In: Automatic Differentiation of Algorithms. Springer, New York, NY
DOI: 10.1007/978-1-4613-0075-5_30; ISBN: 978-1-4612-6543-6

AD Tools and Prospects for Optimal AD in CFD Flux Jacobian Calculations

Mohamed Tadjouddine, Shaun A. Forth, and John D. Pryce

ABSTRACT We consider the problem of linearising the short (approximately 100 lines of) code that defines the numerical fluxes of mass, energy and momentum across a cell face in a finite volume compressible flow calculation. Typical of such formulations is the numerical flux due to Roe, widely used in the numerical approximation of flow fields containing moderate to strong shocks. Roe's flux takes as input 10 variables describing the flow either side of a cell face and returns as output the 5 variables for the numerical flux. We present results concerning the efficiency of derivative calculations for Roe's flux using several currently available AD tools. We also present preliminary work on deriving near optimal differentiated code using the node elimination approach. We show that such techniques, within a source transformation approach, will yield substantial gains for application code such as the Roe flux.

1 Differentiation of Roe's Numerical Flux

In previous work [For98] we have reported on the application of our own operator overloaded, forward AD library to calculate the linearisation of Roe's numerical flux [Roe81]. We validated the linearisation by showing that a finite difference evaluation converged to the AD linearisation as the finite difference step size was reduced. Here we linearise the flux using currently available AD software (ADIFOR [BCH+98], TAMC [Gie97] and ADO1 [PR98]). Results are presented in Table 1 where time(∇F) denotes the CPU time for a Jacobian (and function) evaluation, $\frac{\text{time}(\nabla F)}{\text{time}(F)}$ denotes the ratio of the Jacobian evaluation to the function evaluation in CPU time, and the final column gives the maximum absolute difference in the Jacobian (assuming that the ADIFOR results are correct). Note that the ADIFOR results are for all arrays and loops of fixed length 10 and with the *Performance* exception handling option.

We see that all the AD methods produced the same results to within machine relative precision whereas, as expected, the finite difference results are in error by around the square root of machine precision. The source transformation methods (ADIFOR and TAMC) are substantially faster than the library AD01 using operator overloading. The forward methods

Method	time(∇F)	$\frac{time(\nabla F)}{time(F)}$	Deviation from ADIFOR
Finite Differences (1-sided)	0.12	10.64	4.34E-07
ADIFOR	0.15	13.37	—
TAMC (Forward)	0.13	11.94	4.66E-15
TAMC (Reverse)	0.11	10.28	5.77E-15
AD01 (Forward)	1.55	134.68	7.99E-15
AD01 (Reverse)	0.95	82.90	4.88E-15

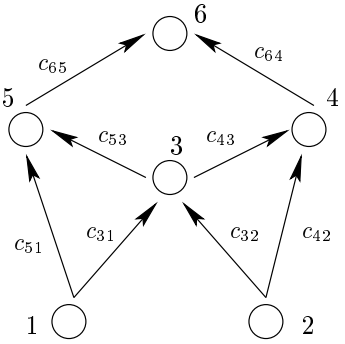
TABLE 1.1. CPU timings (in seconds) on SGI IRIX64 IP27

ADIFOR and TAMC (forward) are comparable in execution time and only slightly slower than the finite differences while giving exact results. The reverse modes of TAMC and AD01 have performed better than the forward ones with this SGI machine. However TAMC reverse performed worse with an older COMPAQ Alpha AXP 250-330 workstation. We attribute this to cache misses during the reverse pass on this smaller cache machine.

In the rest of this paper we consider AD techniques applied to a restricted class of Fortran codes important in CFD applications: subroutines with typically 5 to 30 inputs x_i and outputs y_i and some hundreds v_i of intermediate values, with no loops but allowing branches. The aim is to efficiently compute the Jacobians associated with such subroutines.

