Computational Science and Its Applications - ICCSA 2003
International Conference, Montreal, Canada, May 18-21, 2003, Proceedings, Part II
Lecture Notes in Computer Science , Vol. 2668,
Kumar, V.; Gavrilova, M.L.; Tan, C.J.K.; L'Ecuyer, P. (Eds.) ISBN: 978-3-540-40161-2

# Hierarchical Automatic Differentiation by Vertex Elimination and Source Transformation

Mohamed Tadjouddine[1], Shaun A. Forth[1], and John D. Pryce[2]

[1] Applied Mathematics & Operational Research, ESD
Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, UK
{M.Tadjouddine, S.A.Forth}@rmcs.cranfield.ac.uk
[2] Computer Information Systems Engineering, DOIS
Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, UK
J.D.Pryce@rmcs.cranfield.ac.uk

**Abstract.** We present a hierarchical scheme to extend the applicability of automatic differentiation (AD) by vertex elimination from the basic block level to code with branches and subroutine calls. We introduce the ELIAD tool that implements our scheme. Results from computational fluid dynamics (CFD) flux linearisations show runtime speedup by a typical factor of two over both finite-differencing and traditional forward and reverse modes of AD.

## 1 Introduction

In sensitivity analysis, design optimisation or Newton solvers, we frequently need to compute derivatives of a function represented by a computer program. Finite-differencing is a popular way of calculating derivatives. It is easy to implement but incurs truncation errors that may affect robustness of algorithms that use derivatives. An alternative is to use the algorithms and tools of Automatic Differentiation (AD) [1–3], by which derivatives of a function represented by a computer program are computed without the truncation errors associated with finite differences.

The most efficient way to implement AD in terms of run-time speed is usually *source transformation*; here the original code is augmented by statements that calculate the needed derivatives. ADIFOR [4] and TAMC [5] are well-established tools for this 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 [6–8] is an AD tool also using source transformation but, in contrast to the AD tools listed above, ELIAD uses the vertex elimination algorithm of Griewank and Reese [9]. 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 [6,8].

Section 2 describes AD using the vertex elimination approach. Section 3 presents the source transformation approach and describes the ELIAD tool. Section 4 discusses the hierarchical vertex elimination approach as an efficient way of dealing with branches. Section 5 presents numerical results from CFD flux calculations and Sect. 6 concludes.

## 2 AD by Vertex Elimination

Following [2], we consider a computer program that represents a function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$. An execution of that computer program can be viewed as a sequence of $p+m$ scalar assignments described as follows:

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

where $\phi_i$ represents an *elemental function* and $j \prec i$ means that $x_j$ is used in computing $x_i$. We define the *independent* variables to be those input variables with respect to which we need to compute the derivatives, the *dependent* variables to be those outputs whose derivatives are desired, and the *intermediate* variables to be those whose value depends on an independent and affects a dependent variable. Further, we define an *active* variable to be an independent, intermediate or dependent variable. Without loss of generality, we assume that in (1), $x_1, \ldots, x_n$ 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 and no dependent variable is calculated directly from another dependent variable.

The sequence of assignments in (1) yields the following non-linear system,

$$0 = (\phi_i(\{x_j\}_{j \prec i}) - x_i), \quad i = n{+}1, \ldots, N. \tag{2}$$

Assuming the $\phi_i$ have continuous first derivatives, we can differentiate the non-linear system (2). Writing $\nabla x_i = \left(\frac{\partial x_i}{\partial x_1}, \ldots, \frac{\partial x_i}{\partial x_n}\right)$, we obtain

$$0 = \nabla\phi_i(\{x_j\}_{j \prec i}) - \nabla x_i = \sum_{j \prec i} \frac{\partial \phi_i}{\partial x_j} \nabla x_j - \nabla x_i, \;\; i = n{+}1, \ldots, N. \tag{3}$$

