ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

ANL/MCS-TM-183

Workshop Report on

First Theory Institute on Computational Differentiation

held at Argonne National Laboratory
May 24–26, 1993

edited by

Christian H. Bischof, Andreas Griewank, and Peyvand M. Khademi

Mathematics and Computer Science Division

Technical Memorandum No. 183

December 1993

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Introduction

The first Theory Institute on Combinatorial Challenges in Computational Differentiation was held at Argonne National Laboratory, May 24–26. Computational differentiation (CD) is concerned with tools, techniques, and mathematics for generating, with little human effort, efficient and accurate derivative codes from programs written in such computer languages as C and Fortran.

Organized by Christian Bischof and Andreas Griewank of Argonne's Mathematics and Computer Science Division, the institute brought together many leading developers of CD theory and implementations, along with the following prominent representatives of related areas:

Bruce Char (Drexel University)
Harley Flanders (University of Michigan)
John Gilbert (Xerox PARC)
Robert Grossman (University of Illinois, Chicago)
Kieran Herley (University College, Cork, Ireland)
Erich Kaltofen (Rensselaer Polytech Institute)
Jacques Morgenstern (INRIA Sophia-Antipolis and the University of Nice, France)
Joel Saltz (University of Maryland)
Stephen Watt (IBM Thomas J. Watson Research Center)

This diversity provided for a stimulating meeting and ensured a certain element of suspense. The primary purposes of the meeting were to explore the deep complexity issues that lie at the heart of the computation of derivatives from computer programs and to provide a forum for brainstorming on future research directions, including the applications of automatic differentiation (AD) in scientific computing and the development of AD tools.

What Is Automatic Differentiation?

So-called AD techniques are the basis for all efforts related to computational differentiation; with these techniques, it is possible to determine the sensitivities of certain output variables with respect to certain input variables for any mapping that is described by a computer program. Automatic differentiation relies on the fact that every function is evaluated on a computer as a sequence of elementary operations, such as +, -, and intrinsics such as sin and log. By successively applying the chain rule to the composition of those elementary operations, derivatives can be computed exactly (up to machine roundoff) and in a completely mechanical fashion. These successive chain rule applications traditionally proceed in one of two modes: the forward mode, which propagates derivatives of intermediate values with respect to the inputs, and the reverse mode—a discrete analog of the adjoint equations used in optimal control—which propagates derivatives of outputs with respect to the intermediates. A broad introduction to the field was given by Andreas Griewank ("The Chain Rule Revisited in Scientific Computing," SIAM News, May and July 1991).

Tools for AD produce code for derivative computations either by operator overloading or by explicit augmentation of the original code with statements for derivative computations.

More traditional methods for differentiation, such as divided differences, hand-derivation, and symbolic approaches, are less efficient, more error-prone, or not generally applicable to large codes. AD software packages—some of which are briefly discussed later in this article—are already competitive with the other differentiation methods in terms of efficiency of derivative computation, with many approaches to speedup as yet unexplored.

Speakers Document a Growing Research Area

The talks presented at the institute touched on computer algebra, numerical linear algebra, complexity theory, graph theory, symbolic computing, parallel processing, and compiler design. The diversity of the participants' backgrounds provided a fertile ground for extensive interdisciplinary discussions both during and outside the framework of the formal talks. It emerged in the course of the institute that the research area of computational differentiation had grown from a series of isolated rediscoveries into a cohesive body of basic tenets and that the scope of application areas had been extended. Important characteristics of the field discussed at the meeting included theoretical results confirming the combinatorial complexity of the underlying problems, and the development of general-purpose differentiation tools.

Automatic differentiation can be posed as a graph elimination problem, where the sequence of operations can be represented as the nodes of a computational directed acyclic graph (CDAG) and derivatives are propagated along the edges of the graph. Kieran Herley presented a proof at this workshop that the problem of minimizing "fill-in" of the CDAG during Jacobian accumulation is NP-complete. This is an important result not only because it opens the door to the invention of ingenious heuristics for the implementation of AD, but also because it shows that the efficient computation of derivatives is a hard problem.

Transpositionality, another interesting theoretical result, was discussed by Erich Kaltofen. Some computational targets, such as the solutions of linear systems, can be conveniently characterized as gradients of functionals. Using the fact that the reverse mode allows the computation of the gradient of a function at a cost that is at most five times that of evaluating the function, independent of the number of independent variables, Kaltofen showed that the application of AD to such functionals in some cases results in asymptotic complexity lower than that of direct methods. An example would be parallel matrix inversion performed by differentiating a parallel method for computing the determinant.

Andreas Griewank addressed the basic challenge of the reverse mode of AD, namely, the potentially extremely large memory requirement that results from the need to record each individual transformation that nonlinearly impacts the final result. The memory requirements can be dramatically reduced by the related techniques of multilevel differentiation and recursive checkpointing, at the expense of the moderate increase in the operations count that results from repeated recalculation of intermediates.

John Gilbert presented a survey of combinatorial sparse matrix theory as it relates to AD. He also illustrated Griewank's checkpointing algorithm in terms of a two-color pebble game on a CDAG. Bruce Christianson of Hatfield Polytechnic, England, introduced a novel approach to nested differentiation whereby the reverse accumulation mode is extended to compute successively higher-order derivatives.

Stephen Watt prefaced his theoretical presentation with a look at the potential advantages of analytic derivatives over finite differencing in real-world applications, where the computational efficiency of AD has translated into substantial resource savings. He cited the experience at IBM Burlington, where the use of analytic derivatives in circuit simulations led to an estimated $1,000,000 in monthly savings in terms of CPU time. In his talk on algebraic methods, he showed how, through the specification of canonical and rule-based simplifications, expressions appearing in automatically differentiated code can be re-expressed in ways that lead to more parsimonious computation. Finally, he showed how dataflow analysis techniques can enhance the effectiveness of algebraic methods.

Jacques Morgenstern presented results and methods in algebraic complexity, including ensembles computation, linear forms, and rational functions. Robert Grossman presented an algebra of rooted trees, which allows the symbolic translation of algorithms for manipulating differential operators into assertions about labeled trees, leading to exponential speedup over naive algorithms.

Christian Bischof discussed issues arising from a comprehensive study of the forward-mode approach using ADIFOR, a primarily forward-mode AD tool developed jointly at Argonne and Rice University. He concluded that traditional measures favoring the reverse over the forward mode were too simplistic and that the forward mode, if supported by dynamic data structures and applied in a hierarchical fashion, may be able to deliver derivatives, and, in particular gradients, much less expensively. He also illustrated the need to maintain forward-mode quantities (i.e., derivatives with respect to the input) in the computation of derivatives of iterative processes.

Alan Carle of Rice discussed compile-time analysis in ADIFOR, describing in particular how ADIFOR automatically identifies active variables, that is, those variables on the dependency path from the independent to dependent variables, in the presence of scoping, actual-formal parameter binding, and memory layout. Limiting the number of active variables at compile time leads directly to reduced time and space requirements for derivative computations.

Martin Berz of Michigan State University interpreted computational differentiation in terms of nonstandard arithmetics, by viewing the forward mode of AD in a more general way as analysis on non-Archimedean structures. The methods discussed allow for the practical computation of higher-order derivatives needed in Taylor series and have been applied in the actual design of new accelerators, including the Superconducting SuperCollider in Texas and the electron–positron collider LEP at CERN in Europe.

Joel Saltz discussed the implications of his group's ongoing research on runtime compilation techniques as applied to parallel computations. Such runtime techniques could equally well be applied to efficient sparse derivative computations.

Tools and Applications

The institute also provided a forum for the assessment of the state of the art in AD in terms of tools and applications. Jim Horwedel of Oak Ridge National Laboratory described the latest version of GRESS (the GRadient Enhanced Software System). GRESS was designed to allow the application of AD to large-scale Fortran programs in the nuclear industry without requiring changes in the coding. GRESS implements both the forward and the reverse modes. The modular differentiation technique (MDT) uses both the forward and the reverse modes of GRESS to restrict the growth of execution time and storage requirements.

The aforementioned ADIFOR (Automatic DIfferentiation of FORtran) is a general-purpose tool developed to deal with real-world Fortran 77 codes. ADIFOR, which employs parts of Rice University's ParaScope parallel programming infrastructure, has been successfully applied to a variety of codes in areas ranging from biomedical modeling to turbulent fluid flow.

Ralf Giering of the Max-Plank-Institute for Meteorology in Hamburg, Germany, presented AMC (Adjoint Model Compiler), a reverse-mode Fortran source-to-source translation tool developed to support adjoint code generation for climate modeling codes. Michael Gleicher of Carnegie Mellon University demonstrated his graphical interaction system, which uses AD to provide dynamic functionality for interactively built mechanical models. Harley Flanders demonstrated the implementation of his AD tool for the PC Basic environment.

Also discussed at the workshop was ADOL-C, a tool developed for C/C++ by Andreas Griewank with David Juedes, Duane Yoder, Jean Utke, and other students to compute derivatives of any order in both the forward and the reverse modes. Other AD tools considered were PADRE II, a Fortran precompiler developed by Koichi Kubota at Chuo University in Japan, and the Odyssee tool developed by Nicole Rostaing and Andre Galligo at INRIA Sophia-Antipolis. As part of an effort to aid in the development of robust AD tools, Christian Bischof proposed to create a public-domain database of Fortran programs, input, and derivatives, for use in comparing different AD tools.

The general conclusion from the implementation-focused discussions was that AD software developers are committed to the development of AD software that is ever easier to use and ever more robust. The potential impact of this new technology was apparent at the meeting as the various tool developers talked about the successful application of their tools to promising pilot projects. It is hoped that the expansion of application areas will lead to widening awareness in the scientific and technology circles that AD is the approach of choice for computing derivatives.

