EFFICIENT DERIVATIVE CODES THROUGH AUTOMATIC DIFFERENTIATION AND INTERFACE CONTRACTION: AN APPLICATION IN BIOSTATISTICS*

P. HOVLAND†, C. BISCHOF‡, D. SPIEGELMAN§, AND M. 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 variables with respect to which one wishes to differentiate. Such an "interface contraction," coupled with the associativity of the chain rule for differentiation, allows one to apply automatic differentiation in a more judicious fashion, resulting in much more efficient code for the computation of derivatives. A case study involving the ADIFOR automatic differentiation tool and a program for maximizing a logistic-normal likelihood function developed from a problem in nutritional epidemiology is examined, and performance figures are presented.

Key words. automatic differentiation, computational differentiation, interface contraction, log-likelihood functions, derivatives, ADIFOR

AMS subject classifications. 65-04, 39-04, 49-04, 65-05, 68-04, 68N20, 92B15

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 [15, 24]. When automatic differentiation is applied blindly to a program, the derivative computation can be almost as expensive as divided differences, especially if the so-called forward mode is being used. We show in § 3, however, that if the user has high-level knowledge about the structure of a program, currently available automatic differentiation tools can be employed more judiciously, resulting in codes whose performance rivals those produced by laborious 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 derivatives are desired.

† Computer Science Department, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801, hovland@uiuc.edu.
‡ Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, 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.

While previous studies have addressed how an automatic differentiation tool developer might exploit analogous contractions in a computational graph, we demonstrate how the user of an automatic differentiation tool can easily modify the code produced by this tool to take advantage of interface contraction. Dealing with programs, rather than computational graphs, leads naturally to the simple technique for exploiting chain rule associativity described in §2. The technique depends upon an easily recognized program feature, the "width" of the interface between program units. We show the relevance and practicality of this approach by applying it 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. Section 1 provides a brief introduction to automatic differentiation and explains some of its advantages in comparison with other techniques used for computing derivatives. Section 2 presents interface contraction, relates our work to previous research, and explains how interface contraction can be used to reduce the computational cost of a derivative computation. Sections 3 and 4 describe the results of our experiments using the application from biostatistics. Section 5 provides a summary and suggests directions for future study.

1. 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,

$$(1) \qquad \left. \frac{\partial}{\partial t} f(t) \right|_{t=t_0} \approx \frac{f(t_0 + h) - f(t_0)}{h}$$

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 [10] and Mathematica [29], 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 [15, 24]. 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,

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

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. For problems with a small number of dependent variables, the reverse mode offers theoretically much better performance than the forward mode [15]. An efficient implementation of the reverse mode is rendered challenging by the fact that one needs to remember or recompute any intermediate value that affects the final result nonlinearly. An example is given in [3].

Several tools have been developed that use automatic differentiation for the computation of derivatives [21]. In particular, we mention ADIFOR [2, 3], AMC [13], GRESS [18], Odyssee [25], and PADRE-2 [22] for Fortran programs and ADOL-C [16] and ADIC [6] for C programs. We also mention the GRAD package for Maple, which is part of the Maple Share Library (Maple V Release 4) and provides automatic differentiation in the context of a symbolic manipulator. It should also be noted that, in contrast to the situation described in [26], recent tool developments have resulted in AD-generated derivative code whose runtime performance is in line with theoretical predictions, although the potentially explosive memory requirements of current reverse mode tools may still lead to difficulties.

We employed the ADIFOR source transformation tool in our experiments. ADIFOR (**A**utomatic **D**ifferentiation of **For**tran) [2, 3] 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.[1]

2. Interface Contraction. Automatic differentiation tools such as ADIFOR produce derivative code that typically outperforms divided difference approximations (see, for example, [1, 4, 7, 9, 23]), but, not surprisingly, is usually much less efficient than optimal hand-derived code. We describe a technique using what we call "interface contraction," that can reduce dramatically the runtime and storage requirements for computing derivatives via automatic differentiation. This technique can, at a low level, easily be embedded into automatic differentiation tools, and, at a higher level, allows a programmer to capitalize on his or her knowledge of which subroutines encapsulate the majority of the computation, and of the number of variables passed to these subroutines. In counting the number of variables, we count the number of *scalar* variables; for example, a 10×10 array would be counted as 100 variables.

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

---

[1] The ADIFOR 2.0 system can be obtained via the WWW at http://www.mcs.anl.gov/adifor and http://www.cs.rice.edu/~adifor.