In matrix terms, denoting $c_{i,j} = \begin{cases} \frac{\partial \phi_i}{\partial x_j} & n{+}1 \le i, j \le N \\ 0 & 1 \le i, j \le n \end{cases}$, and $\mathbf{C} = (c_{i,j})_{1 \le i,j \le N}$, the linear system (3) can be rewritten as,

$$(\mathbf{C} - \mathbf{I}_N)\nabla \mathbf{x} = \begin{bmatrix} -\mathbf{I}_n \\ \mathbf{0}_{(p+m)\times n} \end{bmatrix}, \tag{4}$$

where $\mathbf{I}_k$ denotes the $k \times k$ identity matrix. The lower triangular matrix $\mathbf{C}-\mathbf{I}_N$ is called the *extended Jacobian*. The equivalent graph of the extended Jacobian matrix is called the *computational graph* (see the example of [8] for details). Adopting the standard notation [2, p. 161], the strictly lower triangular $N \times N$ matrix $\mathbf{C}$ can be written in block form, (with $\mathbf{L}$ strictly lower triangular),

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 0 \\ \mathbf{B} & \mathbf{L} & 0 \\ \mathbf{R} & \mathbf{T} & 0 \end{bmatrix}. \tag{5}$$

The Jacobian $\nabla\mathbf{F}$ of the function $\mathbf{F}$ is then the Schur complement of $\mathbf{R}$ in $\mathbf{C}-\mathbf{I}_N$ and can be calculated using some form of Gaussian elimination [2]. We use pivot orderings based on heuristics from sparse matrix technology such as the Markowitz criterion studied in [2, 9, 10]. Such heuristics aim to minimise the number of floating point operations at each step of the elimination.

## 3 AD by Source Transformation

An AD tool can be implemented using operator overloading or source transformation (see [2, chap. 5]). Automatic Differentiation by source transformation uses compiler techniques to parse and analyse the original code, then augments the code's statements that calculate real valued variables with additional statements to calculate their derivatives.

### 3.1 Source Transformation Approach

As described in [11], a source transformation AD tool differentiates a program in the following stages:

1. The input program is parsed into an abstract syntax tree, see [12], as in the front-end of a compiler (lexing, parsing, semantic analysis).
2. The abstract syntax tree may be transformed into a semantically equivalent *intermediate representation* suited to applying the AD technique, e.g. see Sect. 4.2.
3. A dependency analysis is performed to determine intermediate variables from user defined independent and dependent variables.
4. The intermediate representation is augmented with derivative calculation statements.
5. Some optimisations [12, 13] (simplification of algebraic expressions, loop optimisations, statement reordering) are performed on the derivative code.
6. The transformed program is output in the source language.

### 3.2 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 to 100 inputs and outputs, some hundreds of intermediate values, branches and, (assumed unrollable) loops. ELIAD is written in Java and uses a front-end and back-end generated by ANTLR [14]. As depicted in Fig. 1, ELIAD enables the user to build up the computational graph of an input code, save it as a file, and send it to a tool for finding elimination sequences, which we term the *sequencer*. A file containing the elimination sequence is passed back to ELIAD, which then generates the corresponding derivative code. The representations of the computational graph and the sequence are described in [15]. The sequencer is currently a separate Matlab program but will eventually be coded in Java to facilitate communication with ELIAD. Nonetheless, ELIAD provides two default sequences that are the forward and reverse orderings of the intermediate vertices.

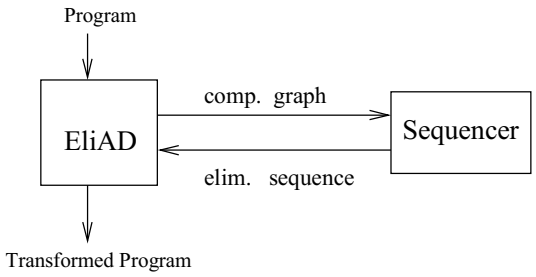


**Fig. 1.** ELIAD framework

**Building the Computational Graph** The ELIAD front-end parses an input program into its abstract syntax tree. In a program transformation framework, the abstract syntax tree is usually processed to construct control or data flow graphs. We use a graph that we term the Abstract Computational Graph (ACG). This can be viewed as a control flow graph in which each basic block is expanded to a computational graph. Unlike the computational graph used in AD by operator overloading and that represents one execution of the program, the ACG takes into account all possible execution paths for all possible values of independent variables. For the restricted class of codes defined in Sect. 3.2, the ACG is a directed acyclic graph describing the chain of operations from the independent to the dependent variables.

