An introduction to automatic differentiation

Arun Verma*
Computer Science Department and Cornell Theory Center, Cornell University, Ithaca NY 14850, USA

1. Introduction

The field of computational science includes problems ranging from modeling physical phenomena, computation of option pricing in finance, and optimal control problems, to inverse problems in medical imaging and geosciences. The solution of problems in a variety of areas in computational science often involves presenting the problem as a numerical optimization problem which in turn requires computing derivatives of numerical functions. In numerical optimization derivatives, usually in the form of gradients, Jacobians and Hessians, are used to locate the extrema of a function; most optimization software include some way of computing the derivatives (exactly or approximately) if not provided by the user.

Automatic differentiation (AD) is an upcoming technology which provides software for automatic computation of derivatives of a general function provided by the user. There are many AD tools which are out, including ADOL-C for C/C++ functions, ADIFOR for FORTRAN and ADMIT-1 and ADMAT for MATLAB. AD is a chain-rule-based technique for evaluating the derivatives with respect to the input variables of functions defined by a high-level language computer program. AD relies on the fact that all computer programs, no matter how complicated, use a finite set of elementary (unary or binary, e.g. sin(·), sqrt(·)), operations as defined by the programming language. The value or function computed by the program is simply a composition of these elementary functions. The partial derivatives of the elementary functions are known, and the overall derivatives can be computed using the chain rule; this process is known as AD\textsuperscript{++}.

Abstractly, the program to evaluate the solution $y$ (an $m$-vector) as a function of $x$ (generally an $n$-vector) has the form:

$$x \equiv (x_1, x_2, \ldots, x_n),$$
$$\downarrow$$
$$z \equiv (z_1, z_2, \ldots, z_p), \quad p > m + n,$$
$$\downarrow$$
$$y \equiv (y_1, y_2, \ldots, y_m),$$

where the intermediate variables $z$ are related through a series of these elementary functions which may be unary,

$$z_k = f_{\text{elem}}^k(z_i), \quad i < k,$$

consisting of operations such as (–, pow(·), sin(·), …) or binary,

$$z_k = f_{\text{elem}}^k(z_i, z_j), \quad i < k, j < k,$$

such as (+, ·, …).

In general, the number of intermediate variables is much larger than the dimensions of the problem, i.e. $p \gg m, n$.

AD has two basic modes of operations, the forward mode and the reverse mode. In the forward mode the derivatives are propagated throughout the computation using the chain rule, e.g. for the elementary step $z_k = f_{\text{elem}}^k(z_i, z_j)$ the intermediate derivative, $dz_k/dx$, can be propagated in the forward mode as:

$$\frac{dz_k}{dx} = \frac{\partial f_{\text{elem}}^k}{\partial z_i} \frac{dz_i}{dx} + \frac{\partial f_{\text{elem}}^k}{\partial z_j} \frac{dz_j}{dx}.$$

This chain rule-based computation is done for all the

*e-mail: verma@CS.Cornell.EDU
intermediate variables \( z \) and for the output variables \( y \), finally yielding the derivative \( dy/dx \).

The reverse mode computes the derivatives \( dy/dz_k \) for all intermediate variables backwards (i.e. in the reverse order) through the computation. For example, for the elementary step

\[ z_k = f^k_{\text{elem}} \text{(the derivative) are propagated as:} \]

\[
\frac{dy}{dz_i} = \frac{\partial f^k_{\text{elem}}}{\partial z_i} \frac{dy}{dz} \quad \text{and} \quad \frac{dy}{dz_j} = \frac{\partial f^k_{\text{elem}}}{\partial z_j} \frac{dy}{dz}.
\]

At the end of computation of the reverse mode the derivative \( dy/dx \) will be obtained. The key is that the derivative propagation is done in reverse manner, hence, you need \( du/dz_k \) in order to compute derivatives \( du/dz_i, du/dz_j \). Initially, \( du/dx \) is initialized to 1.

The reverse mode requires saving the entire computation trace, since the propagation is done backwards through the computation, and hence, the partials

\[
\frac{\partial f^k_{\text{elem}}}{\partial z_j}, \quad \frac{\partial f^k_{\text{elem}}}{\partial z_i}
\]

need to be stored for derivative computation as shown above. Hence the reverse mode can be prohibitive due to memory requirements.

In summary, given a function computation, \( F(x): \mathbb{R}^n \rightarrow \mathbb{R}^m \), the forward mode can compute the Jacobian-matrix product, \( J*V \), and the reverse mode can compute the adjoint product, \( J^T*W \), where \( J = J(x) \) is the Jacobian matrix, \( dF/dx \). Here \( V \in \mathbb{R}^m \times p_1 \), \( W \in \mathbb{R}^n \times p_2 \), where \( p_1 \) and \( p_2 \) are number of columns in \( V \) and \( W \) respectively.

3. AD in action

Consider the following simple program (Figure 1), which computes \( f(x) = (x + x^2)^2 \).

The user function does not need to be in the binary form as showed in Figure 1, it can be any code (say in C or MATLAB or your favorite programming language) and the AD tool will internally view it as a sequence of binary code. For example, you can include complicated code such as

\[ y = \sqrt{\sin(A + \log(B + \cos(C + \tanh(D))))} \].

The AD tool will internally break it up in the binary form as:

\[ t1 = \tanh(D), \]
\[ t2 = \cos(C + t1), \]
\[ t3 = \log(B + t2), \]
\[ t4 = \sin(A + t3), \]
\[ y = \sqrt{t4}. \]

3.1. AD in forward mode

AD can be seen as simply augmenting the function with the derivative statements, as shown in the following derivative function in Figure 2. This is just a view of the sample AD tool output, the user will not need to see this view and this is just to illustrate the working of the AD tool on the sample code above. In the AD tool, the augmented derivative statements are carried out computationally and not physically inserted in the code.

