Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol II,
IMECS 2010, March 17 - 19, 2010, Hong Kong

On the Application of Algorithmic Differentiation to Newton Solvers

Emmanuel M. Tadjouddine [*]

Abstract— **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. Recent years witnessed an intense activity to produce tools enabling systematic calculation of derivatives. Efficient and reliable AD tools for evaluating derivatives have been published. In this paper, we sketch some of the main theory at the heart of AD, review some of the best AD codes currently available and put into context the use of AD for iterative solution methods of nonlinear systems or adjoint equations. Our aim is to direct scientists and engineers confronted with the need of exactly calculating derivatives to the use of AD as a highly useful tool and those AD tools which they could try primarily. Moreover, we show that the use of AD increases the performance of the quadratically convergence solution of a parabolised Navier-Stokes equations.**

Keywords: ***Algorithmic Differentiation, Newton solvers, Adjoint equations, vertex elimination***

1 Introduction

Algorithmic Differentiation (AD) also known as Automatic Differentiation or Computational Differentiation, is a set of algorithms for generating derivatives of functions represented by computer programs of arbitrarily complex simulations. It uses the chain rule of calculus applied to elementary operations in an automated fashion. AD is an old technique, which has been invented and rediscovered many times but still has not reached all its potential users, see [24, 17] for some historical notes.

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

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

in which $\mathbf{R}$ represents the residual vectors in a finite-volume or finite-element calculation, $\mathbf{q}$ represents the set of flow variables at each point of the mesh, and $\mathbf{x}$ is the supplied mesh. Typically, the resolution of (1) is part of a rather large computation of the form

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

To solve the system in (1), one can use the following two approaches:

- explicit time marshing based on Runge-Kutta solvers, with acceleration methods such as enthalpy damping, implicit residual smoothing, and multigrid, see for example [14].
- Newton-like iteration methods involving derivative calculations, see for instance [10, 30].

We focus on the second, which can be written as

$$\mathbf{q}_{n+1} = \mathbf{q}_n - \mathbf{P}(\mathbf{q}_n,\mathbf{x})\mathbf{R}(\mathbf{q}_n,\mathbf{x}) \tag{3}$$

Some CFD algorithms use an approximation of $\mathbf{P}$ but their convergence is asymptotically linear [14]. The Newton-Raphson algorithm defines $\mathbf{P}$ to be $(\frac{\partial \mathbf{R}}{\partial \mathbf{q}})^{-1}$ and exhibits a quadratic convergence. However, Newton-Krylov solvers are of great interest as they are matrix-free solvers [20]. Therefore they use vector-Jacobian or Jacobian-vector products instead of constructing the full Jacobian. The efficiency of vector-Jacobian or Jacobian-vector products is explored in for example [35]. Successful Newton-Krylov methods depend on how good the resulting Newton-Krylov iteration is preconditioned. This preconditioning is usually done using successive Jacobian approximations [20]. In [18], the Jacobian approximation is done using finite differences but one could use AD to evaluate a more accurate Jacobian. It is shown that a mixed finite differencing and AD scheme for evaluating Jacobian-vector products increased the convergence rate and the speedup of a GMRES [25], which is a Krylov-type method solver [18].

[*]Centre for Mathematics & Financial Computing, CSSE, Xi'an Jiaotong-Liverpool University, 111 Ren Ai Road, SIP, Suzhou, Jiangsu Province, P.R. China 215123 Email: emmanuel.tadjouddine@xjtlu.edu.cn

In [5], the nonlinear system of the equation (1) is derived from a system of differential algebraic equations and solved with the following Gauss-Newton iteration:

$$\mathbf{q}_{n+1} = \mathbf{q}_n - \rho_n \mathbf{R}'(\mathbf{q}_n, \mathbf{x}) \dagger \mathbf{R}(\mathbf{q}_n, \mathbf{x}) \tag{4}$$

where $\mathbf{R}'(\mathbf{q}_n, \mathbf{x}) \dagger \mathbf{R}(\mathbf{q}_n, \mathbf{x})$ represents the minimum norm least squares solution of the system

