Computing Derivatives of Computer Programs

Christian Bischof and Martin Bücker

published in

Modern Methods and Algorithms of Quantum Chemistry,
Proceedings, Second Edition, J. Grotendorst (Ed.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. 3, ISBN 3-00-005834-6, pp. 315-327, 2000.

COMPUTING DERIVATIVES OF COMPUTER PROGRAMS

CHRISTIAN H. BISCHOF AND H. MARTIN BÜCKER
Institute for Scientific Computing
Aachen University of Technology
D-52056 Aachen
Germany
E-mail: {bischof,buecker}@sc.rwth-aachen.de

Automatic differentiation is introduced as a powerful technique to compute derivatives of functions given in the form of a computer program in a high-level programming language such as Fortran, C, or C++. In contrast to traditional approaches such as handcoding of analytic expressions, numerical approximation by divided differences, or manipulation of symbolic algebraic expressions by computer algebra systems, automatic differentiation offers the following substantial benefits: it is accurate up to machine precision, efficient in terms of computational cost, applicable to a 1-line formula as well as to a 100,000-line code, and can be produced with minimal human effort.

1 Introduction

Numerical simulations arising in large-scale scientific applications such as quantum chemistry often require the evaluation of derivatives of some objective function. An example is given in this conference proceedings[1] where the need for derivatives in quantum chemical calculations of molecular properties is demonstrated. Derivatives play a crucial role not only in quantum chemistry but in numerical computing in general. Examples include the solution of nonlinear systems of equations, stiff ordinary differential equations, partial differential equations, and differential-algebraic equations. Derivatives are also ubiquitous in the areas of sensitivity analysis of computer models, inverse problems, and (multidisciplinary) design optimization.

Traditionally, such problems with derivatives have been addressed by using techniques of numerical and analytical differentiation as discussed by Gauss[1]. Here, we will discuss another powerful technique called automatic differentiation (AD) for computing derivative information, say, gradients or Hessians. AD has been successfully applied[2,3], it is currently less well known than and sometimes confused with symbolic differentiation. The purpose of this note is to call attention to automatic differentiation, to provide some background information on the technique, and to highlight its advantages over other techniques of differentiation.

To abstract from the particular area of interest, let

$$f : \mathbb{R}^n \to \mathbb{R}^m \qquad \text{with} \qquad \mathbf{x} \mapsto \mathbf{y}$$

denote any vector-valued objective function whose derivatives are sought. We call $\mathbf{x}$ the vector of *independent variables* and $\mathbf{y}$ the vector of *dependent variables.* In large-scale applications, the objective function f is typically not available in analytic form but is given by a computer code written in a high-level programming language such as Fortran, C, or C++. Think of f as a function computed by, say, one of the modules of the TURBOMOLE program system to compute and analyze the electronic structure of molecules[4]. Given such a representation of the objective function $f(\mathbf{x}) = \big(y_1(\mathbf{x}), y_2(\mathbf{x}), \ldots, y_m(\mathbf{x})\big)^T$, computational methods often demand the evaluation of the Jacobian matrix

$$J(\mathbf{x}) := \begin{pmatrix} \frac{\partial}{\partial x_1} y_1(\mathbf{x}) & \ldots & \frac{\partial}{\partial x_n} y_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} y_m(\mathbf{x}) & \ldots & \frac{\partial}{\partial x_n} y_m(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{m \times n} \tag{1}$$

at some point of interest $\mathbf{x} \in \mathbb{R}^n$.

A well-known and widely used approach for the approximation of the Jacobian matrix is the use of divided differences (DD). For the sake of simplicity, we only mention first-order forward DD but stress that the following discussion applies to DD as a technique of numerical differentiation in general. Using first-order forward DD, one approximates the ith column of the Jacobian matrix Eq. (1) by

$$\frac{f(\mathbf{x} + h_i \mathbf{e}_i) - f(\mathbf{x})}{h_i}, \tag{2}$$

where h_i is a suitably chosen step size and $\mathbf{e}_i \in \mathbb{R}^n$ is the ith Cartesian unit vector. An advantage of the DD approach is that the function f needs to be evaluated only at some suitably chosen points. Roughly speaking, f is used as a black-box. The main disadvantage of DD is that the accuracy of the approximation depends crucially on a suitable choice of these points, that is, of the step size h_i. However, any strategy to determine a step size faces the dilemma of mutual influence of truncation and cancellation error: The step size should be small to decrease the error of Eq. (2) in approximating Eq. (1) even if infinite-precision arithmetic were used; the step size should be large to avoid cancellation of significant digits when using finite-precision arithmetic in the computation of Eq. (2).

