Automatic Differentiation: Obtaining Fast and Reliable Derivatives – Fast*

Christian H. Bischof,† Alan Carle,‡
Peyvand M. Khademi,† Gordon Pusch†

Argonne Preprint MCS-P484-1194

to appear in
Proc. of the SIAM Symposium on Control Problems in Industry
San Diego, July 1994

Abstract

In this paper, we introduce automatic differentiation as a method for computing derivatives of large computer codes. After a brief discussion of methods of differentiating codes, we review automatic differentiation and introduce the ADIFOR automatic differentiation tool. We highlight some applications of ADIFOR to large industrial and scientific codes, and discuss the effectiveness and performance of our approach. Finally, we discuss sparsity in automatic differentiation and introduce the SparsLinC library.

1 Introduction

The computation of derivatives plays an essential role in many numerical methods, such as sensitivity analysis, inverse problems, data assimilation, and the emerging field of multidisciplinary design optimization. Typically, one has a model, say F, expressed as a computer code, with a vector-valued input $x \in \mathcal{R}^n$, and output $F(x)$ and one is interested in evaluating $\frac{\partial F(x)}{\partial x}$.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D and Cooperative Agreement No. NCCW-0027, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, bischof@mcs.anl.gov, khademi@mcs.anl.gov, pusch@mcs.anl.gov.

‡Center for Research on Parallel Computation, Rice University, 6100 S. Main St., Houston, TX 77251, carle@cs.rice.edu.

We have identified three basic issues that arise in the computation of derivatives, which can be viewed as the criteria for comparing the relative effectiveness of various methods of differentiation:

- **Compute Time:** the runtime of the derivative code;
- **Reliability:** the correctness and numerical accuracy of the derivative results; and
- **Development Time:** the time it takes to design, implement, and verify the derivative code, beyond the time to implement the code for the computation of the underlying function.

There are four main approaches to computing derivatives:

By Hand: One can differentiate the code for F by hand and thus arrive at a code that also computes the derivatives. Hand-coding of derivatives for a large code is a tedious and error-prone process, in particular as "real" codes are often not well documented. In fact, the effort can take months or years, and in some cases may even be considered prohibitive [6]. However, depending on the skill of the implementer, hand-coding may lead to the most efficient code possible.

Divided Differences: One can approximate the derivative of F with respect to the ith component of x at a particular point x_0 by differencing, for example by a one-sided difference,

$$\frac{\partial F(x)}{\partial x_i}\Big|_{x=x_0} \approx \frac{F(x_0 \pm h * e_i) - F(x_0)}{\pm h}, \tag{1}$$

where e_i is the ith Cartesian basis vector. This approach leads to an approximation of the desired derivatives and has the advantage of having a minimal development time, since all that is needed for the implementation of (1) is the "black box" application of F. However, the accuracy of divided differences is hard to assess, and numerical errors tend to grow with problem complexity (see, e.g., [17]). Further, the computational complexity of the method has a lower bound of n times the time to compute F. These factors make divided differences impractical for the computation of large derivative matrices and gradients.

