The Chain Rule Revisited in Scientific Computing

Andreas Griewank

CRPC-TR91143
1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

The Chain Rule Revisited in Scientific Computing[1]

by

Andreas Griewank
Argonne National Laboratory

One of the few subjects many academics teach successfully to most of their undergraduates is differential calculus. So why would the research offices of the Army and Air Force fund a SIAM-sponsored workshop on *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*? Moreover, why would more than sixty academics, scientists and engineers from all over the world trek in early January to Breckenridge, Colorado, even though most of them showed remarkably little interest in skiing? We can only guess, but by most accounts automatic differentiation proved to be a worthwhile and surprisingly diverse field of scientific research and application.

The bottom-up, or forward, mode of automatic differentiation has been used for at least thirty years, and the mathematically slightly more intriguing top-down, or reverse, mode has attracted steadily increasing interest over the past two decades. In terms of operations counts the reverse mode yields gradient vectors for essentially the same effort that it takes to evaluate the underlying scalar function, no matter how many variables there are. The application of the reverse mode to large problems has been hampered by the potentially huge memory requirement of current implementations, but that obstacle can be removed, as we will explain later.

So — are you computing derivatives symbolically or by taking divided differences? This frequently posed question leaves many proponents of automatic differentiation gasping for air (a largely symbolic gesture at Breckenridge with an elevation of more than 9,000 feet). How can people ask such questions, right after one's lucid exposition of how variations on the good old chain rule will yield any derivative value humans or computers may desire, without any truncation error and practically free of charge? While none of the participants was brave enough to defend the virtues of divided differences, there were several representatives from the computer algebra community. Prof. Char (Drexel University) discussed various techniques for representing and manipulating evaluation procedures in computer algebra packages. Dr. Goldman (University of Twente) showed that on certain test problems of moderate size, an automated search for common terms in the expression trees of gradient components can reduce the evaluation effort to the level achieved by the reverse mode of automatic differentiation. It was generally hoped that for the particular task of differentiation, the flexibility and convenience of symbolic manipulation packages can eventually be combined with the numerical efficiency of automatic differentiation.

Getting Down to Basics

Like computer algebra, automatic differentiation is based on the fact that most functions of practical interest are compositions of basic functions, mostly binary arithmetic operations

[1] This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

and univariate transcendentals. This applies in particular to all functions that are entered symbolically as closed formulas or evaluated by programs written in a high-level computer language (e.g., Fortran or C). The analytic differentiation of these *elementary functions* is usually no problem at all, and their derivatives can be built into the software. For example, the scalar assignments $z = y \cdot x$ and $v = \sin(u)$ can be augmented by the statements $z' = y \cdot x' + y' \cdot x$ and $v' = \cos(u) \cdot u'$, respectively. Here the superscript prime denotes differentiation with respect to some or all independent variables. Moreover, the user can enter other special functions (e.g., quadratures) into the library of elementary functions. He or she merely has to supply formulas or routines for the evaluation of the new elementary function and its derivative(s). For simplicity we will assume in the following discussion that all elementary functions are scalar-valued and have a small number of arguments. The result values of most elementary functions are themselves used as arguments to other elementaries, and we will refer to them as *intermediate variables.* In general one would expect that the numbers N and M of independent and dependent variables of the composite function being evaluated is much smaller than the number T of intermediate variables. Consequently, the time for evaluating the composite function on a given serial computer is roughly proportional to T, which may thus serve as a measure of computational complexity.

Programs, Graphs, Trees, and Formulas

The independent, intermediate, and dependent variables form the vertices of a directed acyclic graph — a representation apparently first considered by Kantorovitch. An arc runs from one variable to another exactly if the value of the latter depends directly on the value of the former. In other words, for each elementary function all argument variables are connected to the vertex corresponding to its result variable. This *computational graph* was extensively used by F. L. Bauer [1] and others (See e.g. [10]) in the context of error estimation, a task that is closely related to the reverse mode, as Prof. Iri (University of Tokyo) explained at the workshop. For lack of space we will not here discuss this important side benefit of automatic differentiation [8]. Numbering the variables in the order that they are computed by the underlying serial computer program, one obtains a topological ordering of the directed graph. Any computation that traverses the vertices according to this original ordering will be called a forward (or bottom-up) sweep, and any computation that traverses the vertices in the opposite order will be called a reverse (or top-down) sweep.

One may associate with each arc the partial derivative of its destination with respect to its origin as an arc-value. The left part of Fig. 1 depicts the computational graph for the program of six assignments listed in the little box on top. If one is interested only in first derivatives, the elementary partials can be evaluated immediately at the current point, so that one obtains a *linearized computational graph.* When a function is defined *explicitly*, the computational graph reduces to the more familiar expression tree. In fact, by replicating intermediate vertices, any computational graph can be expanded to a forest (i.e., a collection of M disjoint trees). The result of this transformation is depicted in the right half of Fig. 1 with $M = 1$, as there is only one dependent variable f. Below the tree in Fig. 1 we see the corresponding algebraic formula. The tree has 18 nodes compared with the 8 nodes of the graph. The problem is that the number of replicas needed for a given intermediate vertex equals the number of distinct directed paths that connect it to dependent variables. Therefore, the attempt to express the dependent variables directly in