Another traditional approach for computing derivatives is handcoding of analytic expressions. Here, an analytic expression for the Jacobian matrix $J(\mathbf{x})$ is identified first and then implemented by hand using any high-level programming language. If care is taken, handcoding results in highly optimized implementations. However, analytic expressions are not always available. Furthermore, handcoding is smooth only for "simple" objective functions, is substantially error-prone, and requires considerable human effort.

Computer algebra systems such as MACSYMA can, in principle, also be used to find an explicit expression for the Jacobian matrix $J(\mathbf{x})$. A disadvantage of symbolic differentiation is that the length of the representation of the resulting derivative expressions increases rapidly with the number n of independent variables. This property is extremely painful when higher-order derivatives are considered. For instance, the Hessian of an objective function of some complexity in more than three variables can easily result in expressions filling several pages. Moreover, symbolic differentiation is inherently inefficient in terms of computing time, because of the rapid growth of the underlying expressions. The reader is referred to an article by Griewank[5] for a more detailed discussion of computing derivatives symbolically. Another computer algebra system, Maple, is unusual in that it does offer the additional option of automatic differentiation. However, the intention of automatic differentiation of Maple procedures is the development of efficient programs in Maple and other programming languages (Fortran, C). On the other hand, in this note we consider automatic differentiation for generating derivatives of large production codes written in virtually any high-level programming language.

Automatic differentiation is another option for computing the Jacobian matrix $J(\mathbf{x})$. Virtually any computer program written in a high-level programming language such as Fortran, C, or C++ can be differentiated by this black-box mechanism. Given a program for the evaluation of the objective function f, this technique generates, in a completely automatic fashion, another computer program, called the extended program, that evaluates $f(\mathbf{x})$ and $J(\mathbf{x})$ simultaneously. The key concept behind AD is the fact that every computation, no matter how complicated, is executed on a computer as a (potentially very long) sequence of a limited set of elementary arithmetic operations such as additions, multiplications, and intrinsic functions such as `sin()` and `cos()`. By applying the chain rule over and over again to the composition of these elementary operations, the extended program can be generated accurately evaluating $f(\mathbf{x})$ and $J(\mathbf{x})$ up to machine precision. AD techniques are discussed in a monograph[6] and a forthcoming book[7]. Differentiating a computer program by AD meets all of the following requirements:

Reliability: The computed derivatives should ideally be accurate to machine precision. If the functional relation between $\mathbf{x}$ and $\mathbf{y}$ 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 be bounded a priori.

Scalability: The approach should give correct results for a 1-line formula as well as a 100,000-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 in which computer models are bound to change frequently.

Handcoding, divided-difference approximations, and symbolic manipulators 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 the next section, we give a brief overview of automatic differentiation. Section 3 discusses issues that arise in the design of software packages implementing the AD technology. In Section 4, we discuss some issues concerning the use of AD tools. In the last section, we summarize AD's advantages and provide pointers to AD tools.

2 Basic Modes of Automatic Differentiation

Traditionally, two basic approaches to automatic differentiation have been employed: the so-called forward mode 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 the literature[5,6,8].

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 $J\,S$ of columns of the Jacobian matrix J. In matrix-matrix multiplication, the symbol S denotes an arbitrary $n \times p$ matrix. The effort required to compute not only the objective function but also $J\,S$ is roughly p times the runtime and memory of the original program. In particular, when $p = 1$ and thus the matrix S reduces to a vector $\mathbf{s}$, we compute the directional derivative

$$J\ \mathbf{s} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}) - f(\mathbf{x})}{h},$$

where h is some step size.

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 must 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 an $m \times q$ matrix W, the reverse mode allows us to compute arbitrary linear combinations $W^T J$ of rows of the Jacobian matrix J with roughly 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 $q = 1$ and thus the matrix W^T reduces to a row vector w^T, we compute the derivative $w^T J$. The reverse mode is particularly attractive for the computation of long gradients, as its operations count does not depend on the number n of independent variables.

