An Application for Semi-Automatic Differentiation

John D. Hobby

Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ, 07974, USA. hobby@bell-labs.com

Summary. A large software project involving the optimization of cellular phone systems required evaluating a complicated function and its derivatives. Due to stringent limitations on run-time and memory consumption, all the derivatives were hand-coded.

We discuss the time and memory costs for the hand-coded derivatives and compare them with the costs of automatic differentiation. We also consider the software development costs for hand-coded derivatives and suggest some approaches for 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. We also consider methods for partial automation while preserving the efficiency of hand-coded derivatives, and we discuss some preliminary efforts in this direction. There need to be more tools for partial automation so that applications such as this one can get at least some of the benefits of automatic differentiation.

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 where derivatives were hand-coded to achieve maximum efficiency. We argue that there need to be more tools for partial automation so that applications such as ours can get at least some of the benefits of automatic differentiation (henceforth referred to as AD).

The application involves a complicated function written in C++ that models cellular phone system performance as a function of numerous parameters. To facilitate optimization via `snopt` [4, 5], 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. This is a real application of significant business importance.

As explained in Section 2, long running times and large numbers of variables appeared to make existing techniques for AD 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. 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 4 investigates partial automation.

2 Time and Memory Comparison with Hand-Coded 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. The hand-coded derivatives had a run time overhead factor near 3.0 with a memory overhead factor that is at most 2.0 and often much less.

With purely forward-mode AD, both factors are near $n + 1$, which is prohibitively expensive for our application where the number of variables n can be 1000 or more.

A purely reverse-mode system such as ADOL-C's reverse mode [7] gives a time factor that does not grow with n, but memory consumption is proportional to run time. C++ operator overloading is very convenient but involves a substantial run-time overhead. Note that this overhead can be reduced significantly if we can parse the input program and generate new code [2]. However, 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.

Systems such as ADIC provide mixtures of the forward and reverse modes [1], but it is generally agreed that the efficiency of hand-coded derivatives is very hard to get with automatic tools. Giering and Kaminski claim that TAF can achieve this efficiency if used with enough care [3], but no equally-good C++ version is available at this time. The fact that the cell-phone application dates back to 1998 and is written in C++ made it especially difficult to find suitable automatic tools.

2.1 Reasons for Better Performance with Hand-Coded Derivatives

It takes a great deal of data to describe how well a cell-phone user can communicate at all possible locations, so the function to be differentiated involves vast numbers of intermediate results. There is also a great deal of combinatorial information to be computed, usually by computing spline functions and checking whether they are nonzero in certain places.

When a function to be differentiated computes both combinatorial data and numerical results, the derivative code can often assume that the combinatorial data has already been computed. Hence, what needs to be differentiated is a completely different function that uses the combinatorial data to recompute the numerical results.

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, 6].

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. In one case where the function involved dynamic programming, the adjoints for the intermediate results required nontrivial data structures, and in another case the adjoints could share storage with some of the intermediate results from the non-derivative computation.

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.

Many of the issues mentioned above have come up in the literature, and some of the automatic tools do allow the user to help in dealing efficiently with such problems, but it is certainly difficult to do this well.

3 Software Development Costs for Hand-Coded Derivatives

When hand-coding 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. As will be explained in Section 4.1, derivative debugging required a separate test harness to determine whether derivative routines for intermediate results are consistent with finite difference computations.

4 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? Surely, some degree of automation should be possible without giving up the efficiency of hand-coded derivatives.

We begin in Sections 4.1 and 4.2 with ideas for debugging tools since debugging occupies so much of the development time for hand-coded derivatives. Next, Section 4.3 discusses what properties a function has to have in order to be "simple" enough for certain methods of efficient AD, and it also explains how common such routines were. Actual techniques for partial automatic differentiation based on these simple functions appear in Section 4.4. Finally, Section 4.5 discusses the provision of special input to facilitate efficient automatic differentiation in otherwise difficult cases.

4.1 Debugging with Finite Differences

The statistics from Section 3 suggest that a significant reduction could have been achieved by a tool that merely simplifies debugging. Without such assistance, a developer of hand-coded derivative routines must build his own tools based on finite differences.

The most obvious way to use finite differences is the "black box" approach of choosing an argument vector v and evaluating the overall function $F(v)$ and its gradient $\nabla F(v)$ as well as $F(v-\bar{v}_i)$ and $F(v+\bar{v}_i)$ for various displacement vectors $\bar{v}_0, \bar{v}_1, \ldots, \bar{v}_k$. Since the cell phone application had 4 types of parameters appearing in v, this was done for with $k = 4$ and $\|\bar{v}_i\|_2 \approx 10^{-7} \|\bar{v}\|_2$ for each i. Typically,

$$2\bar{v}_i \cdot \nabla F(v) \approx F(v + \bar{v}_i) - F(v - \bar{v}_i)$$

to within about a few parts in 10^7, and the extent to which

$$F(v + \bar{v}_i) - F(v) \quad \text{and} \quad F(v) - F(v - \bar{v}_i)$$

agree gives some indication of the accuracy that can be expected. However, it is sometimes hard to know if inaccuracies are due to bugs in the hand-coded derivatives, and even when this is clear, the programmer can deduce little more than that there is some kind of bug somewhere an a 40000 line program.