A consequence of this synergetic relation between tools, applications, and the mathematics of AD will be to bring research issues into sharper focus. Questions being considered include: Is AD something that should eventually be native to compilers as an option, bringing to bear the full weight of dataflow and dependence analysis for better AD implementations? What happens to a fluid flow or ODE solver that is differentiated by means of AD, and how do the automatically derived derivatives correlate with the solution of the adjoint equation, for example? What is the meaning of the code generated by differentiating through iterative methods?

While previous research concentrated mainly on the "automatic" aspects of computational differentiation, these questions provide an indication of the emerging issues in this field, namely, the exploitation of user insight into the structure or mathematical properties of a particular application so that AD tools can be used more efficiently, and the mathematical interpretation of the results of AD for computational paradigms approximating limit processes. These as yet mostly unresolved issues and the apparent potential for improvements in tools supporting computational differentiation promise to make this field an exciting one in the years to come.

Acknowledgments

The editors express their deep appreciation and thanks to Judy Beumer for her tireless work both in the organization of the Institute and in the publication of these proceedings. We also thank Gail Pieper for numerous suggestions which greatly improved the organization and readability of this report.

List of Attendees

Martin Berz
Christian Bischof
Alan Carle
Bruce Char
Bruce Christianson
Mene Doffou
Harley Flanders
Kyoko Fuchi
David Gay
Ralf Giering
John Gilbert
Michael Gleicher
Victor Goldman
Andreas Griewank
Robert Grossman
James Horwedel
Paul Hovland
David Juedes
Erich Kaltofen
R. Baker Kearfott
Peyvand Khademi
Koichi Kubota
Kieran Herley
Jacques Morgenstern
Louis Rall
Joel Saltz
Dimitri Shiriaev
Aaron Ross
William Thacker
Jean Utke
Wolfgang Walter
Weishi Wan
Stephen Watt
Duane Yoder
Toshinobu Yoshida

Infinitely Small Numbers and Almost Infinitely Large Accelerators, or: Automatic Differentiation as Non-Archimedean Analysis

Martin Berz
(berz@vixen.nscl.msu.edu)

Physics Department
Michigan State University
East Lansing, MI 48842

The method of the forward mode of automatic differentiation is viewed in a more general way as analysis on non-Archimedean structures. If one views the tuples of derivative vectors encountered in automatic differentiation as a ring, a total ordering can be introduced that is compatible with the algebraic structure. In this view, the components of the higher derivatives appear as the coefficients of a representation in infinitely small basis vectors or infinitesimals.

Starting from this simple observation, one can generalize the structures to form a real-closed field. In this case, one also obtains infinitely large numbers, and all customary algebra can be performed in this extension of the real numbers. Based on the ordering, it is possible to study questions of convergence; it turns out that the structures are Cauchy-complete. There is another natural type of convergence that lends itself to the introduction of power series, which can be shown to converge within their conventional radius of convergence.

In a similar way, questions of continuity and differentiability can be studied, and in most cases, there are analogs to the standard theorems of analysis, including intermediate and mean value theorems. As a bonus, one obtains the theorem that "differential quotients are derivatives" or more specifically, the value of a derivative can be obtained up to an infinitely small error by "numerically" evaluating the difference quotient using an infinitely small Δx. While providing a nice justification to the concept of differential quotients in a similar way as nonstandard analysis [1, 2], it puts the methods of forward automatic differentiation in a more philosophical and less technical light and even allows the computation of derivatives in cases where conventional automatic differentiation fails [3].

On the practical side, the methods can be applied for the study of weakly nonlinear systems in which the equations of motions as well as their flows can be represented by very quickly converging Taylor series. In the field of optics, one of the subfields of this area, these coefficients have been traditionally known as aberrations and, until the advent of automatic differentiation methods, in particular in the connection with differential algebraic methods, have been next to impossible to compute for higher orders [4].

In the case of repetitive structures studied in accelerator physics, the coefficients manifest themselves in the form of tune shifts and resonances and potentially limit the stability of the motion. Using methods of normal form theory that allow a rigorous analysis of the repetitive properties of weakly nonlinear systems [5], many important questions regarding the long-term behavior can be analyzed in a very clean way. The methods are applied for the actual design for new accelerators, including the Superconducting SuperCollider in Texas.

References

[1] A. Robinson. Non-standard analysis. In *Proceedings Royal Academy Amsterdam, Series A*, volume 64, page 432, 1961.

[2] C. Schmieden and D. Laugwitz. Eine Erweiterung der Infinitesimalrechnung. *Mathematische Zeitschrift*, 69:1–39, 1958.

[3] M. Berz. Automatic differentiation as nonarchimedean analysis. In *Computer Arithmetic and Enclosure Methods*, Amsterdam, 1992. Elsevier Science Publishers.

[4] M. Berz. Arbitrary order description of arbitrary particle optical systems. *Nuclear Instruments and Methods*, A298:426, 1990.

[5] M. Berz. Differential algebraic formulation of normal form theory. In M. Berz, S. Martin and K. Ziegler (Eds.), *Proc. Nonlinear Effects in Accelerators.* IOP Publishing, 1993.

Going Forward

Christian H. Bischof
(bischof@mcs.anl.gov)

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

For a given function $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}^m$ the forward mode propagates $\frac{dh}{dx} \in \mathbf{R}^n$ and the reverse (or adjoint) mode propagates $\frac{dy}{dh} \in \mathbf{R}^m$, where h denotes an intermediary value in the program. Hence, an often-used complexity figure (see, e.g., [8]) for the flop and storage complexity of an automatic differentiation program is $O(r)$ times the flop and storage complexity of evaluating f, where r is the maximal number of nonzeros in any row of the Jacobian $\frac{dy}{dx}$.

We show that this measure is too simplistic and that the forward mode, if supported by dynamic data structures and applied in a hierarchical fashion, may be able to deliver derivatives, and in particular gradients, much less expensively. We also illustrate the need for maintaining derivatives with respect to x, that is, forward-mode quantities, in the computation of derivatives of iterative processes.

Dynamic Data Structures: Coloring techniques can be applied advantageously in the computation of "compressed" sparse Jacobians by using automatic differentiation [1, 5] much as with divided-difference approximations. However, the rows of the Jacobian are, in general, the densest derivative objects ever propagated in the forward mode. Hence, by employing sparse dynamic data structures for the vector operations executed in the forward mode, we can take advantage of hidden structure in the program. An example is given in [3], where on a 190×190 Jacobian the dynamic sparse version required only 4% of the additions and 17% of the multiplications compared with the computation of the 190×28 "compressed" Jacobian. Another class of functions for which this approach can be very favorably applied is so-called "partially separable functions" [11, 12], which, in particular, include functions that have sparse Hessians.

Hierarchical Derivative Schemes: The ADIFOR tool [1], for example, employs a hybrid mode for propagating derivatives. For an assignment statement $w = <$some expression involving $a, b, c >$, say, we employ the reverse mode to compute $\frac{\partial w}{\partial a}, \frac{\partial w}{\partial b}, \frac{\partial w}{\partial c}$, and then employ the forward mode to form

$$\frac{dw}{dx} = \frac{\partial w}{\partial a}\frac{da}{dx} + \frac{\partial w}{\partial b}\frac{db}{dx} + \frac{\partial w}{\partial c}\frac{dc}{dx}K.$$

Note, however, that code of similar complexity would have been created if one had used the forward mode to compute $\frac{\partial w}{\partial a}, \frac{\partial w}{\partial b}, \frac{\partial w}{\partial c}$, since the decrease in complexity does not hinge on the choice of mode applied to the right-hand side of the assignment statement, but rather on the fact that x contains in general more than three entries. Studies exploiting this "contraction" in a larger context are described in [6].

Derivatives of Iteratively Defined Functions: Most CFD codes in aeronautical engineering, for example, compute flow and displacements fields by iterative procedures, which may converge very slowly and often involve discontinuous adjustments of grids or free boundaries. That is, for given x_* we are solving a nonlinear system

$$F(z, x_*) = 0 \tag{1}$$

to find the value $z_* = z(x_*)$ of the function implicitly defined by F. What we wish to compute are the derivatives $z'_* = \frac{dz}{dx}|_{x=x_*}$. Recently, we have been able to show that applying automatic differentiation to the iteration for z will generate a convergent iteration for z' for a wide class of iterative schemes [9]. We have also shown that the convergence of z' may lag behind the convergence of z and that one needs to monitor the convergence of the derivative iteration by computing $|| \frac{\partial F}{\partial x} |_{(z_k, x_*)} ||$. If x is k-vector, this quantity can be computed with an effort that is no more than k evaluations of F. Experiments with the differentiation of iterative procedures are reported in [3, 2, 7]. Note, however, that $\frac{\partial F}{\partial x} |_{(z_k, x_*)}$ is a forward-mode quantity and that, as a result, application of the reverse mode in a fashion crossing the iteration boundaries seems problematic from a numerical point of view.

In summary, we believe that the forward mode is more powerful than ordinarily assumed and that forward-mode-based tools will be competitive with reverse-mode-based tools for a wider class of problems than traditionally assumed. However, we also admit that there are problems for which there is no alternative to the classical reverse mode and that, in general, hybrid forward/reverse mode schemes will be the most efficient (see, for example, [10, 4]).

References

[1] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):1–29, 1992.

[2] Christian Bischof, George Corliss, Larry Green, Andreas Griewank, Kara Haigler, and Perry Newman. Automatic differentiation of advanced cfd codes for multidisciplinary design. ADIFOR Working Note #12, MCS–P339–1192, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[3] Christian Bischof and Andreas Griewank. ADIFOR: A Fortran system for portable automatic differentiation. In *Proceedings of the 4th Symposium on Multidisciplinary Analysis and Optimization, AIAA Paper 92-4744*, pages 433–441. American Institute of Aeronautics and Astronautics, 1992.