The forward mode can be naturally extended to second or 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, since 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[5]. 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 are 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 toward the evaluation of Cartesian derivatives, that is, the partials of certain dependent variables with respect to certain independent variables.

3 Design of 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 effect this transformation, two approaches have been employed:

Operator Overloading: Modern computer languages such as C++ or Fortran 90 make it possible to redefine the meaning of elementary operators. We can, for example, define a type for floating-point numbers that have gradient objects associated with them (let's call this new type `adouble`), and for each elementary operation such as a multiplication, we can define the meaning of the operator "$*$" for variables of type `adouble` as follows. An assignment $z = x * y$ not only computes the product of x and y but also updates the associated gradient object in a product rule fashion $\nabla z = x \nabla y + y \nabla x$. So, each occurrence of a multiplication of two `adouble`s 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. For example, the assignment $z = x * y$ is rewritten into a piece of code that contains not only the computation of z but also an implementation of the vector linear combination $\nabla z = x \nabla y + y \nabla x$, 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 threefold.

Terseness: All that is required for a new data type, such as `adouble`s, 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: 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 package[9].

The drawbacks of operator overloading are the following

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, since 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 depending on the sophistication of the compiler.

Dusty Deck Assimilation: Many existing computer codes are written in languages such as Fortran 77 or ANSI-C that do not support operator overloading. In particular, assimilating large 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 as follows.

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 as long as the chain rule is applied in a basic local fashion. 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 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[10,11] and ADIC[12] 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 the following.

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 rather extensive subroutine library interfaces must be maintained to encapsulate the basic computational kernels. The latter approach, in many ways, is similar to operator overloading, albeit considerably less elegant.

Of course, the relevance of these advantages and disadvantages depends to a great extent on the particular application.

Given the mathematical underpinnings of the concept of derivatives, the "black-box" application of an AD tool usually raises several questions that we briefly 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 by Fischer[13].

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 by Griewank *et al.*[14]. 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.

4 Using Automatic Differentiation Tools

Based on our experience with the ADIFOR[10,11] and ADIC[12] tools for automatic differentiation, this section explores some of the subtler issues related to the use of AD and the implications for numerical software design. In particular, we focus on the issues that arise from the fact that AD differentiates a given computer program step by step, in a fashion that is oblivious of the overall semantics of a program. This "myopic" view gives AD tools the power to deal with programs of arbitrary length, but it also implies that users of AD tools may have to communicate some of their knowledge to an AD tool to arrive at a desired solution. Specifically, we illustrate the issues arising in the context of nondifferentiable language intrinsics such as `max()` and numerical integrators.

4.1 Intrinsic Functions

Since the derivative of $\sin(x)$ with respect to x is given by $\cos(x)$, an AD tool might transform the statement

$$\texttt{y} = \texttt{sin}(\texttt{x})$$

into the derivative statement

$$\nabla\texttt{y = cos(x) * }\nabla\texttt{ x.}$$

Here, the notation ∇`y` denotes the derivatives of variable `y` with respect to some chosen set of variables. In this case, there is no difficulty, since `sin()` is everywhere differentiable.

Most computer languages do, however, contain intrinsic functions that are not differentiable in some points in their domain, as for example the Fortran 77 intrinsics `abs(x)` and `sqrt(x)` when the value of the argument is zero. We call such a point an "exceptional point." We cannot simply claim that the function in question is not differentiable, since a computer program executing such instructions may well represent a smooth function, such as $g(x, y) = \sqrt{x^4 + y^4}$. Moreover, intrinsics may be used to guard against unphysical values of simulation parameters. For example, in a weather model one might see code such as

$$\texttt{rain} = \texttt{max}(\texttt{rain}, 0.0).$$

This statement reflects the fact that rainfall cannot be negative and is intended to convert a small negative number, which may have arisen from floating-point roundoff, to the physically sensible number 0 (i.e., no rain).

