Christian Bischof and Andreas Griewank

Tools for the Automatic Differentiation of Computer Programs

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 basic implementation issues for the design of general purpose tools that can deal with codes from the Fortran and C family, address some frequently asked questions, and provide pointers for further study.

1. Introduction

Let f be a computer model, and denote by $f(x)$ the output produced for a particular input x. Derivatives $\frac{\partial f_j}{\partial x_i}$ are ubiquitous in numerical computing; examples are methods for minimization, the solution of nonlinear systems of equations, or the solution of stiff ordinary differential equations, partial differential equations, or differential-algebraic equations. Derivatives also play a central role in the sensitivity analysis of computer models, where one tries to assess the sensitivity of a computational model to perturbations in its parameters or initial conditions. inverse problems, where one tries to calibrate the initial state of a computer model such that its behavior best matches a series of experimentally acquired data, or (multidisciplinary) design optimization, where one tries to find the optimal setting of input parameters of a computer model with respect to a cost function that quantifies the quality of the overall design.

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 vector of *independent variables* and y the vector of *dependent variables.* In computing derivatives, we should keep the following issues in mind:

Reliability: The computed derivatives should ideally be accurate to machine precision. If the functional relation between x and y is not necessarily smooth the user should get a warning that something might be amiss.

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 as much as possible and in any case a priori bounded.

Scalability: Whatever tool we choose should give correct results for 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 preparing a code for differentiation, in particular in situations where computer models are bound to change frequently.

Handcoding, divided-difference approximations, and symbolic manipulators 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, [5,10]. By applying the chain rule of derivative calculus e.g.,

$$\frac{\partial}{\partial t} f(g(t))\Big|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)\Big|_{s=g(t_0)}\right)\left(\frac{\partial}{\partial t} g(t)\Big|_{t=t_0}\right) \tag{1}$$

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 [8].

In the next section, we give a brief overview of automatic differentiation. Section discusses the major implementation issues that arise in the design of automatic differentiation tools. Section answers some commonly asked questions. Lastly, we discuss remaining challenges and prospects.

2. Automatic Differentiation Modes

Traditionally, two basic approaches to automatic differentiation have been employed: the so-called forward and reverse mode, which date back to the early sixties and seventies, respectively. 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 [2,5,10] 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

$$J = \begin{pmatrix} \frac{\partial\, y(1)}{\partial\, x(1)} & \cdots & \frac{\partial\, y(1)}{\partial\, x(n)} \\ \vdots & & \vdots \\ \frac{\partial\, y(m)}{\partial\, x(1)} & \cdots & \frac{\partial\, y(m)}{\partial\, x(n)} \end{pmatrix}. \tag{2}$$

For an $n \times p$ matrix S, the effort required is roughly 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 store the intermediate values that have been involved in nonlinear operations before they are overwritten or go out of scope. Sometimes some of these intermediates can be recomputed during the reverse sweep but in any case one has to keep a log of the branch directions taken.

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 operations count does not depend on the number of independent variables.

The forward mode can be very naturally extended to second third and even higher derivatives, but the complexity grows like the square or cube p, respectively. Especially for Hessian-vector products a combined forward and reverse sweep is attractive, as it still has essentially the same complexity as a single evaluation of the underlying scalar function. In any case, automatic differentiation produces code that computes derivatives accurate to machine precision [8] The techniques of automatic differentiation are directly applicable to computer programs of arbitrary length containing branches, loops, and subroutines.

The weighting and combining of derivatives through the matrices W and S is very natural and useful for many applications, especially if sparsity in J can be exploited. Unfortunately, many existing AD tools are (like computer algebra packages) still exclusively oriented towards the evaluation of Cartesian derivatives, i.e. the partials of certain dependent variables with respect to certain independent variables.

3. Design Issues for Automatic Differentiation Tools

Automatic differentiation can be viewed as a particular semantic transformation problem: Given a code for computing a function, we would like to generate a code that computes the derivatives of that function. To affect this transformation, two approaches have been employed:

Operator Overloading: Modern computer languages like C++ or Fortran 90 make it possible to redefine the meaning of elementary operators. That is, we can for example define a type for floating point numbers that have gradient objects associated with them (let's call them `adouble`, say), and for each elementary operation such as a multiplication, we can define the meaning of the operator '*' for variables of type `adouble`. If we

define the usual product rule ($z = x * y \rightarrow \nabla z = x\nabla y + y\nabla x$), then each occurrence of a multiplication of two `adoubles` in the code will also effect the update of the associated derivatives in a transparent fashion.

Source Transformation: Another way of changing the semantics of the code is to rewrite it explicitly. That is, for example, the assignment $z = x * y$ is rewritten into a piece of code that not only contains the computation of z, but also an implementation of the vector linear combination $z = x\nabla y + y\nabla x$, implemented either as a do-loop, or as a subroutine call.

Each of these approaches has its advantages and disadvantages. The advantages of operator overloading are

Terseness: All that is required for a new data type, such as `adoubles`, is a new class definition. While such a class definition can be substantial, comprising several thousand lines of code, it hides this complexity from the user of an AD tool.