[4] Christian Bischof and Andreas Griewank. On the development of pseudo-adjoint codes. ADIFOR Working Note MCS-P375-0793, Mathematics and Computer Science Division, Argonne National Laboratory, 1993. To appear in the Proceedings of the Workshop on Computing in the Geosciences.

[5] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. ADIFOR Working Note #2, ANL/MCS–TM–158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[6] Christian H. Bischof and Moe El-Khadiri. Extending compile-time reverse mode and exploiting partial separability in ADIFOR. ADIFOR Working Note #7, ANL/MCS–TM-163, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[7] Lawrence Green, Perry Newman, and Kara Haigler. Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via automatic differentiation. In *Proceedings of the 11th AIAA Computational Fluid Dynamics Conference, AIAA Paper 93-3321.* American Institute of Aeronautics and Astronautics, 1993.

[8] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers, Dordrecht.

[9] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence of iterative equation solvers. ADIFOR Working Note MCS-P333-1192, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[10] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, Penn., 1991.

[11] Andreas Griewank and Philippe L. Toint. On the unconstrained optimization of partially separable objective functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312. Academic Press, London, 1981.

[12] Andreas Griewank and Philippe L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39:119–137, 1982.

Compile-Time Activity Analysis in ADIFOR

Alan Carle
(carle@cs.rice.edu)

Center for Research on Parallel Computation
Rice University
Houston, TX 77251

Efficient computation of derivatives by automatic differentiation requires good compile-time analysis of the programs to be differentiated, good derivative code generation based on the results of compile-time analysis, and good run-time support. This talk will examine the current use of compile-time analysis in ADIFOR (Automatic DIfferentiation of FORtran).

The cost of computing derivatives by the forward or reverse modes of automatic differentiation can be reduced by computing derivatives or adjoints only of "active" variables, those variables that are on a dependence path from the independent to the dependent variables. Compile-time analysis to solve this problem should result in large computational savings when applied to "multipurpose" codes, codes that have been designed to compute a set of functions simultaneously, but that are often used to compute only a single function.

Activity analysis is an interprocedural problem; its solution depends on the actions performed by each of the routines in a program. Complex issues of scoping, actual-formal parameter binding, and memory layout must all be addressed. A series of examples will be presented demonstrating the effect of each of these issues on precise activity analysis. The algorithm currently being used by ADIFOR will then be outlined and critiqued.

Reverse Accumulation of Functions Containing Gradients

Bruce Christianson
(comqbc@herts.ac.uk)

School of Information Sciences, Hatfield Campus
University of Hertfordshire
AL10 9AB England

We extend the technique of reverse accumulation so as to allow efficient extraction of gradients of scalar-valued functions, which are themselves constructed by composing operations that include taking derivatives of subfunctions. The first technique described here relies upon augmenting the computational graph and performs well when the highest order of derivative information required is fourth or fifth order. When higher order is required, an approach based upon interpolation of Taylor series is likely to give better performance, and as a first step in this direction we introduce a transformation mapping reverse passes through an augmented graph onto Taylor-valued accumulations through a forward pass.

It is well known (see, for example, [2]) that reverse accumulation can be used to extract all components of the gradient vector ∇f of any scalar-valued function f for about 3 times the floating-point computational cost of a single evaluation of f, where the constant 3 is independent both of the form of f and of the number of parameters (independent variables). We develop these techniques to derive a simple and elegant way of extracting gradients (and higher derivatives) of functions such as $y = \phi(\mathbf{u}, \nabla f(\mathbf{u}))$ which are constructed by composing operations that include taking gradients of subfunctions. We show how to obtain such information to the same level of accuracy as the function value for f, and at a small constant multiple of the computational cost.

We assume for ease of exposition that reverse accumulation is implemented in the style of [2], by operator overloading but without overloading assignment. Floating-point program variables are redeclared as of type *vary*, where

```
type vary = record (ref_node : pointer to node)
type node = record (opcode : integer; arg1 : vary; arg2 : vary;
    forward_value : real; adjoint_value : real)
```

Evaluation of the function f produces as a side effect a computational graph for f. The reverse accumulation sweep to evaluate ∇f begins by placing the value 1.0 in the adjoint value field of the end node. The reverse sweep then moves backwards through the graph incrementing the adjoint values by appropriate multiples of the operation derivatives, as required by the chain rule. For example, the adjoint accumulation step corresponding to the forward step $v = \sin u$ is $\bar{u} = \bar{u} + \bar{v} \cos u$, and the adjoint accumulation steps corresponding to the forward step $w = u * v$ are $\bar{u} = \bar{u} + \bar{w} * v$; $\bar{v} = \bar{v} + \bar{w} * u$. At the end of the reverse sweep, the adjoint value in each node is the partial derivative of the function value in the end node with respect to the forward value in the given node. In particular, adjoint values in the nodes pointed at by the independent variables correspond to the components of ∇f.

Suppose now that we have some function $\phi(\nabla f(u))$ and we wish to evaluate $\nabla\phi$. This can be done by the very same code that we have just described, by making one crucial change. We redeclare the adjoint value field as

adjoint_value : vary

The reverse accumulation step $a = a + b * c$, where a and b are now *vary*s, is implemented by overloading in such a way that $null + a$ returns a and $null * b$ returns a null pointer.

The effect of this redeclaration is that the reverse sweep now creates an additional segment of the computational graph, recording the calculation of the various operation derivatives and adjoint values. At the end of the reverse sweep,
$x.ref\_node.adjoint\_value.ref\_node.forward\_value$ contains the floating-point adjoint value (derivative component) corresponding to the independent variable x.

Part or all of the computational graph can be swept in this way, and similarly adjoint values, once calculated, can be used in subsequent constructions which can then themselves be reverse-swept. In this case, it is important to reset (to null pointers) the adjoint fields in the part of the graph to be reswept before resweeping. This can be done as a side effect in the course of the previous reverse sweep. Note that this reset operation does not affect the node previously pointed at by the reinitialized field. This approach has been implemented by Kubota [4].

The repeated use of reverse accumulation on a problem of the form

$$f(\mathbf{z}, \nabla g(\mathbf{y}, \nabla h(\mathbf{x})))$$

where $\mathbf{y}, \mathbf{z}$ also depend partially on $\mathbf{x}$, will produce duplicate structures with the same form form as $G(h)$, the graph of h. The number of copies of $G(h)$ is exponential in the depth of gradient nesting. The question therefore arises whether it might be more efficient to store the various coefficients in a single (enlarged) copy of the graph for h. We have shown in [2, §5],[3, §6] that reversal through the reversed graph is equivalent to developing a first-order Taylor series in a single variable forward through the original graph.

It turns out (using similar arguments) that nested reverse traversals amount to maintaining precisely the completely heterogeneous terms of a multivariate Taylor series (i.e., no variable appearing in power two or higher).

For example, if $\mathbf{p} = \nabla_x\, h(\mathbf{x}), \mathbf{q} = \nabla_x\, g(\mathbf{y}, \mathbf{p})$, then we can evaluate $\mathbf{q}$ as follows: build the graph $G(h)$, reverse through $G(h)$ to obtain the values $\bar{\mathbf{x}} = \mathbf{p}$, copy these into the base of the graph for g, build $G(g)$, reverse through $G(g)$ to obtain $\bar{\mathbf{p}} = \nabla_p\, g$, set $\mathbf{x}_1 = \mathbf{x} + \bar{\mathbf{p}}.t$ where t is the (first) Taylor variable, then make a second pass forward and backward through $G(h)$ computing the linear Taylor terms in t. The first-order terms in t for $\bar{\mathbf{x}}_1$ give the value for $\mathbf{q}$. These in turn are built into the base of the graph for f, and the reverse pass through $G(f)$ requires a second pass forward and back through $G(g)$ in a direction $\bar{\mathbf{q}}$ corresponding to the second Taylor variable s. This in turn requires a further pass forward and back through $G(h)$ evaluating the coefficients of the terms of order s and st. The next level of nesting would require passes for terms r, rt, rs, rst, and so on (hence the exponential growth with nesting level).

We have already considered representing a reversal through a previously built graph segment as an explicit computational step (corresponding to a graph node). This could be extended so as to define operations representing the addition of another Taylor variable to the (forward or reverse portion of the) graph. Combining this with the interpolated Taylor series approach [1] holds out the prospect of some time and space savings if the total order of differentiation is high, and this is identified as a promising avenue for future research.

References

[1] Christian Bischof et al., 1992, Structured Second- and Higher-Order Derivatives through Univariate Taylor Series, *Optimization Methods and Software*, to appear

[2] Bruce Christianson, 1992, Automatic Hessians by Reverse Accumulation, *IMA Journal of Numerical Analysis* **12**, 135–150

[3] Bruce Christianson, 1992, Reverse Accumulation and Accurate Rounding Error Estimates for Taylor Series Coefficients, *Optimization Methods and Software* **1**, 81–94

[4] Koichi Kubota, 1989, An Implementation of Fast Automatic Differentiation with C++, Abstracts of the 1989 Spring Meeting of the Operations Research Society of Japan, 175–176 (in Japanese)

Program ODE

Harley Flanders
(usergee3@umichum.bitnet)

Department of Mathematics
University of Michigan
Ann Arbor, MI 48109

ODE is a program under development to show the power of automatic computation of Taylor polynomials (AD) in various applications. It uses the MS-DOS platform, uses protected mode, and requires VGA graphics. The program is menu-driven and quite user friendly. It will be distributed commercially in time.

ODE accepts user input of any function expressible in terms of the usual algebraic operations and the usual transcendental functions used in computer languages.

Its current modules:

1. Computation of a Taylor approximation to any function up to degree 100. Plot of the function and the Taylor polynomial. Comparison of computed values.
2. Ditto for functions implicitly defined by F(x, y) = c.
3. Ditto for inverse functions.
4. Solution of systems of ODE, numerical and graphical, by Taylor polynomials and analytic continuation. Up to 3 dependent variables. Various plane and space plots.
5. Ditto for ODE systems defined implicitly, up to 20 dependent variables.
6. Solution of F(x) = 0 by Newton, Halley, another third-order method, three fourth-order methods, and one fifth-order method, with comparison of convergence.
7. Integration of F(x) over [a, b] by Taylor expansion with adaptive step size.
8. Graph of F(x, y, z) = c. This uses gradient computations.