The function $\max(x, y)$ is not differentiable for $x = y$. However, in the previously described case, it makes sense to define partial derivatives for the exceptional cases as $\frac{\partial \max(x,y)}{\partial x}|_{x=y} := 1.0$ and $\frac{\partial \max(x,y)}{\partial y}|_{x=y} := 0.0$. These definitions do not change ∇`rain` when `rain` is set to zero in the induced AD statement

$$\nabla\texttt{rain} = \frac{\partial \max(x, y)}{\partial x}\nabla\texttt{rain}.$$

However, these definitions would not lead to the desired result if the order of arguments in the `max()` call was reversed, namely,

$$\texttt{rain} = \texttt{max}(0.0, \texttt{rain}).$$

In this case, the derivative of `rain` would be zeroed out when the value of the variable was zero, and it would have been appropriate to exchange the definitions of $\frac{\partial \max}{\partial x}$ and $\frac{\partial \max}{\partial y}$. In other contexts, an argument could also be made for setting $\frac{\partial \max(x,y)}{\partial x}|_{x=y} = 0.5$ and $\frac{\partial \max(x,y)}{\partial y}|_{x=y} = 0.5$, since then automatic differentiation provides a so-called subgradient, which is useful in nonsmooth numerical optimization, as described, for example, by Clark[15].

These examples demonstrate the following points:

i. No choice of derivative values for exceptional points will always be correct.

ii. There is no "automatic" way to decide what sensible choices are.

```
Given: parameter $p$, current time $t$, current solution $x_c \approx x(t,p)$,
       suggested time step $\Delta t$.
1) Compute $x_1 \approx x(t+\Delta t, p)$ using Method A.
2) Compute $x_2 \approx x(t+\Delta t, p)$ using Method B.
3) Compute $\delta = \|x_1 - x_2\|$ for some norm $\| \cdot \|$.
4) if ( $\delta\langle$some given threshold )
     Accept the higher-order of $x_1$ and $x_2$
     and update $t \leftarrow t + \Delta t$.
   else
     $\Delta t = g(\Delta t, \delta)$;
     goto 1)
   endif
```

Figure 1. Simplified Description of a Numerical Integrator

iii. User insight into the problem is essential.

Thus, potential users of AD tools need to be aware of these facts and provide "hints" for an AD tool in the code to be eventually differentiated. Such hints are particularly important for numerical libraries, since these codes typically embody subtle numerics and will be reused often. To this end, the ADIFOR and ADIC systems employ the completely user-customizable ADIntrinsics system for dealing with Fortran and ANSI-C intrinsics. Surprisingly, in most cases the derivatives turn out to be the ones intended without the need for derivatives intrinsics modifications.

4.2 Numerical Integrators

Another problem arises from the fact that an AD tool, when applied to a code embodying a numerical method, will not only differentiate the solution produced by this method, but also take into account the *way by which one arrived at the solution.* As an illustration, consider a parameter-dependent initial value problem

$$\dot{x}(p) = f(x,p,t) \qquad \text{with} \qquad x(t=0) = x_0, \tag{3}$$

where p is a parameter. Figure 1 shows a simplified version of the time-stepping loop of a typical explicit numerical integrator with step size control. In this figure, the notation Method A and Method B is used for two integration methods of different order, and g is some function that adjusts the time step Δt. For simplicity, we ignore the fact that the time step will be adjusted upward if there is a good fit.

If, for a given p, we are interested in $\frac{\partial x}{\partial p}\big|_{t=T}$, where T is the final time, we can employ an AD tool to differentiate this code with respect to p. If we differentiate with respect to p and use ∇ to denote $\frac{\mathrm{d}}{\mathrm{d}\,p}$, the chain rule of differential calculus now implies that

$$\nabla(\Delta t) = \frac{\partial g}{\partial (\Delta t)} \nabla(\Delta t) + \frac{\partial g}{\partial \delta} \nabla \delta.$$

Clearly, $\nabla \delta \neq 0$ in general, since δ depends on x, which in turn depends on p. Thus we have the interesting situation that $\nabla(\Delta t) \neq 0$ when $\frac{\partial g}{\partial \delta} \neq 0$; that is, the computational equivalent of time, Δt, will have a nonzero derivative with respect to the parameter p. Viewed from an analytical perspective, this is nonsense – the values of time and the parameter are not related. From a computational perspective however, it does make sense – depending on the value of the parameter, we may choose a different time discretization. Thus, what we really compute as the final value $x_T(p)$ is

$$x_T(p) = x(t(p), p)|_{t(p)=T}$$

(note the dependence of t on p). Thus, we obtain

$$\nabla x_{t=T} = \left.\frac{\partial x}{\partial t}\right|_{t=T} \cdot \nabla t_{t=T} + \left.\frac{\partial x}{\partial p}\right|_{t=T},$$

and with Eq. (3)

$$\nabla x_{t=T} = f(x_T, p, T) \cdot \nabla t_{t=T} + \left.\frac{\partial x}{\partial p}\right|_{t=T}.$$

