$

In (2), $\mathbf{P}(\mathbf{q}_n)$ may be an approximation to the inverse Jacobian $(\partial\mathbf{R}/\partial\mathbf{q})^{-1}$ and the resulting quasi-Newton algorithm has at best asymptotically linear convergence [2]. Alternatively, in the Newton–Raphson algorithm (exact Newton iteration), **P** is defined as the inverse of the Jacobian $P = (\partial\mathbf{R}/\partial\mathbf{q})^{-1}$ and (2) exhibits quadratic convergence. However, the Newton–Raphson iteration is limited by the amount of memory needed for industrial applications involving high resolution of multidimensional PDE simulations. Newton–Krylov (inexact Newton) solvers are of great interest as they are matrix-free solvers [3, 4]. They can save a great amount of work while having asymptotically quadratic convergence since they use Jacobian-vector or vector-Jacobian products instead of constructing the full Jacobian. However, successful Newton–Krylov methods depend on how well the resulting Newton–Krylov iteration is preconditioned. This preconditioning can be done using a linear combination of local Jacobian approximations [3]. In Reference [5], this is done using finite differences; however, one can use AD for accurate evaluation of these submatrices (local Jacobians).

Newton solvers require derivative information that can be obtained using hand-coding [6], computer algebra systems [7] such as MATHEMATICA or MAPLE, finite-differencing [5], or AD. AD is not symbolic manipulation as performed in computer algebra systems. It systematically augments the floating-point part of a computer program with extra instructions to calculate derivative values using the chain rule. In fact, it facilitates differentiation of functions represented by arbitrarily complex computer programs usually at a cheap cost without the truncation errors associated with finite-differencing.

The last decade witnessed an intense activity in AD tool development. Excellent AD tools that are efficient and reliable have been published. ADIFOR, TAF, TAMC, ADIC, and TAPENADE are well-established tools, which make use of the standard forward or reverse modes [1] of AD; see www.autodiff.org for more information. In this paper, we have used the ELIAD AD tool [8, 9], which uses the vertex elimination algorithm of Griewank and Reese [10] in a source transformation framework. Careful experiments showed that ELIAD produced Jacobian codes running 2–10 times faster than those by ADIFOR or TAMC [9]. In this paper, we detail the linearization of the Osher flux, which is complex as it contains nested branches and subroutine calls for which ELIAD uses a hierarchical approach to efficiently preaccumulate the Jacobian [11]. We show that the ELIAD-generated Jacobians are as fast as hand-coded Jacobians and are an order of magnitude faster than those generated by TAMC, ADIFOR or TAPENADE. Moreover, the associated Newton solver has near-identical convergence (in terms of number of iterations and CPU time) to that using laboriously hand-coded Jacobians.

2. THE IMPNS SOLVER

Implicit multigrid parabolized Navier–Stokes (IMPNS) [12] is a space-marching solver for the prediction of steady, supersonic and hypersonic flows around many objects of interest [13]. This solver can be used to compute inviscid, laminar or turbulent Navier–Stokes flowfields.

In the cross flow plane, inviscid and viscous fluxes are obtained with the Osher scheme [14] and central scheme, respectively. Streamwise fluxes use the Vigneron approximation in conservative form (CF) or Morrison–Korte form (MKF) [15]. IMPNS uses a combination of implicit (space-marching finite-volume scheme on a multiple-block structured grid and pseudo time-marching at each streamwise station) and multigrid methodologies for convergence acceleration. In the implicit schemes, the resulting linear system can be approximated with approximate factorization (AF) and solved exactly or solved approximately using the GMRES algorithm with an AF or an incomplete block LU factorization preconditioning.

3. THE ELIAD TOOL

ELIAD is an AD tool for a restricted class of Fortran programs motivated by application to numerical flux evaluation in CFD. Such subroutines typically have 10–100 inputs and outputs, some hundreds of intermediate values, branches and (assumed unrollable) loops. ELIAD uses the source transformation approach but unlike ADIFOR or TAMC, ELIAD employs the vertex elimination algorithm [10]. This algorithm views the Jacobian as a bipartite graph between output and input vertices, whereby a nonzero entry of the Jacobian is represented by an edge from an input vertex to an output vertex. The Jacobian can be calculated by eliminating, in some order, all intermediate vertices of the computational graph obtained from the input code. In matrix terms, this can be regarded as a Schur complement calculation or the solution of a linear system using some form of Gaussian elimination [1, 9].

