An Application for Semi-Automatic Differentiation (extended abstract)

John D. Hobby
Bell Laboratories—Lucent Technologies

2004

Abstract

A large software project involving the optimization of cellular phone systems required evaluating a complicated function and its derivatives. Stringent limitations on run-time and memory consumption appeared to preclude standard methods for automatic differentiation, so all the derivatives were hand-coded.

We discuss the time and memory costs for the hand-coded derivatives and compare this with the costs of automatic differentiation. We also consider the software development costs for hand-coded derivatives and investigate the problem of adding as much automation as possible subject to the constraint that overall run time and memory consumption is similar to that of the hand-coded derivatives. Initial efforts in this direction are making use of a C++ parser that preserves comments and white space.

1 Introduction

Due to increases in problem size and software complexity, it is still common for large applications to run into run-time and memory limitations. This paper is based on experiences with a large project of this nature involving the optimization of certain parameters so as to improve the performance of cellular phone systems. This is a real application of significant business importance.

The application involves a complicated function that models cellular phone system performance as a function of numerous parameters. To facilitate optimization via `snopt` [3, 4], there are derivatives with respect to four types of parameters. The total number of independent variables can range from a few dozen to more than 1000, depending on the problem size. Typical run times for a function evaluation (without derivatives) on a 1Ghz PC range from 5 seconds for a small problem instance to about 15 minutes for a very large one.

As explained in Section 2, long running times and large number of variables appeared to make existing techniques for automatic differentiation impractical, hence the derivatives were hand-coded. This decision allowed for large problem sizes but limited the number of types of parameters for differentiation due to the additional implementation and debugging effort required for new derivative functions. Section 3 explains that it also led to significant additional software development costs. As explained in Section 4, the offsetting benefit is that the function with hand-coded derivatives typically runs at most 3 times as long as the function without derivatives and uses little additional memory. In order to achieve this with reduced software development costs, Section 5 investigates partial automation.

2 Time and Memory Costs for Automatic Differentiation

Upgrading a function of n variables to compute the function and gradient vector clearly must increase run time and memory consumption by factors ≥ 1. For forward-mode automatic differentiation, both factors are often near $n+1$, which is prohibitively expensive for our application where n can reach 1000. The mixed forward-reverse mode operation of ADIC should be somewhat better if it can handle all the C++ features used in this application [1].

A purely reverse-mode system such as ADOL-C [6] gives a time factor that does not grow with n, but memory consumption is proportional to run time. The time factor could theoretically be close to the value 3 observed for the hand-coded derivatives, but overhead in ADOL-C's overloaded operators understandably makes this hard to achieve. However, Gay showed that a less-general approach can achieve this under some circumstances [2]. Unfortunately, it is impractical to store every elementary operation for a function that takes 15 minutes to evaluate, so the memory factor is prohibitive unless it can be controlled in some way.

3 Software Development Costs for Hand-Coded Derivatives

When doing hand-coded derivatives, it is often convenient to work on the function and the derivative routines simultaneously and put them in a common source file. Since this makes it hard to track development costs, we concentrate on a subsystem where this was not done.

This subsystem computes a few dozen numeric quantities and makes a fairly complex set of combinatorial decisions that are ultimately based on whether certain spline functions are nonzero for certain inputs. It involves transcendental functions, various computations based on the combinatorial decisions, nonlinear extrapolation, and numerical integration.

The initial version of this subsystem was about 4000 lines of code, and the non-derivative computations required 25 days of coding and 8 days of debugging. The derivatives were about 55% of the code and they required 15 days of coding and 35 days of debugging. Thus derivatives required 38% of the initial coding time but this rises to 60% when debugging is included. Derivative debugging required a separate test harness to determine whether derivative routines for intermediate results are consistent with finite difference computations.

4 Time and Memory Savings from Hand-Coded Derivatives

For hand-coded derivatives, the programmer knows a lot about the control flow of the function being differentiated and hence can get many of the advantages of the reverse-mode approach without recording an explicit execution trace. This leaves numeric quantities that do have to be propagated backwards—we refer to them as adjoints as in [2, 5].

