Automatic Differentiation and Numerical Software Design

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

Abstract

Automatic differentiation (AD) tools can generate accurate and efficient derivative code for computer programs of arbitrary length. In some cases, however, the developer of the code to be differentiated may be required to provide additional information to an AD tool to ensure the desired solution. We illustrate these issues with nondifferentiable language intrinsics such as `max()` in the context of computing the Euclidean norm and numerical integrators. In both cases, very little additional information is required to ensure that AD computes the "do-what-I-mean" derivatives. In addition, the provision of such information makes it easy to derive "derivative-enhanced" versions of these codes.

Keywords

Automatic Differentiation, Numerical Integrators, Intrinsics, ADIntrinsics, SparsLinC.

1 INTRODUCTION

Automatic differentiation (AD) tools automate the generation of derivatives of "functions" defined by computer programs (see, for example, the book by Rall (1981) or the article by Griewank (1989). Codes generated by AD tools (see `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools` for an overview of currently available AD tools) compute derivatives that are accurate up to machine precision and can be significantly faster than divided-difference approximations. Thus, AD tools offer a convenient mechanism for providing the derivative codes that are needed in the context of numerical schemes for differential equations, optimization, or inverse problems (see, for example, the proceedings volumes edited by Griewank and Corliss (1991) or Berz et al. (1996).

Based on our experience with the ADIFOR (see Bischof et al. (1992,1994)) and ADIC (see Bischof et al. (1996)) tools for automatic differentiation, this article 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. Lastly, we discuss benefits of using AD tools.

2 DEALING WITH INTRINSICS

Automatic differentiation (AD) augments computer programs with statements for the computation of derivatives by exploiting the fact that every program is composed of simple operations such as additions, multiplications, or intrinsic functions, for which derivatives are known (we call such derivatives "elementary derivatives"). For example, an AD tool might transform the statement

```
y = sin(x)
```

into the derivative statement

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

since $\frac{d\sin(x)}{d\,x} = \cos(x)$. Here ∇`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

```
rain = max(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 $\max(x,y)$ function 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 definition would not lead to the desired result if the order of arguments in the `max()` call was reversed, namely,

```
rain = max(0.0,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, in the book by Clarke (1983).

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.
(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, as 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. For example, in translating a call to a `max` intrinsic, the ADIFOR preprocessor might generate a "pseudocall" like

```
call AD_INTRINSIC_FIRST_MAX_S(t, z, r3_v, r1_p, r2_p)
```

which is expected to return the partial derivative values of the result of a binary `max()` call with respect to its first and second argument in the variables `r1_p` and `r2_p`, respectively.

The ADIntrinsics postprocessor is then called to instantiate this pseudocall based on a translation blueprint. For the `max()` intrinsic, the default blueprint is provided in the file `max.T` and is shown in Figure 1. Here `x` and `y` correspond to the first and second arguments, `z` to the result, `fx` and `fy` to the first-order partials with respect to the first and second argument, and `fxx, fxy,` and `fyy` to the second-order partials. In the so-called performance mode, no error handler is called, whereas otherwise, the pseudocall `call EXCEPTION_HANDLER` is replaced by code setting the value of `fx` to a default value and reporting the fact that `max` was invoked at a point where its arguments had the same value.

The user either can change the default values embedded in such a blueprint, or can define alternative blueprints. For example, the library call

```
call ehsups(7,1,my_default_value)
```

effectively defines `fx = my_default_value` in the instantiation of the `EXCEPTION_HANDLER` call. In this fashion, the user can easily obtain all the three choices mentioned before. If

```
      z = max (x,y)
#finish FVAL
      if (x .gt. y) then
         fx = TYPE(1.0)
         fy = TYPE(0.0)
      else if (x .lt. y) then
         fx = TYPE(0.0)
         fy = TYPE(1.0)
      else
#ifdef PERFORMANCE
         fx = TYPE(0.5)
         fy = TYPE(0.5)
#else
         call EXCEPTION_HANDLER
         fy = TYPE(1.0) - fx
#endif
      endif
      fxx = TYPE(0.0)
      fxy = TYPE(0.0)
      fyy = TYPE(0.0)
```

