ON THE AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS AND AN APPLICATION TO MULTIBODY SYSTEMS

CHRISTIAN H. BISCHOF
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL 60439
bischof@mcs.anl.gov

Abstract. Automatic differentiation (AD) is a methodology for developing sensitivity-enhanced versions of arbitrary computer programs. In this paper, we provide some background information on AD and address some frequently asked questions. We introduce the ADIFOR and ADIC tools for the automatic differentiation of Fortran 77 and ANSI-C programs, respectively, and give an example of applying ADIFOR in the context of the optimization of multibody systems.

1. Introduction

Assume that we have a code for the computation of a function f and $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}^m$, and we wish to compute the derivatives of y with respect to x. We call x the *independent variable* and y the *dependent variable.*

In computing derivatives, we should keep the following issues in mind:

Reliability: The computed derivatives should ideally be accurate to machine precision.

Computational Cost: In many applications, the computation of derivatives is the dominant computational burden. Hence, the amount of memory and runtime required for the derivative code should be minimized.

Scalability: Whatever method we choose should be applicable to a 1-line formula as well as a 50,000-line code.

Human Effort: Derivatives are a means to an end. Hence a user should not spend much time in computing derivatives, in particular in situations where computer models are bound to change frequently.

Handcoding, divided-difference approximations, and symbolic methods traditionally have been used for the computation of derivatives. However, these methods fall short with respect to the previously mentioned criteria. The main drawbacks of divided-difference approximations are their numerical unpredictability and their computational cost. In contrast, both the handcoding and symbolic approaches suffer from a lack of scalability and require considerable human effort.

In this paper, we discuss another approach for computing derivatives, based on automatic differentiation (AD). AD 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, [10, 16]. By applying the chain rule of derivative calculus over and over again 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 [12].

In the next section, we give a brief overview of automatic differentiation. Section 3 introduces the ADIFOR and ADIC AD tools for Fortran 77 and ANSI-C, respectively, and Section 4 answers some commonly asked questions. In Section 5, we report on the application of ADIFOR in the context of the optimization of a multibody system. Lastly, we summarize our results.

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. We briefly summarize the main points about these two approaches; a more detailed description can be found in [4] and the references therein.

The forward mode propagates derivatives of intermediate variables with respect to the independent variables and follows the control flow of the original program. By exploiting the linearity of differentiation, the forward mode allows us to compute arbitrary linear combinations $J \cdot S$ of columns

of the Jacobian

$$J = \begin{pmatrix} \frac{\partial y(1)}{\partial x(1)} & \cdots & \frac{\partial y(1)}{\partial x(n)} \\ \vdots & & \vdots \\ \frac{\partial y(m)}{\partial x(1)} & \cdots & \frac{\partial y(m)}{\partial x(n)} \end{pmatrix}. \tag{1}$$

For an $n \times p$ matrix S, the effort required is roughly $O(p)$ times the runtime and memory of the original program. In particular, when S is a vector s, we compute the directional derivative $J * s = \lim_{h \to 0} \frac{f(x+h*s)-f(x)}{h}$.

In contrast, the reverse mode of automatic differentiation propagates derivatives of the final result with respect to an intermediate quantity, so-called adjoint quantities. To propagate adjoints, one must be able to reverse the flow of the program, and remember or recompute any intermediate value that nonlinearly affects the final result. In particular, one must remember the intermediate values taken by variables that are overwritten, and keep a log of the branch directions taken. Also, changing a "+" to a "*" in the computer code can have profound ramifications for the complexity of the generated reverse mode code, while it does not have much effect for the forward mode.

For a $q \times m$ matrix W, the reverse mode allows us to compute the row linear combination $W \cdot J$ with $O(q)$ times as many floating-point operations as required for the evaluation of f. In a straightforward implementation, however, the storage requirements may be proportional to the number of floating-point operations required for the evaluation of f, as a result of the tracing required to make the program "reversible." When W is a row vector w, we compute the derivative $\frac{\partial (w^T * J)}{\partial x}$. The reverse mode is particularly attractive for the computation of long gradients, as its floating-point complexity does not depend on the number of independent variables.

In either case, automatic differentiation produces code that computes derivatives accurate to machine precision [12]. The techniques of automatic differentiation are directly applicable to computer programs of arbitrary length containing branches, loops, and subroutines.