Projected modules:

9. Two-body problem
10. Three-body problem
11. Singular systems of ODE
12. System of DAEs (differential algebraic equations)
13. Initial value Cauchy problem

Adjoint Code Generation

Ralf Giering
(giering@dkrz-hamburg.dbp.de)

Max-Planck-Institut für Meteorologie
Bundesstrasse 55, D-20146 Hamburg, Germany

Adjoint models are increasingly in development for use in meteorology and oceanography for data assimilation, model tuning, sensitivity analysis, and determination of singular vectors. The adjoint model computes the gradient of a cost function with respect to control variables. Direct coding of the adjoint of a complex CFD model is extremely time consuming and subject to errors. Automatic generation of adjoint models would greatly help. For this purpose a tool has been developed.

The automatic generation of adjoint code is a special case of automatic differentiation of algorithms in backward mode, where the dependent function is a scalar. Out of a Fortran subroutine calculating a cost function f

$$\begin{array}{rcl} f \,:\, R^N & \Longrightarrow & R \\ x & \Longrightarrow & y \end{array}$$

a Fortran subroutine is generated computing

$$f' \,:\, R \Longrightarrow R^N \,,\; f'_i := \frac{\partial y}{\partial x_i}.$$

The method used is based on a few principles:

- Every active variable (inside dependency tree) has a corresponding adjoint variable.
- For every common block containing active variables, an adjoint common block is created.
- For every subroutine (function) that calculates active variables, an adjoint structure is generated.
- The active output variables of a structure are the adjoint input variables of the corresponding adjoint structure and vice versa. In addition to this, input variables of the structure needed for the adjoint calculations are also input variables of the adjoint structure. This checkpointing ensures that each adjoint structure could be generated separately, by knowing only the active variables.

The body of each structure is analyzed, and for every statement the input and output variables are determined. After constructing the corresponding adjoint statements according to specified rules, the variables of the original code needed for the adjoint calculations are determined. These may be indices, passive variables, or active variables. Code is included, where necessary, to calculate these values as they are calculated in the original code. The user may use directives to create code to store the values during the calculation of the cost function and to restore them during the adjoint calculations.

A Fortran program consists mainly of four kinds of statements. The generation of the adjoint statements is shortly described:

assignment

$$X \longleftarrow g(X, A, B, ...)$$

The expression on the right hand-side of an assignment is symbolically differentiated with respect to every active variable that occurs in the assignment. For each of them an adjoint assignment is generated (adA = adjoint of A)

$$adA \longleftarrow adA + adX * \frac{\partial g}{\partial A}$$

The last assignment contains the assignment to the adjoint variable of the left-hand side variable of the original code.

$$adX \longleftarrow adX * \frac{\partial g}{\partial X}$$

conditional statement For a conditional statement

IF *condition* **THEN** *statement A* **ELSE** *statement B*

the adjoint statement is

IF *condition* **THEN** *adjoint statement A* **ELSE** *adjoint statement B*

loop In case there is any recursive assignment inside a loop, the adjoint loop has to take the reverse order. The number of passes has to be provided.

sequence of statements The adjoint of the statements are arranged in reverse order. Code is included in front to provide variables, which are needed.

subroutine call The corresponding adjoint subroutine is called.

A peculiar problem may arise from multiple assignment to a variable within a structure if the same variable is needed for the adjoint calculations. In this case a warning is given to the user.

For the iterative calculation of a nonlinear implicit function, a special adjoint code can be generated assuming that the iteration converged. This code avoids the storing of variables at each iteration.

In general the user has to choose between recalculating or storing of variables. The former takes additional computer time; the later requires memory or disk space. This decision cannot be made automatically.

A Tour of Combinatorial Sparse Matrix Technology

John R. Gilbert
(gilbert@parc.xerox.com)

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Graph theory has been ubiquitous as a tool for sparse matrix computation since Seymour Parter described fill during Cholesky factorization as a transformation on graphs in 1961 [26]. Graphs conveniently capture the path structure that is important in Gaussian elimination; they expose the issues of locality that affect efficiency on machines with hierarchical or distributed memory; and they bring to bear a well-developed set of efficient algorithms and data structures.

This talk surveys some combinatorial sparse matrix theory that relates (sometimes tangentially!) to automatic differentiation. We consider four topics: directed graphs and sparse matrix algorithms; the minimum fill problem; graph partitioning; and a game that models the space/time tradeoff in reverse mode.

Directed Graphs and Sparse Matrix Algorithms

If A is an n by n nonsingular matrix with nonzero diagonal elements, its directed graph has n vertices and an edge (i, j) for each off-diagonal nonzero a_{ij}. Permuting the rows and columns of A to block triangular form (so that a system of linear equations can be solved by block back-substitution, factoring only the irreducible diagonal blocks) is the same as finding the strongly connected components of the graph. Tarjan's linear-time strong components algorithm [30] was one of the first applications of depth-first search to numerical computation. The irreducible diagonal blocks of an arbitrary matrix are independent of the choice of nonzero diagonal in the matrix [6].

If in addition A has an LU factorization without pivoting, Rose and Tarjan [28] give a simple characterization of the nonzero structure of the factors in terms of paths in the graph of A. Several people have studied this so-called "directed filled graph" [7, 14, 21].

Consider a linear system $Ax = b$ in which both the matrix A and the vector b are sparse. The nonzero positions of b can be thought of as a set of vertices of the graph of A. The nonzero positions of x correspond to exactly those vertices from which there exist paths in the graph to vertices of b [11]. This makes possible an efficient algorithm to solve triangular systems with sparse right-hand sides, which in turn permits LU factorization with partial pivoting of an arbitrary nonsingular matrix in time proportional to the number of nonzero arithmetic operations [15].

Jacobian accumulation by automatic differentiation is related to computing a Schur complement in a sparse triangular matrix by Gaussian elimination [18]. Consider the graph of the computation of outputs $y_1, \ldots, y_m$ from inputs $x_1, \ldots, x_n$ by way of intermediate values $z_1, \ldots, z_p$. Let C be the matrix of that graph, with diagonal elements equal to -1

and off-diagonal elements equal to the local partial derivatives. Then C is triangular, and the Jacobian is the m by n Schur complement $J = C_{xy} - C_{zy}C_{zz}^{-1}C_{xz}$ obtained by eliminating all the intermediate vertices z_i. In this setting, the forward, reverse, and Markowitz modes of automatic differentiation correspond to different elimination orders. Schur complements in sparse matrices have been studied in a graph-theoretic setting [8].

Approximately Minimum Fill

Kieran Herley showed at this workshop that minimizing the fill in the computational graph during Jacobian accumulation is NP-complete. Fill and other cost measures for sparse Gaussian elimination have been studied extensively, especially in the setting of undirected graphs and Cholesky factorization of symmetric, positive definite matrices [10]. Minimum degree (the symmetric version of the Markowitz heuristic) is very effective in practice in reducing fill and operation count, though there are simple examples where it performs arbitrarily badly [3].

Nested dissection, a divide-and-conquer heuristic based on separators in the graph, usually does not perform quite as well as minimum degree in practice. However, nested dissection is the only method known that can produce provable guarantees on the various cost measures. For example, the Leighton-Rao algorithm [22] produces separators within $O(\log n)$ of optimal for any bounded-degree graph in polynomial time; using these separators in nested dissection guarantees that fill, operation count, frontsize, treewidth, pathwidth, and elimination tree height are all within a polylogarithmic factor of minimum [1, 4]. Thus, at least in the undirected case, the quality of a graph's elimination orders is intimately connected to the quality of its separators.

Graph Partitioning

Recent interest in separators has been sparked by the desire to partition computational graphs for distributed-memory parallel machines. Random graphs (in a suitable sense) do not have good separators [23], but several useful classes do, including trees [19], planar graphs [23], graphs of bounded genus [12], graphs that forbid a fixed set of minors [2], and chordal graphs [16]. In practice, many heuristic methods have been used to find separators [9, 20, 24]; recently suggested methods use ingredients as diverse as the spectrum of the Laplacian matrix of the graph [27] and the geometry of an underlying finite element mesh [25].

Modular automatic differentiation and Griewank's checkpointing algorithm can be viewed as partitioning the computational graph. However, it is probably not useful in this context to use a general-purpose partitioner on the computational graph. Rather, the key research question is likely to be how to use the call graph and/or some version of the dataflow graph (such as static single-assignment form [5]) to partition the computational graph suitably (at appropriate subroutine or basic block boundaries, for example) without actually forming it.

Reverse Mode as a Pebble Game

Griewank's checkpointing algorithm [17] can be described in terms of a one-person game on a directed acyclic graph. The player has a number of "pebbles" in two colors, black and white, which can be placed on the vertices of the graph according to the following rules:

- Black rule: A black pebble may be placed on a vertex if all its immediate predecessor vertices hold black pebbles. (Thus a black pebble may be placed on a source, or input variable, at any time).
- White rule: A black pebble on a vertex may be replaced with a white pebble if all its immediate successor vertices hold white pebbles. (Thus a black pebble on a sink, or output variable, may be changed to a white pebble at any time.)

Placing a black pebble on a vertex corresponds to computing a value; placing a white pebble corresponds to computing a derivative. The goal of the game is to place white pebbles on all the input vertices, while using as few pebbles (that is, as little space) and as few placements (that is, as little time) as possible.

