Efficient Derivative Codes through Automatic Differentiation and Interface Contraction: An Application in Biostatistics

Christian Bischof
Mario Cassella
Paul Hovland
Donna Spiegelman

CRPC-TR95514-S
January 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Also available as *MCS-P491-0195* from Argonne National Laboratory, Mathematics and Computer Science Division.

Efficient Derivative Codes through Automatic Differentiation and Interface Contraction: An Application in Biostatistics*

by

Paul Hovland,† Christian Bischof,† Donna Spiegelman,‡ and Mario Casella.§

Abstract

Developing code for computing the first- and higher-order derivatives of a function by hand can be very time-consuming and is prone to errors. Automatic differentiation has proven capable of producing derivative codes with very little effort on the part of the user. Automatic differentiation avoids the truncation errors characteristic of divided-difference approximations. However, the derivative code produced by automatic differentiation can be significantly less efficient than one produced by hand. This shortcoming may be overcome by utilizing insight into the high-level structure of a computation. This paper focuses on how to take advantage of the fact that the number of variables passed between subroutines frequently is small compared with the number of the variables with respect to which we wish to differentiate. Such an "interface contraction," coupled with the associativity of the chain rule for differentiation, allows us to apply automatic differentiation in a more judicious fashion, resulting in much more efficient code for the computation of derivatives. A case study involving a program for maximizing a logistic-normal likelihood function developed from a problem in nutritional epidemiology is examined, and performance figures are presented. We conclude with some directions for future study.

1 Introduction

Many problems in computational science require the evaluation of a mathematical function, as well as the derivatives of that function with respect to certain independent variables. Automatic differentiation provides a mechanism for the automatic generation of code for the computation of derivatives, using the program for the evaluation of the function as input [9, 19]. However, when automatic differentiation is applied without insight into the program being processed, the derivative computation can be almost as expensive as divided differences, especially if the so-called forward mode is being used. Nonetheless, in Section 4, we show that if the user has high-level knowledge about the structure of a program, automatic differentiation can be employed more judiciously, resulting in codes whose performance rivals those produced by many person-hours of hand-coding. In particular, we show how one can exploit "interface contraction," that is, instances where the number of variables passed between subroutines is small compared with the number of variables with respect to which

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, the National Aerospace Agency under Purchase Order L25935D, the U.S. Department of Defense through an NDSEG fellowship, the Centers for Disease Control through Cooperative Agreement K01 OH00106, the National Institutes of Health through Cooperative Agreement R01 CA50597, and the National Science Foundation through NSF Cooperative Agreement No. CCR-9120008.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, {hovland, bischof}@mcs.anl.gov.

‡Departments of Epidemiology and Biostatistics, Harvard School of Public Health, 677 Huntington Avenue, Boston, MA 02115, stdls@gauss.med.harvard.edu.

§Department of Epidemiology, Harvard School of Public Health, 677 Huntington Avenue, Boston, MA 02115, stmbc@gauss.med.harvard.edu.

derivatives are desired. Combining our knowledge of this interface contraction with the chain rule for differentiation enables us to compute derivatives considerably more efficiently than is possible using "black-box" automatic differentiation. We discuss this technique and describe how it was applied to a program segment that maximizes a likelihood function used for statistical analyses investigating the link between dietary intake and breast cancer.

The organization of this paper is as follows. The next section provides a brief introduction to automatic differentiation and explains some of its advantages in comparison with other techniques used for computing derivatives. Section 3 presents interface contraction and explains how it can be used to reduce the computational cost of a derivative computation. Sections 4 and 5 describe the results of our experiments using the application from biostatistics. Section 6 provides a brief summary and discusses possible directions for future study.

2 Automatic Differentiation

Traditionally, scientists who wish to compute the derivatives of a function have chosen one of two approaches—derive an analytic expression for the derivatives and implement this expression as a computer program, or approximate the derivatives using divided differences, for example,

$$\frac{\partial}{\partial t} f(t)\Big|_{t=t_0} \approx \frac{f(t_0+h)-f(t_0)}{h} \tag{1}$$

for small h. The former approach suffers from being tedious and prone to errors, while the latter can produce large errors if the size of the perturbation is not carefully chosen; even in the best case, half of the significant digits will be lost. For problems of limited size, symbolic manipulators, such as Maple [7], are available. These programs can simplify the task of deriving an expression for derivatives and converting this expression into code, but they are typically unable to handle functions that are large or contain branches, loops, or subroutines.

