Proceedings of the Second International Conference in Computational Science,
Vol 2330, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2002

Performance Issues for Vertex Elimination Methods in Computing Jacobians using Automatic Differentiation

Mohamed Tadjouddine, Shaun A. Forth, John D. Pryce, & John K. Reid

AMOR Group, ESD
Cranfield University (RMCS Shrivenham),
Swindon SN6 8LA, UK
M.Tadjouddine@rmcs.cranfield.ac.uk

Abstract. In this paper, we present first results from EliAD, a new automatic differentiation tool. EliAD uses the **Eli**mination approach for **A**utomatic **D**ifferentation first advocated by Griewank and Reese (*Automatic Differentiation of Algorithms*, SIAM, 1991 p126-135). EliAD implements this technique via source-transformation, writing new Fortran code for the Jacobians of functions defined by existing Fortran code. Our results are from applying EliAD to the Roe flux routine commonly used in computational fluid dynamics. We show that we can produce code that calculates the associated flux Jacobian approaching or in excess of twice the speed of current state-of-the-art automatic differentiation tools. However, in order to do so we must take into account the architecture on which we are running our code. In particular, on processors that do not support *out-of-order execution*, we must reorder our derivative code so that values may be reused while in arithmetic registers in order that the floating point arithmetic pipeline may be kept full.

1 Introduction

In scientific computation, there is a frequent need to compute first derivatives of a function represented by a computer program. One way to achieve this is to use Automatic Differentiation (AD) [1–3], which allows for the computation of derivatives of a function represented by a computer program. 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], Odyssée [5], and TAMC [6] are well-established tools for this which make use of the standard forward and reverse modes of AD.

We have developed a new AD tool EliAD [7,8] which also uses source transformation. EliAD is written in Java and uses a parsing front-end generated by the ANTLR tool [9]. In contrast to the AD tools listed above, EliAD uses the vertex elimination approach of Griewank and Reese [10,2], later generalised to edge and face eliminations by Naumann [11,12]. Here, we consider only the vertex elimination approach, which typically needs less floating point operations

to calculate a Jacobian than the traditional forward and reverse methods implemented by ADIFOR, Odyssée or TAMC. We introduce the vertex elimination approach in section 2.

In section 3 we present results of applying our tool to the Roe flux [13] computation which is a central part of many computational fluid dynamics codes. We used various ways to sequence the elimination. We ran the resulting derivative codes on various machine/compiler combinations and at various levels of compiler optimisation. We found that execution times were not always in proportion to the number of floating point operations and that this effect was very machine-dependent.

Section 4 discusses the performance of certain elimination strategies regarding such machine dependent issues as cache and register utilisation and then points out the importance of statement ordering on certain processors. Section 5 concludes.

2 Automatic Differentiation By Elimination Techniques

AD relies on the use of the chain rule of calculus applied to elementary operations in an automated fashion. The input variables x with respect to which we need to compute derivatives are called *independent* variables. The output variables y whose derivatives are desired are called *dependent* variables. A variable which depends on an independent variable, and on which a dependent variable depends, is called an *intermediate* variable.

To illustrate the elimination approach, consider the code fragment comprising four scalar assignments in the left of equation 1.

$$\begin{array}{ll} x_3 = \phi_3(x_1, x_2) & \nabla x_3 = c_{3,1}\nabla x_1 + c_{3,2}\nabla x_2 \\ x_4 = \phi_4(x_2, x_3) & \nabla x_4 = c_{4,2}\nabla x_2 + c_{4,3}\nabla x_3 \\ x_5 = \phi_5(x_1, x_3) & \nabla x_5 = c_{5,1}\nabla x_1 + c_{5,3}\nabla x_3 \\ y = \phi_6(x_4, x_5) & \nabla y = c_{6,4}\nabla x_4 + c_{6,5}\nabla x_5 \end{array} \tag{1}$$

Denoting

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

we may use standard rules of calculus to write the linearised equations to the right of equation (1). These linearised equations describe the forward mode of AD and equivalent code would be produced by ADIFOR or TAMC enabling Jacobian calculation on setting $\nabla x_1 = (1, 0)$ and $\nabla x_2 = (0, 1)$. The Computational Graph (CG) for derivative calculation is sketched in Figure 1. Vertices 1, 2 represent the independents x_1, x_2; vertices 3, 4, 5 the intermediate variables x_3, x_4, x_5; and vertex 6 the dependent y. The edges are labelled with the local derivatives of the equivalent code statements given to the right of equation (1) as shown.