3. Automatic Differentiation Tools

We are involved in the development of the ADIFOR (jointly with Rice University) and ADIC tools, which provide automatic differentiation functionality for Fortran 77 and ANSI-C, respectively, The ADIFOR 2.0 system is mature, and reference [4] lists 25 references reporting on the use of ADIFOR in various application domains, on codes of up to 60,000 lines. ADIC, in contrast, is in the prototype phase, but has been successfully applied to codes of up to 10,000 lines. ADIFOR and ADIC employ a source transformation approach directly rewriting the source code. This approach requires

```
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 1. Sample Segment of an ADIFOR-generated Code

considerable compiler infrastructure, and ADIFOR and ADIC employ the ParaScope [8] and Sage++ [7] compiler environments developed at Rice and Indiana University, respectively. For references to other automatic differentiation tools, see [4].

ADIFOR and ADIC employ a hybrid forward/reverse-mode approach to generating derivatives. For each assignment statement, they use 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 employ the forward mode to propagate overall derivatives. For example, ADIFOR transforms the Fortran statement

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

into the code segment shown in Figure 1.[1] Note that none of the common subexpressions $x(i) * x(j)$ are recomputed in the reverse-mode section for $\frac{\partial \mathtt{y}}{\partial \mathtt{x(i)}}$. The variable `g$p$` denotes the number of (directional) derivatives being computed. For example, if `g$p$` = 5, and `g$x(1:5,1:5)` is the 5×5 identity matrix (i.e., `g$x(i,j)` = $\frac{\partial x(i)}{\partial x(j)}$), 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 `g$p$` = 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 1 implicitly computes $\frac{dy}{ds} = \frac{dy}{dx}\frac{dx}{ds}$ without ever forming $\frac{\partial y}{\partial x}$ explicitly.

[1] The dollar sign indicates ADIFOR-generated variables. ADIFOR 2.0 could use any other character instead, taking care not to generate duplicate names.

ADIFOR and ADIC provide the directional derivative computation possibilities associated with the forward mode of automatic differentiation. We also mention that both ADIFOR and ADIC can transparently exploit sparsity in derivative computations by replacing the dense vector loop in Figure 1 with a call to a SparsLinC routine [4, 5], which, as a byproduct of the computation, will automatically compute the sparsity pattern of large sparse Jacobians.

None of these AD tools require any knowledge of the application domain. Hence, unlike handcoding or symbolically assisted approaches, automatic differentiation enables derivatives to be updated easily when the original code changes. Information on these tools as well as application highlights and reports can be found on the world-wide web at

`http://www.mcs.anl.gov/autodiff/index.html`.

4. Frequently Asked Questions

Given the mathematical underpinnngs of the concept of derivatives, the "ignorance" with which one can apply an AD tool usually provokes some of the questions that we try briefly to address here.

Question: How do you know that the code represents a globally differentiable function?

Answer: We don't. AD computes the derivative defined by the sequence of assignment statements executed in the course of a function evaluation. Hence, for a branch (if-statement), which potentially introduces a nondifferentiability, AD will compute a one-sided directional derivative. This problem is further discussed in [9].

Question: How do you deal with intrinsics?

Answer: Some intrinsics functions, such as `abs()` and `sqrt()`, are not differentiable in all points of their domain. At these points, ADIFOR invokes the ADIntrinsics system [4] to provide a (user customizable) default value, and prints a warning message. The ADIC prototype uses a similar, although less refined, mechanism.

Question: What happens when you differentiate through iterative processes?

Answer: It depends. AD generates a new iteration, and it is not clear a priori whether the new iteration will converge and what it will converge to, although empirically, AD leads to the desired result. However, derivative convergence may lag, or derivatives may diverge. For some commonly used approaches for solving nonlinear systems of equations, this issue is discussed in [11]. This problem clearly requires more re-

search, but the emergence of robust AD tools has made it possible to tackle this problem for sophisticated numerical methods.

5. An Example: The Iltis All-Terrain Vehicle

The dynamic and kinetic behavior of vehicles can be modeled through multibody systems. Optimization techniques can then be employed to improve the design of such a vehicle with respect to comfort, ride, and handling. For an overview of this field as well as the methods employed, see [1].

In general, the motion of a multibody system can be described as follows:

$$\left.\begin{array}{rcl} \dot{y} & = & v(t,y,z,p) \\ M(t,y,p)\,\dot{z} + k(t,y,z,p) & = & q(t,y,z,p) \end{array}\right\}, \tag{2}$$

where $\dot{y} = \frac{\partial y}{\partial t}$ is the derivative with respect to the time t, M is the mass matrix, k are the coriolis forces, q the external forces, y generalized position coordinates, z generalized velocity coordinates, and p the design parameters.