To help localize bugs, the "black box" finite difference checker for the cell phone application also checked 10 types of intermediate results. Most of these were scaler quantities with derivatives with respect to the argument vector v, but two of them were vectors, each component of which depended on a few other intermediate results in a manner for which derivatives were available. This could often indicate which of a few major subsystems were responsible for a bug, but it required 1200 lines of debugging code that was nontrivial to set up and maintain, as well as additional data structures and code to compute quantities and derivatives that were only useful for debugging.

More fine-grained finite-difference tests required a special test harness capable of taking subroutines meant to exist as part of a large program, finding interesting input for them, and running them and the corresponding derivative routines separately under controlled conditions. (See Figure 1.) Doing this for 19 subroutines, many of which involved derivatives of multiple quantities or derivatives with respect to multiple parameters required more than 4000 lines of debugging code.

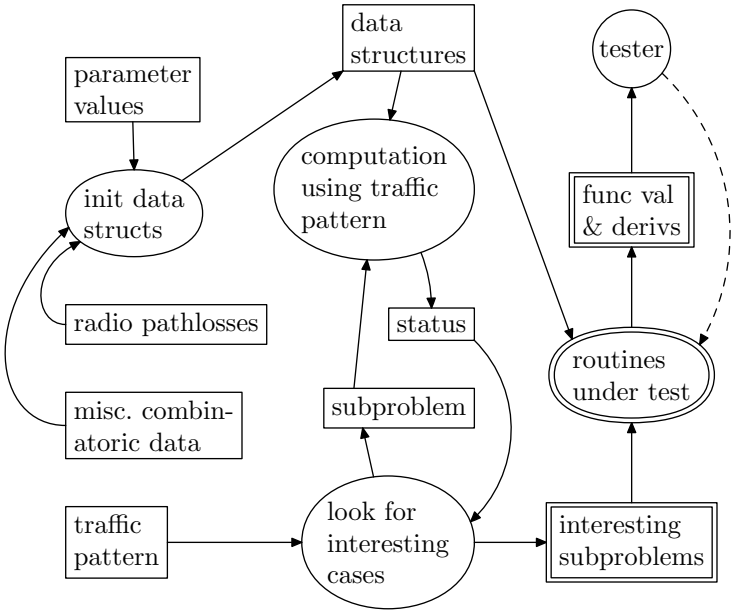

Fig. 1. Major components of a test harness for using finite differences to verify individual derivative routines.

In spite of the great effort to provide finite difference tests that could localize bugs to specific subroutines, the 19 derivative routines tested represented only a small fraction of the total: examination of the source code revealed 133 routines that compute derivatives of specific functions, and many more for which the quantity being differentiated was only part of one or more larger functions.

4.2 Debugging Tools

Although finite difference tests have been around for a long time, there should be more tools for constructing and maintaining debugging code like that described in Section 4.1.

For the "black box" approach, what required 1200 lines of debugging code was making the 10 types of intermediate results and their purported derivatives available for testing. There are plenty of tools for checking a function against its gradient, but that only automates the easy part of the task. The hard part seems problem-specific and hard to automate, so it may be best to concentrate on automating the fine-grained, "white box" approach.

Automating the test harness illustrated in Figure 1 requires a programming discipline whereby the programmer can specify which derivative routines correspond to which original functions. Some kind of random sampling has to be used to select specific invocations from the running program, then all the input and data structures needed by the routine to be tested would have to be checkpointed for use in the separate testbed. Issues that have to be dealt with are the desire for "interesting subproblems," and the difficulty in taking subroutines from one program and guaranteeing that they will compile and run in another program.

For the cell phone application, finding "interesting subproblems" required classifying invocations of the routine being tested based on the combinatorial properties of the input, and then making sure that all the combinatorial classes were represented. In general, the programmer would probably have to provide code for classifying invocations if there are rare but important cases that simple random sampling might miss.

It might also be possible to avoid some of these difficulties by using theorem-proving techniques to verify hand-coded derivatives. This may have the potential for providing assurances that the derivative routines are fully debugged.

4.3 Routines that are Simple Enough for Efficient Automatic Differentiation

In order for a function to be simple enough for efficient AD, it is desirable 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 surely qualifies, and it is also not be too hard to handle function calls and loops that are not order dependent.