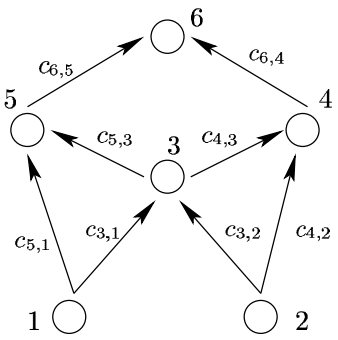


Fig. 1. An example computational graph

The corresponding matrix representation gives a linear system,

$$\begin{bmatrix} -1 & & & & \\ & -1 & & & \\ c_{3,1} & c_{3,2} & -1 & & \\ & c_{4,2} & c_{4,3} & -1 & \\ c_{5,1} & & c_{5,3} & & -1 \\ & & & c_{6,4} & c_{6,5} & -1 \end{bmatrix} \begin{bmatrix} \nabla x_1 \\ \nabla x_2 \\ \nabla x_3 \\ \nabla x_4 \\ \nabla x_5 \\ \nabla y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{2}$$

to solve for the derivatives, with zero entries in the matrix omitted for clarity.

We may interpret the elimination approach via either the computational graph or the matrix representation. Via the graph the Jacobian $\frac{\partial y}{\partial (x_1, x_2)}$ is determined by eliminating intermediate vertices from the graph until it becomes bipartite, see [2, 10, 11]. For example vertex 5 may be eliminated by creating new edges $c_{6,1} = c_{6,5} \times c_{5,1}$, $c_{6,3} = c_{6,5} \times c_{5,3}$ and then deleting vertex 5 and all the adjacent edges. We might then eliminate vertex 4 then 3, termed a reverse ordering.

In terms of the matrix representation, vertex elimination is equivalent to choosing a diagonal pivot from rows 3 to 5, in some order. At each step we eliminate all the coefficients under that pivot. For example by choosing row 5 as the pivot row we add the multiple $c_{6,5}$ of row 5 to row 6 and so produce entries $c_{6,1} = c_{6,5} \times c_{5,1}$, $c_{6,3} = c_{6,5} \times c_{5,3}$ in row 6. We see that this is identical to the above computational graph description. By then choosing rows 4 and 3 as pivots we are left with the Jacobian in elements c_{61} and c_{62}.

There are as many vertex elimination sequences as there are permutations of the intermediate vertices. We choose a sequence using heuristics from sparse matrix technology aimed at reducing fill-in, such as the Markowitz criterion studied in [10, 11]. By reducing fill-in such techniques have the desired side-effect of choosing an ordering which minimises the number of floating point operations at each elimination step.

3 Numerical Results

A typical application is the Roe flux routine [13]. This computes the numerical fluxes of mass, energy and momentum across a cell face in a finite-volume compressible flow calculation. 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 (200 lines of code).

We used EliAD to differentiate the Roe flux code with different vertex elimination orderings. Those orderings are Forward, Reverse, Markowitz (Mark), VLR, Markowitz Reverse Bias (Mark RB), and VLR Reverse Bias (VLR RB) eliminations; see for instance [11, 14] for details. The (reverse bias) variants of Markowitz/VLR use the last node with the minimum Markowitz/VLR cost instead of the first in the conventional implementations [14].

From the input code we generated derivative codes in one of the two following manners.

1. Statement level: local derivatives are computed for each statement, no matter how many variables x_i appear in its right-hand side.
2. Code list: local derivatives are computed for each statement in a rewritten code that has at most two variables x_i in each right-hand side.