$$\mathbf{R}'(\mathbf{q}_n, \mathbf{x})q = \mathbf{R}(\mathbf{q}_n, \mathbf{x}),$$

and ρ_n is a factor seeking to increase the convergence region of the iteration. This Newton iteration needs the calculation of the Jacobian $\frac{\partial \mathbf{R}}{\partial \mathbf{q}}$. Loosely speaking, Newton solvers have asymptotic superlinear or quadratic convergence but required derivative calculation. It is not our intention to present a detailed study of the convergence properties of Newton solvers. However, we present a case study from CFD by which the use of AD in a Newton-Raphson solver ensure quadratic convergence. We mainly focus on using AD as a powerful tool to systematically calculate the derivative required by a Newton solver.

There are different ways of calculating derivatives including hand-coding [33], using computer algebra systems [32] such as `Mathematica` or `Maple`, using finite-differencing, or AD [24, 15, 2, 17, 6]. AD is not about symbolic manipulation as performed in computer algebra systems such as `Maple`. It systematically augments the floating-point part of a computer program with extra instructions calculating derivatives. It is an efficient way of calculating derivatives in terms of accuracy and runtime. In the AD terminology, input variables with respect to which we need to compute the derivatives are called *independents*, and outputs whose derivatives are desired are called *dependents*.

Excellent AD tools that are efficent and reliable have been published (see `www.autodiff.org`). ADIFOR, ODYSSÉE, TAMC, and ADIC are well-established tools, which make use of the standard forward and reverse modes of AD. The forward mode propagates directional derivatives along the flow of the program. The reverse mode first computes the function, then calculates the sensitivities of the dependent variables with respect to the intermediate and independent variables in the reverse order to their calculation in the function. The sensitivities of the dependent to the independent variables give the desired derivatives.

ELIAD [29, 10, 11] is an AD tool that also uses source transformation but, in contrast to the AD tools listed above, ELIAD uses the vertex elimination algorithm of Griewank and Reese [16]. The Jacobian code created requires fewer floating-point operations than that obtained by the traditional forward and reverse AD methods as implemented by ADIFOR or TAMC. The vertex elimination AD algorithm efficiently exploits the sparsity of the Jacobian calculation. Careful experiments showed ELIAD produced Jacobian codes running 2 to 10 times faster than those by ADIFOR or TAMC, see [11, 23]. As shown in [10], the application of ELIAD to flux calculations in a finite-volume parabolised Navier-Stokes space-marched flow-solver ensured quadratic convergence for the associated Newton solver and increased overall performance compared with traditional forward or reverse AD or sparse finite-differencing for Jacobian assembly.

2 Adjoint Iterative Solvers

Adjoint iterative solvers [14] using AD exploit the fact that it is cheaper to calculate the transposed Jacobian when the number of the dependent variables is lower compared to the number of independent variables, see for example [17] for further details. The Newton iteration becomes

$$\mathbf{q}_{n+1}^T = \mathbf{q}_n^T - \mathbf{P}(\mathbf{q}_n, \mathbf{x})^T \mathbf{R}(\mathbf{q}_n, \mathbf{x})^T \tag{5}$$

and since the transposition is an adjoint operator, $\mathbf{q}^T$ can be denoted as $\bar{\mathbf{q}}$. Typically, in design optimization, we wish to minimize a scalar cost function of many parameters. A gradient-based algorithm [22] uses the sensitivities of the cost function to the design parameters. One can use the reverse mode AD to systematically evaluate such quantities as the gradient of the cost function. This is cheaper as the cost of the gradient evaluation is independent from the number of design parameters.