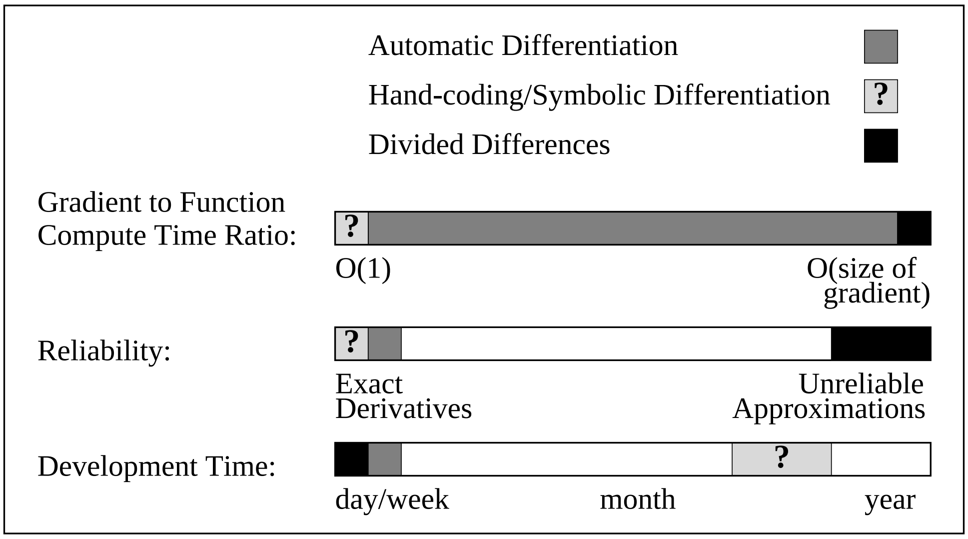

Figure 1: Comparing Differentiation Methods by Various Criteria

Symbolic Differentiation: Symbolic manipulators like Maple, Macsyma, or Reduce provide powerful capabilities for manipulating algebraic expressions but are, in general, unable to deal with constructs such as branches, loops, or subroutines that are inherent in computer codes. Therefore, differentiation using a symbolic manipulator still requires considerable human effort to break down an existing computer code into pieces digestible by a symbolic manipulator and to reassemble the resulting pieces into a usable derivative code.

Automatic Differentiation: Automatic differentiation techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and cos (see, for example, [19, 25]). By repeated application of the chain rule of derivative calculus to the composition of those elementary operations, one can compute, in a completely mechanical fashion, derivatives of F that are correct up to machine precision [22]. The techniques of automatic differentiation are directly applicable to computer programs of arbitrary length containing branches, loops, and subroutines.

Figure 1 shows a schematic comparison of the methods of differentiation along the previously mentioned criteria for the case of computing the gradient of a scalar-valued function. Note that we have grouped symbolic differentiation with hand-coding, as the postprocessing manipulation performed on codes generated by symbolic differentiators often amounts to nontrivial hand-coding. Effectively, this makes symbolic differentiation very similar to hand-coding in terms of the development time and reliability criteria. Note also that in Figure 1 we have labeled the shading representing these two methods with a "?" to emphasize that both correctness and efficiency are contingent upon the code designer's skill and not guaranteed by virtue of the methodology.

We have expressed compute time as a ratio of gradient to function runtimes. Provided memory constraints are not exceeded, a hand-coded gradient can be computed in a constant multiple of the function runtime [20], whereas a straightforward implementation of divided differences would have a linear dependency on n. In contrast to these, there is a large range for runtimes of derivative codes generated by automatic differentiation. This variance is due to a number of factors which will be discussed in the ensuing sections.

In summary, automatic differentiation addresses the need for computing derivatives of large codes accurately, irrespective of the complexity of the model. In fact, the intent behind the title of this paper is to convey that, based on the three criteria identified in Figure 1, automatic differentiation is often the best-of-all-worlds solution to the problem of computing derivatives. In cases where derivatives are infeasible or too expensive to code by hand, automatic differentiation is the most viable alternative, since both the numerical reliability of its results and its runtime efficiency surpass those of divided differences.

In the next section, we review the forward and reverse modes of automatic differentiation. In Section 3, we briefly describe the ADIFOR tool for automatic differentiation of Fortran 77 programs, and provide a brief account of recent experiences with ADIFOR applications. Section 4 contains a discussion of sparsity in this context and an introduction to SparsLinC, a library for the exploitation of sparsity in automatic differentiation. Lastly, we summarize our discussion.

2 Automatic Differentiation

Traditionally, two approaches to automatic differentiation have been developed: the so-called forward and reverse modes. These modes are distinguished by how the chain rule is used to propagate derivatives through the computation. In either case, automatic differentiation produces code that computes the values of the derivatives accurate to machine precision. Here, we discuss briefly issues impacting the computational complexity of each mode, and refer the reader to [3, 12] for a detailed treatment of both these modes.

The Forward Mode: The forward mode accumulates the derivatives of intermediate variables with respect to the independent variables, corresponding to the forward sensitivity formalism [14, 15]. Here, derivatives are computed much in the way that the chain rule of differential calculus is usually taught.