A vertex of the ACG represents an active program variable, that is, a variable whose derivative value is not a priori zero. The independent[dependent] vertices are the ones that have no predecessors[successors]. An edge of the ACG, say from vertex $j$ to $i$, represents the dependency $j \prec i$ (defined in Sect. 2) and is labelled by the symbolic expression for local partial derivative $\frac{\partial x_i}{\partial x_j}$.

**Generating the Derivative Code** To generate a derivative code, ELIAD is given a file containing an elimination sequence provided by the sequencer of Fig. 1. Using the abstract syntax tree of the input code and the information kept

in the abstract computational graph, the original code is interspersed with assignments that compute local partial derivatives. Then the elimination sequence is used to eliminate intermediate vertices until the computational graph becomes bipartite. A vertex is eliminated by connecting each of its predecessors to all its successors. New edges are labelled by the product of existing edges linking the corresponding successors and predecessors. If the edge already exists, its label is updated by adding the product. This results in a series of scalar assignments that compute new entries or update existing entries of the extended Jacobian.

A difficulty in using the vertex elimination approach in the AD source transformation framework is dealing with branching. We use a hierarchical elimination scheme similar to [16] to overcome this difficulty.

## 4 Hierarchical Vertex Elimination

An IF block construct is viewed as a supernode of the ACG whose inputs and outputs are determined using a read/write analysis. In the overall elimination process, computational graphs within the supernodes are first rendered bipartite subgraphs whose vertices can then be eliminated along with the remaining intermediate vertices to complete the Jacobian accumulation.

### 4.1 Elimination of Supernodes

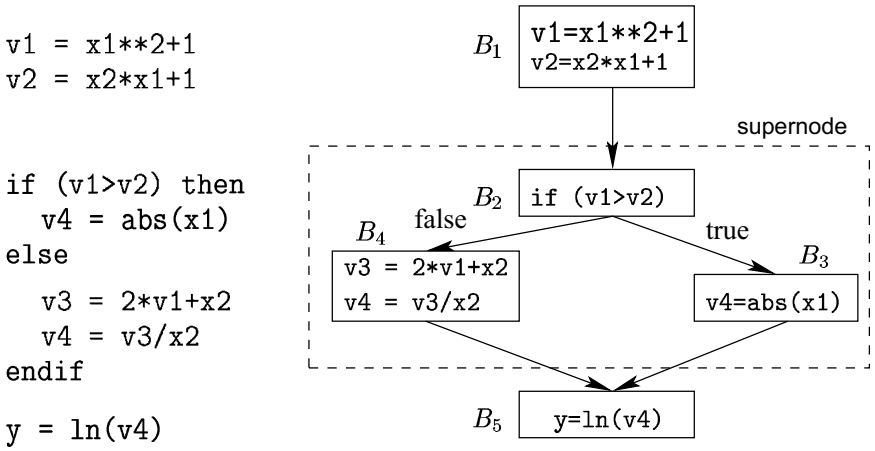


**Fig. 2.** Fortran code (left) and control flow graph (right)

Figure 2 shows a code fragment with an IF construct, and its flow graph. In the ACG, the basic blocks $B_1, B_3, B_4, B_5$ are expanded to their computational graph. Figure 3 shows the computational graph of the basic block $B_4$. Each basic block is analysed to determine its inputs and outputs. The inputs are

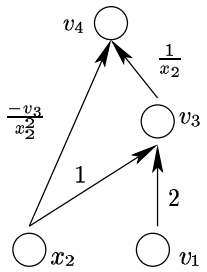


**Fig. 3.** The computational graph of the block $B_4$

those active variables that are written (assigned to) before, and read (appear on the right hand side of an assignment) within, the basic block. The outputs are those active variables that are written in the basic block and read thereafter. The inputs [outputs] of the IF construct (supernode) are the union of inputs [outputs] of its branches. This analysis corrects that of [17].

The elimination of a supernode takes three logical stages that we describe by reference to Fig. 2.