Then, we applied the different vertex elimination strategies to the resulting computational graphs (62 intermediate vertices for the first case and 208 intermediates for the second case). We ran the subsequent derivative codes on a Silicon Graphics (SGI), Compaq Alpha(ALP) and two SUN machines with associated compilers as described in Table 1. Table 2 shows the CPU times required by the

Platform	Processor				Compiler	
	Processor	CPU	L1-Cache	L2-Cache	Compiler	Options
SGI	R12000	300MHz	64KB	8MB	f90 MIPSPro 7.3	-Ofast
ALP	EV6	667MHz	128KB	8MB	Compaq f95 5.4	-O5
SUN	Ultra10	440MHz	32KB	2MB	Workshop f90 6.0	-fast
NAG	Ultra10	440MHz	32KB	2MB	Nagware f95 4.0a	-O3 -native
FUJ	Ultra1	143MHz	32KB	0.5MB	Fujitsu f90 5.0	-Kfast

Table 1. Platforms (Processors and Compilers)

calculation of the original function on the different platforms.

	SGI	ALP	SUN	NAG	FUJ
time(F)	0.8	0.5	0.9	1.8	6.2

Table 2. Roe flux CPU timings (in μ seconds) of the function on different platforms

Table 3 summarises the ratio between the timings of the Jacobian and the original function by using different methods and platforms. Methods starting with VE are those using a vertex elimination strategy from our AD tool. We denoted FD for finite differences, SL for statement level and CL for code list. For TAMC, ftl denotes the forward mode and ad denotes the reverse mode.

Timings are based on $10,000$ evaluations for each method and an average of 10 runs of the executables from the different machines we used. We have checked the accuracy by evaluating the largest difference from the corresponding derivative found by ADIFOR [4]. This was less than 10^{-14} in all cases except finite differences, where it was about 10^{-7}, in line with the truncation error associated with this approximation.

Table 3 shows that on the SGI and ALP platforms the vertex elimination

No.	Method	$\frac{\text{time}(\nabla F)}{\text{time}(F)}$					$\frac{\text{FLOPs}(\nabla F)}{\text{FLOPs}(F)}$	No. lines
		SGI	ALP	SUN	NAG	FUJ		of code
1	FD (1-sided)	12.1	12.6	12.6	13.3	11.9	11.4	284
2	VE Forward(SL)	7.5	5.5	13.2	12.2	10.2	8.4	1534
3	VE Reverse(SL)	6.5	4.6	8.8	8.8	6.9	6.9	1318
4	VE Mark(SL)	6.4	5.1	9.8	10.4	6.7	7.2	1267
5	VE Mark RB(SL)	6.6	5.0	14.1	10.2	6.6	7.2	1261
6	VE VLR(SL)	6.0	4.5	8.9	9.1	6.1	6.5	1172
7	VE VLR RB(SL)	6.2	4.6	9.0	9.6	6.0	6.5	1170
8	VE Forward(CL)	7.3	5.2	18.7	19.2	14.9	12.1	3175
9	VE Reverse(CL)	6.7	4.7	10.7	9.9	9.5	8.9	2433
10	VE Mark(CL)	6.6	5.0	9.3	11.7	11.2	7.9	2058
11	VE Mark RB(CL)	6.6	5.3	10.2	11.0	10.9	7.7	2012
12	VE VLR(CL)	6.4	5.0	10.1	11.1	10.9	7.8	2116
13	VE VLR RB(CL)	7.2	5.4	10.8	12.4	12.1	8.4	2145
14	ADIFOR	15.9	9.8	31.4	50.5	14.4	15.0	823
15	TAMC-ftl	14.4	10.2	11.9	56.7	14.9	19.9	580
16	TAMC-ad	12.2	8.5	13.2	42.0	10.0	11.9	951

Table 3. Ratios of Jacobian to Function CPU timings and floating point operations on various platforms, and numbers of lines of code.

approach for computing Jacobians is about twice as fast as both conventional AD tools and finite differences. On the SUN, NAG and FUJ platforms the vertex elimination approach does not reach the same level of efficiency. We observed that derivative codes for the code list version are often slightly slower than their counterparts differentiated at statement level for ALP and SGI and sometimes much slower on SUN, NAG and FUJ.