Let us consider a code with variables `x`, an array of size n and `y`, an array of size m, and say we are interested in computing the Jacobian $\frac{\partial y}{\partial x}\Big|_{x=x_0}$ (i.e., `x` contains the inputs, and `y` the outputs). Let us also introduce the notation ∇`s` to denote the derivative object associated with the program variable `s`. The forward mode generates a derivative code that essentially mirrors the control structure and flow of the original code, and augments it with additional statements derived from the application of the chain rule to each assignment or expression.

For example, the short code segment

```
do i = 1, n
    y(1) = 2*x(i) + 5
    y(2) = x(i)*y(1)
enddo
```

could be augmented as follows in the forward mode.

```
do i = 1, n
    ∇y(1) = 2*∇x(i)
    y(1) = 2*x(i) + 5
    ∇y(2) = x(i)*∇y(1) + y(1)*∇x(i)
    y(2) = x(i)*y(1)
enddo
```

One can easily convince oneself that by initializing ∇`x(i)` to the i-th canonical unit vector of length n, on exit each ∇`y(i)` contains the gradient $\frac{\partial y(i)}{\partial x(1:n)}$.

Forward-mode code is easy to generate, for the most part preserves any parallelizable or vectorizable structures within the original code, and is readily generalized to higher-order derivatives [7] (in this paper, however, our discussions are restricted to first-order derivatives). If we wish to compute n directional derivatives, then running forward-mode code requires at most on the order of n times as much time and memory as the original code.

The Reverse Mode: In contrast to the forward mode, the reverse mode propagates adjoints, that is, the derivatives of the final values with respect to intermediate variables, corresponding to the adjoint sensitivity formalism [14, 15]. To propagate adjoints, we have to be able to reverse the flow of the program and remember or recompute any intermediate value that nonlinearly impacts the final result.

The reverse mode is difficult to implement owing to memory requirements. In extreme cases, a reverse-mode implementation can require memory proportional to the number of floating-point operations executed during the run of the original program for the tracing of intermediate values and branches. However, the derivative runtime is roughly m times that of the function when computing m linear combinations of the rows of the Jacobian. This is particularly advantageous for gradients, since then $m = 1$. Hence, in the case of gradient computations, the reverse mode provides a lower bound on runtime complexity.

3 The ADIFOR (Automatic DIfferentiation of FORtran) Tool

There have been various implementations of automatic differentiation, an extensive survey of which can be found in [24]. In this section, we briefly introduce the ADIFOR tool and highlight three applications.

A "source transformation" approach to automatic differentiation has been explored in the ADIFOR [3, 5], ADIC [10], and Odyssee [26, 27] tools. ADIFOR and Odyssee transform Fortran 77 code and ADIC transforms ANSI-C code. By applying the rules of automatic differentiation, these tools generate new code that, when executed, computes derivatives without the overhead associated with trace interpretation schemes. ADIFOR and ADIC mainly use the forward mode. In contrast, Odyssee employs the reverse mode.

Given a Fortran subroutine (or collection of subroutines) describing a "function," and an indication which variables in parameter lists or common blocks correspond to "independent" and "dependent" variables with

```
r$1 = x(1) * x(2)
r$2 = r$1 * x(3)
r$3 = r$2 * x(4)
r$4 = x(5) * x(4)
r$5 = r$4 * x(3)
r$1bar = r$5 * x(2)
r$2bar = r$5 * x(1)
r$3bar = r$4 * r$1
r$4bar = x(5) * r$2
```

Reverse Mode for computing $\frac{\partial \mathtt{y}}{\partial \mathtt{x(i)}}$:

$$\mathtt{r\$jbar} = \frac{\partial \mathtt{y}}{\partial \mathtt{x(i)}},\ i = 1, \ldots, 4$$

$$\mathtt{r\$3} = \frac{\partial \mathtt{y}}{\partial \mathtt{x(5)}}$$

```
do g$i$ = 1, g$p$
   g$y(g$i$) = r$1bar * g$x(g$i$,1)
             + r$2bar * g$x(g$i$,2)
             + r$3bar * g$x(g$i$,3)
             + r$4bar * g$x(g$i$,4)
             + r$3 * g$x(g$i$, 5)
enddo
```