ELIAD takes as input a source code, builds up its computational graph composed of input, intermediate and output vertices, and uses heuristics from sparse matrix technology such as the Markowitz strategy to find elimination sequences that reduce the number of operations required to accumulate the Jacobian. The elimination sequence is then used to generate the corresponding derivative code. The good news is that the ELIAD approach exploits sparsity of the Jacobian calculation and always requires less flops than the conventional AD forward and reverse modes as detailed in References [9, 10].

4. FLUX JACOBIAN CALCULATIONS

The IMPNS code contains flux routines coded as vector functions $\mathbf{F}$ with n independent variables and m dependents: Osher $(n=10, m=5)$, viscous $(n=10, m=5)$, and Vigneron

```
if (cond1) then
 call sub1(q1,f)
 if (cond2) then
  call sub2(q1,f1)
  f = f-f1
 else
  call sub3(q2,f1)
  f = f+f1
  ...
 endif
endif
```

$\Longrightarrow$

```
if (cond1) then
 call sub1_xc(q1,f,xc_f_q1)
 if (cond2) then
  call sub2_xc(q1,f1,xc_f1_q1)
  xc_f_q1 = xc_f_q1-xc_f1_q1
  xc_f_q2 = 0.d0
  f = f-f1
 else
  call sub3_xc(q2,f1,xc_f1_q2)
  xc_f_q2 = xc_f1_q2
  f = f+f1
  ...
endif
```

Figure 1. A code fragment (left) and its differentiated form (right).

$(n=5, m=5)$. These flux routines need to be linearized for the Newton solver as their Jacobians are calculated for each cell face of the finite-volume mesh at each Newton iteration. Typically, these codes contain nested branches and subroutine calls. A difficulty of the ELIAD approach lies in dealing with branches. This is done hierarchically and is described in Reference [11]. A nested branch similar to that found in our Osher code is given in Figure 1 on the left and its ELIAD-generated Jacobian code is on the right. The variables $q1$, $q2$, $f1$, and f are typically 4 arrays of length 5. The variables $q1$ and $q2$ are vectors of flow variables, f the flux and $f1$ the flux contribution to be added or subtracted to f. The variables `cond1` and `cond2` are boolean expressions whose value depends on local characteristic wave speed. The ELIAD-generated code on the right of Figure 1 is simplified for clarity using `Fortran 95` array operations. While ELIAD takes advantage of the structure of the code, building up the 5×10 flux Jacobian from the 5×5 local Jacobians of subroutines `sub1`, `sub2` and `sub3`, ADIFOR, TAMC, or TAPENADE's forward vector modes cannot do so and will propagate derivative vectors of length 10 throughout.

We generated Jacobian codes using various AD tools (ADIFOR, TAMC, TAPENADE, and ELIAD), ran the Jacobian codes using careful randomized data and compared the results with those of hand-coded Jacobians. Random pressures and densities in the range (0.5,1) were generated so that the resulting sound speeds would be realistic. Directional Mach numbers in the range $(-4,4)$ were used to compute velocities so as to take account of all branches in the Osher scheme. We performed 50 000 evaluations for each Jacobian calculation and average the CPU time on: a SUN Blade with 600 MHz CPU using the SUN Workshop compiler (Blade1000); a COMPAQ Alpha DS20E workstation with 667 MHz CPU using the COMPAQ f95 compiler (Alpha); and a Pentium III with 700 MHz CPU using COMPAQ Visual Fortran (PIII). Table I summarizes the ratio between the timings of the Jacobian and the original function. It shows that the ELIAD Jacobian code is relatively faster (up to 2 times) than conventional AD or finite differencing across platforms and occasionally outperforms hand-coded Jacobians.