Denoting by FLOPs(F) and FLOPs(∇F) the numbers of floating point operations required to compute respectively the function and its Jacobian, we obtained the ratios also shown in Table 3 from the Roe flux test case. We see that on the SGI and ALP platforms the vertex elimination derivative code ran faster than the

ratio of floating point operations predicts. On the SUN, NAG and FUJ platforms the reverse is true.

4 Performance Issues

The elimination strategies we used are based on criteria aimed at minimising the number of floating point multiplications required to accumulate the Jacobian. Van der Wijngaar and Saphir [15] have shown that, on RISC platforms, neither the number of floating point operations nor even the cache miss rate are sufficient to explain performance of a numerical code; and processor-specific instructions (which can be seen from the generated assembly code) and compiler maturity affect performance.

4.1 Floating Point Performance

If we consider the derivative code speed in terms of floating point operations per clock cycle as shown in Figure 2, all the elimination strategies are performing

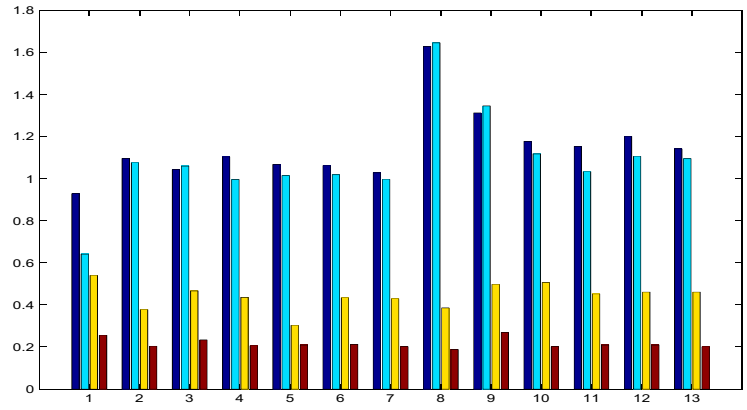


Fig. 2. Number of floating point operations per cycle performed by FD (number 1) and the elimination strategies $2-13$ (numbered as they appear in Table 3); the bars represent respectively results (from left to right) obtained on the SGI, ALP, SUN, and NAG data.

over 1 floating point operation per cycle on the SGI and ALP platforms. The Compaq EV6 and R12000 can perform up to 2 floating point operations per clock cycle [16] and a throughput of in excess of one floating point operation per clock cycle may be considered highly satisfactory. On the SUN, NAG and FUJ platforms a throughput of less than half a floating point operation per clock cycle is achieved. Again the theoretical maximum is 2. It was initially thought that cache misses may explain the poor timing ratios on these platforms.

4.2 Cache Misses

On the SGI, the outputs from SGI Origin's SpeedShop profiler showed that the derivative codes fit in the instruction cache and they differed from each other only by few cache misses in the primary/secondary data caches. The Compaq Alpha EV6 processor we used has caches comparable to those of the MIPS R12000 (see Table 1). Therefore, we do not expect cache misses to be crucial for the SGI and ALP platforms.

The Sparc Ultra 10 has relatively small caches compared with the R12000 and EV6. On the SUN and NAG platforms we noticed a performance degradation mainly of the code list method. Here, the forward elimination from the code list showed particularly poor performance. Profiling with the Sun Workshop 6.2 tools, we observed a 28% instruction cache miss rate for this forward elimination strategy. Using a SUN Blade 1000 (600 MHz) with double the primary cache size, that percentage came down to 10%. Interestingly the timing ratios remained similar to those from the Ultra 10 indicating that although instruction cache misses were occurring, the root cause of the poor performance lay elsewhere.