Forward Mode:
Assembling ∇y from $\frac{\partial \mathtt{y}}{\partial \mathtt{x(i)}}$ and $\nabla x(i)$, $i = 1, \ldots, 5$.

```
y = r$3 * x(5)
```

Computing function value

Figure 2: Sample Segment of an ADIFOR-generated Code

respect to differentiation, ADIFOR analyzes the program to determine which statements in the code have to be augmented with derivative computations, and then produces Fortran 77 code that computes the derivatives of the dependent variables with respect to the independent ones. ADIFOR produces portable Fortran 77 code and accepts almost all of Fortran 77; in particular, it can deal with arbitrary calling sequences, nested subroutines, common blocks, and equivalences. The ADIFOR-generated code tries to preserve vectorization and parallelism in the original code, and employs a consistent subroutine-naming scheme which allows for code tuning, the exploitation of domain-specific knowledge, and the use of vendor-supplied libraries.

ADIFOR employs a hybrid forward-/reverse-mode approach to generating derivatives. For each assignment statement, it uses the reverse mode to generate code that computes the partial derivatives of the result with respect to the variables on the right-hand side, and then employs the forward mode to propagate overall derivatives. For example, the single Fortran statement

$$\mathtt{y} = \mathtt{x(1)} * \mathtt{x(2)} * \mathtt{x(3)} * \mathtt{x(4)} * \mathtt{x(5)}$$

gets transformed into the code segment shown in Figure 2. Note that none of the common subexpressions $x(i) * x(j)$ are recomputed in the reverse mode section for $\frac{\partial \texttt{y}}{\partial \texttt{x(i)}}$.

The variable gp denotes the number of (directional) derivatives being computed. For example, if gp $= 5$, and g$x(1:5,1:5) is initialized to equal $\frac{\partial x(i)}{\partial x(j)}$ (which is a 5×5 identity matrix), then upon execution of these statements, g$y(1:5) equals $\frac{dy}{dx}$. On the other hand, assume that we wished only to compute derivatives with respect to a scalar parameter s, so gp $= 1$, and, on entry to this code segment, g$x(1,i) $= \frac{\partial x(i)}{\partial s}$, $i = 1, \ldots, 5$. Then the do-loop in Figure 2 implicitly computes $\frac{dy}{ds} = \frac{dy}{dx}\frac{dx}{ds}$ without ever forming $\frac{\partial y}{\partial x}$ explicitly. Note that the cost of computing y is amortized over all the derivatives being computed, and hence the ADIFOR approach is more efficient than the normal forward mode or a divided-difference approximation when more than a few derivatives are computed at the same time.

We see that ADIFOR-generated code provides a directional derivative computation capability [8]: Instead of simply producing code to compute the Jacobian J, ADIFOR produces code to compute $J * S$, where the "seed matrix" S is initialized by the user. Hence, if S is the identity, ADIFOR computes the full Jacobian; whereas if S is just a vector, ADIFOR computes the product of the Jacobian by a vector.

The running time and storage requirements of the ADIFOR-generated code are proportional to the number of columns of S, which equals the gp variable in the sample code above. However, ADIFOR-generated code typically runs two to four times faster than one-sided divided-differences approximation when one computes more than 5–10 derivatives at one time. This is due to the reverse/forward hybrid mode, and also the dependence analysis that tries to avoid computing derivatives of expressions that do not affect the dependent variables. We also note that in order to take full advantage of reduced complexity of ADIFOR-generated code, it is advantageous to compute several directional derivatives at the same time—so the ADIFOR-generated code may require significantly more memory than the original simulation code.

ADIFOR has been successfully applied to codes from various domains of science and engineering, an extensive list of which can be found in [5]. We highlight three of them here.

Groundwater Transport Models: In order to evaluate the accuracy and runtime performance of ADIFOR-generated derivative codes in comparison with divided differences and hand-coded derivatives, a comparative study was done on two groundwater codes developed at Cornell University: ISOQUAD, a two-dimensional finite-element model of groundwater transient flow and transport, and TLS3D, a three-dimensional advection/diffusion model [11]. Each code was over 2,000 lines long. The hand-derived derivative code of ISOQUAD took several months to develop; no hand-coded derivative of TLS3D was available for comparison.