Figure 1 ADIntrinsics Translation Blueprint for `max()` Intrinsic

a change of default values is not sufficient, the user can insert directives into the source code to instruct the ADIntrinsics postprocessor to use a user-supplied translation template instead of the default one. For example, the directive

```
c       AD_EXCEPTION_OVERRIDE_INTRINSIC_ONCE(MAX,MYMAX)
```

will instruct the postprocessor to consult a user-generated file called `mymax.T` instead of the default file `max.T` for the next textual occurrence of a `AD_INTRINSIC_FIRST_MAX` pseudocall. Thus, the ADIntrinics system is an open and complete system for dealing with the intrinsics issue, and because of its standalone nature it can be used by any other AD tool. In fact, our intention was to spare other developers of AD tools the considerable effort that went into the development of this system. The use of the ADIntrinsics system for Fortran 77 intrinsics is described in detail in the ADIFOR user guide (Bischof et al., (1995a)), the design philosophy the paper by Mauer et al. (1996).

To illustrate, let us consider the computation of the Euclidean norm $z = \sqrt{x^2 + y^2}$. A numerically sensible way of doing this is shown in Figure 2. This function is differentiable except for $x = y = 0$. However, automatically differentiating with respect to `x` and `y`, we note that we might attempt to compute the derivatives of `abs()` when its argument is zero, and of `max()` when both its arguments have the same value, even when x and y are not both zero. By default, the ADIntrinsics system would invoke the error handler, which

```
xabs = abs(x)
yabs = abs(y)
w = max(xabs,yabs)
if (w .eq. 0.0) then
   z = 0.0
else
   z = w*sqrt( (xabs/w)**2 + (yabs/w)**2 )
endif
```

Figure 2 Computation of Euclidean Norm with Scaling

```
C     AD_EXCEPTION_BEGIN_IGNORE
      xabs = abs(x)
      yabs = abs(y)
      w = max(xabs,yabs)
C     AD_EXCEPTION_END_IGNORE
      if (w .eq. 0.0) then
         z = sqrt(w)
      else
C     AD_EXCEPTION_LEVEL(PERFORMANCE)
         z = w*sqrt( (xabs/w)**2 + (yabs/w)**2 )
C     AD_EXCEPTION_LEVEL(DEFAULT)
      endif
```

Figure 3 Computation of Euclidean Norm Annotated for Subsequent Automatic Differentiation

would report these exceptions to the user. However, we know that, unless $x = y = 0$, this computation represents a differentiable function and that, independent of the value of `w`, we will obtain the same result. Thus, as shown in Figure 3, we turn off exception reporting via directives, and we trigger an invocation of the ADintrinsics error handler at the point of nondifferentiability by replacing z = 0 with z = sqrt(w). We also know that no point of nondifferentiability can be encountered in the computation of `z` in the "else" branch, so we use the so-called performance mode in this part of the code. Lastly, we reset the exception-handling mechanism to its default state. When translated by an ADIntrinsics-aware AD tool, the generated derivative code will report an exception only at $x = y = 0$.

These examples illustrates that, in general, very little effort is required to deal with the intrinsics issue when the code is developed, while subsequent users will in all likelihood not have the knowledge to deal with these subtle issues in a suitable fashion.

Given: parameter p, current time t, current solution $x_c \approx x(t,p)$,
suggested time step Δt.
1) Compute $x_1 \approx x(t+\Delta t, p)$ using Method 1.
2) Compute $x_2 \approx x(t+\Delta t, p)$ using Method 2.
3) Compute $\delta = \|x_1 - x_2\|$ for some norm $\|\cdot\|$.
4) If $\delta <$ 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 4 Simplified Description of a Numerical Integrator

3 NUMERICAL PARADIGMS

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, Figure 4 shows a simplified version of the time-stepping loop of a typical explicit numerical integrator with stepsize control for a parameter-dependent initial value problem

$$\dot{x}(p) = f(x,p,t), x(t=0) = x_o. \tag{1}$$

Here p is a parameter, and g is some function that adjusts the time step. Methods (1) and (2) are two integration methods of different order. For simplicity, we ignored the fact that the time step will be adjusted upwards if there is a good fit.