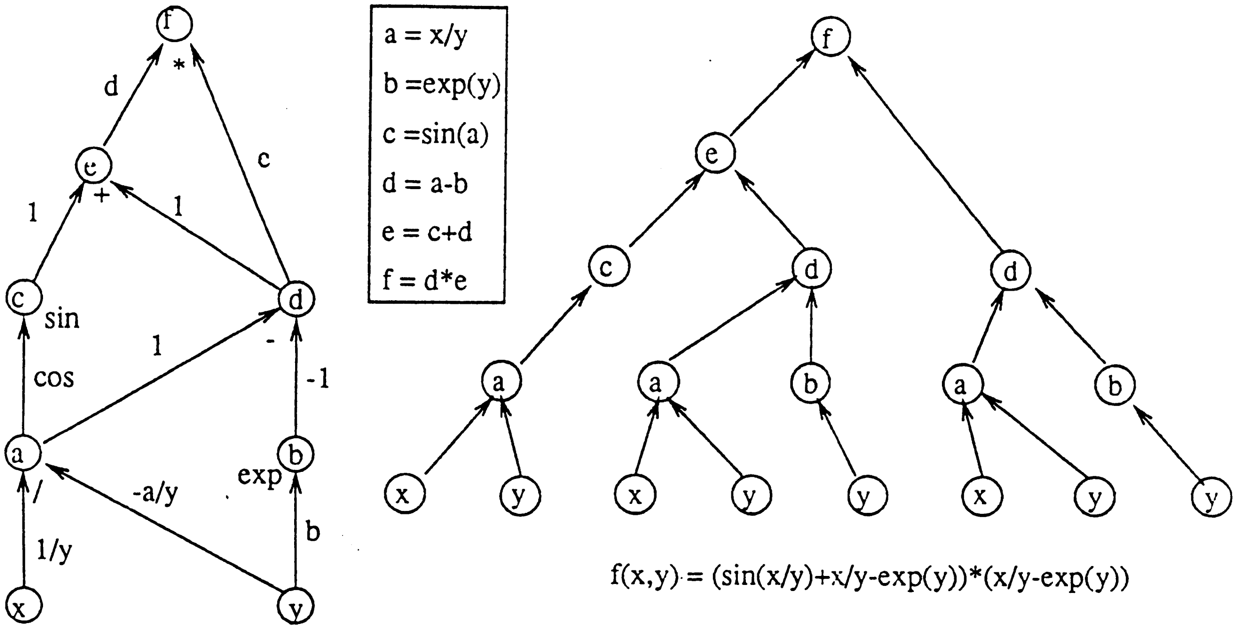

Figure 1: Representations of an evaluation algorithm with one dependent and two independent variables

terms of the independent variables often fails because of exponential growth of the memory requirement. Computer algebra systems such as Maple systematically avoid the replication of common subexpressions until asked to output a given function *explicitly*.

Philosophically, one might argue that the customary identification of the notion of an explicit representation with that of a formula (i.e., a linear string of algebraic operators) is outdated. If we had been brought up to represent and manipulate functional relations in the form of two-dimensional *formulas*, we would probably find the graph in Fig. 1 much more *readable* than the corresponding one-dimensional representation as a conventional algebraic expression. One needs only to enlarge the computational graph of Fig. 1 slightly to make the corresponding formula run over several lines, if not pages. It is a common experience that such symbolic *output* rarely provides analytic insights, and usually represents a poor procedure for evaluating the function numerically. Certainly, modern computer graphics and printing technology allow us to enter and retrieve two-dimensional data, so why don't we think — and possibly even program — that way?

Differentiation as Arithmetic

Possibly as a result of our training in calculus classes, we mostly think of differentiation as a tedious but straightforward process to obtain algebraic expressions for derivatives from the formula of the original undifferentiated function. In this effort the chain rule plays a key role, typically doubling the length of each nonlinear term to which it is applied. Everybody knows that the resulting formula collection tends to contain many common expressions. Consequently, the joint evaluation of a function and all its partials of interest (in a more or less ingenious fashion) is usually a lot cheaper than the sum of the costs for evaluating the function and each partial separately. As we will see, this effect is typical and can be exploited in a systematic fashion.

The computational graph of a function can be extended to a graph for a particular partial by appending a few auxiliary nodes to each original vertex. The size of the resulting computational graph is only a small multiple of the original, but it contains many diamonds created by the chain rule. Consequently, the corresponding expression tree is likely to be, by orders of magnitude, larger than the undifferentiated original. For example, the $(T+1)$-fold application of a nonlinear elementary function such as sine yields a univariate function, whose computational graph is simply a tree with $T+1$ nodes stacked on top of each other. This situation is depicted for $T = 3$ in Fig. 2. From each original vertex we may spawn a node for the cosine and one for the partial of the intermediate variable with respect to the only independent variable. The resulting graph for the overall partial has $3\,(T+1)$ nodes and can thus also be evaluated in $\mathcal{O}(T)$ time. However, if this graph is expanded into a tree, there are $\mathcal{O}(T^2)$ vertices, which make the formula pretty much unreadable.