The conventional noncheckpointing reverse mode corresponds to placing a black pebble on every node in forward topological order, and then replacing each black pebble with a white one in reverse topological order. Griewank's checkpointing algorithm corresponds to a pebbling strategy that leaves black pebbles on a logarithmic number of cuts in the graph, repebbling as necessary to move a cut of white pebbles up from the outputs to the inputs; the time/space analysis says that the total number of pebbles is only $O(\log n)$ times the largest cut, and the total number of placements is only $O(\log n)$ times the number of vertices. It might be possible to design more efficient reverse-mode algorithms for particular computations by considering pebbling strategies for their graphs.

The black-only pebble game has been studied as a model of register allocation in compilers [29]; a two-color pebble game with a rather different white rule has been studied as a model of space-time tradeoffs in nondeterministic computation [13].

References

[1] Ajit Agrawal and Philip Klein. Cutting down on fill using nested dissection: Provably good elimination orderings. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.

[2] Noga Alon, Paul Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and applications. In *Proceedings of the 22th Annual Symposium on Theory of Computing*. ACM, 1990.

[3] Piotr Berman and Georg Schnitger. On the performance of the minimum degree ordering for Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 11:83–88, 1990.

[4] Hans Bodlaender, Hjálmtýr Hafsteinsson, John R. Gilbert, and Ton Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. In *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 570, pages 1–12. Springer-Verlag, 1992.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[6] A. L. Dulmage and N. S. Mendelsohn. Graphs and matrices. In Frank Harary, editor, *Graph Theory and Theoretical Physics*, pages 167–227. Academic Press, 1967.

[7] Stanley C. Eisenstat and Joseph W. H. Liu. Exploiting structural symmetry in unsymmetric sparse symbolic factorization. Technical Report CS–90–12, York University, 1990. To appear in *SIAM Journal on Matrix Analysis and Applications.*

[8] Stanley C. Eisenstat and Joseph W. H. Liu. Structural representations of Schur complements in sparse matrices. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation.* Springer-Verlag, 1993.

[9] Alan George and Joseph W. H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis*, 15:1053–1069, 1978.

[10] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[11] John R. Gilbert. Predicting structure in sparse matrix computations. Technical Report 86–750, Cornell University, 1986. To appear in *SIAM Journal on Matrix Analysis and Applications.*

[12] John R. Gilbert, Joan P. Hutchinson, and Robert Endre Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5:391–407, 1984.

[13] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM Journal on Computing*, 9:513–524, 1980.

[14] John R. Gilbert and Joseph W. H. Liu. Elimination structures for unsymmetric sparse *LU* factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993. Also available as Xerox PARC report CSL 90–11.

[15] John R. Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9:862–874, 1988.

[16] John R. Gilbert, Donald J. Rose, and Anders Edenbrandt. A separator theorem for chordal graphs. *SIAM Journal on Algebraic and Discrete Methods*, 5:306–313, 1984.

[17] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[18] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, Penn., 1991.

[19] Camille Jordan. Sur les assemblages de lignes. *Journal Reine Angew. Math.*, 70:185–190, 1869.

[20] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[21] D. J. Kleitman. A note on perfect elimination digraphs. *SIAM Journal on Computing*, 3:280–282, 1974.

[22] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431. IEEE, 1988.

[23] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[24] Joseph W. H. Liu. A graph partitioning algorithm by node separators. *ACM Transactions on Mathematical Software*, 15:198–219, 1989.

[25] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation.* Springer-Verlag, 1993.

[26] S. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3:119–130, 1961.

[27] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11:430–452, 1990.

[28] Donald J. Rose and Robert Endre Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34:176–197, 1978.

[29] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4:226–248, 1975.

[30] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–159, 1972.

Differential Methods in Graphical Interaction

Michael Gleicher
(gleicher@cs.cmu.edu)

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Traditional methods for manipulating graphical objects have been limited by simple controls that can be employed only one at a time. By employing constrained optimization to couple interactive controls to the parameters of graphical objects, these restrictions can be removed. Such techniques can allow users to control objects by specifying attributes that are computed as functions of the parameters, and also allow multiple attributes to be specified simultaneously, as constraints.

Although the graphics applications require solving very standard numerical problems, challenges are introduced by the interactive nature of the applications. First, the numerics must be transparent, since the user is most likely interested in solving a graphical problem, not in numerical optimization. Second, the system must be dynamic, as the user will continually be changing the set of constraints. Finally, it is important that the numerics be fast, not only to achieve a dynamic state, but also to achieve frame rates fast enough to provide the illusion of continuous motion, which is critical to usability.

Because the applications are dynamic, the functions that represent the constraints and objective functions are not known at compile time. I have created a C++ toolkit called *Snap-Together Math*, which allows the functions to be dynamically defined from smaller pieces. Classes of objects representing primitive function types are defined at compile time by using automatic code-generation tools. At run time, these objects are hooked together to form expression graphs. These graphs are traversed to evaluate the functions and their derivatives. The derivative evaluations are a form of forward-mode automatic differentiation.

For the needed performance, four techniques are used. First, caching is used extensively to avoid recomputing. Second, the sparse nature of the derivatives and matrix calculations is exploited, yielding not only faster performance, but lower computational complexity. Third, the system attempts to solve smaller problems by partitioning and freezing subsets of the constraints. Finally, in an interactive setting we are able to trade accuracy for performance, for example, using larger tolerances in our iterative solving algorithms.

Checking the Memory

Andreas Griewank
(griewank@math.tu-dresden.de)

Argonne National Laboratory
and Technical University Dresden

Being a discrete analog of the adjoint or costate equations used in optimal control, the basic reverse, or top-down, mode of computational differentiation [2] yields gradients at very low operations count. However, it has a potentially extremely large memory requirement. This effect is particularly noticeable on time-dependent problems, where the memory requirement of the basic reverse mode grows in proportion to the number of time steps. The proportionality factor achieved by automatic differentiation tools tends to be much larger than that achieved by the careful hand-coding of adjoints through the judicious selection of the quantities that need to be saved. Fortunately, by the related techniques of multilevel differentiation and recursive checkpointing, the memory requirement can be dramatically reduced at the expense of a moderate increase in the operations count that results from the repeated recalculation of intermediates.

Multilevel differentiation, originally proposed by Volin et al. [3], is already employed in some hand-coded adjoints and some automated adjoint code generators like the ACM system presented at this institute by Ralph Giering. Each subroutine of the original code is interpreted as a superelementary function that can be called (repeatedly) in a direct mode and once in an (adjoint) version. For the repeated direct calls the actual parameters and certain global variables must be either restored from memory or recalculated by the calling routine.

A tight identification of these input sets is crucial for the efficacy of the multilevel differentiation approach. It can be based on user-supplied directives or interprocedural dependency analysis performed by a (pre)compiler. Consequently, the exact memory requirement is hard to predict. The same observation is true for a related scheme, where the parameters and global variables altered by the subroutine are saved and restored. In either case the number of times that the forward version of a particular subroutine is called equals exactly the number of its predecessors in the calling tree. On time-dependent problems with a calling tree of bounded depth, multilevel differentiation does not overcome the proportionality between memory requirement and the number of time steps.

Checkpointing can be applied to any sequence of elementary transformations on a given *core* memory. The basic reverse mode involves recording each individual transformation on a separate *disk* memory and then playing this *tape* backward to calculate the gradient. Rather than recording the whole calculation in one forward sweep on the tape, one can generate it piece by piece from the end by restarting the calculation repeatedly from appropriately placed checkpoints. For the *binomial* checkpointing scheme proposed in [1] one obtains the following complexity result. Given a bound t on the number of repetitions and a bound s on the number of snapshots kept on disk at any time, one can reverse a calculation, whose full tape would have taken up to

$$\binom{s+t}{s} = \frac{(s+t)!}{s!\,t!} = \binom{s+t}{t}$$

times the disk space needed for one snapshot of the core. For time-dependent problems this observation means that the spatial complexity ratio s and the temporal complexity ratio t can both be limited to a logarithm of the number of time steps. Here, each ratio compares the complexity of a gradient calculation with that of the underlying scalar function evaluation. The binomial scheme can be interpreted as multilevel differentiation with an artificially created calling tree, whose depth grows with the length of the calculation (i.e., the size of its tape). When suitable implementations become available, gradient-based optimization and parameter estimation methods will be applicable to computer models of almost arbitrary complexity.

References

[1] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, 1 (1992), pp. 35–54.

[2] Masao Iri, *History of automatic differentiation and rounding estimation*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss, eds., SIAM, Philadelphia, 1991, pp. 1–16 .

[3] Yu. M. Volin and G. M. Ostrovskii, *Automatic computation of derivatives with the use of the multilevel differentiation technique*, Computers and Mathematics with Applications, Vol. 11, no. 11 (1985), pp. 1099–1114.

Trees and Differentiation

Robert Grossman
(grossman@eecs.uic.edu)

Department of Mathematics
University of Illinois at Chicago
Chicago, IL 60680

Let R denote the polynomial algebra $k[x_1, \ldots, x_N]$, and consider the formal symbols F_j defined by

$$F_j = \sum_{\mu=1}^{N} a_j^\mu \frac{\partial}{\partial x_\mu}, \quad j = 1, \ldots, M$$

as first-order differential operators with coefficients a_j^μ in R. Elements in the free associative algebra $A = k{<}F_1, \ldots, F_M{>}$ on $F_1, \ldots, F_M$ may then be interpreted as higher-order differential operators generated by the $F_j's$.

Let B denote the vector space whose basis is the set of finite, rooted trees labeled with the symbols $\{F_1, \ldots, F_M\}$. It turns out that B is a Hopf algebra and the map

$$\phi : A \longrightarrow B$$

, which takes the generators F_j to the tree consisting of a root with a single child labeled with F_j, can be extended to a Hopf algebra homomorphism. It also turns out that B, as well as A, measures R to itself. With this structure, algorithms for manipulating differential operators symbolically are translated into assertions about labeled trees. The result is algorithms that can be exponentially faster than naive ones.