2 Elimination Techniques

An alternative to the conventional forward and reverse modes of AD is the elimination approach [GR91, Nau99, Gri00]. To illustrate this technique,



$$\mathbf{C} = \begin{bmatrix} -1 & & & & & \\ & -1 & & & & \\ c_{31} & c_{32} & -1 & & & \\ & c_{42} & c_{43} & -1 & & \\ c_{51} & & c_{53} & & -1 & \\ & & & c_{64} & c_{65} & -1 \end{bmatrix}$$

FIGURE 1. An example of CG (left) and its matrix representation (right)

consider the graph sketched in Figure 1, which shows a Computational Graph (CG) for derivative calculation and its (sparse) matrix representa-

tion. The nodes 1 and 2 represent the inputs x_1 and x_2; nodes 3, 4, and 5 the intermediates; and node 6 the output y. The matrix represents the equations $\mathbf{C}\dot{\mathbf{v}} = -(\dot{x}_1, \dot{x}_2, 0, \ldots, 0)^T$, that is:

$$\begin{array}{ll} \dot{v}_1 = \dot{x}_1 & \dot{v}_4 = c_{42} * \dot{v}_2 + c_{43} * \dot{v}_3 \\ \dot{v}_2 = \dot{x}_2 & \dot{v}_5 = c_{51} * \dot{v}_1 + c_{53} * \dot{v}_3 \\ \dot{v}_3 = c_{31} * \dot{v}_1 + c_{32} * \dot{v}_2 & \dot{v}_6 = y = c_{64} * \dot{v}_4 + c_{65} * \dot{v}_5 \end{array}$$

The coefficients $c_{i,j}, 1 \leq i, j \leq 6$ represent partial derivatives $\frac{\partial v_i}{\partial v_j}$. The Jacobian $\frac{\partial y}{\partial (x_1, x_2)}$ is determined by eliminating intermediate nodes or edges from the graph until it becomes bipartite. In terms of the matrix representation, node elimination is equivalent to successively choosing a diagonal pivot element from rows 3 to 5, eliminating all the coefficients under that pivot and leaving the Jacobian as elements c_{61} and c_{62}. Adopting the notations $\prec$ and $\prec^*$ of [Gri00], we define the Markowitz and VLR costs at an intermediate node v_j respectively as follows:

$$\begin{aligned} \text{mark}(v_j) &= |\{i : i \prec j\}||\{k : j \prec k\}| && (1.1) \\ \text{VLR}(v_j) &= \text{mark}(v_j) - |\{i : i \prec^* j\}||\{k : j \prec^* k\}|. && (1.2) \end{aligned}$$

Ordering the elimination process using heuristics based on choosing the node with minimum Markowitz or VLR cost generally gives a further improvement [Nau99].

2.1 Application of node elimination methods to Roe flux

We have written and used a Fortran 90 AD module using operator overloading to build up the CG of the Roe flux code. The graph consists of 221 nodes and 342 edges. Then, we applied the standard node elimination methods (Forward, Reverse, Markowitz, VLR). We also employed what we term *reverse bias* variants of Markowitz and VLR which use the last node with the minimum Markowitz/VLR cost as opposed to the first in the conventional implementations.

Figure 2 shows the elimination sequences taken by these 6 methods and Table 1.2 the required number of multiplications. Classical forward AD requires 3420 multiplications and reverse 1710. Hence the elimination techniques use up to some 40% fewer multiplications than reverse AD and 70% fewer than forward AD for this problem. We observe that Markowitz has

Forward	Reverse	Mark.	VLR	Mark.(rev. bias)	VLR (rev.bias)
1959	1335	1031	1075	998	1073

TABLE 1.2. Number of multiplication of the different strategies

taken fewer operations than the other methods and incorporating the reverse bias has improved Markowitz more than it did VLR. Furthermore the

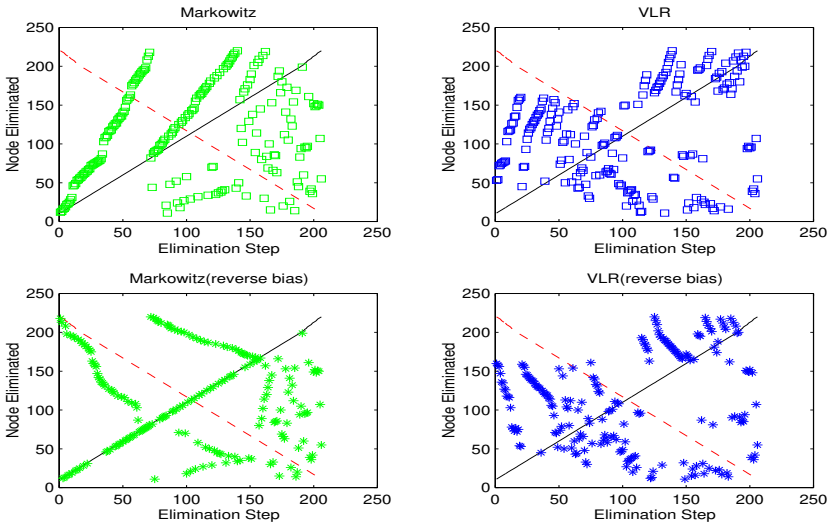


FIGURE 2. Elimination sequence for different strategies applied to the Roe Flux test case (forward/reverse given by solid/dashed line in each graph)

behaviour of the Markowitz (reverse bias) looks like a hybrid of forward and reverse eliminations.