An alternative to these techniques is automatic differentiation [9,19]. Automatic differentiation techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a sequence of elementary operations, such as addition and multiplication, and elementary functions, such as square root and log. By applying the chain rule, for example,

$$\frac{\partial}{\partial t} f(g(t))\Big|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)\Big|_{s=g(t_0)}\right)\left(\frac{\partial}{\partial t} g(t)\Big|_{t=t_0}\right) \tag{2}$$

repeatedly to the composition of those elementary operations, one can compute derivatives of f exactly and in a completely mechanical fashion.

For example, the short code segment

```
y = sin(x)
z = y*x + 5
```

could be augmented to compute derivatives as

```
y = sin(x)
∇y = cos(x)*∇x
z = y*x + 5
∇z = y*∇x + x*∇y
```

where ∇var is a vector representing the derivatives of var with respect to the independent variable(s). Thus, if **x** is the scalar independent variable, then ∇**x** is equal to 1 and ∇**z** represents $\frac{\partial z}{\partial x}$. This example uses the so-called forward mode of automatic differentiation, wherein derivatives of intermediate variables with respect to the independent variables are propagated. There is also a reverse mode of automatic differentiation, which propagates derivatives of the dependent variables with respect to the intermediate variables.

Several tools have been developed that use automatic differentiation for the computation of derivatives [16]. In particular, we mention GRESS [13], PADRE-2 [17], Odyssee [20], and ADIFOR [2] for Fortran programs and ADOL-C [11] and ADIC [4] for C programs. We employed the ADIFOR tool in our experiments.

ADIFOR (Automatic Differentiation of Fortran) [2] provides automatic differentiation for programs written in Fortran 77. Given a Fortran subroutine (or collection of subroutines) describing a "function," and an indication of which variables in parameter lists or common blocks correspond to "independent" and "dependent" variables with respect to differentiation, ADIFOR produces portable Fortran 77 code that allows the computation of the derivatives of the dependent variables with respect to the independent ones.

3 Interface Contraction

Automatic differentiation tools such as ADIFOR produce derivative code that typically outperforms divided difference approximations (see, for example, [1, 3, 5, 6, 18]), but, not surprisingly, is usually much less efficient than a hand-derived code probably could be.* We introduce a technique, called "interface contraction," that can dramatically reduce the runtime and storage requirements for computing derivatives via automatic differentiation. This technique takes advantage of a programmer's understanding of which subroutines encapsulate the majority of the computation and knowledge of the number of variables passed to these subroutines. In counting the number of variables, we count the number of *scalar* variables; that is, a 10×10 array would be counted as 100 variables. We now introduce interface contraction in detail.

Consider a function $f : \mathbf{R}^n \rightarrow \mathbf{R}^p$, and denote its cost by $C(f)$. The "cost" of a computation is usually measured in terms of memory and floating-point operations, and the following argument applies to either. If the derivatives of $f(x)$ with respect to x, $\frac{\partial f}{\partial x}$, are computed using the so-called forward mode of automatic differentiation, the additional cost of computing these derivatives is approximately $n \times C(f)$, because the derivative code includes operations on vectors of length n. Now, suppose that $f(x)$ can be interpreted as the composition of two functions $h : \mathbf{R}^m \rightarrow \mathbf{R}^p$ and $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$ such that $f(x) = h(g(x))$. To compute $\frac{\partial g}{\partial x}$ and $\frac{\partial h}{\partial g}$ independently, the computational cost is approximately $nC(g)$ and $mC(h)$, respectively. Thus, the total cost of computing $\frac{\partial f}{\partial x}$ is $nC(g)+mC(h)$ plus the cost of performing the matrix multiplication, $\frac{\partial h}{\partial g} \times \frac{\partial g}{\partial x}$. The cost of the matrix multiplication will usually be small compared with the cost of computing the partial derivatives. If $m < n$, then the cost of computing derivatives is reduced by using this method. Furthermore, if $C(h) \gg C(g)$, this method has a cost of approximately $mC(f)$, which may be substantially less than $nC(f)$. The value of m is said to be the width of the "interface" between g and h. Hence, if m is less than n, we have what we call interface contraction. An even more pronounced effect can be seen in the case of Hessians, since the cost of computing derivatives using the forward mode is quadratic in the number of independent variables. Hence, the potential speedup due to interface contraction is $(n/m)^2$ rather than just n/m.