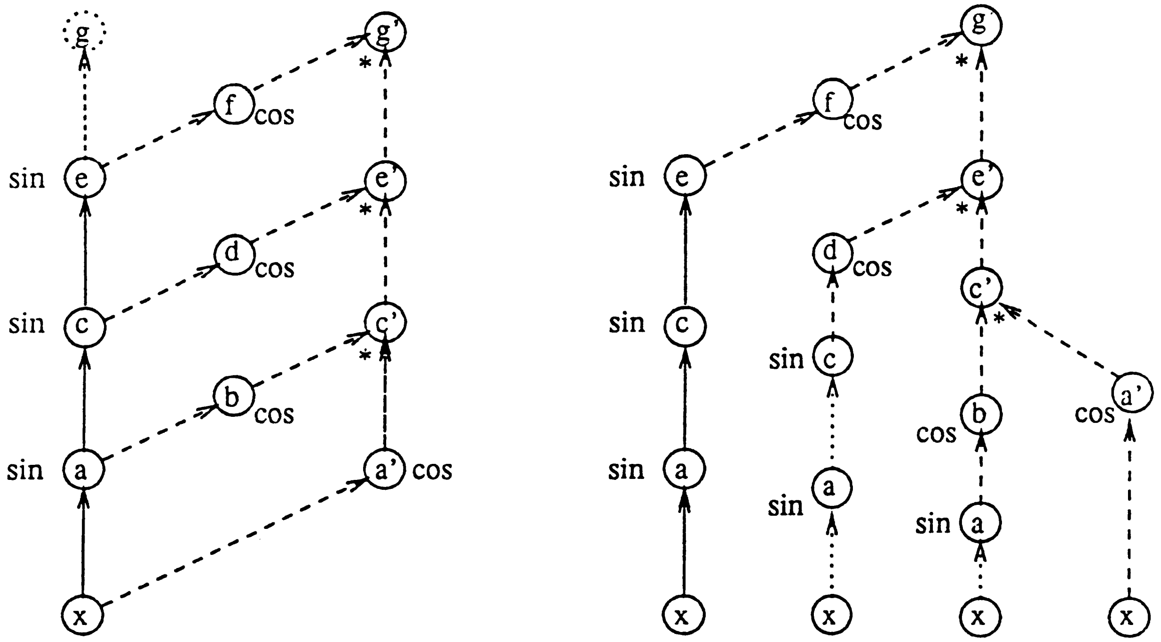

Figure 2: Graph and tree for derivative g' of $g = \sin(\sin(\sin(\sin(x))))$

Even the extension of the graph is not really a good idea, because it has to be done repeatedly for each partial of interest — usually at least the N gradient components. Instead, one should interpret the graph as a program for performing elementary operations on a set of truncated Taylor series, which form a ring in the algebraic sense. For example, one may replace the original argument x by a linear function $x + tv$, where t is a scalar parameter and x, v are both fixed N-vectors. Then one can propagate, through the computational graph, pairs of real numbers (z, z') representing the value of each intermediate variable and its first partial derivative with respect to t evaluated at $t = 0$. The resulting second component at dependent variable nodes represents their directional derivative along the tangent v in the domain of the function. Algebraically, these pairs form a normed ring to which the elementary arithmetic operations and standard univariate functions are easily extended. The same is true if we replace the original scalar variables by higher-order and/or multivariate Taylor series, for example, triplets consisting of the value, gradient, and Hessian of an intermediate variable with respect to all or some of the independent variables.

Again, all elementary operations and functions can be extended to these finite-dimensional real algebras by using the familiar rules of differentiation and power series manipulation. Obviously, the computational effort per node grows with the order of the highest Taylor coefficients and the number of independent variables.

In some way the manipulation of truncated Taylor series leads us back to the field of computer algebra, where efficient methods for combining and simplifying polynomials have been thoroughly investigated. In particular, we may utilize special methods for multiplying and composing polynomials, which at least asymptotically are significantly faster than the naive approach [6]. The relative efficiency of the various methods depends on the particular computing environment, the form of the truncated Taylor series, and the question of whether its coefficients are propagated all at once or recursively, as in the ODE case. It should be noted that, in contrast to the fully symbolic case, higher-order terms occurring during the combination of Taylor series are immediately truncated so that the complexity of their data representation does not increase. The key difference between fully symbolic differentiation and differential arithmetic is that the latter technique applies the chain rule to numbers rather than to formulas. In this way any embellishment or complication of the original computational graph can be avoided.

Applications of Differential Arithmetic

Differential arithmetic or algebra, as sketched above, has been implemented and used in many contexts during the past three decades (see e.g., [11]). Univariate Taylor series of orders up to a hundred have been used extensively for the numerical solution of ordinary differential equations with guaranteed error bounds [5]. Gradients and Hessians with respect to all independents are automatically calculated by using differentiation arithmetic in several integrated packages for optimization and nonlinear equation solving. Taylor series of orders up to ten and with up to seven variables have been used for beam tracing in optical systems and magnetic fields [3]. In all these applications, derivative values are obtained without any truncation errors, inaccuracies that always affect divided-difference approximations and render them practically useless for third and higher derivatives. The following list contains some of the applications for which participants at the Breckenridge workshop used automatic differentiation techniques:

- Exploratory orbit analysis for satellites.
- Simulation of the Superconducting Supercollider.
- Sensitivity analysis in economics.
- Nuclear reactor analysis and design.
- Environmental impact studies.
- Chemical process optimization.
- Low-level radioactive waste disposal.