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 $h : \mathbf{R}^m \to \mathbf{R}^p$ and $g : \mathbf{R}^n \to \mathbf{R}^m$ 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 second 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.

Our technique for exploiting interface contraction can be viewed as a reinterpretation, in a source program context, of a computational graph-based technique, described by Iri [20] as a "vertex cut" and used by Volin and Ostrovskii [27] in their multilevel differentiating technique. Looking for narrow interfaces in a program corresponds to looking for small separators in the computational graph. The former is simple, effective, and can be done by a programmer. The latter is more complicated, often requiring a sophisticated graph algorithm [14], and must be done within an automatic differentiation tool. In §2.1, we revisit the ADIFOR translation strategy for assignment statements, which combines both forward and reverse mode in an interface contraction context. In §2.2, we motivate the application of interface contraction to the composition of subroutines, assuming black-box tools exist for generating derivative code for the constituent subroutines. The latter instance of interface contraction demonstrates that automatic differentiation coupled with the insight of a knowledgeable programmer can provide much better performance than automatic differentiation alone.

2.1. Microscopic Interface Contraction. A simple case of interface contraction occurs for every scalar assignment statement involving at least two variables on the right-hand side. As a simple example, consider the following statement, which computes the product of five terms:

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

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 = $ n. Thus, if we compute $\overline{y} = \frac{\partial f}{\partial y}$ first, we can compute $\nabla f = \overline{y} \times \nabla y$ more efficiently. For computing the derivatives of scalar functions, the reverse mode of automatic differentiation is more efficient than the forward mode [15, 24], 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 yibar 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, 3]. 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, due to interface contraction, even using the forward mode to compute $\overline{y}$ would offer performance superior to that of globally using the straight forward mode.