If, for a a given p, we are interested in $\frac{\partial x}{\partial p}|_{t=T}$, 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 $\frac{d}{dp}$, the chain rule of differential calculus now implies that

$$\nabla(\Delta t) = \frac{\partial g}{\partial(\Delta t)}\nabla(\Delta t) + \frac{\partial g}{\partial \delta}\nabla\delta. \tag{2}$$

Clearly, $\nabla\delta \neq 0$ in general, as δ depends on x, which in turn depends on p. Thus we have the interesting situation that, when $\frac{\partial g}{\partial \delta} \neq 0$, the computational equivalent of time 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

$$x_T(p) = x(t(p), p)|_{t(p)=T} \tag{3}$$

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

$$\nabla x_{t=T} = \frac{\partial x}{\partial t}|_{t=T} \nabla t_{t=T} + \frac{\partial x}{\partial p}, \tag{4}$$

and with (1)

$$\nabla x_{t=T} = f(x_T, p, T) \nabla t_{t=T} + \frac{\partial x}{\partial p}|_{t=T}. \tag{5}$$

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 $\frac{\partial x}{\partial p}|_{t=T}$ which is the result desired by most users.

(ii) If Δt would have been zero at every step, we would have $\nabla t_{t=T} = 0$ and thus $\nabla x_{t=T} = \frac{\partial x}{\partial p}|_{t=T}$, as desired by the user. By default, this happens in methods using a fixed step size. This case is also discussed in the paper by Sandu at al. (1995).

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

$$\frac{\partial x}{\partial p}|_{t=T} = \nabla x_{t=T} - f(x_T, p, T) \nabla t_{t=T}. \tag{6}$$

These issues are discussed in more detail in the forthcoming paper by Eberhard and Bischof (1996).

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 really 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 in the forthcoming paper by Eberhard

and Bischof (1996), 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 only governed by the behavior of x. Thus, the derivatives obtained by (2) or (3) will be consistent, but they may not be as accurate as those obtained by solving the sensitivity equations ($x_p = \frac{\partial x}{\partial p}$)

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

alongside the original ODE (1). It is easy to add the norm of $\nabla\delta$ to the guard for stepsize 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 in the paper by Griewank et al. (1993).

4 CONCLUDING REMARKS

The preceding sections may suggest that AD tools are mainly an additional burden for numerical software developers. However, AD tools can greatly simplify software interfaces that require derivatives. While many numerical codes currently provide an option for the user to provide his own routine for differentiation, the integration of an AD tool can facilitate the process (see, for example, the user's guide by Liu and Tits (1996)). In addition to accurate derivatives, AD tools can also provide, in a fashion that is transparent to the user, information about the zero/nonzero structure of derivative matrices (see Bischof et al. (1995b)). That is, for a vector-valued function $F : x \mapsto y$, we can compute both the value and the nonzero structure of of $\frac{dF}{dx}|_{x=x_o}$, for arbitrarily chosen values x_o. 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, which is described by Mesnier (1995) and accessible at URL `http://www.mcs.anl.gov/home/otc/index.html`.

AD is intended to save work (for handcoding of derivatives) and avoid hassle (caused by numerical difficulties due to inaccurate derivatives). Even though AD tools are still in their infancy, they already can compute derivatives faster than divided difference approximations (see the references in the ADIFOR 2.0 paper (Bischof et al., (1994)), and there are examples where the availability of fully accurate derivatives was essential for numerical robustness and convergence (see, for example, the papers by Hovland et al. (1995), Eberhard (1996), and Ibsais and Ajjarapu (1996)). By taking AD considerations into account in the development of their software, library developers can easily develop "sensitivity-enhanced" versions of their codes using AD tools. Some needed features (such

as intrinsics handling) are already supported; others (such as selective disabling of differentiation or the automatic insertion of code that uses derivatives) are still being discussed. The AD tool developers community is dependent on feedback by potential users to provide the right extensions.