We note that a similar approach to interface contraction was mentioned by Iri [15] as a "vertex cut" when considering automatic differentiation as applied to a computational graph representation of a program.

3.1 Microscopic Interface Contraction

A simple case of interface contraction occurs for every complex assignment statement in a program. Consider a simple example, the following statement, which computes the product of five terms:

```
f = y(1) * y(2) * y(3) * y(4) * y(5)
```

*In most cases, no hand-derived derivative code was available for comparison.

Using temporaries r1, r2, and r3, we could rewrite this as

```
r1 = y(1) * y(2)
r2 = y(3) * r1
r3 = y(4) * r2
f = y(5) * r3
```

and apply the forward mode of automatic differentiation to yield, for n independent variables,

```
r1 = y(1) * y(2)
∇r1(1:n) = y(1) * ∇y(1:n,2) + y(2) * ∇y(1:n,1)
r2 = y(3) * r1
∇r2(1:n) = y(3) * ∇r1(1:n) + r1 * ∇y(1:n,3)
r3 = y(4) * r2
∇r3(1:n) = y(4) * ∇r2(1:n) + r2 * ∇y(1:n,4)
f = y(5) * r3
∇f(1:n) = y(5) * ∇r3(1:n) + r3 * ∇y(1:n,5)
```

But, if we notice that this single statement is a scalar function of a vector $y \in \mathbf{R}^5$, which is itself a function of n independent variables, we have the situation described above, where $p = 1$, $m = 5$, and $n = \mathtt{n}$. Thus, if we compute $\overline{y} = \frac{\partial f}{\partial y}$ first, we can compute $\nabla \mathtt{f} = \overline{y} \times \nabla \mathtt{y}$ more efficiently. For computing the derivatives of scalar functions, the reverse mode of automatic differentiation is more efficient than the forward mode [9, 19], so we can use it to compute the values of $\overline{y}$. The code for computing $\overline{y}$ using the reverse mode is

```
r1 = y(1) * y(2)
r2 = r1 * y(3)
r3 = r2 * y(4)
f = r3 * y(5)
y5bar = r3
r2bar = y(4)*y(5)
y4bar = r2 * y(5)
r1bar = y(3)*r2bar
y3bar = r1 * r2bar
y2bar = y(1) * r1bar
y1bar = y(2) * r1bar
```

where each y*i*bar represents $\overline{y}(i)$. We can then perform the matrix multiplication

```
∇f(1:n) = y1bar * ∇y(1:n,1) + y2bar * ∇y(1:n, 2)
        + y3bar * ∇y(1:n, 3) + y4bar * ∇y(1:n, 4)
        + y5bar * ∇y(1:n, 5)
```

This hybrid mode of automatic differentiation is employed by ADIFOR [2]. We see that interface contraction is mainly responsible for the lower complexity of ADIFOR-generated code compared with divided-difference approximations. We also note that, for a moderate number of variables on the right-hand side, we would still come out ahead if we used the forward mode to compute $\overline{y}$, instead of the reverse mode.

3.2 Macroscopic Interface Contraction

An analogous situation exists for larger program units, in particular, subroutines. Suppose we have a subroutine **subf** that computes a function $z = f(x)$ and that simply calls two subroutines, **subg** and **subh**, as follows:

```
subroutine subf(x,z,n)
integer n
real x(n),z(n)
real y(2)

call subg(x,y,n)
call subh(y,z,n)

return
end
```

Subroutine `subg` computes `y` from `x` ,and `subh` computes `z` from `y`. Applying ADIFOR to this subroutine would result in the creation of subroutines `g$subf`, `g$subg`, and `g$subh`, each of which would work with derivative vectors of length n, representing derivatives with respect to x. If we process `subh` separately to get a subroutine `g$subh` that computes $\frac{\partial z}{\partial y}$, use `g$subg` to compute $\frac{\partial y}{\partial x}$, and then multiply $\frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$, we get $\frac{\partial z}{\partial x}$ as before. However, the additional computational cost of `g$subh` is no longer n times the cost of `subh` but merely two times the cost of `subh` (`y` is a vector of length 2, so $m = 2$). If `subh` is computationally demanding, this may be a great savings.