4.3 Statement Ordering in the Derivative Code

The SGI and ALP processors may perform 2 floating point operations per clock cycle through their floating point pipeline provided that the floating point operation being performed uses data currently in a register and does not require any data from the output of an operation ahead of it and still in the pipeline. The SGI platform has a latency of 2 clock cycles before the result of an operation is available whereas for the ALP platform this is 4 clock cycles [16]. This may explain the slightly better results from the SGI compared to the ALP in Figure 2.

Apart from their larger cache sizes the other main difference between the SGI/ALP platforms and the SUN/NAG/FUJ platforms is that the SGI and ALP processors support *out-of-order execution* [16]. This technique involves maintaining queues of floating point operations ready to be performed and if the one at the head of the current queue requires a value currently being processed in the floating point pipeline or not currently in a register, then the processor switches to another queue. Use of this technique reduces the importance of instruction scheduling by the compiler.

On the Ultra10 and Ultra1 processers of the SUN, NAG and FUJ platforms there is no out-of-order execution. The optimising compiler must perform more intensive optimisation of instruction scheduling in order to maintain good use of the floating point pipeline. The derivative codes we produce are large (see final column of table 3), contain no loops or branching and hence comprise one large basic block. We therefore expected optimising compilers to schedule floating point instructions effectively since there are no complications regarding the control flow of the program. Since this is obviously not occuring for the SUN, NAG and FUJ platforms, we conjectured that the local optimisation of instruction scheduling performed by the compiler might not be able to maintain a

good throughput in the floating point pipeline since statements using the same variables might be separated by several hundred lines of source code. Since an elimination sequence produces a certain set of elimination statements that may be placed in any order that respects their dependencies, we additionally conjectured that we might be able to reorder the statements in the derivative code to make better use of variables currently in registers. To assess the impact of statement ordering in the derivative code, we perturbed the order of statements without altering the data dependencies within the code. We reordered the assignment statements with the aim of using each assigned value soon after its assignment. We therefore used a modified version of the depth first search algorithm [17]. Namely, we regarded the statements in the derivative code as the vertices of an acyclic graph, with the output statements at the top and an edge from s to t if statement t uses the output from statement s. Then, we arranged the statements in the order produced by a depth-first traversal of this graph. The results are shown in Table 4. Comparing this with Table 3, we can see that

Method	$\frac{time(\nabla F)}{time(F)}$				
	SGI	ALP	SUN	NAG	FUJ
VE Forward(SL)	6.7	5.4	9.3	7.8	7.4
VE Reverse(SL)	5.2	4.6	7.7	7.2	6.2
VE Mark(SL)	5.8	5.2	8.6	7.3	6.1
VE Mark RB(SL)	5.8	5.1	7.9	6.8	6.0
VE VLR(SL)	6.4	4.6	8.2	6.3	5.6
VE VLR RB(SL)	6.8	4.5	7.5	6.2	5.6
VE Forward(CL)	6.2	5.2	8.2	13.3	8.5
VE Reverse(CL)	6.4	5.0	8.8	7.5	7.4
VE Mark(CL)	6.6	5.1	7.8	7.3	6.1
VE Mark RB(CL)	6.3	5.4	8.6	7.5	6.1
VE VLR(CL)	7.1	4.9	7.5	6.8	6.2
VE VLR RB(CL)	6.8	5.8	8.1	8.1	6.4

Table 4. Timings of the reordered derivative codes from the Roe flux test case on different platforms

statement reordering has improved performance on the platforms SUN, NAG and FUJ but it hardly made any difference on the ALP or SGI. For instance, looking at the SUN column of Table 4, the statement reordering has improved the generated code using the forward elimination from the code list by 56%. In this case, we observed that the resulting reordered derivative has 34% less loads and 72% less stores than the original derivative code. This indicates that the compiler was not performing an efficient usage of registers for the original derivative code. We now see that the Jacobian to function CPU time ratios have all significantly improved and we are approaching twice the efficiency of the best of the conventional techniques, AD or finite-differencing, of Table 3.