Iterative solutions of adjoint equations can be found by an "intelligent" application of the reverse mode AD. This could be done by adjoining the last iterations of equation (3) [17]. If the Newton iteration (3) converges after N iterations, and if we evaluate the quantity in equation (2), then we have:

$$\bar{\mathbf{x}} = \bar{\mathbf{y}} \frac{\partial \mathbf{F}(\mathbf{q}_N, \mathbf{x})}{\partial \mathbf{x}}, \quad \bar{\mathbf{q}} = \bar{\mathbf{y}} \frac{\partial \mathbf{F}(\mathbf{q}_N, \mathbf{x})}{\partial \mathbf{q}} \tag{6}$$

The convergence issues of iterative solvers as well as their adjoints have been investigated in for example [12, 7]. It is usually found that the convergence of the derivative iterates is R-linear and possibly Q-linear under some nice assumptions (regularity, contractivity) on the original iterates. The references [12, 7, 17] should be consulted for further details. In [7], it is shown that reverse mode AD enables us to adapt the number of iterations so as to reach the required degree of accuracy and then achieve derivative convergence.

Current implementations of the reverse mode AD use either storage (e.g., TAMC) or recomputation (e.g., ODYSSÉE, TAPENADE) schemes for the required variables in order to reverse the derivative calculation. This leads to memory or time requirement bottlenecks for large-scale applications. TAF, which is currently a commercial tool, provides a certain number of directives allowing to exploit knowledge of the code to increase efficiency. To tackle these complexity issues, we usually use checkpointing schemes inspired by the so-called strategy of "divide and conquer" to achieve an acceptable tradeoff between memory and time requirements, see for instance [17].

3 Principles and Techniques of Algorithmic Differentiation

A computer program that evaluates a function of n inputs and m outputs can be viewed as a sequence of $p+m$ scalar assignments described as follows:

$$x_i = \phi_i(\{x_j\}_{j \prec i}), \qquad i = n+1, \ldots, N = n+p+m, \qquad (7)$$

where ϕ_i represents an *elemental function* and $j \prec i$ means that x_j is used in computing x_i. We define *intermediate* variables to be those whose value depends on an independent and affects a dependent variable. Assuming that in (7), $x_1, \ldots, x_n$ are the independents, $x_{n+1}, \ldots, x_{n+p}$ are the intermediates, and $x_{n+p+1}, \ldots, x_{n+p+m}$ are the dependents that are mutually independents, the sequence of assignments in (7) yields the following non-linear system,

$$0 = (\phi_i(\{x_j\}_{j \prec i}) - x_i), \quad i = n+1, \ldots, N. \qquad (8)$$

Assuming the ϕ_i have continuous first derivatives, we can differentiate the non-linear system (8). Writing $\nabla \mathbf{x}_i = \left(\frac{\partial x_i}{\partial x_1}, \ldots, \frac{\partial x_i}{\partial x_n}\right)$, we obtain

$$\begin{aligned} 0 &= \nabla \phi_i(\{x_j\}_{j \prec i}) - \nabla \mathbf{x}_i \\ &= \textstyle\sum_{j \prec i} \frac{\partial \phi_i}{\partial x_j} \nabla \mathbf{x}_j - \nabla \mathbf{x}_i, \;\; i = n+1, \ldots, N. \end{aligned}$$

The conventional forward Automatic Differentiation consists in evaluating each intermediate and dependent variable x_i simultaneously with its directional derivative

$$\nabla \mathbf{x}_i = \sum_{j : j \prec i} \frac{\partial \phi_i}{\partial x_j} \nabla \mathbf{x}_j \;\; i = n+1, \ldots, N.$$

in a given direction $p \in I\!R^n$. The conventional reverse mode Automatic Differentiation consists in

1. performing a forward sweep to calculate the values of variables x_i until the evaluation of the function $\mathbf{F}$ is complete.
2. performing a reverse sweep calculating directional derivatives

   $$\nabla \mathbf{x}_k = \sum_{j : k \prec j} \frac{\partial \phi_k}{\partial x_j} \nabla \mathbf{x}_j \;\; k = n+p, \ldots, 1$$

   with respect to each intermediate and independent variable. The quantities

   $$\nabla \mathbf{x}_k = \left(\frac{\partial x_k}{\partial x_{n+p+1}}, \ldots, \frac{\partial x_k}{\partial x_N}\right),$$

   are sometimes called *adjoint* vectors.