The exploitation of interface contraction in this example is illustrated in Figure 1. The width of an arrow corresponds to the amount of information passing between and computed within the various subroutines. When automatic differentiation is applied to the whole program ("Before"), the gradient objects have length n. Thus, large amounts of data must be computed and stored, resulting in large runtimes and memory requirements. If interface contraction is exploited by processing `subg` and `subh` separately ("After"), the amount of data computed within `g$subh` is greatly reduced, resulting in reduced computational demands within this subroutine. If `subh` is expensive, this approach results in greatly improved performance and reduced memory requirements.

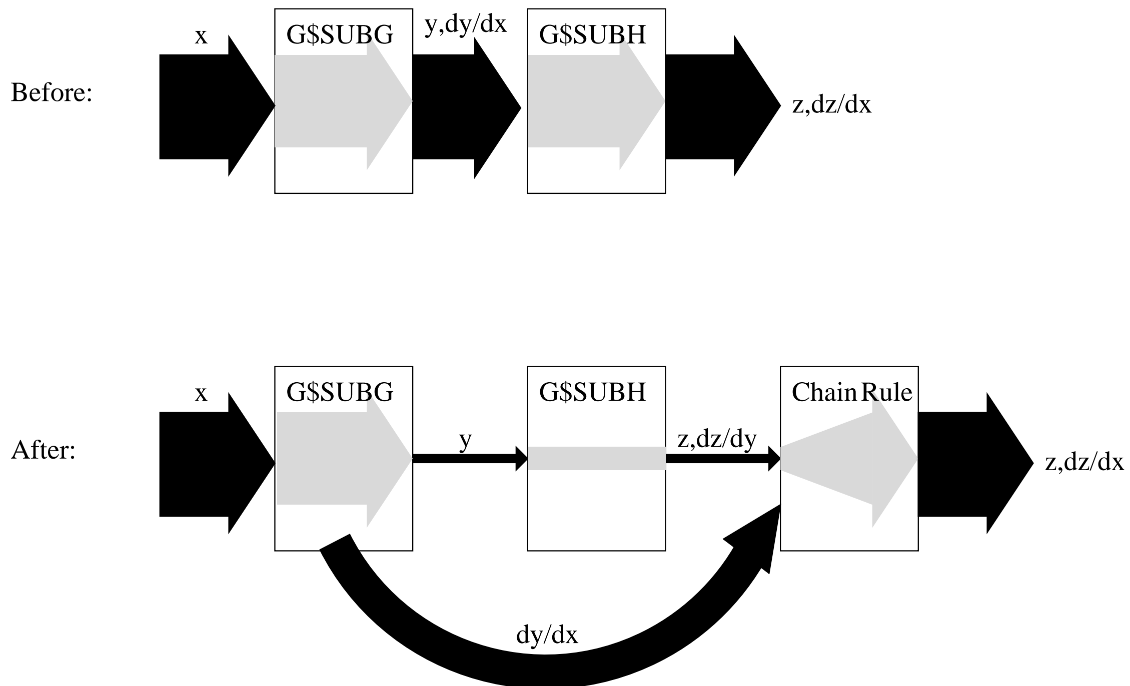

Figure 1: Schematic of subroutine calls before and after interface contraction

Note that this process requires a little more work on the user's part than simply applying a "black-box" automatic differentiation tool. While previously we just applied the automatic differentiation tool to **subf** and the subroutines it called, we now must apply it separately to **subg** and **subh**, and we must provide the matrix-matrix multiply "glue" code as well. Nonetheless, this approach produces in a short amount of time, and with a fairly low likelihood of human error, a derivative code with significantly lower complexity than that derived from "black-box" application of an automatic differentiation tool.