Interestingly we see that VLR and VLR (reverse bias) perform no elimination at the top and bottom of the CG for approximately the first third of their elimination sequences. We explain this as follows. The VLR cost (see equation 1.2) involves a fixed value for each node which is given by the product of the number of input/output nodes it is eventually connected to. If we assume that a node v_j in the centre will be connected to all inputs and outputs, then in our case (10 inputs and 5 outputs) it will have a constant cost of $\mathrm{mark}(v_j) - 5 \times 10$. Nodes close to the outputs will only be connected to a few (e.g. 2) outputs but to all the 10 inputs and have a higher (e.g. $\mathrm{mark}(v_j) - 2 \times 10$) constant cost. Similarly nodes close to inputs will have a higher constant cost again (e.g. $\mathrm{mark}(v_j) - 2 \times 5$). Thus when selecting the node with minimum cost for elimination, VLR is heavily biased for many steps towards the nodes in the centre of the CG.

3 Development of Source a Transformation Tool

We are developing an AD tool to efficiently compute Jacobians of a restricted class of functions via source transformation. The input code is parsed to get the Abstract Syntax Tree (AST) using the freeware ANTLR translator [Pa00]. Two passes through the AST allow us to build an Abstract Computational Graph (ACG) of the program from which we compute the Jacobian via an elimination approach.

3.1 Building the abstract computational graph

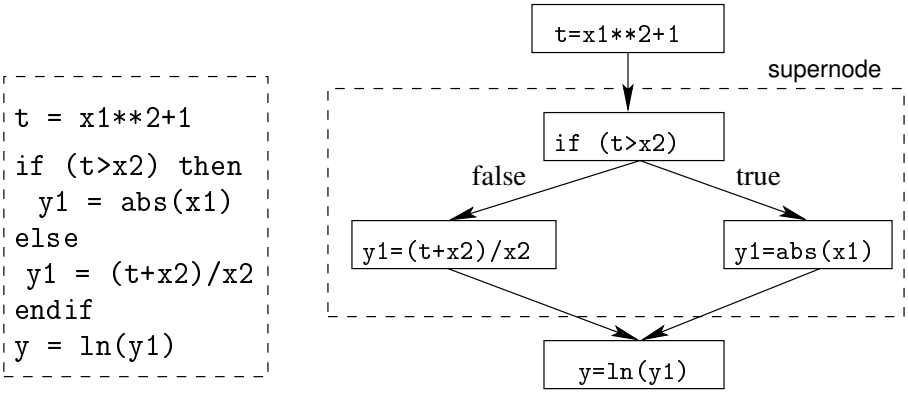


FIGURE 3. Fortran code (left) and Control Flow Graph (right)

Unlike the CG that represents one execution of the program, the ACG takes into account all execution paths. For the class of codes above with single assignment, the derived ACG is a DAG describing the chain of operations from the data inputs to the outputs. The ACG is viewed as a flowchart i.e. a control flow graph in which each basic block is expanded to a computational graph (see Figures 3, and 4). It is built by using the AST to rewrite the input code as a code list [Gri00]. Then we analyse the code to top-down propagate the active variables and compute the local partial derivatives.

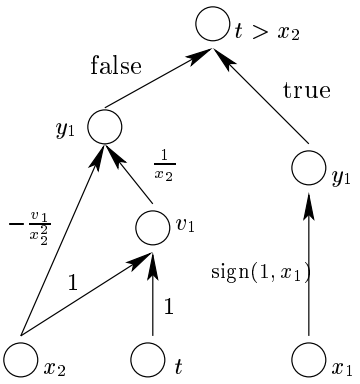


FIGURE 4. The subgraph related to the IF block (supernode)

3.2 Dealing with IF blocks

A novel feature of our work is that we intend to deal with IF blocks. They are viewed as supernodes, which we term *branching nodes* whose inputs and outputs are determined using a read/write analysis. The inputs are the union of variables imported (read and not written) into the IF block. The outputs are the union of variables written in the IF block and exported to (read from) another region of the program. This may be seen as an extension of Bischof and Haghighat's hierarchical AD [BH96] from the

subroutine level to the basic block level. The branching node represents a DAG whose reduction gives rise to a bipartite graph that connects all possible inputs to all possible outputs according to the value of the test controlling the IF block (see Figures 4 and 5). From Figure 4, we eliminate nodes at the lowest level, so that we get an augmented bipartite graph as in Figure 5 (right) with one of 2 sets of local derivatives used depending on the branch taken.