In the case of ISOQUAD, ADIFOR-generated code produced derivatives that agreed with the validated handwritten code to the order of the machine precision, but executed in much less time than the (imprecise) divided-differences method. In particular, for a version of the problem with 126 independent variables, the ratio of runtime of the divided differences to the runtime of the original function was 126; for the ADIFOR-generated code, the ratio was 17; for the hand-coded code, it was 5.

Likewise, in the case of TLS3D, ADIFOR-generated derivatives took 5 to 7 times less time to compute than divided differences. The speedy construction of derivative-computing codes through automatic differentiation was considered significant because this would greatly accelerate the transfer of general techniques developed for using water resource computer models such as optimal design, sensitivity analysis, and inverse modeling problems to field problems.

CFD Airfoil Design: A joint project with NASA Langley Research Center involved the augmentation of coupled codes with sensitivity derivatives. Automated multidisciplinary design of aircraft requires the optimization of complex performance objectives with respect to a number of design parameters and constraints. The effect of these independent design variables on the system performance criteria can be quantified in terms of sensitivity derivatives of the individual discipline simulation codes. The NASA Langley design involves the coupling of the CFD code TLNS3D (a 3-D thin-layer Navier-Stokes code with a multigrid solver) with the WTCO grid generator.

Neither of these codes provide hand-coded derivatives—the code being deemed too complicated to differentiate by hand—and divided differences were shown to be numerically inaccurate, despite several attempts with different perturbation sizes; hence ADIFOR was used to generate the desired sensitivities [16]. In the case of the iterative solver, a post-ADIFOR modification to the derivative code was needed in order for the stopping criterion in the sensitivity code to monitor not only function convergence, but also sensitivity convergence [21]. Sensitivities computed by ADIFOR were validated, thus showing the effectiveness of automatic differentiation in computing derivatives of iterative solvers.

Sensitivity-Enhanced MM5 Mesoscale Weather Model: The Fifth-Generation PSU/NCAR mesoscale weather model (MM5) [18] incorporates most processes known to be relevant in meteorology. We are working on the development of a sensitivity-enhanced version of the code, which may be used to investigate, for example, the sensitivity of model behavior with respect to sensor placement, data coverage, or model resolution. The ability to compute derivatives then allows us to develop a linear approximation to the model (typically referred to by the weather community as the tangent linear model, or TLM) and to use this as a predictor of change.

ADIFOR expects code that complies with the Fortran 77 standard. MM5 does not comply with this standard; in particular, it makes much use of "pointer variables." We circumvented this difficulty by developing an MM5-specific tool to map the pointer handling to standard-conforming Fortran77 code acceptable to ADIFOR, and to remap ADIFOR's output to obtain the desired sensitivity-enhanced code [9].

Given the size and complexity of the code, automatic differentiation is the only viable approach for doing this sensitivity study. Our work has demonstrated that automatic differentiation can generate results equivalent to a tangent linear model for sophisticated weather models, with minimal recourse to laborious and error-prone hand-coding. We have compared the derivatives computed by ADIFOR-generated code with those computed by divided differences and find good agreement everywhere except in intense thunderstorm regions. This result was expected, since storms involve strong nonlinear effects.

4 Exploiting Sparsity in Automatic Differentiation

Computationally, the most expensive kernel of derivative codes generated by forward-mode approaches is the linear combination of vectors operation (for an ADIFOR example, note the do-loop implementation of this operation in Figure 2). We can define the operation as follows:

$$w = \sum_{i=1}^{k} \alpha_i v_i, \tag{2}$$

where w and the v_i are gradient vectors, the α_i are the scalar multipliers, and k is the arity. The length of the vectors, which we denote as p, is equal to the number of directional derivatives being computed. In Figure 2, gp is the runtime value of p.