The derivation of specialized algorithms for numerically integrating the flow of the non-linear system

$$\dot{x}(t) = F(x(t)), \quad x(0) = x^0 \in \mathbf{R}^N$$

leads to computations in the algebras A and B. In particular, the element

$$\exp tF = \sum_{i=0}^{\infty} \frac{t^i}{i!} F^i$$

and its image $\phi(\exp tF)$ turn out to be grouplike elements in the appropriate power series algebras constructed from A and B. Finding efficient numerical algorithms is equivalent to computing other grouplike elements with various desirable properties in these algebras. In this talk, we survey efficient symbolic algorithms for computations of this type and indicate some connections to automatic differentiation.

Jacobian Accumulation by Vertex Elimination

Kieran T. Herley
(STCS8125@bureau.ucc.ie)

Department of Computer Science
University College
Cork, Ireland

The computation of $f(x_1, \cdots x_n) = (y_1, \cdots, y_m)$ for some computable function $f : \mathbf{R}^n \rightarrow \mathbf{R}^m$ may be modeled in terms of its constituent elementary operations (additions, subtractions *etc.*) by means of a suitably labeled directed acyclic graph G_f. In this framework, the vertices of indegree zero and vertices of outdegree zero in the graph represent the independent and dependent quantities of the computation, respectively; the other intermediate vertices (each labeled with an elementary operation) denote the individual steps and the intermediate quantities produced thereby; and the edges of the graph capture the computational dependencies between the various quantities involved in the calculation.

The Jacobian matrix $J_f = [\partial y_j / \partial x_i]$ for f may be calculated from G_f by associating a suitable weight c_{xy} with each edge (x, y) and then *eliminating* the intermediate vertices of the graph one at a time as follows. To eliminate vertex v, (i) add $c_{uv} \cdot c_{vw}$ to the existing weight of each edge (u, w) where u is an uneliminated predecessor of v and w is an uneliminated successor of v (or create a new edge, referred to as a *fill edge*, with this weight if no such edge previously existed), and (ii) mark v *eliminated.*

The order in which the vertices are eliminated does not affect the final result: the final weight of the edge (x_i, y_j) is the Jacobian entry $\partial y_j / \partial x_i$, but strongly influences the computational resources required for its calculation. Many of the techniques such as forward mode, backward mode, and Markowitz methods that have been proposed and studied in the automatic differentiation literature for the efficient calculation of Jacobians can be interpreted, for a suitable choice of vertex elimination order, as a computation of the above type. Thus we have the question of whether, for a given function f, it is possible to determine an optimum elimination order, one that allows the above vertex elimination procedure to be carried out as efficiently as possible. Adopting the number of fill edges created during vertex elimination as a cost criterion, we adapt a result due to Rose and Tarjan to establish the NP-completeness of the following problem: Given a directed acyclic graph G and a positive integer K, is there an elimination order for the intermediate vertices of G that creates at most K fill edges?

This result suggests that the problem of determining the optimum elimination order (at least with respect to the minimum fill cost criterion) is as hard as a host of other well-studied computational problems such as the Travelling Salesman problem and Boolean satisfiability, and hence is unlikely to have an efficient (polynomial time) algorithm.

Reverse Automatic Differentiation of Modular Fortran Programs

J. E. Horwedel
(jqh@ornl.gov)

Computing Applications Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831

The calculation of derivatives necessary for sensitivity analysis or for the optimal solution of systems of nonlinear equations continues to be an important research objective. Several software systems have been developed for implementing automatic differentiation of computer programs. The forward mode of automatic differentiation is efficient for calculating derivatives for a large number of dependent variables with respect to a few independent variables. As the number of independent variables increases, the computational complexity, as measured in execution time and memory requirements, renders the forward mode impractical. The reverse mode or adjoint approach is efficient for derivatives of a few dependent variables with respect to thousands of independent variables; however, available memory and disk storage generally limit the application of the reverse mode to problems with less than a few million floating-point assignments. The fundamental problem with the reverse mode of automatic differentiation is that the accumulation of derivatives is required. A code that uses 3 minutes of execution time to perform 50 million floating-point assignments could easily need more than one gigabyte to store the accumulated derivatives [1, 2, 4].

GRESS (the GRadient Enhanced Software System) was designed to apply automatic differentiation to large-scale FORTRAN programs in the nuclear industry without requiring changes to the coding [4, 5, 6]. GRESS provides two methods for calculating and reporting derivatives. The CHAIN option implements the forward mode of automatic differentiation. The ADGEN option incorporates the reverse mode or adjoint sensitivity analysis methods to calculate derivatives.

A modular differentiation technique (MDT) is discussed that uses both forward and reverse modes to restrict the growth of execution time and storage requirements, thus extending the size of problems to which automatic differentiation can be applied. MDT is implemented using GRESS and provides a compromise between the forward mode with its computational limitations and the reverse mode with its excessive memory or storage requirements. The effectiveness of the MDT in propagating derivatives through a computer program rests on the degree of modularity in the program. Most existing large-scale Fortran programs do not have the degree of modularity necessary to apply MDT in an automated fashion. The approach described is to provide the basic tools to allow one to implement MDT on a module-by-module basis in an existing code or in the development phase for a new code.

A module can be considered to be any sequence of Fortran statements. Any module can be represented by a computational graph. As an example consider the following formula for DIST. A computational graph for this formula is shown in Figure 1. The squares in the computational graph represent arithmetic assignment statements. The reverse mode of automatic differentiation requires the accumulation of derivatives for every floating point assignment that is dependent on a declared parameter. In modular form the DIST formula could be coded as a Fortran subroutine or function with the Y array as input and DIST as the calculated result.

A computer program can be represented as a sequence of modules each with its own computational graph. Each module is assumed to have input and output. Most large Fortran programs are modular in design; however, common blocks provide a mechanism by which modules can share global variables that are not provided on the link to the module. When processing a module, we have to be concerned only with global variables that are accessed or stored during the execution of a module. Though a single module may have thousands of common block variables, only a subset may actually be used as dependent or independent variables. Variables that are used can be determined during execution of the module.

Figure 2 shows the computational graph for a computer program with global variables available to modules. The digits on the links indicate the number of input and output variables for each module. Module A has four inputs and six outputs, B has seven inputs and four outputs, and C has seven inputs and three outputs.

MDT is designed to work with each module independently. Once a module is completed, then either forward or reverse mode is used to calculate the derivatives of the output from the module with respect to the input. Only the derivatives of the output with respect to the input need to be stored. The decision to use forward or reverse mode does not have to be made a priori; it can be determined when the module is finished by the GRESS program.

For MDT to be feasible, the number of variables on the links between modules must be small compared with the number of variables within the modules. The more modular a code system, the more effectively one could implement MDT. A module can be as simple as a subroutine or function; however, the composition of a module is arbitrary. For example, in a code that does hundreds of iterations, each iteration could be treated as a module. Though in the long term, completely automating MDT is recommended, the intent in this paper is to test MDT with existing technology. To demonstrate MDT, we selected a sample problem with a main program and two subroutines. Each subroutine is called per iteration in the main program. The number of iterations can be varied. Four parameters and one dependent variable are retained after each iteration. Three methods were used to process the sample problem: (1) the GRESS ADGEN option implementing reverse mode on the entire program; (2) MDT treating each iteration as a module; and (3) MDT treating each subroutine as a module. The sample problem selected is the test program provided on the GRESS distribution diskette. Of importance in this paper is that there are two subroutines and no global variables. Shown in Figure 3 is a plot of the maximum amount of memory required to store derivatives using each method as a function of the number of iterations. Method 1 is provided for comparison, since ADGEN requires the accumulation of derivatives for every arithmetic assignment statement.

The results clearly demonstrate the fundamental problem with the reverse mode of automatic differentiation; that is, the memory required to store derivatives is proportional to execution time. Interestingly, Method 3 also shows a linear growth, though not as steep as Method 1. For this application, as the number of iterations increases, Method 2 would be the most feasible in terms of memory requirements. Memory requirement using Method

2 increases by 52 bytes per iteration. With an iterative code using Method 2, the expected increase per iteration would be the size of the module frame used to link each iteration.

The results shown in Figure 3 demonstrate that MDT is both practical and feasible. Though the sample problem is very limited in that it does not include common blocks and does require hand intervention in identifying modules, I am very encouraged by the results. Automating the procedure so that common block variables and variables on argument lists can be automatically included as dependent or independent variables is conceptually straightforward. However, having the flexibility of allowing the user to identify modules is also desirable.

The conclusion that Method 2 would be best can be made only for this application. The comparison between two different implementations of MDT raises the question as to whether it would be viable to automatically process a code to determine which method would be most appropriate. Much of the information required may not be available until execution of the model. It may be more feasible to develop tools to enable the user to implement MDT in a semiautomated fashion.

References

[1] J. E. Horwedel. Matrix Reduction Algorithms for GRESS and ADGEN. *ORNL/TM-11261*, Martin Marietta Energy Systems, Inc., Oak Ridge Natl. Lab., 1989.

[2] J. E. Horwedel, R. J. Raridon, and R. Q. Wright. Automated Sensitivity Analysis of an Atmospheric Dispersion Model. *Atmospheric Environment*, 26A, no. 9, 1992.

[3] A. Griewank. Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation. *Preprint MCS-P228-0491*, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[4] J. E. Horwedel, B. A. Worley, E. M. Oblow, and F. G. Pin. GRESS Version 2.0 User's Manual. *ORNL/TM-11951*, Martin Marietta Energy Systems, Inc., Oak Ridge Natl. Lab., 1991.

[5] E. M. Oblow, F. G. Pin, and R. Q. Wright. Sensitivity Analysis Using Computer Calculus: A Nuclear Waste Application. *Nucl. Sci. Eng.*, 94, 46, 1986.

[6] B. A. Worley, R. Q. Wright, F. G. Pin, and W. V. Harper. Application of an Automated Procedure for Adding a Comprehensive Sensitivity Calculation Capability to the ORIGEN2 Point Depletion and Radioactivity Decay Code. *Nucl. Sci. Eng.*, 94, 180, 1986.