5. RESULTS

To see the effect of our various Jacobian evaluation techniques within the IMPNS solver, we calculate the steady flow about a sharp ogive-nosed cylindrical body [13] with inflow

Table I. Efficiency of flux Jacobian calculations: ratio CPU($\mathbf{J}(\mathbf{F})$)/CPU($\mathbf{F}$).

	Osher flux			Viscous flux		
Technique	Blade1000	Alpha	PIII	Blade1000	Alpha	PIII
Finite difference	12.8	11.8	9.4	10.2	10.1	7.6
ADIFOR (forward)	11.8	26.6	4.9	5.9	6.3	5.6
TAMC (forward)	18.1	29.7	4.0	5.7	5.2	4.6
TAMC (reverse)	19.8	26.5	4.2	7.6	4.4	4.7
TAPENADE (forward)	18.5	19.3	4.3	7.5	5.8	5.2
ELIAD	10.6	7.1	3.2	4.8	4.1	3.3
HAND-CODED	4.3	8.6	2.1	4.4	4.4	3.5

	Vigneron flux:CF			Vigneron flux:MKF		
Technique	Blade1000	Alpha	PIII	Blade1000	Alpha	PIII
Finite difference	5.5	5.1	4.7	4.7	3.8	3.7
ADIFOR (forward)	6.4	4.2	4.8	4.5	2.6	3.2
TAMC (forward)	6.1	3.1	3.5	4.2	2.9	2.7
TAMC (reverse)	8.7	5.0	5.2	5.7	2.9	3.3
TAPENADE (forward)	5.4	3.0	4.3	3.8	2.6	3.0
ELIAD	3.8	2.3	3.8	3.2	2.1	2.3
HAND-CODED	2.6	2.0	2.6	1.9	1.4	1.8

conditions given by a Mach number of 2, Reynolds number 1.2×10^6 (based on body diameter) and using the Baldwin–Lomax turbulence model with Degani–Schiff corrections. A grid with 84 cells normal to the body, 72 circumferentially and 60 along the body was used. In Figure 2, we see computed cross-flow pitot pressure contours at a distance nine diameters downstream of the nose; note the boundary-layer separation and large vortex.

Table II gives overall solution statistics, total number of nonlinear iterations and CPU time, when we solve the 60 nonlinear systems associated with each cross-flow mesh using a 2-level multigrid acceleration of our Newton-like iteration with CFL number of 25 and linear solvers via GMRES with block incomplete LU decomposition with zero fill. Different rows of the table correspond to different techniques for generating flux Jacobian code. Runs are performed on different platforms Blade1000, Alpha, and PIII as in Section 4. Table II shows the number of iterations unchanged across platforms but, as might be expected, hand-coding gives the best overall performance, and though the ELIAD-generated results are only up to 2% slower while TAMC-generated Jacobians are up to 6% slower.

6. CONCLUSIONS

In this paper, we have shown that the system Jacobian used by the linear solver involved in a Newton solver can be obtained using AD to obtain flux Jacobians and assemble the system residual Jacobian at each Newton iteration. The resulting Jacobian code is nearly as efficient as hand-coding but for a fraction of the human effort, and gives comparable overall performance for the solution of the 3-D parabolized Navier–Stokes equations using the IMPNS code.

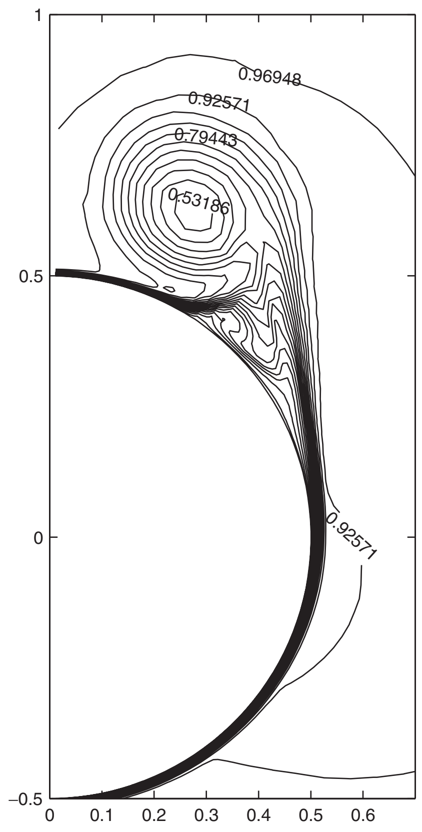

Figure 2. Cross-flow Pitot-pressures at nine diameters.