4 An Application of Interface Contraction

The macroscopic version of interface contraction can be used advantageously in the development of derivative codes for two log-likelihood functions used for biostatistical analysis. These functions were motivated by a problem in nutritional epidemiology investigating the relationships of age and dietary intakes of saturated fat, total energy, and alcohol with the four-year risk of developing breast cancer in a prospective cohort of 89,538 nurses [21]. The likelihood functions take into account measurement error in total energy and binary misclassification in saturated fat and alcohol intake, and include an integral that is evaluated numerically by using the algorithm of Crouch and Spiegelman [8]. The Hessian and gradient are needed for optimization of these nonlinear likelihood functions, and the Hessian is again needed to evaluate the variance-covariance matrix of the resulting maximum likelihood estimates. Two likelihood functions were fit to these data, a 33-parameter function and a 17-parameter function.

Although the current version of ADIFOR does not support second derivatives, we were able to apply ADIFOR twice to produce code for computing the Hessian (see [14] for more details). This method for computing second derivatives is somewhat tedious, and not optimal from a complexity point of view, but does produce correct results. Direct support for second derivatives will be provided in a future release of ADIFOR.

The initial results using ADIFOR exhibited linear increase in computational complexity for gradients and quadratic increase for Hessians as expected. As a result, computing the Hessian for the 33-parameter function required approximately six hours on a SPARCstation 20. However, this problem turns out to be very suitable for exploiting interface contraction. The subroutine defining the function to be differentiated, **lglik3**, calls subroutine **integr**, whose computational expense dominates the computation (approximately 85% of the overall computation time is spent in **integr**). The input variable for subroutine **lglik3**, **xin**, is a vector of length 17 (33), while the length of the input variable for subroutine **integr**, **x**, is 2. The output variables for the routines are **xlogl** (a scalar) and **f** (a scalar), respectively. So, using the notation of our previous example, we have $n = 17$ (33), $m = 2$, and $p = 1$. The difference between this situation and the one examined in Section 3.2 is that rather than being preceded by another subroutine call, **integr** is surrounded by code and is within a loop. In addition, ADIFOR is used to generate code for computing a Hessian, rather than a gradient, as was the case in our previous examples.

Modifying the ADIFOR-generated routines for computing Hessians to accommodate interface contraction is straightforward. The subroutine **integr** and the subroutines it calls are processed separately from **lglik3**, making it possible to compute derivatives with respect to **x** instead of **xin**. Since we are computing Hessians, this reduces computational complexity for the derivatives for **integr** by a factor of approximately $\frac{(17*17)}{(2*2)} \approx 75$ for the 17-parameter problem and $\frac{(33*33)}{(2*2)} \approx 273$ for the 33-parameter problem.

5 Experimental Results

Figures 2 and 3 summarize performance results for the 17- and 33-parameter problems, respectively, on a SPARCstation 20 at the Channing Laboratory, Harvard Medical School, and Brigham and Women's Hospital, Boston, Massachusetts. This workstation is equipped with a 50 MHz Super-

Task	Method	Time	#feval	Factor	Max Error
Func	Analytic	10.94	1.0	-	-
Grad	Analytic	36.65	3.35	1.00	0
	Black-Box AD	186.70	17.07	5.09	6.5e-14
	ADIFOR & Interface Contr.	54.32	4.97	1.48	6.1e-14
	Analytic & Interface Contr.	49.31	4.51	1.34	6.1e-14
Hess	Analytic	185.80	16.98	1.00	0
	Black-Box AD	5636.00	515.17	30.30	9.9e-11
	ADIFOR & Interface Contr.	609.70	55.73	3.28	9.9e-11
	Analytic & Interface Contr.	608.90	55.66	3.28	9.9e-11

Figure 2: Experimental results for the 17-parameter problem

Task	Method	Time	#feval	Factor	Max Error
Func	Analytic	11.67	1.0	-	-
Grad	Black-Box AD	394.25	33.78	4.15	0
	Interface Contraction	104.56	8.96	1.10	8.3e-14
	Analytic Interface	95.00	8.14	1.00	6.1e-14
Hess	Black-Box AD	21130.00	1810.62	7.09	0
	ADIFOR & Interface Contr.	3049.10	261.28	1.02	7.4e-14
	Analytic & Interface Contr.	2979.30	255.30	1.00	2.7e-13