Most of these applications were quite large, so that their numerical solution required substantial resources. No matter how the truncated Taylor arithmetic is implemented, its cost grows rapidly with the number of independent variables. In particular, the calculation of a full gradient typically requires N times as many operations and N times as much storage as the evaluation of the underlying scalar function. For dense Hessians, both computational costs grow by a factor of N^2. Since these cost increases are similar to those for approximating gradients and Hessians by differencing, they may seem a fair price to pay. However, in the alternative reverse mode of automatic differentiation, one can obtain gradients and Hessians for an N-th of the cost, though using a possibly larger memory.

Vertex Elimination on Linearized Graphs

When the linearized computational graph contains no intermediate vertices at all, its arcs represent exactly the nonzero entries of the Jacobian matrix. Otherwise it can be seen that the partial derivative of one dependent variable with respect to a particular independent variable is given by the sum of the values of all paths that connect the latter to the former [10]. Here, the value of a directed path is the product over the values of its arcs. The naive evaluation of these determinant-like sums of products would be grossly inefficient even if the graph were a tree. Instead one may use the following version of the chain rule to successively eliminate each intermediate vertex, which we will call the pivot node, in any particular order. A similar technique of successive graph modifications was proposed by T. Yoshida [17].

Consider all arcs connecting a predecessor of the pivot node to one of its successors. Increment the value of the arc by the value of the path running from the predecessor through the pivot to the successor. If a predecessor/successor pair was previously not directly connected, introduce an arc and initialize its value to that of the path through the pivot node. The elimination of the central node in Fig. 3 generates four new (dashed) arcs, increments two, and eliminates six. The arcs are annotated by letters representing their values, which are all immediately calculated as real numbers.

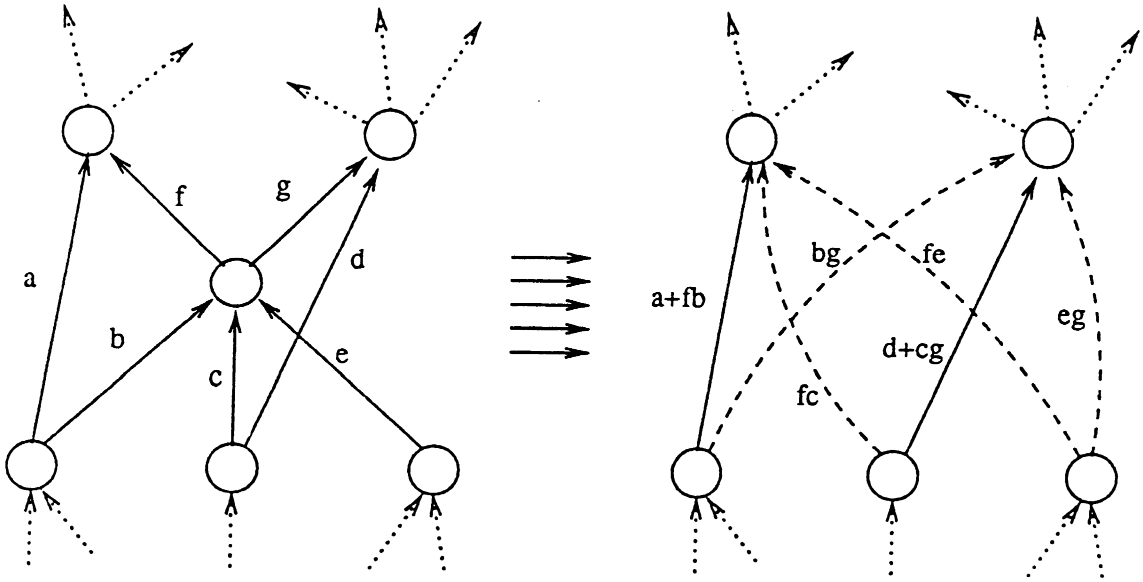

Figure 3: Elimination of an intermediate vertex

Topologically, this vertex elimination rule is exactly the same as the one used in Gaussian elimination with symmetric pivoting [12]. Here, however, there are no divisions, and the pivot node may be chosen with the sole aim of limiting fill-in (i.e., the total number of newly created arcs). The number of arithmetic operations needed to eliminate a certain node is equal the product of the number of its in-degree and out-degree, which count the number of its predecessors and successors, respectively. This so-called Markowitz degree is easy to compute and represents a bound on the local fill-in. For an optimal operations count, one would wish to eliminate the intermediate vertices in an order that minimizes the sum of all Markowitz counts at the pivot nodes. As is the case for the closely related sparse Gaussian elimination problem [12], it is conjectured that the combinatorial task of finding such a minimum ordering is in general NP-hard. However, heuristics like the local Markowitz rule may be quite efficient, and in special cases minimum orderings are easily characterized.