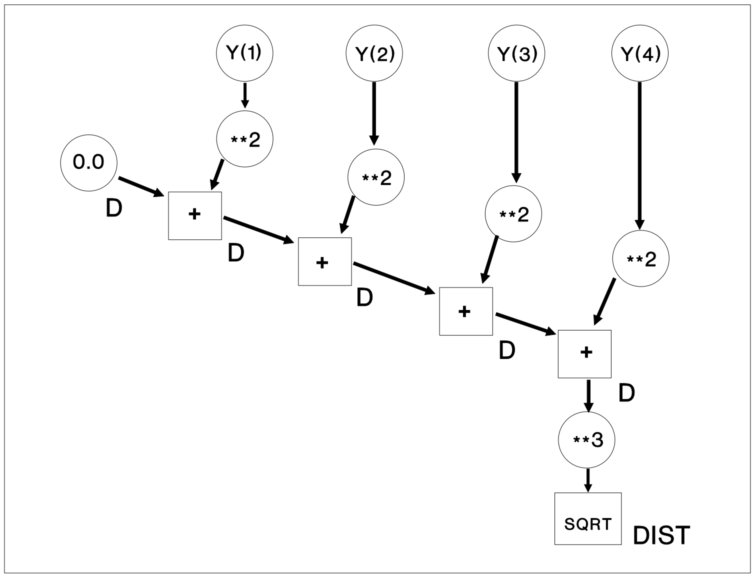

Figure 1: Computational graph for a given module

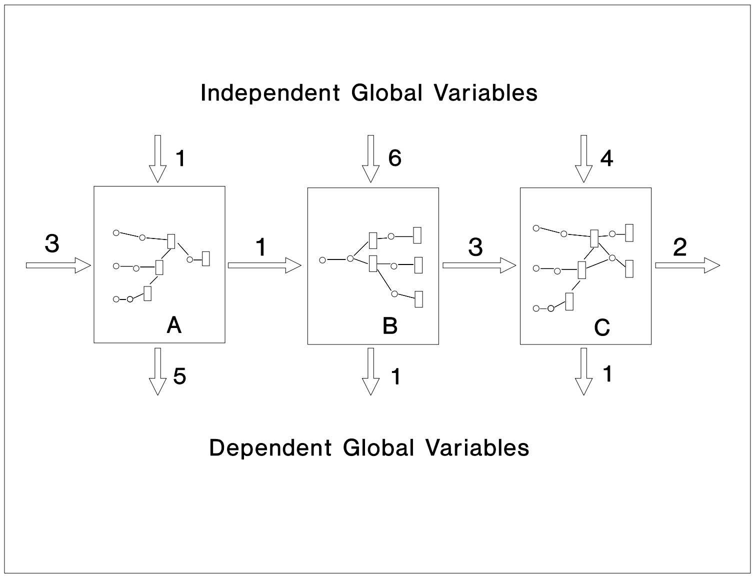

Figure 2: Computational graph for program with common blocks

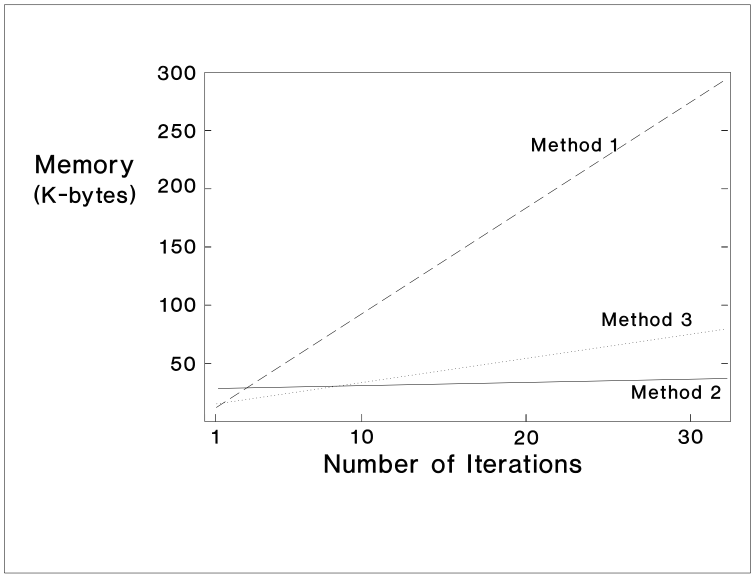

Figure 3: Memory required to store derivatives using MDT

Computational Differentiation and Algebraic Complexity Theory

Erich Kaltofen
(kaltofen@cs.rpi.edu)

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180-3590

The reverse mode of automatic differentiation allows within a constant cost factor the computation of the gradient of a multivariate, single-valued, function that is given by an arithmetic circuit. Indeed, a circuit can be constructed whose number of nodes does not exceed 4 times the number of nodes in the original circuit. Furthermore, it can be arranged that the depth of the circuit is within a constant of the original circuit as well [8], [6]. Griewank [4] has also shown that the sequential space complexity can be kept within a logarithmic factor while increasing the time complexity by only a logarithmic factor. This result has been used for several algebraic complexity estimates:

1. Baur and Strassen [2] show that the complexity of computing the determinant of an arbitrary non-singular matrix is asymptotically no less than that of the inverse, because for a square matrix A we have

$$(-1)^{i+j}\frac{\partial\,\mathrm{Det}(A)}{\partial A_{j,i}} = \mathrm{Det}(A)\,(A^{-1})_{i,j}.$$

The recent so-called processor efficient parallel algorithms of poly-logarithmic time for computing the inverse of a non-singular matrix are based on this reduction (Kaltofen and Pan 1991 and 1992). Automatic differentiation is the only way known to me to compute inverses within the given time and processor count constraints.

2. Furthermore, Baur and Strassen employ the gradient contruction to show that the complexity of computing the sum $x_1^n + \cdots + x_n^n$ is within a constant of computing the individual $(n+1)^{\mathrm{st}}$ powers $x_1^{n+1}, \ldots, x_n^{n+1}$ which by the Strassen's degree bound is $\Theta(n \log n)$[12].

3. The transposition principle asserts that for any (possibly structured) matrix A and any vector b the problems of computing $A \cdot b$ and $A^{\mathrm{tr}} \cdot b$ are of the same asymptotic complexity. Proven explicitly by Kaminski, Kirkpatrick, and Bshouty [9] by reversing the flow in the circuit for computing $A \cdot b$, the principle is also a simple consequence of reverse mode: for

$$f(x_1, \ldots, x_n) = ((\,x_1 \quad \ldots \quad x_n\,) \cdot A^{\mathrm{tr}}) \cdot b \qquad \text{we have} \qquad \begin{pmatrix} \partial_{x_1} f \\ \vdots \\ \partial_{x_n} f \end{pmatrix} = A^{\mathrm{tr}} b.$$

One application is when $A = V^{\mathrm{tr}}$ is a transposed Vandermonde matrix, a problem needed in sparse polynomial interpolation [3] and polynomial factoring [10]. Shoup's explicit algorithm, however, is of linear space complexity and needs no divisions, unlike the one obtained from the fast multipoint polynomial evaluation problem $V \cdot b$ (see [1], §6) and the transposition principle. Shoup [11] also uses this principle in the construction of a fast method for computing the minimum polynomial of an element in an algebraic number field.

4. Similarly, the problems $A^{-1} \cdot b$ and $(A^{\mathrm{tr}})^{-1} \cdot b$ have the same asymptotic complexity. But again the explicitly derived algorithm for the Vandermonde case $(V^{\mathrm{tr}})^{-1} \cdot b$ by Kaltofen and Lakshman [5] has the better linear space complexity.

As it turns out, higher derivatives are much more complex to compute. The following clever reduction of the product of two $n \times n$ matrices B and C to the trace of the Hessian has been communicated to me by T. Lickteig. Let A be a third $n \times n$ matrix. Then

$$\mathrm{Trace}(ABC) = \left(\frac{\partial^2}{\partial x_1^2} + \cdots + \frac{\partial^2}{\partial x_n^2}\right) (\mathbf{x}^{\mathrm{tr}} ABC\mathbf{x}), \quad \text{where} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix};$$

note that the argument to the trace of the Hessian, $\mathbf{x}^{\mathrm{tr}}ABC\mathbf{x}$, can be computed in $O(n^2)$ time. However, by reverse mode, we can compute

$$\frac{\partial \mathrm{Trace}(ABC)}{A_{i,j}} = (BC)_{j,i}$$

Therefore, any method for finding the trace of the Hessian within a factor $g(n)$ gives a matrix multiplication algorithm of $O(g(n)\, n^2)$ arithmetic steps.

Finally, computing multiple partial derivatives is known to be as hard as counting the number of satisfying assignments in a Boolean formula [14]: Consider

$$P(x_1, \ldots, x_n, Z_{1,1}, \ldots, Z_{n,n}) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} x_j Z_{i,j} \right)$$

Then

$$\frac{\partial^n P}{\partial x_1 \cdots \partial x_n} = \mathrm{Permanent}(Z).$$

Therefore, given a transformation that computes $\partial^n/(\partial x_1 \cdots \partial x_n)$ within a factor of $h(n)$ leads to an algorithm to compute the permanent with $O(h(n)\, n^2)$ arithmetic operation. By Valiant's proof [13] that the permanent is $\#\mathcal{P}$-complete, $h(n)$ is likely exponential in n.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Algorithms.* Addison and Wesley, Reading, Mass., 1974.

[2] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Comp. Sci.*, 22:317–330, 1983.

[3] J. Canny, E. Kaltofen, and Yagati Lakshman. Solving systems of non-linear polynomial equations faster. In *Proc. ACM-SIGSAM 1989 Internat. Symp. Symbolic Algebraic Comput.*, pages 121–128, 1989.