Table II. Solution statistics using BILU(0) preconditioning.

	Blade1000		Alpha		PIII	
Linearization	No. of iter.	CPU	No. of iter.	CPU	No. of iter.	CPU
TAMC (forward)	1634	768.6	1634	591.5	1634	1232.2
ELIAD	1634	730.6	1634	562.1	1634	1194.6
HAND-CODED	1636	721.5	1636	558.4	1636	1166.2

Algorithmic differentiation is a tool that is likely to be integrated in the next generation of computer software packages because of the ubiquitous requirement of derivatives in scientific computation. By presenting the use of AD in Newton solvers, we hope to provide some insights on what AD is capable of, raise awareness of AD by pointing out currently available tools, and reach out to engineers and practitioners in the wider area of computational science.

REFERENCES

1. Griewank A. *Evaluating Derivatives*: *Principles and Techniques of Algorithmic Differentiation*, Frontiers in Applied Mathematics. SIAM: Philadelphia, 2000.
2. Giles MB. On the iterative solution of adjoint equations. In *Automatic Differentiation*: *From Simulation to Optimization*, Corliss G *et al*. (eds), Computer and Information Science. Springer: Berlin, 2001; 141–147.
3. Knoll DA, Keyes DE. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *Journal of Computational Physics* 2004; **193**:357–397.
4. Qin N, Ludlow DK, Shaw ST. A matrix-free preconditioned Newton/GMRES method for unsteady Navier–Stokes solutions. *International Journal for Numerical Methods in Fluids* 2000; **33**(3):223–248.
5. Hovland PD, McInnes LC. Parallel simulation of compressible flow using automatic differentiation and PETSc. *Parallel Computing* 2001; **27**(4):503–519.
6. Venkatakrishnan V. Newton solution of inviscid and viscous problems. *AIAA Journal* 1989; **27**(7):885–891.
7. Vanden KJ, Orkwis PD. Comparison of numerical and analytical Jacobians. *AIAA Journal* 1996; **34**(6): 1125–1129.
8. Forth SA, Tadjouddine M. CFD Newton solvers with ELIAD, an elimination automatic differentiation tool. *Proceedings of the Second International Conference on CFD*. Springer: Sydney, 2003; 134–139.
9. Forth SA, Tadjouddine M, Pryce JD, Reid JK. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM TOMS* 2004; **30**(3):266–299.
10. Griewank A, Reese S. On the calculation of Jacobian matrices by the Markowitz rule. *Automatic Differentiation of Algorithms*: *Theory*, *Implementation*, *and Application*. SIAM: Philadelphia, 1991; 126–135.
11. Tadjouddine M, Forth SA, Pryce JD. Hierarchical automatic differentiation by vertex elimination and source transformation. In *Computational Science and its Applications—ICCSA 2003*, Kumar V *et al*. (eds), Lecture Notes in Computer Science, vol. 2668. Springer: Berlin, 2003; 115–124.
12. Ludlow DK. IMPNS User Manual. *CoA Report NFP-0113*, Cranfield University, Beds, UK, 2000.
13. Birch TJ, Qin N, Jin X. Computation of supersonic viscous flows around a slender body at incidence. *In the 12th Applied Aerodynamics Conference*, 2–22 June 1994, Colorado Springs, CO, *AIAA 94-1938*.
14. Osher S, Soloman F. Upwind difference schemes for hyperbolic systems of conservation laws. *Mathematics of Computation* 1982; **38**:339–374.
15. Morrison JH, Korte JJ. Implementation of Vigneron's streamwise pressure gradient approximation in the PNS equations. *AIAA Journal* 1992; **30**(11):2774–2776.