Retreat into the Forest

Suppose the computational graph is in fact a forest. Since vertex elimination as described above can never join disconnected subgraphs, we may then consider one particular tree, whose root represents a single dependent variable. In other words, we are trying to compute the gradient of a scalar component function with respect to all independent variables. Since the root is the only successor of all its immediate neighbors, each of them can be eliminated at a cost proportional to its in-degree. It can be seen that the in-degree of all other intermediate vertices remains unaffected; hence, eliminating the nodes opposite to their original ordering yields the gradient at a cost proportional to the sum of all original in-degrees, which is bounded by a small multiple of T. Thus we see that the time complexity of calculating the gradient in this *top-down*, or *reverse*, fashion is proportional to the cost of evaluating the underlying component function by itself. Because the computational graph was assumed to be a forest, this proportionality carries over to the whole vector function, whose Jacobian can thus be obtained by the reverse mode for little more than the cost of evaluating the function itself.

Unfortunately, the differentiation procedures of most computer algebra systems do just the opposite: they accumulate derivatives in the forward mode. Moreover, rather than calculating their numerical values at the current argument, the derivatives are formed as algebraic expressions of usually rapidly growing complexity. If the intermediate vertices in a tree are eliminated numerically in their original topological ordering, all out-degrees remain the same, but some in-degrees must be greater than one to begin with, and all are likely to increase further as a result fill-in. Consequently, the total number of arithmetic operations is likely to be much larger than the sum o f the original out-degrees, which is also proportional to T. The same may apply for the reverse mode if the computational graph is not a tree, because several component functions are computed by using common expressions. By reversing all arcs in a linearized computational graph, one obtains another (linear) computational graph for which the reverse elimination is equivalent to the forward mode on the original graph.

As an example, let us consider the function that is the multiplicative product of its N independent variables. The i-th component of the gradient is simply the product of the first $i-1$ variables and the last $N-i$ variables. If these initial and terminal subproducts are computed recursively, the product function and its full gradient are evaluated at the

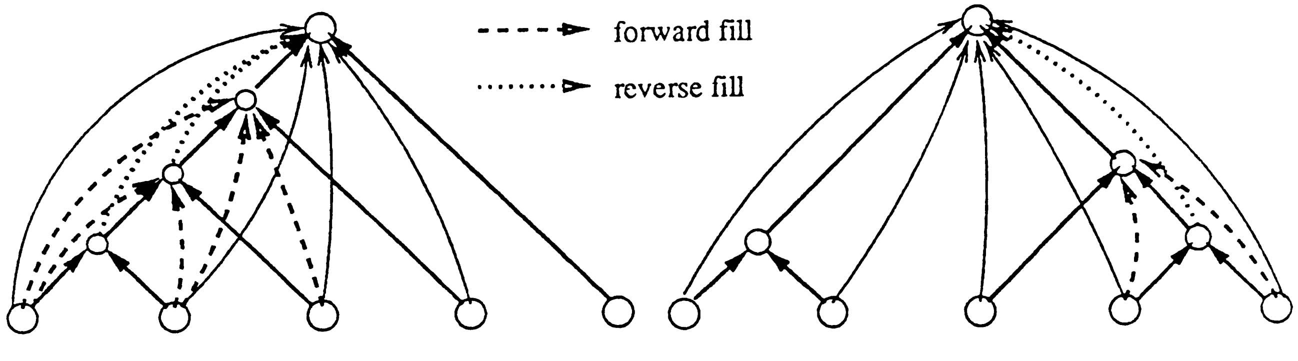

Figure 4: Elimination-fill on unbalanced and balanced tree for product function

total expense of $3N - 1$ multiplications. As was already demonstrated by Speelpenning [13], this seemingly ingenious scheme for differentiating the product amounts to reverse elimination on the computational graph. In contrast, the forward mode involves exactly $N(N-1)/2$ multiplications. Here we have tacitly assumed that the product is evaluated by a simple loop, so that the graph has $T = N - 2$ linearly ordered intermediate vertices, as depicted in the left half of Fig. 4 for the case $N = 5$. Because of their commutativity, the multiplications can be arranged differently for example, in a binary tree of height $\log_2 N$ as depicted in the right half of Fig. 4. In both halves the thick solid arcs represent original dependencies, whereas the dashed and dotted arcs indicate the fill-in generated by forward and reverse elimination, respectively. The thin solid arcs connecting the independents to the single dependent variable represent the gradient components and must therefore be generated by either method. Whereas the reverse mode generates at most one fill-in arc per node, the forward mode creates many more, especially on the unbalanced tree depicted on the left. For general N, balancing the product tree to minimize its height improves the complexity of the forward mode to $N \log_2$, which is still slightly worse than the $\mathcal{O}(N)$ complexity of the reverse mode. Finally, it should be noted that any attempt to create a separate expression or procedure for each gradient component must result in an evaluation cost of order $N * (N-1)$, as each independent variable enters into all but one of the gradient components.

Going Back in Time

The remarkable fact that the ratio between the time-complexity of gradients and the underlying functions does not depend at all on the number of independent variables N has been known — though not widely understood — for quite some time. As explained by Prof. Prof. Iri and Prof. George Cybenko (CSRD University of Illinois), the top-down, or reverse, mode of automatic differentiation was already being used in the early seventies: by Seppo Linnainmaa [9] for the estimation of rounding errors and by Paul Werbos [16] for training neural networks by backpropagation. In 1980 B. Speelpenning (a Ph.D. student of W. Gear) wrote a Fortran precompiler that implements the reverse mode quite efficiently [13]. In 1983 a paper of W. Baur and V. Strassen [2] appeared in Theoretical Computer Science that established for rational functions a complexity bound of 3 in terms of multiplications. Since then, several groups of researchers have contributed to the theory and developed various software implementations.