Let us illustrate the AD technique by considering the code fragment comprising five scalar assignments in the equation (9).

$$\begin{aligned} x_4 &= \log(x_1 x_2) \\ x_5 &= x_2 x_3^2 - 2 \\ x_6 &= 3x_4 + \tfrac{x_2}{x_3} \\ x_7 &= x_4^2 + x_5 - x_2 \\ x_8 &= \sqrt{x_6} - x_5 \end{aligned} \qquad (9)$$

Writing

$$\nabla x_i = \left(\frac{\partial x_i}{\partial x_1}, \frac{\partial x_i}{\partial x_2}, \frac{\partial x_i}{\partial x_3}\right) \text{ and } c_{i,j} = \frac{\partial x_i}{\partial x_j},$$

we use standard rules of calculus to write the linearised equations to the right of equation (10). These linearised equations describe the forward mode of AD and equivalent code would be produced by ADIFOR or TAMC enabling Jacobian calculation on setting $\nabla x_1 = (1, 0, 0)$, $\nabla x_2 = (0, 1, 0)$, and $\nabla x_3 = (0, 0, 1)$.

$$\begin{array}{ll} x_4 = \log(x_1 x_2) & \nabla x_4 = c_{4,1} \nabla x_1 + c_{4,2} \nabla x_2 \\ x_5 = x_2 x_3^2 - 2 & \nabla x_5 = c_{5,2} \nabla x_2 + c_{5,3} \nabla x_3 \\ x_6 = 3x_4 + \frac{x_2}{x_3} & \nabla x_6 = c_{6,4} \nabla x_4 + c_{6,3} \nabla x_3 + c_{6,2} \nabla x_2 \\ x_7 = x_4^2 + x_5 - x_2 & \nabla x_7 = c_{7,5} \nabla x_5 + c_{7,4} \nabla x_4 + c_{7,2} \nabla x_2 \\ x_8 = \sqrt{x_6} - x_5 & \nabla x_8 = c_{8,6} \nabla x_8 + c_{8,5} \nabla x_5 \end{array} \qquad (10)$$

The corresponding matrix representation of the equivalent code statements given to the right of equation (10) gives a linear system,

$$Mu = v \qquad (11)$$

to solve for the derivatives u, wherein

$$M = \left[\begin{array}{ccc|ccc|cc} -1 & & & & & & & \\ & -1 & & & & & & \\ & & -1 & & & & & \\ \hline c_{4,1} & c_{4,2} & & -1 & & & & \\ & c_{5,2} & c_{5,3} & & -1 & & & \\ & c_{6,2} & c_{6,3} & c_{6,4} & & -1 & & \\ \hline & c_{7,2} & & c_{7,4} & c_{7,5} & & -1 & \\ & & & & c_{8,5} & c_{8,6} & & -1 \end{array}\right]$$

with zero entries in the matrix omitted for clarity,

$$u = \begin{bmatrix} \nabla x_1 \\ \nabla x_2 \\ \nabla x_3 \\ \nabla x_4 \\ \nabla x_5 \\ \nabla x_6 \\ \nabla x_7 \\ \nabla x_8 \end{bmatrix} \quad \text{and} \quad v = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The forward mode AD would be equivalent to solving the linear system using a forward substitution. The reverse mode AD would be equivalent to transposing the linear system, apply a backward substitution, and set the adjoint vectors $\bar{x}_7 = (1,0)$, and $\bar{x}_8 = (0,1)$ to calculate $(\bar{x}_1, \bar{x}_2, \bar{x}_3)$ as the transposed of the Jacobian $\partial\,(x_7, x_8)/\partial(x_1, x_2, x_3)$.

The elimination approach, see for example [17] takes a different viewpoint and may be interpreted via either the matrix representation or the so-called computational graph, see [11, 23, 27] for more details. In the graph interpretation viewpoint, the Jacobian $\partial\,(x_7, x_8)/\partial(x_1, x_2, x_3)$ is determined by eliminating intermediate vertices of the Computational graph until it becomes bipartite. In terms of the matrix representation, vertex elimination is equivalent to choosing a diagonal pivot from rows 4 to 6, in some order. At each step we eliminate all the coefficients under that pivot by elementary row operations. For example by choosing row 6 as the pivot row, we add the multiple $c_{8,6}$ of row 8 to row 6 and so produce entries $c_{8,2} = c_{8,6} \times c_{6,2}$, $c_{8,3} = c_{8,6} \times c_{6,3}$, $c_{8,4} = c_{8,6} \times c_{6,4}$ in row 8. By then choosing rows 5 and 4 as pivots we are left with the Jacobian in the submatrix comprising of elements $c_{7,1}$, $c_{7,2}$, $c_{7,3}$, $c_{8,1}$, $c_{8,2}$, and $c_{8,3}$.