2.2. Macroscopic Interface Contraction. An analogous situation exists for larger program units, in particular, subroutines. Suppose we have a subroutine subf that computes a function $z = f(x)$ by calling 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
```

16

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.
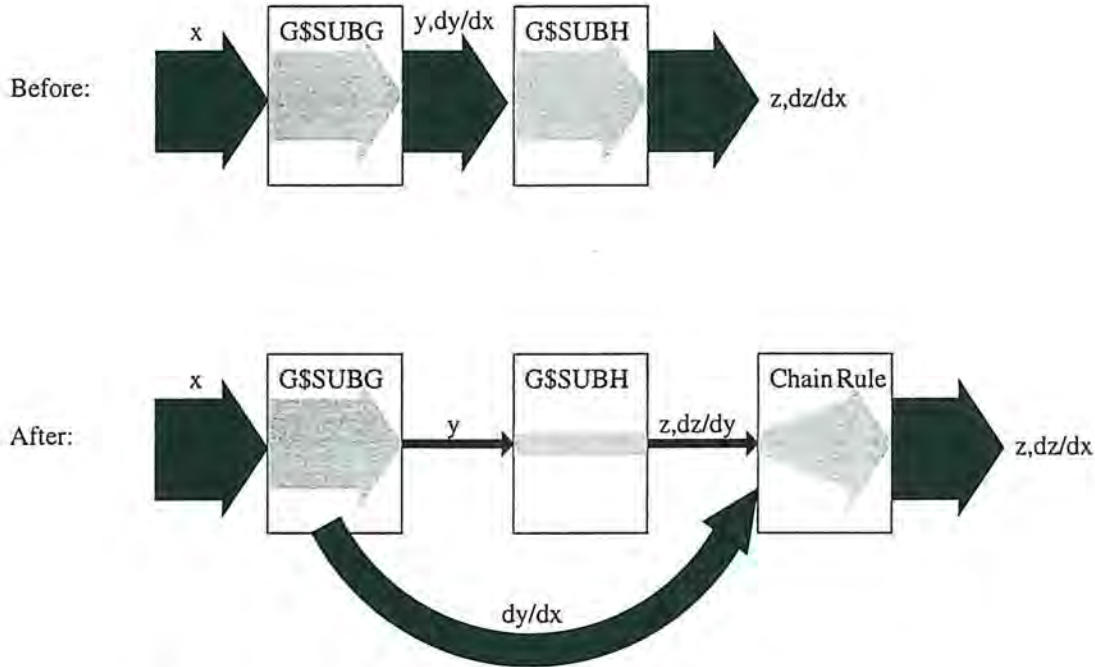

FIG. 1. *Schematic of subroutine calls before and after interface contraction*

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.

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.

We have described macroscopic interface contraction in terms of the forward mode. However, interface contraction is equally applicable to derivative code generated using the reverse mode. It can also be applied to code generated using both modes. In fact, given a reverse mode tool and a forward mode tool capable of generating compatible derivative code, an excellent way to exploit a narrow interface is to use the reverse mode for code before the interface and the forward mode for code after the interface.

3. 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 [28]. 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 [11]. Newton's method with a line search globalization is employed for the optimization of these nonlinear likelihood functions, and hence the Hessian and gradient are needed at every optimization step. The Hessian is also 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 [19] 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. It is planned that 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 §2.2 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 ap-

TABLE 1
Experimental results for the 17-parameter problem

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

TABLE 2
Experimental results for the 33-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
	ADIFOR & Interface Contr.	104.56	8.96	1.10	8.3e-14
	Analytic & Interface Contr.	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

proximately $\frac{(17*17)}{(2*2)} \approx 75$ for the 17-parameter problem and $\frac{(33*33)}{(2*2)} \approx 273$ for the 33-parameter problem.

4. Experimental Results. Tables 1 and 2 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 SuperSPARC 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 tables. 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`.

To facilitate understanding of these techniques by programmers who might wish to use them, as well as to provide other automatic differentiation researchers with a resource for testing new ideas, we have made the source code for computing the 17 parameter log-likelihood function, and for computing its gradient using each of the four techniques mentioned above, available via the World Wide Web at `http://www.mcs.anl.gov/Projects/autodiff/interface.html`.

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 in less than 1.5 times the compute time required by 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 [17]. 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 90% of execution time for computing the second derivatives of subroutine `integr` and 10% of execution in the rest of the program, the interface contraction version will require $.10+.90*\frac{1}{75} = .112$ the execution time of the black-box version, for a speedup factor of about 9. In the same way, the speedup due to interface contraction is smaller for the 33-parameter problem, due to a greater percentage of execution time spent outside of `integr`.

Certainly, the interface contraction technique will not provide such large improve-

ments in performance for all programs. However, our results demonstrate that contractions occur naturally in real scientific programs and that the theoretical reductions in complexity translate into real gains in computational performance. Furthermore, interface contraction used in conjunction with a reverse mode tool could provide even better performance.

5. 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." This technique for exploiting chain rule associativity arises naturally when considering interfaces between program units in a source translation environment, where derivative code is generated by explicitly rewriting program source. A knowledgeable programmer can easily recognize "narrow" interfaces, and by applying this simple technique, develop efficient derivative code using currently available automatic differentiation tools. Our application of the interface contraction technique to a biostatistics application demonstrates that for real scientific applications, 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.

The exploitation of chain rule associativity and program structure is what makes automatic differentiation challenging. Interface contraction is but one example of a technique capitalizing on both. Other examples are the exploitation of derivative sparsity [5, 12], and their use in pseudo-adjoint schemes [8]. We expect that more techniques exploiting particular program structures beyond the traditional forward and reverse mode dichotomy will be developed as automatic differentiation tools mature and enter the mainstream of computational science.

REFERENCES

[1] B. AVERICK, J. MORÉ, C. BISCHOF, A. CARLE, AND A. GRIEWANK, *Computing large sparse Jacobian matrices using automatic differentiation*, SIAM Journal on Scientific Computing, 15 (1994), pp. 285–294.

[2] C. BISCHOF, A. CARLE, G. CORLISS, A. GRIEWANK, AND P. HOVLAND, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 11–29.