[4] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Preprint MCS-P228-0491, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., May 1991.

[5] E. Kaltofen and Yagati Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In *Proc. ISSAC '88*. pages 467-474, Springer Lect. Notes Comput. Sci. 358, 1988.

[6] E. Kaltofen and V. Pan. Processor efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Ann. ACM Symp. Parallel Algor. Architecture*, pages 180–191, ACM Press, 1991.

[7] E. Kaltofen and V. Pan. Processor-efficient parallel solution of linear systems II: The positive characteristic and singular cases. In *Proc. 33rd Annual Symp. Foundations of Comp. Sci.*, pages 714–723, 1992.

[8] E. Kaltofen and M. F. Singer. Size efficient parallel algebraic circuits for partial derivatives. In *Proc. IV International Conference on Computer Algebra in Physical Research*, edited by D. V. Shirkov, V. A. Rostovtsev, and V. P. Gerdt, pages 133–145, World Scientific Publ., Singapore, 1991.

[9] M. Kaminski, D. G. Kirkpatrick, and N. H. Bshouty. Addition requirements for matrix and transposed matrix products. *J. Algorithms*, 9:354–364, 1988.

[10] V. Shoup. A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic. In *Proc. 1991 Internat. Symp. Symbolic Algebraic Comput.*, edited by S. M. Watt, pages 14–21, ACM Press, 1991.

[11] V. Shoup. Fast construction of irreducible polynomials over finite fields. In Proc. 4th Annual ACM-SIAM Symp. on Discrete Algor., pages 484–492, ACM and SIAM, New York, N.Y., and Philadelphia, Penn., 1993.

[12] V. Strassen. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numer. Math.*, 20:238–251, 1973 (in German).

[13] L. Valiant. The complexity of computing the permanent. *Theoretical Comp. Sci.*, 8:189–201, 1979.

[14] L. Valiant. Reducibility of algebraic projections. *L'Enseignement mathématique*, 28:253–268, 1982.

Algebraic Complexity

Jacques Morgenstern
(jacques.morgenstern@sophia.inria.fr)

INRIA, 2004 route des Lucioles
06565 - Sophia Antipolis, France

Some Results and Methods in Algebraic Complexity (short survey)

1. Ensembles computation:

   Given a family of subsets $\{E_i\}_{i \in J}$ of subsets of a finite set E, can it be reached by union steps (inclusive or), disjoint unions, or symmetric differences (exclusive or) in less than r steps? NP-complete for the two first cases. Unknown for the third case.

2. Linear forms:

   a) Is $v \in Q^n$ a linear combination of length $\leq r$ of $m > n$ given vectors $v_1, v_2, \ldots, v_n$? NP-complete?

   b) Can you compute a set of t linear forms in less than r binary linear combinations $(\lambda f + \alpha g)$? Known for $t = 2$. May be indivisible in general over the integers.

   c) Lower bound in terms of the determinant if the scalars used are bounded.

3. Rational functions:

   m functions of n variables. $\phi : \mathbb{C}^n \rightarrow \mathbb{C}^m$. The degree of group $\phi \subseteq C^n \times C^m$ leads to a lower bound on the number of multiplications or divisions.

Short Description of Odyssée in Our SAFIR Group at INRIA-University of Nice

We develop a software for automatic differentiation of Fortran programs in the direct or reverse mode. We use a strongly typed polymorphic "ML" language, CAML, and the Fortran programs are read, syntactically analyzed; an abstract syntax tree is produced and transformed, and a Fortran code for the derivatives is then generated.

Runtime Compilation and Automatic Differentiation

Joel Saltz
(saltz@cs.umd.edu)

Computer Science Department
University of Maryland
College Park, MD 20742

Our research focuses on the development of methods to make it possible to produce portable compilers that generate efficient multiprocessor code for irregular scientific problemss (i.e., problems that are unstructured, sparse, adaptive, or block structured).

We work closely with applications scientists and engineers whose problem areas include computational fluid dynamics, computational chemistry, computational biology, structural mechanics, and electrical power grid calculations. Key aspects of the research associated with irregular scientific problems focuses on the development of portable runtime support libraries that (1) coordinate interprocessor data movement, (2) manage the storage of, and access to, copies of off-processor data, (3) support a shared name space, and (4) couple runtime data and workload partitioners to compilers. Researchers employ this runtime support in distributed-memory compilers. The runtime support is also used to port applications codes to a variety of multiprocessor architectures. This compiler research involves the development of methods to reduce interprocessor communication costs and to reduce the overheads associated with runtime preprocessing ([2], [1]).

It seems likely that analogous runtime optimizations could be used in a number of contexts in the area of automatic differentiation. When we embed an irregularly distributed array onto a multiprocessor architecture, each processor's program must manage an arbitrary subset of a global name space. In a more general context, our techniques make it possible to efficiently manage data and computations that are associated with arbitrary subsets of a global name space. This capability could be very useful in automatic differentiation codes such as ADIFOR, as it could provide the basis for developing a general method able to allocate data and computations associated with derivative objects in a sparse fashion. The automatic differentiation tool would first identify active variables (as is now done by ADIFOR). It seems likely that, in many cases, only a subset of the indices of an array of active variables will actually be active at a given point in program execution. The tool would then generate code that would, *at runtime*, identify the active *index sets* associated with variable arrays. The tool would also generate code to carry out the automatic differentiation that would allocate the memory needed to store derivative objects only for the active index sets associated with active variables. The compiler could also generate code that uses knowledge of active index sets to reduce the amount of computation needed to compute derivatives.

Another method that may be useful in the context of automatic differentiation would be a particularly aggressive form of stripmining whose goal would be to reduce the memory requirements associated with maintaining derivative objects for active arrays. The key here is that many codes execute sequences of loops, each of which sweeps over some set of large array variables. Assume we can reorder the computation so that the sequence of loops sweep over roughly conforming subsets of active arrays. In other words, we would carry out a stripmine transformation, the result of which would be a code that would consecutively carry out computations for conforming patches of different arrays. We should be able to take advantage of the fact that data access patterns associated with these sweeps are often rather local. In that case, by carrying out runtime preprocessing, we may be able to determine that many active variable indices are live only during a particular stripmine iteration.

References

[1] R. Das and J. H. Saltz. Program slicing techniques for compiling irregular problems. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, Ore., August 1993.

[2] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*. IEEE Computer Society Press, November 1993.

Algebraic Simplification and Automatic Differentiation

Stephen M. Watt
(smwatt@watson.ibm.com)

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

This talk presents algebraic simplification techniques of computer algebra and optimizing compilers that can be usefully applied in the field of automatic differentiation.

We preface the presentation with an example of how analytic derivatives improve upon finite differencing in a real-world application: IBM Burlington has several mainframes devoted to simulating new semiconductor technologies before they are manufactured. Converting the device models with a custom piece of automatic differentiation software employing algebraic simplifications led to an estimated $1,000,000 in monthly savings in terms of CPU time.

We note that the meaning of algebraic simplification depends on the class of expressions and on the measure of simplicity. Even for the relatively simple class of univariate polynomials, there is no single best definition: a factored or expanded representation might be "simpler" on a case-by-case basis. A well-defined subproblem of simplification is to ask whether an expression is equivalent to zero. This question is easier and sufficient for many purposes. Depending on the class of expressions, however, even this question can be provably unsolvable.

Some classes of expressions admit "normal" or "canonical" forms. Simplification to a normal form converts a zero-equivalent expression to 0. Simplification to a canonical form converts mathematically equivalent expressions to a unique representative. Conversion to these forms requires exact coefficient arithmetic, so implementations usually convert floating-point numbers to rational numbers beforehand.

In computer algera, various normal and canonical forms are used for polynomials and rational functions. The class of functions can be extended by treating $\sin(x)$, $\cos(x)$, $\sin(2x)$, etc., as new indeterminates. In this setting, many identities are algebraic relations (e.g., $\sin^2(x) + \cos^2(x) - 1 = 0$). There may be a structure theorem for a class of functions that can be used to express a given set of extensions in terms of an algebraically independent set. This leads to expressions that are in one sense simpler, even though the resulting form is usually more lengthy. It is often more convenient to retain a set of algebraic relations and to simplify expressions modulo this set. An important technique in computer algebra is to simplify an expression modulo a set of polynomials of this sort. We describe how this can be done using Gröbner bases. We then describe other simplification techniques including factorization, functional decomposition, radical transformation and rule-based methods.

Various computational techniques of computer algebra are based on algebraic properties. One standard method is to solve and combine several simpler problems. An example of this is the computation of several modular images that are combined by Chinese remaindering.

One might distinguish between “black box” and “clear box” function representations. In a “black box” representation, one can compute only values of a function — one cannot see any explicit expression structure. This form is the standard in numerical computing, but it has only recently found effective use in computer algebra. In a “clear box” representation, a function is represented as an expression tree or sequence. This is the standard form in computer algebra, but it is relatively uncommon in numeric computing, with automatic differentiation being one example.

Compiler optimizations can be seen as algebraic simplification of “clear box” function representations. We describe a number of these optimization techniques applicable to automatic differentiation. We begin with a description of basic blocks and flow graphs. We then give a detailed description, global data-flow analysis and illustrate how it can be used to eliminate variables or eliminate common subexpressions. We also discuss transformation of programs to static single assignment form and data structure elimination.

In conclusion, we present a list of “dos” and “don’ts”:

- *Do* exploit algebraic relationships (e.g., sin, cos, matrix ops.).
- *Do not* blindly trust loosely defined simplifications.
- *Do* use term orderings to eliminate more expensive function calls.
- *Do not* expect miracles from expression simplification.
- *Do* use data-flow analysis for dependency information, common subexpression elimination, etc.
- *Do not* expect compiler providers to include specific automatic differentiation methods.
- *Do* ask compiler providers to architect application-oriented back doors into their optimizers.