An even older, closely related tradition is that of adjoint differential equations in optimal control. Whenever a state vector evolves according to an ordinary differential equation, one may simultaneously integrate its partial with respect to a particular input parameter as a solution to an auxiliary linear differential equation. As explained by Prof. Evtushenko (Soviet Academy of Sciences, Moscow), the simultaneous integration of this impulse equation in time corresponds to the forward mode of automatic differentiation. On the other hand, if one is interested in the gradient of some final performance measure with respect to many input parameters, the integration of the adjoint equation backward in time is much more efficient. The reverse mode discussed above is simply a discrete analog of this well-established costate approach. The two methods share an obvious disadvantage, namely, the apparent need to store the trace of the forward integration or evaluation. Also, as explained at the workshop by Prof. Burns (Virginia Polytechnic Institute and State University), discretizations of control problems must satisfy additional consistency requirements in order to yield convergent costates. As reported by Prof. Talagrand (Laboratoire de Météorologie Dynamique, Paris), and Prof. Navon (Florida State University), handcoded adjoints have been used extensively in four-dimensional data assimilation for short- and medium-term weather modeling. P.C. Shah (Schlumberger Well-Services) described the application of adjoint equations for distributed parameter estimation in models of building structures and petroleum reservoirs. W. C. Thacker (Atlantic Oceanographic and Meteorological Laboratory), discussed the task of automating the coding of adjoints without sacrificing too much computational efficiency. [14].

Checking the Memory

So far, we have considered only the temporal complexity in comparing the forward and reverse mode. The memory requirement of the forward mode is relatively easy to predict. Suppose we wish to calculate an $M \times N$ Jacobian, whose rows contain at most $n \leq N$ nonzero elements. If we exclude the possibility of exact cancellations, it follows that none of the vertices can ever have an in-degree greater than n during the forward elimination process. As a consequence, the forward Jacobian accumulation requires no more than n times the storage and operations count of the underlying function evaluation. Similarly $m \leq M$, the maximal number of nonzeros in any column of the Jacobian, bounds the operations count for the reverse accumulation relative to the function evaluation. However, whereas the forward mode can be propagated simultaneously with the function evaluation, the reverse elimination can begin only after the function has been fully evaluated. Moreover, all intermediate variable values enter into the reverse accumulation and must therefore be kept in memory, unless they can be calculated. Speelpenning was apparently the first to notice that the data representing the computational graph can be generated and accessed in exactly the opposite order. Hence this potentially very large data set can be manually or automatically paged out to disk without unreasonable run-time penalties. On the other hand, the randomly accessed memory requirement for the reverse mode can be limited to m times that of the function evaluation itself. Speelpenning's precompiler and some more recent implementations split the storage requirement accordingly into sequentially and randomly accessed memory, abbreviated as RAM and SAM in Table 1. Nevertheless, there is reason for concern, because these systems typically generate some 15–25 bytes of

SAM for each elementary operation during the function evaluation.

On a weather-related PDE problem, a comparison of three automatic differentiation packages with a handcoded adjoint program showed that they used nearly a hundred times more memory than was really needed. The handcoded program stored only the trace of the time-dependent solution function itself and recalculated all the other intermediates during the reverse sweep. In contrast, the automatic packages stored the large number of intermediate quantities that were generated during each time step: thus their SAM requirement was proportional to T. Theoretically one can see quite easily that by using *multilevel differentiation* as proposed by Volin and Ostrovskii [15], the storage requirement of the reverse mode can be restricted to grow like a root or even the logarithm of T. To this end, slices of the graph need to be recalculated from suitably chosen checkpoints at which a snapshot of all variable values has been taken during a previous forward evaluation. Most operating systems utilize such checkpointing to allow restarts after an exception and to swap user programs in and out on a timesharing system. Unfortunately, this approach has not yet been implemented for automatic differentiation, but its existence should make the reverse mode eventually a viable proposition even for very large problems.

Complexity of Jacobians

Basic complexity bounds for accumulating Jacobians by various methods are listed in Table 1. The first row serves the purpose of defining the quantities $T, R,$ and S, which represent the run-time, core, and disk space needed by the original function evaluation program. Similarly, the first entry in the second row may be viewed as defining the integer $n \leq N$. This bound represents a suitable number of groups into which the columns of the Jacobian can be partitioned for the purpose of differencing. The columns in each group must be pairwise structurally orthogonal, which can be ensured by the graph coloring approach of Coleman and Moré [4]. The same algorithm can be applied to the sparsity pattern of the transposed Jacobian yielding a corresponding number $m \leq M$ of row groups. The usually somewhat smaller effective dimensions $n \leq \bar{n}, m \leq \bar{m}$, still denote the maximal number of nonzero entries in the columns and rows of the Jacobian, respectively.