Figure 3: Experimental results for the 33-parameter problem

SPARC processor and 48MB of memory. All code was compiled by using Sun's f77 compiler with the "-O" option. The times provided are the total cpu time in seconds, averaged over several runs. The column labelled "#feval" shows the number of function evaluations that could be completed in that time. The cpu time required for a given method divided by the execution time of the fastest method is reported in the "Factor" column. The maximum (over all components of the gradient or Hessian) relative error is reported in the "Max Error" column, compared with values obtained by the method for which the error is listed as 0 in the figures. The column labeled "Task" indicates whether the function (func), its gradient (grad), or its Hessian (Hess) is being computed. We used four methods to compute derivatives:

Analytic: For the 17-parameter problem, code was developed over the course of two years of person-time to compute the derivatives analytically. No such code exists for the 33-parameter problem.

Black-Box Automatic Differentiation: This method corresponds to the code generated by ADIFOR. Thus, even for the smaller (17-parameter) problem, the code generated via automatic differentiation is much slower than the code generated by hand. However, it also took almost no person-time to develop.

To implement the interface contraction (IC) approach, we used two different versions of the derivative code for `integr`:

Analytic and Interface Contraction: Code already existed for analytically computing the derivatives of the output variables of `integr` with respect to its input variables, and we employed this code to compute the derivatives of `integr`.

ADIFOR and Interface Contraction: We employed the derivative code generated by ADIFOR for `integr`.

As a rough estimate, it can be expected that, for n independent variables, computing derivatives by using the forward mode of automatic differentiation requires n and n^2 the cost of computing the function for gradient and Hessian, respectively. For our problems, this would amount to the equivalent of 17 (33) and 289 (1089) function evaluations to compute the gradient and Hessian of the 17- (33-) parameter function. As can be seen from the #feval column, the gradient computed by ADIFOR conforms well to this rough estimate, while the Hessian computation performs worse by a factor of less than two. This is due to the fact that the Hessian code (which was derived by applying ADIFOR to the ADIFOR-generated first-order derivative code) cannot internally exploit the symmetry inherent in Hessian computations.

The exploitation of interface contraction significantly improved performance. There is little difference in the performance of the two implementations of interface contraction with analytic and ADIFOR-generated derivative code for `integr`, respectively. The fact that the ADIFOR-generated Hessian code for `integr` does not exploit symmetry is of little importance here since we compute only a 2×2 Hessian. For the 17-parameter problem, interface contraction allows us to obtain the gradient by a factor less than 1.5 slower than the hand-derived version, corresponding to the time taken by about five function evaluations. The gradient for the 33-parameter problem is obtained at the cost of about eight function evaluations. The 17×17 (33×33) Hessian is obtained at the cost of about 55 (260) function evaluations.

The fact that interface contraction does worse, in comparison with the analytic approach, for Hessians is due to an effect akin to that described by Amdahl's law [12]. For the 17-parameter problem, the cost of the black-box version of the first- (second-) derivative code of `integr` can be expected to be approximately $\frac{17}{2} = 8.5$ ($\frac{(17*17)}{(2*2)} \approx 75$) times that of the interface contraction version. The speedup of about 2.8 (10) that we measured is due to the fact that some small amount of time is spent in parts of the code outside of `integr`. In the interface contraction version of the derivative code, the ratio of time outside of the derivative code for `integr` to time within increases, resulting in a smaller speedup. For example, if the black-box version spends 85% of execution time for computing the second derivatives of subroutine `integr` and 15% of execution in the rest of the program, the interface contraction version will require $.15 + .85 * \frac{1}{75} = .161$ the execution time of the black-box version, for a speedup factor of about 6.2. By the same token, interface contraction leads to a smaller decrease in computation time for the 33-parameter problem, as the portion of the execution time outside of `integr` increases.