5 Conclusions

Automatic differentiation via vertex elimination allows us to produce Jacobian code which, in conjunction with an optimising compiler, efficiently uses a floating point pipeline on processors that support out-of-order execution. On other platforms tested and that supported a floating point pipeline we found that the compiler was not able to schedule the floating point operations with enough register re-use to allow the floating point processor to perform efficiently. Application of a simple statement reordering strategy based on depth first traversal enabled the compiler to optimise the resulting derivative code more effectively.

We conclude that, in conjunction with necessary statement reordering on some platforms, we have an AD tool that produces Jacobian code approaching or exceeding twice the efficiency of the current state-of-the-art for the Roe flux test problem. We are currently testing and evaluating the tool's performance on a number of test problems taken from the MINPACK-2 optimisation test suite [18].

Acknowledgements

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

References

1. Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penn., 1991.
2. Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, Penn., 2000.
3. George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors. *Automatic Differentiation: From Simulation to Optimization.* Computer and Information Science. Springer, New York, 2001.
4. Christian H. Bischof, Alan Carle, Paul D. Hovland, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0 user's guide (Revision D). Technical report, Mathematics and Computer Science Division Technical Memorandum no. 192 and Center for Research on Parallel Computation Technical Report CRPC-95516-S, 1998. See `www.mcs.anl.gov/adifor`.
5. Christèle Faure and Yves Papegay. Odyssée user's guide, version 1.7. Technical Report 0224, INRIA, Unité de Recherche, INRIA, Sophia Antipolis, 2004 Route des Lucioles, B.P. 93, 06902, Sophia Antipolis Cedex, France, September 1998. Available via `http://www.inria.fr/safir/SAM/Odyssee/odyssee.html`.
6. 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.
7. M. Tadjouddine, J.D. Pryce, and S.A. Forth. On the implementation of AD using elimination methods via source transformation. AMOR report 2000/08, Cranfield University (RMCS Shrivenham), Swindon, SN6 8LA, UK, Nov. 2000.

8. Mohamed Tadjouddine, Shaun A. Forth, John D. P ryce, and John K. Reid. On the Implementation of AD using Elimination Methods via Source Transformation: Derivative code gener ation. AMOR Report 01/4, Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, England, 2001.
9. T. Parr, J. Lilly, P. Wells, R. Klaren, M. Illouz, J. Mitchell, S. Stanchfield, J. Coker, M. Zukowski, and C. Flack. ANTLR Reference Manual. Technical report, MageLang Institute's jGuru.com, January 2000. Available via `http://www.antlr.org/doc/index.html`.
10. Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Griewank and Corliss [1], pages 126–135.
11. 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.
12. Uwe Naumann. Elimination techniques for cheap Jacobians. In Corliss et al. [3], chapter 29, pages 241–246.
13. Philip L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.
14. Mohamed Tadjouddine, Shaun A. Forth, and John D. Pryce. AD tools and prospects for optimal AD in CFD flux Jacobian calculations. In Corliss et al. [3], chapter 30, pages 247–252.
15. Rob F. Van der Wijngaart and William C. Saphir. On the efficacy of source code optimizations for cache-based processors. NAS Technical Report NAS-00-014, 2000.
16. Stefan Goedecker and Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM Philadelphia, 2001.
17. Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental algorithms.* Adison-Wesley, 1997.
18. Brett M. Averick and Jorge J. Moré. User guide for the MINPACK-2 test problem collection. Technical Memorandum ANL/MCS-TM-157, Argonne National Laboratory, Argonne, Ill., 1991. Also issued as Preprint 91-101 of the Army High Performance Computing Research Center at the University of Minnesota.



https://dspace.org/

Performance Issues for Vertex Elimination Methods in Computing Jacobians using Automatic Differentiation.

Tadjouddine, Mohamed; Forth, Shaun A.; Pryce, John D.; Reid, John K.

2002
Springer
https://doi.org/10.1007/3-540-46080-2_113
Tadjouddine M, Forth SA, Pryce JD, Reid JK. Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation. In: Proceedings of the 2nd International Conference in Computational Science, Volume 2330, Lecture Notes in Computer Science, Springer, Berlin, 2002

Downloaded from a DSpace repository. https://dspace.org/