The prefixes S and M of the three procedures Forward, Reverse, and Reverse^k specify whether the variant calculates the whole Jacobian in one single sweep or in groups of columns or rows using multiple sweeps over the computational graph. As we see, the multiple forward sweep has exactly the same complexity as the Coleman-Moré [4] differencing scheme, where small factors that depend on the computing system but not the particular problem are ignored. The single forward method may save some computing time, but does require substantially more storage, since all scalars in the original variable space are replaced by N-vectors with up to n nonzero entries. The multiple and single reverse methods may be interpreted as the corresponding forward methods applied to the reversed graph, that is, the graph obtained by reorienting all arcs without changing thei r values. By turning the graph upside down in this way, one interchanges the role of the independent and dependent variable vertices. By reevaluating the original function, one can recreate the graph from the independent variables during each forward sweep, but this procedure is not possible during reverse sweeps. Therefore the sequential-access memory requirement for the two simple reverse schemes is larger by T (i.e., the space needed for storing the linearized computational graph at the current argument). This severe limitation is avoided by the as yet only theoretically conceived procedures denoted by Reverse^k with $k > 1$. If one

Table 1: Complexity of Jacobian evaluation, $n \leq \bar{n} \leq N$ and $m \leq \bar{m} \leq M$

Mode	OPS	RAM	SAM
Function	T	R	S
Differencing	$\bar{n}\, T$	R	S
M-Forward	$\bar{n}\, T$	R	S
S-Forward	$n\, T$	$n\, R$	S
M-Reverse	$\bar{m}\, T$	R	$S + T$
S-Reverse	$m\, T$	$m\, R$	$S + T$
M-Reverse^k	$\bar{m}\, T\, k$	R	$S + R\, k\, (T/R)^{1/k}$
S-Reverse^k	$m\, T\, k$	$m\, R$	$S + R\, k\, (T/R)^{1/k}$

accepts an increase in the number of operations by a fixed factor k, the corresponding SAM requirement is limited essentially by the k-th root of the original run-time T. For the particular choice $k = ln(T/R)$, total storage and computing time both grow logarithmically in the ratio T/R.

As we mentioned above, the vertices in the linearized graph can be eliminated in any order, and one can easily construct examples for which all forward and reverse modes are much less efficient than orderings that begin the elimination in the middle of the graph rather than at one end. An experimental implementation of the Markowitz criterion has produced promising results on some test problems, but the pivot search over the whole graph is quite costly, in terms of both logical operations and core memory. Further investigations should yield relaxations of the Markowitz-rule that limit fill-in as much as possible without randomly accessing the whole graph structure.

Parallelism and and Other Future Challenges

Much remains to be done in all three categories of theory, implementation, and application. As we indicated above, the accumulation of Jacobians poses a host of combinatorial optimization problems even on a serial machine, and it has only very recently been considered in a parallel context, as discussed by Dr. Bischof (Argonne National Laboratory) at the workshop. Moreover, in many applications, Jacobians are used only as a means for calculating Newton steps, and it has been shown that these vectors can sometimes be computed more efficiently without accumulating the Jacobian at all [7]. Prof. Dixon demonstrated that the computational graph of gradients computed in the reverse mode is in some sense symmetric, a strong property that should certainly be exploited in the evaluation of Hessians and the direct calculation of Newton steps. Since it represents data dependencies on the scalar level, the computational graph allows (in theory) the determination of optimally concurrent evaluation schedules for both function and derivatives. For practical purposes, many implementation issues — for example, appropriate granularity and maintenance of vectorizability — need to be resolved. The numerical analysis of nonlinear bifurcations and the solution of differential algebraic equations would greatly benefit from efficient and accurate methods for evaluating derivatives of implicit functions. Except for the contributions of Dr. Lawson (JPL) and Prof. Flanders (University of Michigan), in the univariate case, the automatic differentiation community has so far not given much attention to implicit functions: in particular, no basic complexity estimates have appeared in the literature. Many participants at the Breckenridge workshop felt that the currently insufficient distribution of efficient, reliable, and user-friendly software is the main obstacle to the wider application of automatic differentiation. A large variety of implementations does, in fact, exist, mostly in the form of precompilers for Fortran codes, overloading utilities in Ada or C++, and integrated optimization packages with symbolic function entry. A survey of 28 such software packages will appear in the Proceedings of the workshop, which are to be published by SIAM and edited by George Corliss (Marquette University) and Andreas Griewank (Argonne National Laboratory). A draft version cluding all relevant contacts can be obtained by e-mail from the survey's author, David Juedes, at the e-mail address juedes@atanoff.cs.iastate.edu . For various reasons, none of the current implementations are entirely satisfactory, and all require a large amount of extra storage in the reverse mode. As we indicated above, that limitation can be overcome by checkpointing techniques that are probably best implemented at the compiler level.

References

[1] F. L. Bauer (1974). *Computational Graphs and Rounding Errors.* SINUM, Vol. 11, no. 1, pp. 87–96.

[2] W. Baur and V. Strassen (1983). "The Complexity of Partial Derivatives", *Theoretical Computer Science*, Vol. 22, pp. 317–330.

[3] M. Berz (1990). *Computational Aspects of Design and Simulation: COSY INFINITY.* Nuclear Instruments and Methods, A298:473.

[4] T. F. Coleman and J. J. Moré (1983). *Estimation of Sparse Jacobian Matrices and Graph Coloring Problems* . SINUM, Vol. 20, pp. 187–209.

[5] G. F. Corliss and Y. F. Chang (1982). *Solving Ordinary Differential Equations using Taylor Series.* ACM TOMS, Vol. 8, pp. 114–144.