The cell phone application had 77 functions for which there were corresponding derivative routines, and 50 of them met this definition of "simple enough": the function and its derivative had the same *if* tests and executed the same loops in the same order (but 6 of the 50 routines functions had extra *if* tests to handle degeneracies).

Of the 27 non-simple derivative routines, 17 required control flow reversals and 10 required two or more different types of derivatives in separate routines with major changes in control flow. Six of the control flow reversals were due to loops where each iteration depended on the previous one, and 11 were due calling various other user-defined functions in a manner that required control flow reversal.

4.4 Automatic Differentiation of Sufficiently Simple Routines

We have seen that many functions can be differentiated efficiently with no significant changes in control flow. Automating just those derivatives would significantly speed software development and allow the user to handle high-level decisions such as adding another pass over the data structures so that the automatically-generated could achieve low time and memory costs. A rough prototype has been developed for the purpose of experimenting with these ideas. It requires the user to insert special comments before each routine to be differentiated, saying what to differentiate with respect to what. Then the system parses the input program and outputs a new version with generated derivative routines added.

The first stage of processing is a C++ parser that preserves comments and white space. It uses a proprietary C/C++ grammar for the SGLR parser [9] that was developed by D. Waddington as part of a research project on software transformation. Then it looks at the type declarations to build up a hash table containing type information, and passes function definitions preceded by the special comments to the AD package. The result of AD is a generated parse tree for the derivative routine's function declaration. After merging this with the original parse tree, the merged parse tree can be converted back into C++.

The prototype handles AD for a single function by proceeding as follows:

1. Construct a control-flow graph, and decide which basic blocks must precede or follow which others.
2. Within each basic block, convert the parse tree into a series of expression trees with variables replaced by expression trees for their values. (This generates common subexpressions so that the expression trees become a DAG.)
3. For each basic block where there are expression trees that represent the final values of quantities to be differentiated, convert each such expression tree into an expression tree for the derivative as follows:

a) Mark the root node with an adjoint of 1 since we need 1 times its derivative with respect to the independent variable x.
b) Find a node that is marked with adjoints for each of its parents, generate expression trees that multiply by its partial derivatives, and use them as the adjoints with which its children must be marked.
c) Repeat the previous step until all nodes have been processed. Whenever an instance of the variable x is encountered, add its adjoint to the expression tree for the the derivative result.

4. Convert all the expression trees back into parse trees, creating temporary variables for common subexpressions.

Note that there is no control flow reversal, so it only works if loop iterations do not depend on previous ones and the loops can thus be thought of as parallelizable. The order of execution of basic blocks must also be predictable enough in order to know what other basic blocks to refer to in order to find the expression tree for a variable's value at the start of a basic block. The advantages are that this procedure gives the full benefits of reverse mode AD while generating a full set of expression trees that can be simplified via common subexpression elimination and other techniques.

4.5 Special Input for Automatic Differentiation

Under some circumstances, it may be necessary for the user to provide special routines as input for AD. 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 efficient automation for the case where iterative methods are used to solve a nonlinear system of equations, the special routine provided by the user should be a computation of the residual. Then of there could be an option for a redundant run-time check that verifies that the residual is small at the computed solution. Although some AD tools cope with such iterations by techniques such as ignoring all but the last few iterations, better efficiency should be obtainable via the less automatic approach of requiring the user to provide a residual function. For the iterations found in the cell phone application, such residual functions would have been easy to write.

5 Conclusion

We have analyzed a large application where stringent limitations on run time and memory consumption would have presented a difficult case for AD. Some form of partial automation would probably have been useful, but a long series of project deadlines allowed no time to develop this.

In order to prevent projects such as this one from missing out entirely on the benefits of AD, there needs to be more of a range of approaches that cover the full spectrum from completely manual and nearly automatic. While almost all AD tool providers recommend significant manual intervention in order to achieve good performance, it is easy for a practitioner to decide that he cannot afford to go that far away from the manual approach.

The semiautomatic tool discussed in Section 4.4 is currently in a very preliminary state. It is intended to do derivatives approximately the same way a human programmer would, but it certainly has more limitations than a human programmer. It should probably be extended so as to relax the control flow restrictions in a manner consistent with this philosophy. Other problems are that it does not cope well with C++ templates and exceptions and it should probably be merged with existing tools, perhaps via OpenAD [8].

References

1. Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.
2. David M. Gay. Automatic differentiation of nonlinear AMPL models. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 61–73. SIAM, Philadelphia, PA, 1991.
3. Ralf Giering and Thomas Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002, Augsburg, Germany*, 2002.
4. 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.
5. 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.
6. Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, pages 1–78, 2003.
7. 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.
8. Jean Utke. OpenAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 2004.
9. 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.