There are as many row orderings as there are permutations of the intermediate rows in the extended Jacobian. The forward ordering consists in eliminating rows 4, 5, 6 in that order. The reverse ordering will eliminate rows 6, 5, 4 respectively. In addition to the forward or reverse ordering, we use orderings based on heuristics from sparse matrix technology such as the Markowitz criterion studied in [15]. The main advantage of this approach is that it exploits the sparsity of the calculation at compilation time and allows for minimizing the number of FLOPs to accumulate the Jacobian.

4 Algorithmic Differentiation Tools

Recent works on Automatic Differentiation has focused on producing tools to differentiate high level languages such as FORTRAN, C, or MATLAB. There are at least two methods for implementing an AD tool:

- *operator overloading* in which the basic arithmetic operators and intrinsic functions are overloaded to allow for the propagation of derivatives. The user must change the types of all active variables by hand as there is no automated analysis of the source code.
- *source transformation* in which the original source code is augmented by new statements, which compute derivative values resulting in a new transformed source code. This uses compiler technology and allows for sophisticated analyses in order to optimize the derivative code.

Let us now list some of the available tools that are found be reliable and efficient, see `www.autodiff.org` for further information.

4.1 AD for Fortran 77

To differentiate a Fortran 77 program, the following AD tools are available:

- Adifor implements an overall forward mode in which statements are augmented using the reverse mode. It is said to be overall forward but statement level reverse [3]. This enables directional derivative calculations with the help of a seed matrix. Recently, the AD reverse mode has been implemented in the version 3 of Adifor.
- Odyssée implements algorithms for directional derivative calculations (Jacobian-vector and vector-Jacobian products) using the AD forward mode and reverse mode.
- Tamc implements algorithms allowing for directional derivative or Jacobian calculations using forward/reverse modes of AD. In contrast to Odyssée's reverse, which stores all modified variables, Tamc's reverse recomputes required values of the modified variables.
- Tapenade, the Odyssée's successor allows for calculating directional derivatives using AD forward/reverse modes and Jacobians using the forward mode. Tapenade's reverse mode uses a storage strategy, which stores only the set of required variables in a conservative way.

Adifor, Odyssée, Tamc and Tapenade use the source transformation approach.

4.2 AD for Fortran 95

Some AD tools allow to differentiate Fortran 95 codes (e.g., TAF, AD01) or some limited extensions of Fortran 77 to Fortran 95 (e.g., Tapenade).

- TAF, the Tamc's successor provides efficient algorithms calculating directional derivatives, Jacobians, and Hessians. The TAF reverse mode uses a recomputation strategy but it allows the user to use directives (including storage directives) based on his or her knowledge of the original code in order to enhance the efficiency of the derivative code.
- AD01 provides ways to calculate derivatives of any order in forward mode and up to second derivatives in reverse mode.

TAF is a source transformation tool whereas AD01 uses operator overloading. An ongoing project [21] aims at integrating AD as part of the NAG F95 compiler.

4.3 AD for C/C++

Most of the available AD tools for C/C++ used the operator overloading approach. ADOL-C and FADBAD++ use operator overloading and allow to calculate first and higher derivatives. ADIC uses the source transformation approach and enables differentiating codes written in ANSI C, see [4].

4.4 AD for MATLAB

ADMAT uses the object oriented features of MATLAB to implement forward and reverse mode by operator overloading. MAD exploits MATLAB's high level operations to efficiently implement the AD forward mode. ADIMAT employs a mixed source transformation/operator overloading framework implementing AD algorithms.