If AD is kept in mind when writing software, numerical software developers can easily enhance the functionality of their software by providing derivative-enhanced versions of their codes as well. We believe this to be a considerable bonus, since this feature may greatly enhance the potential usability of this software, for example when a program requiring an integrator solver is ultimately to be embedded in an inverse problem or optimization context. However, AD needs to be kept in mind when developing codes, and interaction with developers of AD tools is needed to arrive at mutually satisfactory solutions.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under contract W-31-109-Eng-38. This work was partially completed while the author was visiting the Institute of Scientific Computing, ETH Zürich, Switzerland.

REFERENCES

Berz, M., Bischof, C., Corliss, G., and Griewank, A. (1996). *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia. To appear.

Bischof, C., Carle, A., Corliss, G., Griewank, A., and Hovland, P. (1992). ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29.

Bischof, C., Carle, A., Khademi, P., and Mauer, A. (1994). The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs. Preprint MCS-P481-1194, Mathematics and Computer Science Division, Argonne National Laboratory. Also Technical Report CRPC-TR94491, Center for Research on Parallel Computation, Rice University.

Bischof, C., Carle, A., Khademi, P., Mauer, A., and Hovland, P. (1995a). ADIFOR 2.0 user's guide (Revision C). Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory. Also Technical Report CRPC-95516-S, Center for Research on Parallel Computation, Rice University.

Bischof, C., Khademi, P., Bouaricha, A., and Carle, A. (1995b). Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation. Preprint MCS-P519-0595, Mathematics and Computer Science Division, Argonne National Laboratory. Also Technical Report CRPC-TR95583, Center for Research on Parallel Computation, Rice University. Accepted for publication in Optimization Methods and Software.

Bischof, C., Roh, L., and Mauer, A. (1996). ADIC — A tool for the automatic differentiation of C programs. Private Information.

Clark, F. (1983). *Optimization and Nonsmooth Analysis*. John Wiley and Sons, New York.

Eberhard, P. (1996). Analysis and optimization of complex multibody systems using advanced sensitivity analysis methods. ICIAM/GAMM 95: Issue 3: Applied Stochastics and Optimization, pages 40–43. Special Issue of Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM).

Eberhard, P., and Bischof, C. (1996). Automatic differentiation of numerical integration algorithms. Private Information.

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

Griewank, A., Bischof, C., Corliss, G., Carle, A., and Williamson, K. (1993). Derivative convergence of iterative equation solvers. *Optimization Methods and Software*, 2:321–355.

Griewank, A., and Corliss, G. (1991). *Automatic Differentiation of Algorithms*. SIAM, Philadelphia.

Hovland, P., Bischof, C., Spiegelman, D., and Casella, M. (1995). 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. To appear in SIAM J. Scientific Computing.

Ibsais, A., and Ajjarapu, V. (1996). The role of automatic differentiation in power system analysis. In *IEEE PES Winter Power Meeting, Paper No. 96 WM 328-5 PWRS, Baltimore, Maryland, January 21-25.*

Liu, M. D., and Tits, A. L. (1996). User's guide for ADIFFSQR version 1.0.

Mauer, A., Bischof, C., and Carle, A. (1996). The ADIntrinsics system for handling automatic differentiation exceptions. Private Information.

Mesnier, M. P. (1995). The network-enabled optimization system server. Technical Memorandum ANL/MCS-TM-210, Mathematics and Computer Science Division, Argonne National Laboratory.

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

Sandu, A., Carmichael, G. R., and Potra, F. A. (1995). Sensitivity analysis for atmospheric chemistry models via automatic differentiation. Technical Report 73, Dept. of Mathematics, University of Iowa.