1. The partial derivatives of the outputs of $B_3$ [$B_4$] with respect to the inputs of $B_4$ [$B_3$] are initialised to zero.
2. Local partial derivatives in $B_3$ and $B_4$ are calculated and generated at statement level.
3. All intermediate vertices of $B_3$ and $B_4$ are eliminated and resulting statements are generated.

```
IF (v1>v2) THEN
  dv4byx2 = 0.              !initialisations
  dv4byv1 = 0.
  dv4byx1 = sign(1.,x1)     !local partial derivatives
  v4 = abs(x1)
ELSE
  dv4byx1 = 0.              !initialisation
  dv3byx2 = 1.              !local partial derivatives
  dv3byv1 = 2.
  v3 = 2*v1+x2
  dv4byx2 = -v3/x2**2
  dv4byv3 = 1./x2
  v4 = v3/x2
  dv4byx2=dv4byx2+dv4byv3*dv3byx2  !Elimination of
  dv4byv1 = dv4byv3*dv3byv1        !intermediate vertices
ENDIF
```

**Fig. 4.** Elimination of the supernode

This augments $B_3$ and $B_4$ with new statements that initialise or calculate derivatives and reduces the subgraphs within the supernode to a bipartite graph be-

tween its inputs and outputs. For the example of Fig. 2, the read/write analysis allows us to determine that $x_1, x_2, v_1$ are the inputs and $v_4$ the output, of the supernode. The computational graph of $B_4$ has one intermediate $v_3$ but the computational graph of $B_3$ has no intermediates. The elimination of the supernode yields the code of Fig. 4.

### 4.2 Rewriting Branches in Canonical Form

To ensure safe derivative augmentation, the IF constructs need to be rewritten in a canonical form prior to differentiation. To generate correct derivative code by AD using the vertex elimination approach, an important issue arises with variables that may be overwritten in a branch. To illustrate this, consider the code fragment on the left of Fig. 5. The variable `t` is overwritten if the condition

```
ORIGINAL CODE                    ITS CANONICAL FORM

                                 t = sin(x)
t = sin(x)                       if (t>0.5) then
if (t>0.5) then          ⟶        t = t/x
  t = t/x                        else
endif                            t = t
y = 2+t**2                       endif
                                 y = 2+t**2
```

**Fig. 5.** Overwrite variable `t` made explicit by adding extra statements

`t>0.5` is true. The variable `y` uses the value of `t` defined by either `t = sin(x)` or `t = t/x`. We could use conditionals to represent the dependency from instances of `t` to `y` but this will render the ACG construction and exploitation unnecessarily complex. We ensure that `y` uses the correct instance of `t` from within the IF construct by adding the assignment `t = t` when the condition is false.

Nested branches are rewritten in canonical form and hierarchically eliminated so that supernodes are eliminated starting from the innermost to the outermost. Each supernode elimination yields a local Jacobian, whose entries are then used to recursively complete the overall elimination.

### 4.3 Interprocedural Differentiation

A subroutine call is seen as a supernode with a set of inputs and outputs. It therefore gives rise to a bipartite graph which connects its outputs to its inputs. To enhance the efficiency of the vertex elimination approach, an interprocedural dependency analysis is performed at compile time. Our analysis assumes:

- No subroutine has an ENTRY statement and each procedure returns only from its final executable statement.
- All subroutine source codes are available for analysis.

The analysis uses the well-known *call graph*, which is a directed graph representing the calling relationships between the procedures of the program [13]. A construction algorithm for the call graph can be found in [13].

The call graph is traversed bottom up to perform dependency and activity analyses for each subroutine. Again, this is done hierarchically starting from the lowest level and recursively combining the analyses at a higher level. Finally, the call graph is traversed top down to take into account the context and consequently refine the analyses. This interprocedural analysis allows us to avoid calculating derivatives that can be identified a priori as being always zero.

For each subroutine, the analyses are performed using its control flow graph and result in:

- A dependency graph which shows the dependencies of each subroutine variable with respect to the subroutine inputs.
- An activity graph which shows the set of active variables associated with each basic block of the control flow graph of the subroutine.