5 Application to a 2-D Parabolised Navier-Stokes Solver

A finite-volume Parabolised Navier Stokes space-marched flow-solver is studied in [10]. The flux Jacobian was calculated using hand-coded approximation, one-sided finite-differencing, and the AD tools ELIAD and TAMC. Figure 1 gives performance details in terms of convergence rate and runtimes. Except from the approximate linearization, quadratic convergence was observed and the use of ELIAD generated code improved the overall performance. This improvement is due to the fact that the ELIAD generated code was slightly faster than that of TAMC. We can make the following two observations. First, we can see that accurate derivatives can improve the convergence rate of a Newton-type solver. Second, faster derivative can improve overall performance of the solver.

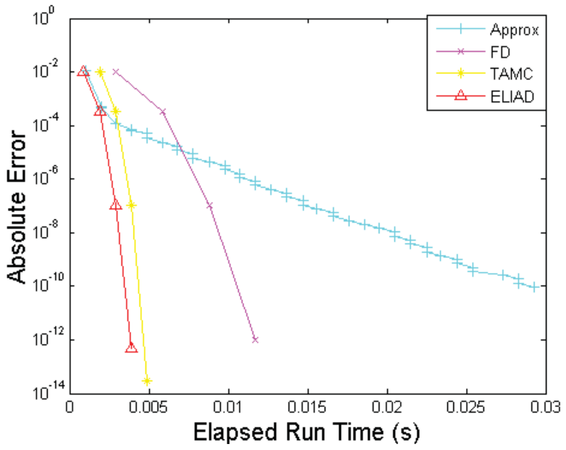

Figure 1: CPU time and convergence rate for the Flow Solver

6 Conclusions

Algorithmic Differentiation is a tool which is likely to be integrated in the next generation of computer software systems because of its growing implication in scientific computation. Its application to Newton solvers is one of the many applications such as data assimilation [13], weather forecasting [31], CFD [9, 10, 30], parameter identification [3, 4], sensitivity analysis [1, 26], and Engineering [28]. These applications show that AD is gaining in popularity. By presenting the use of AD in Newton solvers, we hope to provide some insights on what AD is capable of, facilitate the discovery of AD by pointing out most of currently available material, and reach out engineers and practioners in the wider area of computational science.

References

[1] Barhen, J. and Reister, B. 2003. Uncertainty Analysis Based on Sensitivities Generated Using Automatic Differentiation. In Proceedings of ICCSA 2003, pp:70-77.

[2] Berz, M., Bischof, C., Corliss, G., Griewank, A., eds. 1996: *Computational Differentiation: Tech., Appli., and Tools.* Philadelphia, SIAM.

[3] Bischof, C., Carle, A., Khademi, P., Mauer, A. 1996: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. IEEE Computational Science & Engineering **3** pp:18–32

[4] Bischof, C., Roh, L., Mauer, A. 1997: ADIC — An Extensible Automatic Differentiation Tool for ANSI-C. Software–Practice and Experience **27** pp:1427–1456.

[5] Campbell, S.L and Hollenbeck, R. 1996: Automatic Differentiation and Implicit Differential Equations. In Proceedings of AD 1996, pp:215-227.

[6] Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U., eds. 2001: *Automatic Differentiation of Algorithms: From Simulation to Optimization.* Springer.

[7] Christianson, B. 1994: Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326.

[8] Drexler, K.E 1987. *Engines of Creation: the Coming Era of Nanotechnology.* Bantam, New York.

[9] Forth, S.A. and Evans, T. P. 2001. Aerofoil Optimisation via AD of a Multigrid Cell-Vertex Euler Flow Solver. In [Corliss et al., 2001], chapter 17, pp:149–156.

[10] Forth, S.A. and Tadjouddine, M. 2003: CFD Newton Solvers with ELIAD, An Elimination Automatic Differentiation Tool. In Int. Conf. on CFD. Springer, Sydney, pp:134-139.

[11] Forth, S.A., Tadjouddine, M., Pryce, J.D. and Reid, J.K.: 2004. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Transactions on Mathematical Software*, 30(3):266–299.

[12] Gilbert, J.C. 1992: Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21.