Note that ∇x and ∇t will have been computed by the AD-generated derivative code. We observe the following:

i. Depending on how the time discretization was chosen, we will obtain different values for $\nabla t_{t=T}$ and thus for $\nabla x_{t=T}$. Most certainly, we will *not* obtain $\left.\frac{\partial x}{\partial p}\right|_{t=T}$ which is the result desired by most users.

ii. If Δt would had been zero at every step, we would have $\nabla t_{t=T} = 0$ and thus $\nabla x_{t=T} = \left.\frac{\partial x}{\partial p}\right|_{t=T}$, as desired by the user. By default, this happens in methods using a fixed step size. This case is also discussed by Sandu *et al.*[16]

iii. Independent of how the time discretization was chosen, we can recover the desired solution as

$$\left.\frac{\partial x}{\partial p}\right|_{t=T} = \nabla x_{t=T} - f(x_T, p, T) \cdot \nabla t_{t=T}. \tag{4}$$

These issues are discussed in more detail by Eberhard and Bischof[17].

Note that approaches (ii) and (iii) are really geared toward the library developer and the sophisticated AD user, respectively. When an integrator code is written, it is probably feasible to indicate the places where the next time step is assigned and to indicate that an AD tool should treat this statement as constant with respect to differentiation, resulting in the assignment of a zero gradient. Current AD tools do not have such facilities built-in yet, but will so soon. At any rate, unless the developer of the integrator provides this information, the considerable sophistication of these codes makes it difficult for others to extract this information from the code.

While one might take the attitude that this was not an issue, given the "fix" (iii), this is not really the case. Even when $\frac{\partial x}{\partial p}$ is well behaved, ∇t and ∇x can become very large and can overflow. Furthermore, the user of an AD tool may well be unaware of these issues, or may not be able to localize the problem since the integrator may be buried under other layers of software. However, as shown by Eberhard and Bischof[17], if the final time is prescribed, we are likely to obtain

$\nabla t_{t=T} = 0$, and everything works out; we suspect that this situation has happened in quite a few AD applications.

We note that while (ii) and (iii) will result in the right derivatives $\frac{\partial x}{\partial p}$, there is no guarantee that the derivatives will be obtained at the same accuracy as the solution x, since the guard of the if-statement governing acceptance or rejection of a step will *not* be augmented by AD, and thus still will be governed only by the behavior of x. Thus, the derivatives obtained by Eq. (4) will be consistent, but they may not be as accurate as those obtained by solving the sensitivity equation

$$\dot{x}_p = \frac{\partial f}{\partial x} x_p + \frac{\partial f}{\partial p},$$

where $x_p = \frac{\partial x}{\partial p}$, alongside the original differential equation Eq. (3). It is easy to add the norm of $\nabla \delta$ to the guard for step size control, but an AD tool cannot be expected to do so without user guidance. Similar issues also arise in the context of automatic differentiation of iterative solvers for nonlinear equations and are discussed by Griewank *et al.*[14].

5 Concluding Remarks

This note was meant to give a brief introduction to automatic differentiation. We briefly discussed the advantages of this powerful technique in contrast to the better-known approaches of numerical, analytic, and symbolic differentiation. Broadly speaking, automatic differentiation saves work in comparison with handcoding of analytic derivatives and, by computing accurate derivatives, avoids the hassle caused by inaccurate numerical differentiation. We reviewed the forward and reverse modes of automatic differentiation, gave some background on design issues of automatic differentiation tools, and discussed some subtle issues involved in using these tools.