Both graphs are used to construct the ACG for each subroutine as described in Sect. 3.2. The resulting ACG is then used to generate the corresponding derivative code as in Sect. 3.2. The ACG supernode associated to a subroutine call is built up by connecting the vertices representing its active outputs to those of its active inputs. The supernode edges are labelled by the corresponding components of the local Jacobian resulting from the differentiation of that called subroutine.

## 5 Numerical Results

In [7], 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. Table 1 shows the ratio of CPU time for computing Jacobian $\nabla F$ to that for computing the function $F$ (with $n$ independents and $m$ dependents), for (a) the Roe flux with MUSCL interpolation ($n = 20$, $m = 5$), (b) the Vigneron Flux ($n = 5$, $m = 8$) and (c) the viscous flux ($n = 10$, $m = 4$). The runs are from a COMPAQ Alpha DS20E workstation with 667 MHz CPU, 128 KB L1-Cache and 8 MB L2-cache. The Roe flux code contains branches (Harten's entropy fix). All calculations were performed for $10,000$ random sets of inputs repeated 10 and 50 times respectively. We used the forward and reverse modes of TAMC, 1-sided finite-differencing and ELIAD with Markowitz ordering. In Table 1, timings in brackets are obtained with the TAMC's "-pure" option, which removes function calculation. This saves some floating-point operations and a similar capability will be developed for ELIAD. The ELIAD generated code is twice as fast as the next fastest AD technique (TAMC reverse) for Roe, (TAMC forward) for Vigneron and (TAMC reverse) for viscous fluxes. The TAMC(reverse) is slower for the Vigneron flux as it propagates 8 directional derivatives instead of 5 for

**Table 1.** Efficiency of flux Jacobian calculations

(a) Roe flux, CPU($F$) = $7.31 \times 10^{-7}$ $s$

| technique | CPU($\nabla F$)/CPU($F$) | Error($\nabla F$) |
|---|---|---|
| TAMC(forward) | 18.5 (18.3) | 0.0 |
| TAMC(reverse) | 9.1 (7.6) | $3.6 \times 10^{-16}$ |
| Finite Difference | 24.1 | $4.0 \times 10^{-6}$ |
| ELIAD | 4.7 | $5.6 \times 10^{-15}$ |

(b) Vigneron flux CPU($F$) = $1.93 \times 10^{-7}$ $s$

| technique | CPU($\nabla F$)/CPU($F$) | Error($\nabla F$) |
|---|---|---|
| TAMC(forward) | 3.8 (3.0) | 0.0 |
| TAMC(reverse) | 8.9 (7.5) | $4.4 \times 10^{-16}$ |
| Finite Difference | 8.3 | $1.3 \times 10^{-6}$ |
| ELIAD | 2.2 | $4.4 \times 10^{-16}$ |

(c) Viscous flux Jacobian, CPU($F$) = $4.22 \times 10^{-7}$ $s$

| technique | CPU($\nabla F$)/CPU($F$) | Error($\nabla F$) |
|---|---|---|
| TAMC(forward) | 5.8 (5.4) | 0.0 |
| TAMC(reverse) | 3.6 (3.6) | 0.0 |
| Finite Difference | 13.4 | $4.5 \times 10^{-8}$ |
| ELIAD | 1.8 | $5.6 \times 10^{-17}$ |

TAMC(forward). We also show the maximum error in the entries of the Jacobians, assuming TAMC(forward) is exact. The ELIAD Jacobians are correct to machine precision. The error in the 1-sided finite-difference approximation is in line with its truncation error.

## 6 Conclusions

We have described a hierarchical vertex elimination approach that efficiently deals with branches and subroutine calls. This approach may be seen as an extension of the hierarchical approach of [16] from the subroutine level to the IF block level. It allows us to systematically generate Jacobian code valid for all possible execution paths, that is near optimal within each level of the hierarchy.

We have implemented this approach in the ELIAD tool. For the Roe code which contains branches and other non-branching CFD codes, the ELIAD generated Jacobian code executes about twice as fast as that generated using traditional forward and reverse mode AD tools. This run-time improvement is mainly due to the full sparsity exploitation enabled by the elimination approach.