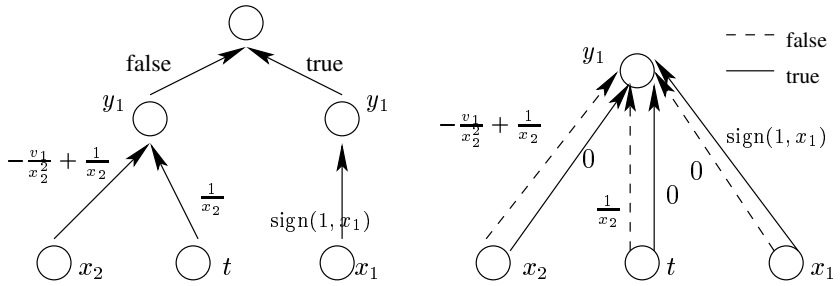


FIGURE 5. Elimination process to build up the bipartite graph from a supernode

3.3 Jacobian accumulation

Generation of the source text that computes the Jacobian proceeds as follows:

1. Apply the node elimination process from the ACG to build up the bipartite graph. The innermost branching nodes will be treated first, providing local Jacobians whose entries will then be used to recursively complete the overall elimination.
2. Generate derivative code consisting of the input code interspersed with derivative calculations obtained from the node elimination process positioned so as to attempt to maximise cache performance.

4 Conclusions

We have applied several different AD software systems ADIFOR, ADO1, and TAMC, to Roe's numerical flux. For such dense code with more inputs than outputs, reverse mode has been shown to be more efficient than forward provided the machine cache is sufficiently large. Source transformation is shown to be consistently superior to operator overloading. For a restricted class of codes, we have introduced the Abstract Computational Graph to implement the node elimination methods via source transformation. Preliminary results using operator overloading indicate that using elimination strategies such as Markowitz or VLR will give a substantial improvement over the conventional forward and reverse modes.

5 References

[BCH+98] Christian H. Bischof, Alan Carle, Paul Hovland, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0 users' guide (revision D). Technical Report ANL/MCS-P263-0991, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439 USA, 1998. Available via `http://www.mcs.anl.gov/adifor/`.

[BH96] Christian Bischof and Mohammad Haghighat. Hierarchical approaches to automatic differentiation. Technical Report CRPC-TR96647, Center for Research on Parallel Computation, Rice University, April 1996.

[For98] S.A. Forth. Automatic differentiation for flux linearisation. AMOR Report 98/1, Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, England, 1998. Poster Presentation at *16th International Conference on Numerical Methods in Fluid Dynamics*, July 6th-10th, 1998, Arcachon, France.

[Gie97] Ralf Giering. Tamgent linear and adjoint model compiler. Users Manual Manual Version 1.2, TAMC Version 4.8, Center for Global Change Sciences, Department of earth, Atmospheric, and Planetary Science, MIT, Cambridge, MA 02139, USA, December 1997.

[GR91] A. Griewank and S. Reese. On the Calculation of Jacobian Matrices by the Markowitz rule for Vertex Elimination. *G. Corliss and A. Griewank Ed., Automatic Differentiation: Theory, implementation, and applications, SIAM*, pages 126–135, 1991.

[Gri00] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, 2000.

[Nau99] Uwe Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs.* PhD thesis, Technical University of Dresden, December 1999.

[Pa00] T. Parr and al. ANTLR Reference Manual. Technical report, MageLang Institute's jGuru.com, January 2000. Available via `http://www.antlr.org/doc/index.html`.

[PR98] J.D. Pryce and J.K. Reid. ADO1, a Fortran 90 code for automatic differentiation. Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England, 1998. Available via `ftp://matisa.cc.rl.ac.uk/pub/reports/prRAL98057.ps.gz`.

[Roe81] P.L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.