For problems where the gradient vectors in the above operation are known to be mostly sparse, approaches that exploit sparsity can dramatically reduce the runtime and memory requirements for the derivative computation. Two classes of such problems that arise in large-scale optimization are computing sparse Jacobians and computing gradients of partially separable functions. Sparse Jacobians, as the name suggests, occur where many of the dependent variables are expected to have a zero dependency on the independent variables. Partially separable functions [23] can be represented in the form

$$f(x) = \sum_{i=1}^{np} f_i(x), \tag{3}$$

where each of the component functions f_i has limited support. This implies that the gradients ∇f_i are sparse even though the final gradient ∇f is dense. It can be shown [23] that any function with a sparse Hessian is partially separable.

One approach for exploiting sparsity is the "compressed Jacobian" approach. Given the sparsity pattern of the Jacobian, this approach derives a graph coloring that identifies which columns of the Jacobian can be computed with the same directional derivative. The full Jacobian is then mapped onto a corresponding compressed Jacobian (in ADIFOR this mapping is implemented via the initialization of the seed matrix). This effective reduction of p results in reduced runtimes and memory requirements [2]. Given some code rewriting (which in some cases can be nontrivial), this approach is also applicable to the computation of gradients of partially separable functions [13].

An alternative approach exploits sparsity in a transparent fashion, that is, without the a priori knowledge of the sparsity pattern of the Jacobian required for graph coloring. If the initial seed matrix is sparse (e.g., the identity), then if one ignores exact numerical cancellation, the left-hand side vector w in (2) has no fewer nonzeros than any of the vectors on the right-hand side. Hence, if the final derivative objects, which correspond to a row of the Jacobian J or a component gradient ∇f_i, are sparse, all intermediate vectors must be sparse. That is, by employing algorithms and data structures tailored for sparsity, sparsity in derivative calculations can be exploited transparently. Note that the sparsity structure of J or ∇f_i is computed as a by-product of the derivative computation.

The SparsLinC (Sparse Linear Combination) library [4, 12] addresses the case in which p is large, and most of the vectors involved in vector linear combination are sparse. It provides support for sparse vector linear combination in a fashion that is well suited to the use of this operation in the context of automatic differentiation. SparsLinC, which is written in ANSI C, encompasses the following:

Three Data Structures for Sparse Vectors: SparsLinC has different data structures for a vector containing only one nonzero, a few nonzeros, or several nonzeros.

Efficient Memory Allocation Scheme: SparsLinC employs a "bucket" memory allocation scheme which (in effect) provides a buffered memory allocation mechanism.

Polyalgorithm: SparsLinC adapts to the dynamic growth of the derivative vectors by transparently switching between the three vector representations, thus efficiently representing vectors that grow from a column of the identity matrix (often occurring in the seed matrix) to a dense vector. Also, special support is provided for the "+=" operation, $w = \alpha_1 * w + \alpha_2 * v$, which occurs frequently when computing gradients of partially separable functions, as suggested by (3).

Full-Precision Support: single- and double-precision computations are provided for both real- and complex-valued computations.

Figure 3 shows plots contrasting results obtained from ADIFOR-generated code both with and without SparsLinC support (the plots labeled as "Sparse" are from the runs with SparsLinC support). The problems, DGL2 (2-D Ginsburg-Landau model for homogeneous superconductors) and DMSA (mininal surface area problem), are taken from the MINPACK-2 test problem collection [1] of large-scale optimization problems. Both problems are partially separable functions. We are interested in computing gradients for a range of grid sizes. Note that in Figure 3, which is a square "log-log" plot, the nonsparse gradients exhibit linear behavior with respect to p. Notice also that in the SparsLinC-supported