6 Conclusions

Although automatic differentiation is an excellent tool for developing accurate derivative codes quickly, the resultant code can sometimes be significantly more expensive than analytic code written by a programmer. We introduced an approach for reducing derivative complexity that capitalizes on chain rule associativity and what we call "interface contraction." By combining the technique of automatic differentiation with techniques capitalizing on high-level program-specific information, such as interface contraction, a user can create codes in the span of a few days that rival the performance of codes that require months to develop by hand. Application of this approach to a biostatistics problem validated the promise of this approach.

Computational differentiation (CD) is an emerging discipline that, in addition to "black-box" automatic differentiation, exploits insight into program structure and underlying algorithms. Its goal is the generation of efficient derivative codes with minimal human effort, Future research in CD will further explore how a computational scientist can exploit knowledge about a program (such as interface contraction) or an underlying algorithm (such as the solution of a nonlinear system of equations [10]) to reduce the cost of computing derivatives. Research in CD will also lead to the

development of a tool infrastructure to make it easier to employ this knowledge in building derivative codes.

Acknowledgments

We thank Alan Carle for his instrumental role in the ADIFOR project, Andreas Griewank for stimulating discussions, and Eugene Demidenko for his valuable assistance in program development. We thank Dr. Frank Speizer for making available the facilities of the Channing Laboratory for performing the analyses described in this paper.

References

[1] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[2] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[3] Christian Bischof, George Corliss, Larry Green, Andreas Griewank, Kara Haigler, and Perry Newman. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Journal on Computing Systems in Engineering*, 3(6):625–638, 1992.

[4] Christian Bischof and Andrew Mauer. Private communication. Argonne National Laboratory, 1995.

[5] Christian Bischof, Greg Whiffen, Christine Shoemaker, Alan Carle, and Aaron Ross. Application of automatic differentiation to groundwater transport models. In Alexander Peters et al., editor, *Computational Methods in Water Resources X*, pages 173–182, Dordrecht, 1994. Kluwer Academic Publishers.

[6] Alan Carle, Lawrence Green, Christian Bischof, and Perry Newman. Applications of automatic differentiation in CFD. In *Proceedings of the 25th AIAA Fluid Dynamics Conference, AIAA Paper 94-2197*. American Institute of Aeronautics and Astronautics, 1994.

[7] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Michael B. Monagan, and Stephen M. Watt. *MAPLE Reference Manual.* Watcom Publications, Waterloo, Ontario Canada, 1988.

[8] E. A. C. Crouch and D. Spiegelman. The evaluation of integrals of the form $\int f(t)e^{-t^2}dt$. Application to logistic-normal models. *Journal of the American Statistical Association*, 85:464–469, 1990.

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

[10] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence of iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[11] Andreas Griewank, David Juedes, and Jay Srinivasan. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Preprint MCS-P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, 1990.

[12] J. J. Hack. Peak vs. sustained performance in highly concurrent vector machines. *Computing*, 19(9):9–19, 1986.

[13] Jim E. Horwedel. GRESS: A preprocessor for sensitivity studies on Fortran programs. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 243–250. SIAM, Philadelphia, 1991.

[14] Paul Hovland, Christian Bischof, and Alan Carle. Using ADIFOR to compute Hessians. Technical Report ANL/MCS–TM–192, Mathematics and Computer Science Division, Argonne National Laboratory, 1995 (in press).

[15] Masao Iri. History of Automatic Differentiation and Rounding Estimation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 1–16. SIAM, Philadelphia, 1991.

[16] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–330, Philadelphia, 1991. SIAM.

[17] Koichi Kubota. PADRE2, a FORTRAN precompiler yielding error estimates and second derivatives. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 251–262. SIAM, Philadelphia, 1991.

[18] Seon Ki Park, Kelvin Droegemeier, Christian Bischof, and Tim Knauff. Sensitivity analysis of numerically-simulated convective storms using direct and adjoint methods. In *Preprints, 10th Conference on Numerical Weather Prediction, Portland, Oregon*, pages 457–459. American Meteorological Society, 1994.

[19] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[20] Nicole Rostaing, Stephane Dalmas, and Andre Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[21] W. C. Willett, et al. Dietary fat and fiber in relation to risk of breast cancer. *Journal of the American Medical Association*, 268:2037–2044, 1992.