Hand-coding the derivatives allowed a complex mixture of the forward- and reverse-mode approaches. In fact, there were several types of intermediate results for which it was advantageous to propagate derivatives with respect to those quantities in forward mode, but in no case was there ever any forward-mode propagation for derivatives with respect to the main input variables since there can be up to $n = 1000$ of them as explained above.

The hand-coded derivative routines also involved dramatic changes in control flow. Where the original function looped over large data structures, computing $O(n)$ quantities needed for subsequent computations, a vector v of derivatives of the final result with respect to those $O(n)$ quantities could only be computed after the loop. To avoid excessive storage for adjoint vectors, the entire loop was repeated, this time with derivative computations that depend on v. Furthermore, whenever the original function contained a loop that adds up a large number of quantities, the fact that addition is commutative and associative made it unnecessary to propagate adjoints in reverse mode.

Another benefit was that when the original function used iterative methods to solve a nonlinear system, the derivative routines could ignore the iteration and do implicit differentiation based on the system of equations being solved.

5 Doing as Much Automation as Possible

Is it possible to reduce the fraction of development time spent on derivatives to much less than 60% with run time and memory consumption close to that of hand-coded derivatives? The statistics from Section 3 suggest that a significant reduction could have been achieved if there were a tool that verifies hand-coded derivatives in such a way as to make it clear exactly where the bugs are. It was difficult to get such detailed information via finite difference tests, so perhaps a symbolic analysis tool is needed.

Another option is for the user to handle high-level decisions such as adding another pass over the data structures, and then use an automatic system for functions that are simple enough to allow for automatic differentiation with low time and memory costs. Inserting this user-supplied information via special comments allows the overall process to be administered with the aid of C++ parser that preserves comments and white space. This is being done via a proprietary C/C++ grammar for the SGLR parser [7]. The grammar was developed by D. Waddington as part of a research project on software transformation.

In order to be simple enough for efficient automatic differentiation, it is necessary to be able to use reverse mode with no memory. A function that calls no user-defined functions relevant to differentiation and has "if" statements but no loops should qualify, and it would probably not be too hard to handle some function calls and loops that are not order dependent. Such functions were observed to be common enough that it would help a lot to differentiate them automatically.

Under some circumstances, it may be necessary for the user to provide special routines as input for automatic differentiation. For instance, the hand-coded derivative routines for the subsystem described in Section 3 take the combinatorial decisions from the data structures produced by a previous call to the non-derivative version. Hence, what was differentiated was a special version that assumes all the combinatorial decisions have already been made.

In order to do any automation for the case where iterative methods are used to solve a nonlinear system of equations, the special routine provided by the user would probably be a computation of the residual, but this might not be sufficient.

6 Conclusion

We have analyzed a large application where stringent limitations on run time and memory consumption presented a difficult case for automatic differentiation. Some form of partial automation would probably have been useful, but a long series of project deadlines allowed no time to develop this. The semiautomatic methods discussed in Section 5 are not ready to use at the time of this writing, and they do not cope well with C++ templates and exceptions.

References

[1] Chirstian Bischof, Lucas Roh, and Andrew Mauer-Oats. ADIC: An extensible automatic differentiation tool for ANSI-C. *Software Practice and Experience*, 27(12):1427–1456, December 1997.

[2] D. M. Gay. Automatic differentiation of nonlinear AMPL models. In A. Griewank and G. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 61–73. SIAM, 1991.

[3] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Journal on Optimization*, 12:979–1006, 2002.

[4] Philip E. Gill, Walter Murray, and Michael A. Saunders. User's guide for SNOPT 6.1: A Fortran package for large-scale nonlinear programming. Technical Report NA 02–2, Dept. of Math., UC San Diego, 2002.

[5] Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, pages 1–78, 2003.

[6] Andreas Griewank, David Juedes, H. Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[7] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J.Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Compiler Construction 2001*, pages 365–370. Springer Verlag, 2001. LNCS 2027.