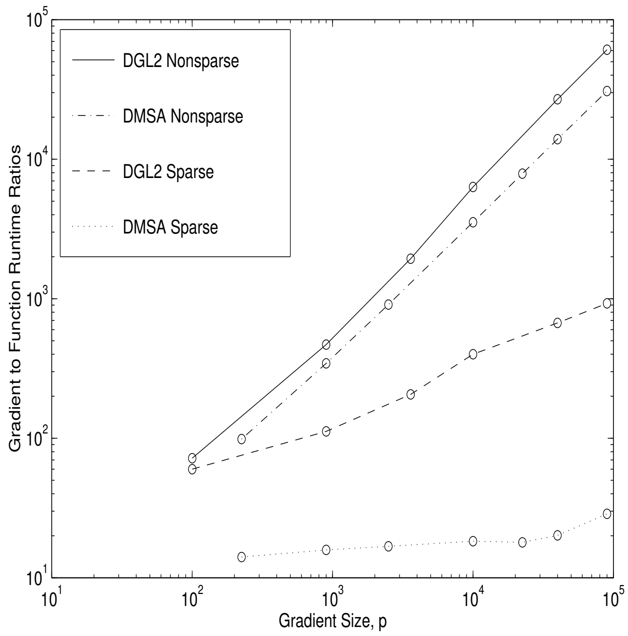

Figure 3: Sparse vs Non-Sparse Gradient to Function Runtime Ratios

computations the complexity of derivative computation is clearly less than linear, particularly in the case of DMSA where, for a segment of gradient sizes, the gradient computation runtime appears as a constant times the function runtime.

5 Conclusions

In this paper, we gave a brief introduction to automatic differentiation. We outlined the criteria for comparing methods of differentiating codes and showed that in many cases automatic differentiation is the method of choice. We reviewed the forward and reverse modes of automatic differentiation and discussed the ADIFOR automatic differentiation tool. We presented three examples where the application of ADIFOR for generating derivative codes has significant scientific and/or engineering impact. Finally, we reviewed exploitation of sparsity in automatic differentiation and briefly introduced the SparsLinC library in this connection.

At the time of this writing, we anticipate the forthcoming release of ADIFOR 2.0 [5], our newest version of ADIFOR in which the SparsLinC library is fully integrated.

References

[1] B. M. Averick, R. G. Carter, J. J. Moré, and G. L. Xue. The MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-150 (revised), Mathematics and Computer Science Division, Argonne National Laboratory, June 1992.

[2] 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.

[3] 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.

[4] Christian Bischof, Alan Carle, and Peyvand Khademi. Fortran 77 interface specification to the SparsLinC library. Technical Report ANL/MCS-TM-196, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[5] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew 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.

[6] 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.

[7] Christian Bischof, George Corliss, and Andreas Griewank. Computing second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software*, 2:211–232, 1993.

[8] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS-TM-158,

Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[9] Christian Bischof, Peyvand Khademi, and Timothy Knauff. ADIFOR strategies related to pointer usage in MM5. Technical Report ANL/MCS-TM-187, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[10] Christian Bischof and Andrew Mauer. Unpublished information. Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[11] 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.

[12] Christian H. Bischof. Automatic differentiation, tangent linear models and pseudo-adjoints. Preprint MCS-P472-1094, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[13] Christian H. Bischof and Moe El-Khadiri. Extending compile-time reverse mode and exploiting partial separability in ADIFOR. Technical Report ANL/MCS-TM-163, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[14] D. G. Cacuci. Sensitivity theory for nonlinear systems, I: Nonlinear functional analysis approach. *Journal of Mathematical Physics*, 22(12):2794–2802, 1981.

[15] D. G. Cacuci. Sensitivity theory for nonlinear systems, II: Extension to additional classes of responses. *Journal of Mathematical Physics*, 22(12):2803–2812, 1981.

[16] 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.

[17] Phillip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, London, 1981.

[18] G. A. Grell, J. Dudhia, and D. R. Stauffer. A description of the fifth-generation Penn State/NCAR mesoscale weather model (MM5). Technical Report NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado, June 1994.

[19] Andreas Griewank. The chain rule revisited in scientific computing. Preprint MCS-P27-0491, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

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

[21] 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.

[22] 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, 1991.

[23] 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, London, 1981. Academic Press.

[24] 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.

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

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

[27] Nicole Rostaing-Schmidt and Eric Hassold. Basic functional representation of programs for automatic differentiation in the Odyssee system.

In Francois-Xavier Le Dimet, editor, *High-Performance Computing in the Geosciences*, Dordrecht, 1994. Kluwer Academic Publishers.