An efficient method for optimizing a multibody system is the adjoint variable method developed by Bestle and Eberhard [2], which requires the derivatives $\frac{\partial M_{mn}}{\partial t}$, $\frac{\partial M_{mn}}{\partial y_i}$, $\frac{\partial M_{mn}}{\partial p_k}$, $\frac{\partial (k_m - q_m)}{\partial y_i}$, $\frac{\partial (k_m - q_m)}{\partial z_j}$, and $\frac{\partial (k_m - q_m)}{\partial p_k}$. In [13], Häußermann applied the first version of ADIFOR [3] to several multibody systems and compared it with symbolic approaches and with approximations of derivatives via divided differences.

However, application of ADIFOR 1.0 to the so-called Iltis problem, a benchmark problem modeling an all-terrain vehicle [15], proved to be somewhat laborious. ADIFOR 1.0 was unable to process the subroutine of several thousand lines describing the equations of motion that had been generated with the NEWEUL [14] package. The problem had to be split by hand, a somewhat laborious and error-prone process.

With the new ADIFOR 2.0 system, however, one can now process the code as is. Differentiating with respect to 20 parameters, one obtains the results shown in Table 1. Computations were performed on a Silicon Graphics Indigo with 32 MB RAM and a 100 Mhz MIPS R4000 microprocessor. Here "Iltis" refers to the original code, and "Iltis.AD" refers to the code generated by ADIFOR 2.0. We see that the memory required by the ADIFOR-generated code increases by a factor of 6.7, whereas runtime increases by a factor 20, the same cost increase one would also experience with divided-difference approximations. In most cases, however, ADIFOR-generated code outperforms one-sided divided-difference approximations, typically by a factor 1.5 to 3, and by a factor of 7.4 in the best case so far [6]. Code expansion is considerable because of the somewhat unusual

TABLE 1. Results of Applying ADIFOR 2.0 to the Iltis Problem

	Iltis.AD	Iltis	Ratio
Memory (MB)	3.52	0.52	6.7
Runtime (sec)	42.6	2.13	20.0
Lines of code	71,887	11,172	6.4

structure of the NEWEUL-generated code. The number of lines of code increases by a factor of 6.4, and the resulting length of the .AD versions of the NEWEUL-generated files prevented compilation on an HP workstation. In our experience, code expansion by a factor 2 to 3 is typical. The generated code accurately computes the desired derivatives, whereas the study by Häußerman shows that this is not necessarily the case for divided difference approximations.

6. Conclusions

This paper gave a brief introduction into automatic differentiation. We reviewed the forward and reverse mode of automatic differentiation, answered some commonly asked questions, and introduced the ADIFOR and ADIC automatic differentiation tools. We also presented results on applying ADIFOR 2.0 to the Iltis multibody benchmark problem, which showed that reliable and efficient derivatives can be computed by using AD with minimal recourse to laborious and error-prone hand coding.

Acknowledgments

We thank Peter Eberhard for providing us with the Iltis code and for performing the benchmark runs. We also thank Peter Eberhard and Dieter Bestle for introducing us to multibody system optimization. Lastly, we thank Ralf Knösel for processing the Iltis code with ADIFOR 2.0.

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Aerospace Agency under Purchase Order L25935D; and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

References

1. Dieter Bestle. *Analyse und Optimierung von Mehrkörpersystemen.* Springer, Berlin,

1994.

2. Dieter Bestle and Peter Eberhard. Analyzing and optimizing multibody systems. *Mechanical Structures and Machinery*, 20(1):67–92, 1992.
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, 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.
5. Christian Bischof and Andrew Mauer. ADIC – A tool for the automatic differentiation of C programs. Preprint MCS-P499-0295, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.
6. Christian Bischof, Greg Whiffen, Christine Shoemaker, Alan Carle, and Aaron Ross. Application of automatic differentiation to groundwater transport models. In Alexander Peters et al., editors, *Computational Methods in Water Resources X*, pages 173–182. Kluwer Academic Publishers, Dordrehct, 1994.
7. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Goutwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. SAGE++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*. IEEE, 1994.
8. D. Callahan, K. Cooper, R. T. Hood, K. Kennedy, and L. M. Torczon. ParaScope: A parallel programming environment. *International Journal of Supercomputer Applications*, 2(4):84–99, December 1988.
9. Herbert Fischer. Special problems in automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 43 – 50. SIAM, Philadelphia, Penn., 1991.
10. Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Amsterdam, 1989.
11. 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.
12. 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.
13. Uli Häußermann. Automatische Differentiation zur Rekursiven Bestimmung von Partiellen Ableitungen. STUD-102, Institut B für Mechanik, Universität Stuttgart, 1993.
14. E. Kreuzer and G. Leister. Programmsystem NEWEUL'90. Technical Report Anleitung AN-24, Institut B für Mechanik, Universität Stuttgart, 1991.
15. G. Leister and W. Schiehlen. Benchmark-beispiele des DFG-schwerpunktprogramms dynamic von mehrkörpersystemen. Technical Report Zwischenbericht ZB-64, Band 2, Institut B für Mechanik, Universität Stuttgart, 1991.
16. Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.