[3] C. BISCHOF, A. CARLE, P. KHADEMI, AND A. MAUER, *The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs*, 1994. Preprint MCS-P481-1194, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR94491, Center for Research on Parallel Computation, Rice University.

[4] C. BISCHOF, G. CORLISS, L. GREEN, A. GRIEWANK, K. HAIGLER, AND P. NEWMAN, *Automatic differentiation of advanced CFD codes for multidisciplinary design*, Journal on Computing Systems in Engineering, 3 (1992), pp. 625–638.

[5] C. BISCHOF, P. KHADEMI, A. BOUARICHA, AND A. CARLE, *Computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation*, Preprint MCS-P519-0595, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[6] C. BISCHOF AND A. MAUER, *ADIC – a tool for the automatic differentiation of C programs*. to appear, 1994.

[7] C. BISCHOF, G. WHIFFEN, C. SHOEMAKER, A. CARLE, AND A. ROSS, *Application of automatic differentiation to groundwater transport models*, in Computational Methods in Water Resources X, A. Peters, ed., Dordrecht, 1994, Kluwer Academic Publishers, pp. 173–182.

[8] C. H. BISCHOF, *Automatic differentiation, tangent linear models and pseudo-adjoints*, Preprint MCS-P472-1094, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[9] A. CARLE, L. GREEN, C. BISCHOF, AND P. 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.

[10] B. W. CHAR, K. O. GEDDES, G. H. GONNET, M. B. MONAGAN, AND S. M. WATT, *MAPLE Reference Manual*, Watcom Publications, Waterloo, Ontario Canada, 1988.

[11] E. 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 (1990), pp. 464–469.

[12] L. C. W. DIXON, Z. MAANY, AND M. MOHSENINIA, *Automatic differentiation of large sparse systems*, Journal of Economic Dynamics & Control, 14 (1990). Presented at IFAC on Dynamic Modelling & Control of National Economies, Edinburgh, July, 1989. Also appeared as Technical Report NOC TR223, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K., July, 1989.

[13] R. GIERING, *Adjoint model compiler, manual version 0.2, AMC version 2.04*, tech. report, Max-Planck Institut für Meteorologie, August 1992.

[14] J. R. GILBERT, J. P. HUTCHINSON, AND R. E. TARJAN, *A separator theorem for graphs of bounded genus*, J. Algorithms, 5 (1984), pp. 391–407.

[15] A. GRIEWANK, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, Amsterdam, 1989, Kluwer Academic Publishers, pp. 83–108.

[16] A. GRIEWANK, D. JUEDES, AND J. 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.

[17] J. J. HACK, *Peak vs. sustained performance in highly concurrent vector machines*, Computing, 19 (1986), pp. 9–19.

[18] J. E. HORWEDEL, *GRESS: A preprocessor for sensitivity studies on Fortran programs*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 243–250.

[19] P. HOVLAND, *Using ADIFOR to compute Hessians*, Tech. Report MCS–TM–195, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[20] M. IRI, *History of automatic differentiation and rounding estimation*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 1 − 16.

[21] D. JUEDES, *A taxonomy of automatic differentiation tools*, in Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss, eds., Philadelphia, 1991, SIAM, pp. 315–330.

[22] K. KUBOTA, *PADRE2, a FORTRAN precompiler yielding error estimates and second derivatives*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 251–262.

[23] S. K. PARK, K. DROEGEMEIER, C. BISCHOF, AND T. KNAUFF, *Sensitivity analysis of numerically-simulated convective storms using direct and adjoint methods*, in Preprints, 10th Conference on Numerical Weather Prediction, Portland, Oregon, American Meteorological Society, 1994, pp. 457–459.

[24] L. B. RALL, *Automatic Differentiation: Techniques and Applications*, vol. 120 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 1981.

[25] N. ROSTAING, S. DALMAS, AND A. GALLIGO, *Automatic differentiation in Odyssee*, Tellus, 45a (1993), pp. 558–568.

[26] E. SOULIE, *User's experience with Fortran compilers for least squares problems*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 297–306.

[27] Y. M. VOLIN AND G. M. OSTROVSKII, *Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis*, Computers and Mathematics with Applications, 11 (1985), pp. 1099 − 1114.

[28] W. WILLETT, D. HUNTER, AND M. S. ET AL., *Dietary fat and fiber in relation to risk of breast cancer*, Journal of the American Medical Association, 268 (1992), pp. 2037–2044.

[29] S. WOLFRAM, *Mathematica Reference Guide (for Mathematica Version 2)*, Addison-Wesley, Reading, Mass., 1992.

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

P. Hovland, C. Bischof, D. Spiegelman, M. 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 variables with respect to which one wishes to differentiate. Such an "interface contraction," coupled with the associativity of the chain rule for differentiation, allows one to apply automatic differentiation in a more judicious fashion, resulting in much more efficient code for the computation of derivatives. A case study involving the ADIFOR (Automatic Differentiation of Fortran) tool and a program for maximizing a logistic-normal likelihood function developed from a problem in nutritional epidemiology is examined, and performance figures are presented.

Key words. automatic differentiation, computational differentiation, interface contraction, log-likelihood functions, derivatives, ADIFOR

AMS Subject Classifications. 65-04, 39-04, 49-04, 65-05, 68-04, 68N20, 92B15

DOI. 10.1137/S1064827595281800