Flexibility: If we want to change an implementation strategy associated with a particular class, the source code remains unaffected. All that changes is the class definition itself. So for example, whether we compute first or second order derivatives is reflected in the class definition, but not in the code being differentiated.

Full Access to Runtime Information: As mentioned previously, the reverse mode of AD requires the ability to reverse the partial flow of program execution. One way to do this is to use operator overloading to generate a tape that logs all the operations actually performed, and use this tape as the input for a derivative interpreter, which then can compute any derivatives desired using either the forward or reverse mode of automatic differentiation. This approach is, for example, chosen in the ADOL-C [7]

The drawbacks of operator overloading are

Lack of Transparency: While it is aesthetically pleasing that the source code does not change, even though its meaning does, it does not aid in debugging, as one has to deduce the meaning of the operations implied by the source code and the associated class definitions.

Implementation Overhead: The actions associated with a class definition can be viewed as an implied subroutine call, and although much progress has been made recently in the compilation of operator overloading, the runtime overhead of this technique can be substantial.

Dusty Deck Assimilation: Many existing computer codes are written in languages such as Fortran 77 or ANSI-C which do not support operator overloading. In particular for large codes, assimilating such codes into the supposedly backwards compatible Fortran 90 or C++ languages turns out to be a thorny task.

On the other hand, the advantages of the source transformation approach are

Simplicity of Generated Code: Since the derivative code is spelled out exactly, usually in the same language as the input code, it is easier to follow the actions of the derivative code. This simplicity also facilitates compiler optimizations and hence faster execution of the generated code.

Dusty Deck Assimilation: The source transformation approach requires traditional compiler infrastructure such as parsers, generators and manipulators of intermediate languages, and unparsers. These kind of tools are readily available for languages such as Fortran 77 or ANSI-C, at least in the commercial world.

Variable Scope: Operator overloading inherently sees one elementary operation at a time. Source transformation approaches, on the other hand, have access to the context of a particular computation, and hence have more flexibility in applying derivative rules. For example, the ADIFOR [1,2] and ADIC [3] tools view a program as a sequence of assignment statements, applying the reverse mode at this level, and the forward mode overall.

The disadvantages of the source transformation approach are

Implementation Complexity: Source transformation approaches, at least at the moment, require considerable tool infrastructure, in particular for the processing of language-dependent features. Also, the lack of a standardized language description makes changing the semantics of a particular automatic differentiation tool a potentially rather involved task.

Code Expansion or Subroutine Interface Swell: A "pure" source transformation approach is infeasible when the action associated with a particular statement exceeds a certain level of complexity. In this case, either the length of the generated code grows too large for a compiler to digest, or alternatively, rather extensive subroutine library interfaces need to be maintained to encapsulate the basic computational kernels. The latter approach, in many ways, is similar to operator overloading, albeit considerably less elegant.

It depends to a great extent on the particular application to what extent the above-mentioned advantages and disadvantages are relevant. Hence, we encourage the reader to have a look at the collection of automatic differentiation tools at

http://www.mcs.anl.gov/Projects/autodiff/AD_Tools

This collection gives a short description of some available automatic differentiation tools and provides pointers how to obtain access to these tools.

4. Frequently Asked Questions

Given the mathematical underpinnings 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 [4].

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. Some tools invoke an extension handler flagging such occurrences, others ignore such occurrences.

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 [6]. This problem clearly requires more research, but the emergence of robust AD tools has made it possible to tackle this problem for sophisticated numerical methods.

5. Conclusions

This paper gave a brief introduction into automatic differentiation. We reviewed the forward and reverse mode of automatic differentiation, gave some background on implementation issues, and answered some commonly asked questions.

Automatic differentiation is a technology in its infancy. The emergence of robust AD tools for general purpose computer languages such as Fortran 77, Fortran 90, C, and C++ is putting these tools within the reach of many computational practitioners. At the same time, mathematical modeling and design techniques for nonlinear processes usually require derivatives, and the computational horse power typically available today makes such schemes feasible for most computational practitioners.

Much remains to be done in the development of better AD tools. Currently, all AD tools choose one or the other of the two basic implementation approaches mentioned previously. However, the mostly complementary lists of advantages and disadvantages of operator overloading and source transformation suggests that implementations combining both approaches will be the promising approach in the long term. Lastly, the associativity of the chain rule of differential calculus leaves great leeway in how derivatives are actually computed. New approaches are slowly emerging (see, for example, [9]), but much remains to be explored.

6. References

1 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.

2 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.

3 Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC – An Extensible Automatic Differentiation Tool for ANSI-C, Preprint ANL/MCS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

4 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.

5 Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

6 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.

7 Andreas Griewank, David Juedes, and Jay 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.

8 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.

9 Paul Hovland, Christian Bischof, Donna Spiegelman, and Mario Casella. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. Preprint MCS-P491-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

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

Addresses: Christian H. Bischof, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, `bischof@mcs.anl.gov`.

The work of this author was supported by the Mathematical, Information, and Computational Science 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 by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

Andreas Griewank, Institut für Wissenschaftliches Rechnen, Technische Universität Dresden, Mommsenstr. 13, D-01062 Dresden, `griewank@math.tu-dresden.de`.