[13] Giering, R., Kaminski, T. 1998: Recipes for Adjoint Code Construction. ACM Trans. on Math. Software **24** pp:437–474

[14] Giles, M.B. 2001: On the Iterative Solution of Adjoint Equations. In *Automatic Differentiation: From Simulation to Optimization*, Springer, pages:141–147.

[15] Griewank, A., Corliss, G. 1991: *Automatic Differentiation of Algorithms*. SIAM.

[16] Griewank, A., Reese, S. 1991: On the calculation of Jacobian matrices by the Markowitz rule. In [Griewank and Corliss, 1991], pp:126–135

[17] Griewank, A. 2000: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM.

[18] Hovland, P.D. and MacInnes, L.C. 2001: Parallel Simulation of Compressible Flow Using Automatic Differentiation and PETSc. Parallel Computing Volume 27, number 2, 2001.

[19] Keyes, D.E., Hovland, P.D., McInnes, L.C. and Samyono, W. 2001. Using Automatic Differentiation for Second-order Matrix-free Methods in PDE-Constrained Optimization. In [Corliss et al., 2001], chapter 3, pp:33–48.

[20] Knoll, D.A. and Keyes, D.E. 2002: Jacobian-free Newton-Krylov Methods: A Survey of Approaches and Applications, J. Comp. Phys., submitted.

[21] Naumann, U., Christianson, B., Riehme, J., and Gendler, D. 2008: *Differentiation Enabled Fortran Compiler Technology*, University of Aachen, Germany; University of Hertfordshire, UK; Numerical Algorithms Group Ltd, UK.

[22] Nocedal, J. and Wright, S.J. 1999: *Numerical Optimization*. Springer Series in OR.

[23] Pryce, J.D. and Tadjouddine, E.M. 2008: Fast automatic differentiation Jacobians by compact LU factorization. *SIAM J. Scientific Computing*, 30(4):1659–1677.

[24] Rall, L.B. 1981: *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

[25] Saad, Y. and Schultz, M.H. 1986: GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. SIAM J. Sci. Stat. Comput., **7**, pp:856-869.

[26] Tadjouddine, E.M. 2009: Algorithmic Differentiation Applied to Economics. In Proceedings of the IMECS 2009, Hong Kong, 18-20 March 2009, pages: 2199-2204.

[27] Tadjouddine, E.M. 2008: Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal*, 51(6):688–699.

[28] Tadjouddine, M., Forth, S.A., Keane, A.J. 2005: *Adjoint Differentiation of a Structural Dynamics Solver*. In Automatic Differentiation: Applications, Theory, and Implementations. Bücker et al (Eds), Vol. 50, Lecture Notes in Computational Science and Engineering, Springer, pages:309-319

[29] Tadjouddine, M., Forth, S.A., Pryce, J.D., Reid, J.K. 2002: Performance Issues for Vertex Elimination Methods in Computing Jacobians using Automatic Differentiation. In [Sloot et al. 2002], pp:1077–1086.

[30] Tadjouddine, M., Forth, S.A., and Qin, N. 2005: *Elimination AD Applied to Jacobian Assembly for an Implicit Compressible CFD Solver*. International Journal for Numerical Methods in Fluids, 2005, Volume 47, Issue 10-11, pages:1315–1321, Special Issue: 8th ICFD Conference on Numerical Methods for Fluid Dynamics, John Wiley & Sons, Ltd.

[31] Talagrand, O. 1991. The Use of Adjoint Equations in Numerical Modelling of the Atmospheric Circulation. In [Griewank and Corliss, 1991], SIAM, pp:169–180.

[32] Vanden, K.J. and Orkwis, P.D. 1996: AIAA Journal **34**, 6.

[33] Venkatakrishnan, V. 1989: AIAA Journal **27**.

[34] Venkatakrishnan V. 1993. On the Accuracy of Limiters and Convergence to Steady State. AIAA, 93-0880-CP.

[35] Walther A., Griewank, A. and Best A. 1999: Multiple Vector-Jacobian Products are Cheap, Applied Numerical Mathematics, vol. 30, pp:367-377.