[6] R. J. Fateman (1974). *Polynomial Multiplication, Powers, and Asymptotic Analysis: Some Comments.* SIAM J. Comput., Vol. 3, pp. 196–213.

[7] A. Griewank (1990). *Direct Calculation of Newton Steps without Accumulating Jacobians*, in "Large-Scale Numerical Optimization", T. F. Coleman and Yuying Li, eds., SIAM, pp. 115–137.

[8] M. Iri, T. Tsuchiya, and M. Hoshi (1988). *Automatic Computation of Partial Derivatives and Rounding Error Estimates with Applications to Large-Scale Systems of Nonlinear Equations. Journal of Computational and Applied Mathematics*, Vol. 24, pp. 365–392.

[9] S. Linnainmaa (1976). *Taylor Expansion of the Accumulated Rounding Error.* BIT, Vol. 16, pp. 146–160.

[10] W. Miller and C. Wrathall (1980). *Software for Roundoff Analysis of Matrix Algorithms.* Academic Press, New York.

[11] L. B. Rall (1981). *Automatic Differentiation - Techniques and Applications*, Springer Lecture Notes in Computer Science, Vol. 120, Springer-Verlag, Berlin.

[12] D. J. Rose and R. E. Tarjan (1978). *Algorithmic Aspects of Vertex Elimination on Directed Graphs.* SIAM J. Appl. Math., Vol. 34, pp. 177–197.

[13] B. Speelpenning (1980). *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

[14] W. C. Thacker (1990). *Large Least-Squares Problems and the Need for Automating the Generation of Adjoint Codes*, in Lecture Notes in Mathematics 26, Computational Solution of Nonlinear Systems of Equations, American Mathematical Society, pp. 645–677.

[15] Yu. M. Volin and G. M. Ostrovskii (1985). *Automatic Computation of Derivatives with the Use of the Multilevel Differentiating Technique.* Computers and Mathematics with Applications, Vol. 11, pp. 1099-1114.

[16] P. J. Werbos (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Ph.D. dissertation, Department of Statistics, Harvard University, Cambridge, Massachusetts.

[17] T. Yoshida (1987). *Derivation of a Computational Process for Partial Derivatives of Functions Using Transformations of a Graph.* Transactions of IPSJ , Vol. 11, pp. 1112–1120.

Mathematics and Computer Science Division
Building 221
Argonne National Laboratory
Argonne, Illinois 60439-4844

Recent Preprints:

H. G. Kaper and M. K. Kwong, "On Two Conjectures Concerning the Multiplicity of Solutions of a Dirichlet Problem," MCS-P211-0191.

Kenneth W. Dritz, "Ada Solutions to the Salishan Problems," MCS-P212-0291.

I. B. Tjoa and L. T. Biegler, "Simultaneous Strategies for Data Reconciliation and Gross Error Detection of Nonlinear Systems," MCS-P213-0291.

I-Liang Chern, "A Control Volume Method on an Icosahedral Grid for Numerical Integration of the Shallow-Water Equations on the Sphere," MCS-P214-0291.

Ian Foster, William Gropp, and Rick Stevens, "The Parallel Scalability of the Spectral Transform Method," MCS-P215-0291.

Man Kam Kwong, "On the Unboundedness of the Numer of Solutions of a Dirichlet Problem," MCS-P216-0291.

Xiaolong Yang, Norman J. Zabusky, John F. Hawley, and I-Liang Chern, "Vorticity Generation and Evolution in Shock-Accelerated Density-Stratified Interfaces," MCS-P217-0291.

D. Levine, D. Callahan, and J. Dongarra, "A Comparative Study of Automatic Vectorizing Compilers," MCS-P218-0391.

W. W. McCune, "Single Axioms for the Left Group and Right Group Calculi," MCS-P219-0391.

W. W. McCune, "Automated Discovery of New Axiomatizations for the Left Group and Right Group Calculi," MCS-P220-0391.

Mark Jones and Paul Plassmann, "Fortran Subroutines to Compute Improved Incomplete Cholesky Factorizations," MCS-P221-0391.

Larry Wos, "The Problem of Choosing the Type of Subsumption to Use," MCS-P222-0391.

Larry Wos, "The Problem of Choosing the Representation, Inference Rule, and Strategy," MCS-P223-0391.

Lorenz T. Biegler and James B. Rawlings, "Optimization Approaches to Nonlinear Model Predictive Control," MCS-P224-0391.

Christian H. Bischof and Ping Tak Peter Tang, "Robust Incremental Condition Estimation," MCS-P225-0391.

Stephen J. Wright, "Interior Point Methods for Optimal Control of Discrete-Time Systems," MCS-P226-0491.

Andreas Griewank, "Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation," MCS-P228-0491.

Stephen Wright, "Stable Parallel Elimination for Boundary Value ODEs," MCS-P229-0491.

Gui-Qiang Chen, "Hyperbolic Systems of Conservation Laws with Symmetry," MCS-P230-0491.

Larry Wos, "Automated Reasoning and Bledsoe's Dream for the Field," MCS-P231-0491.

Larry Wos and Robert Veroff, "Automated Reasoning in Relation to Logic," MCS-P232-0491.