An ongoing difficulty is that the computational graph construction requires static analyses that are in general conservative. Generally, safe overestimations are used to ensure correctness and efficiency [18]. Typically, loops require array region analyses to maintain the efficiency of this approach. Future work will investigate techniques that deal with loops within our framework.

**Acknowledgments**

We thank EPSRC and UK MOD for funding this project under grant GR/R21882.

## References

1. Griewank, A., Corliss, G.: Automatic Differentiation of Algorithms. SIAM, Philadelphia (1991)
2. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia (2000)
3. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U., eds.: Automatic Differentiation of Algorithms: From Simulation to Optimization. Springer (2002)
4. Bischof, C., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. IEEE Computational Science & Engineering **3** (1996) 18–32
5. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. ACM Trans. on Math. Software **24** (1998) 437–474
6. Forth, S.A., Tadjouddine, M., Pryce, J.D., Reid, J.K.: Jacobian Code Generated by Source Transformation and Vertex Elimination is as Efficient as Hand-coding. ACM Trans. on Math. Software (2002) (submitted).
7. Forth, S.A., Tadjouddine, M.: CFD Newton Solvers with ELIAD, An Elimination Automatic Differentiation Tool. In: Int. Conf. on CFD. Springer-Verlag, Sydney (2003) (to appear).
8. Tadjouddine, M., Forth, S.A., Pryce, J.D., Reid, J.K.: Performance Issues for Vertex Elimination Methods in Computing Jacobians using Automatic Differentiation. In Sloot, P.M., Tan, C.K., Dongarra, J.J., Hoekstra, A.G., eds.: ICCS, Part II. Volume 2330 of LNCS. Springer, Amsterdam (2002) 1077–1086
9. Griewank, A., Reese, S.: On the calculation of Jacobian matrices by the Markowitz rule. [1] 126–135
10. Naumann, U.: Elimination techniques for cheap Jacobians. [3] 241–246
11. Bischof, C., Roh, L., Mauer, A.: ADIC — An Extensible Automatic Differentiation Tool for ANSI-C. Software–Practice and Experience **27** (1997) 1427–1456
12. Aho, A., Sethi, R., Ullman, J.: Compilers: principles, techniques, and tools. Addison-Wesley (1986)
13. Zima, H., Chapman, B.: Supercompilers for Parallel and Vector Computers. Addison-Wesley Publishing Company (1991)
14. Parr, T., Lilly, J., Wells, P., Klaren, R., Illouz, M., Mitchell, J., Stanchfield, S., Coker, J., Zukowski, M., Flack, C.: ANTLR Reference Manual. Technical report, MageLang Institute's jGuru.com (2001) See `http://www.antlr.org/doc/`.
15. Naumann, U., Forth, S.A., Tadjouddine, M., Pryce, J.D.: A Standard Interface for Elimination Sequences in Automatic Differentiation (Version 1). AMOR Report 01/3, Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, UK (2001)
16. Bischof, C.H., Haghighat, M.R.: Hierarchical approaches to automatic differentiation. In Berz, M., Bischof, C., Corliss, G., Griewank, A., eds.: Computational Differentiation: Tech., Appli., and Tools. SIAM, Philadelphia, Penn. (1996) 83–94
17. Tadjouddine, M., Forth, S.A., Pryce, J.D.: AD tools and prospects for optimal AD in CFD flux Jacobian calculations. [3] 247–252
18. Creusillet, B., Irigoin, F.: Exact vs. Approximate Array Region Analyses. In Sehr, D.C., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A., eds.: Lang. and Comp. for Parallel Computing. Volume 1239 of LNCS. Springer (1997) 86–100

# Hierarchical automatic differentiation by vertex elimination and source transformation

Tadjouddine, Mohamed

2003-01-01T00:00:00Z

Tadjouddine M, Forth SA, Pryce JD. (2003) Hierarchical automatic differentiation by vertex elimination and source transformation. Computational Science and Its Applications - ICCSA 2003, International Conference, 18-21 May 2003, Montreal, Canada. Proceedings, Part II, Lecture Notes in Computer Science, Volume 2668, pp.115-124

http://dx.doi.org/10.1007/3-540-44843-8_13