Even though automatic differentiation tools are still in their infancy, under a wide range of circumstances they already can compute derivatives faster than divided difference approximations[11]. Furthermore, there are examples where the availability of fully accurate derivatives was essential for numerical robustness and convergence[18,19,20]. Another advantage of automatic differentiation tools that we did not discuss in this note is their ability to provide, in a fashion that is transparent to the user, information about the zero/nonzero structure of derivative matrices.[21] This information is required to solve linear systems involving the Jacobian, and the automatic detection of the sparsity pattern avoids the error-prone task of having the user specify the sparsity pattern. This feature is provided in ADIFOR and ADIC through the SparsLinC library and is used, for example, in the NEOS (Network-enabled Optimization Server) problem-solving environment[22], which is accessible at `http://www-neos.mcs.anl.gov/`.

The emergence of robust automatic differentiation tools applicable to functions defined by computer programs in general-purpose computer languages such as Fortran 77, Fortran 90, C, and C++ is putting these tools within the reach of many computational practitioners in any field requiring derivatives, including quantum chemistry. The web site at `http://www.sc.rwth-aachen.de/Research/AD/subject.html` gives a short description of some available automatic differentiation tools and provides pointers for obtaining these tools.

Acknowledgments

This work was completed while the second author was visiting the Mathematics and Computer Science Division, Argonne National Laboratory, supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

1. J. Gauss, in *Modern Methods and Algorithms of Quantum Chemistry*, edited by J. Grotendorst, (John von Neumann Institute for Computing, Jülich, Germany, 2000).
2. M. Berz, C. Bischof, G. Corliss, and A. Griewank, *Computational Differentiation: Techniques, Applications, and Tools* (SIAM, Philadelphia, 1996).
3. A. Griewank and G. Corliss, *Automatic Differentiation of Algorithms* (SIAM, Philadelphia, 1991).
4. R. Ahlrichs, M. Bär, M. Häser, H. Horn, and C. Kölmel, Chemical Physics Letters **162**, 165 (1989).
5. A. Griewank, in *Mathematical Programming: Recent Developments and Applications*, pages 83–108 (Amsterdam, 1989, Kluwer Academic Publishers).
6. L. B. Rall, *Automatic Differentiation: Techniques and Applications*, volume **120** of *Lecture Notes in Computer Science* (Springer Verlag, Berlin, 1981).
7. A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (SIAM, Philadelphia, to appear).
8. C. Bischof, A. Carle, P. Hovland, P. Khademi, and A. Mauer, ADIFOR 2.0 user's guide (Revision D), Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1998 (also CRPC Technical Report CRPC-95516-S).
9. A. Griewank, D. Juedes, and J. Utke, ACM Transactions on Mathematical Software **22**, 131 (1996).
10. C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, Scientific Programming **1**, 11 (1992).
11. C. Bischof, A. Carle, P. Khademi, and A. Mauer, IEEE Computational Science & Engineering **3**, 18 (1996).
12. C. Bischof, L. Roh, and A. Mauer, Software–Practice and Experience **27**, 1427 (1997).
13. H. Fischer, in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, edited by A. Griewank and G. F. Corliss, pages 43–50 (SIAM, Philadelphia, Penn., 1991).
14. A. Griewank, C. Bischof, G. Corliss, A. Carle, and K. Williamson, Optimization Methods and Software **2**, 321 (1993).

15. F. Clark, *Optimization and Nonsmooth Analysis* (John Wiley and Sons, New York, 1983).
16. A. Sandu, G. R. Carmichael, and F. A. Potra, Atmospheric Environment **31**, 475 (1997).
17. P. Eberhard and C. Bischof, Mathematics of Computation **68**, 717 (1999).
18. P. Hovland, C. Bischof, D. Spiegelman, and M. Casella, SIAM Journal on Scientific Computing **18**, 1056 (1997).
19. P. Eberhard, in *ICIAM/GAMM 95: Issue 3: Applied Stochastics and Optimization*, edited by O. Mahrenholtz, K. Marti, and R. Mennicken, pages 40–43 (1996), Special Issue of Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM).
20. A. Ibsais and V. Ajjarapu, IEEE Transactions on Power Systems **12**, 592 (1997).
21. C. Bischof, P. Khademi, A. Bouaricha, and A. Carle, Optimization Methods and Software **7**, 1 (1996).
22. J. Czyzyk, M. P. Mesnier, and J. J. Moré, IEEE Computational Science and Engineering **5**, 68 (1998).