![Figure 2. AD of the sample function.](image-url)

In the above program \( \bullet \) stands for the derivative, \( du/dx \). Since \( x \) is independent, \( \tau = (dx/dx) + 1 \). In general \( \tau \) stands for the tangent direction. Consider the input, \( x = 2, \tau = 1 \), the program returns:

\[
\begin{aligned}
z &= x*x = 2*2 = 4, \\
t &= 2*x*x = 2*2*1 = 4, \\
w &= x + z = 2 + 4 = 6, \\
\nu &= \tau + t = 1 + 4 = 5, \\
y &= w*w = 6*6 = 36, \\
\psi &= 2*w*\nu = 2*6*5 = 60.
\end{aligned}
\]

The function of an AD tool is illustrated best using a flowchart as shown in Figure 3 corresponding to the simple MATLAB program in Figure 1. We have given simple scalar values to the variables for the purpose of illustration. The values of the derivatives are propagated along with the values of the variables as shown in the flowchart. The value variables are represented by \( x, z, w, y \) and the derivatives by \( dx/dx, dz/dx, dw/dx, dy/dx \).
3.2. *AD in reverse mode*

In the reverse mode, the flow of the computation is reversed. The AD tool functions in the manner as shown in Figure 4. After the full function computation, the values of \( x, w, z \) are saved and used in the reverse computation of derivative.

Here \( \overline{y} \) stands for the derivative \( dy/du \), hence \( \overline{y} = dy/dy = 1 \). In general \( \overline{y} \) could be a general adjoint direction. If we input the same value of \( x = 2 \), and an initial derivative \( \overline{y} = 1 \), we get the following computation:

\[
\begin{align*}
\overline{w} &= 2 \times 2 \times \overline{y} = 2 \times 6 \times 1 = 12, \\
\overline{x} &= \overline{w} = 12, \\
\overline{z} &= \overline{x} + 2 \times x \times \overline{z} = 12 + 2 \times 2 \times 12 = 60.
\end{align*}
\]

Hence, you get the same answer with the reverse mode as well.

### 4. Complexity of AD

One key advantage of AD is that it allows an a priori bound on the cost of evaluating certain derivative objects in terms of the cost for evaluating the function itself. Consider a general nonlinear \( F(x): \mathbb{R}^n \rightarrow \mathbb{R}^m \). Let \( \alpha(\cdot) \) denote the temporal complexity or computational cost to carry out a certain operation and \( S(\cdot) \) denote the spatial (memory) complexity.

Cost of basic forward and reverse mode

- **Forward mode**: \((x, \forall x \in \mathbb{R}^m) \rightarrow (F(x), JF)\)
  
  Work cost: \( \alpha F, JF = p F, \alpha F \)

- **Reverse mode**: \((x, \forall x \in \mathbb{R}^m) \rightarrow (F(x), J^T W)\)
  
  Work cost: \( \alpha F, J^T W = p F, \alpha F \)

Space cost: \( S(F, JF) = S(F) \).

\( \alpha F, \nabla f = 5 \alpha f \), \( S(f, \nabla f) = \text{numIvars}(f) \).

#### 4.1. AD vs finite difference

In this section we illustrate the key differences between AD and finite differences. AD computes the derivatives exactly (up to machine precision) while finite differences incur truncation errors. The size of the step needed for finite difference \((h)\) varies with the current value of \( x \) (the independents) making the problem of choosing \( h \) very difficult. AD on the other hand, is automatic and time need not be spent in choosing step-size parameters, etc. Also, a caveat is that, if the function computation itself is not accurate (i.e. it has roundoff errors), these will appear in the AD process as well (ditto for stability questions). If the function computation is accurate and numerically stable then so will be the AD process.

AD is also traditionally faster than finite difference, since...
AD can take advantage of the problem structure (description of problem structure and its advantages is outside the scope of this article)\textsuperscript{3,4}.

For example, consider a simple MATLAB computation of inverting a $200 \times 200$ dense matrix. The function computation takes 0.46 s on a SUN Ultra SPARC workstation. AD takes a total of 0.78 s to compute the derivative of the inverse (w.r.t. one independent), while finite difference takes 0.92 s (basically equivalent to two function computations).

5. Extensions and summary

We have illustrated the working of the bare-bone AD tool with some examples. In many problems, working with this bare-bone functionality is not adequate; e.g. often the Jacobian matrices associated with large-scale nonlinear problems are sparse, which requires a layer of sparsity exploitation technology above AD. The Bi-coloring method by Coleman and Verma is a very efficient way to compute sparse derivative matrices\textsuperscript{8}.

Many large-scale optimization applications (e.g. inverse problems) are very complex in nature. It becomes impractical to consider the function evaluation of such problems as a ‘black-box’ function since the computation is structured in some manner, going through a set of defined structured steps, i.e. problem structure. It pays to expose the problem structure in the computation to be able to compute the derivatives efficiently, thus making the problem solution practical\textsuperscript{3–5}.

AD technology has been applied to a variety of applications, in particular some recent work has been in the computational finance area\textsuperscript{12}, seismic inversion\textsuperscript{13}, and shape optimization\textsuperscript{14}.

In general, the full framework of the AD technology should be rightly seen as a layered view shown in Figure 5. AD forms the backbone of the computational ladder shown, driven from the top with real-world problems. Often, the real-world problems are translated to an optimization sub-problem. The AD tools allow fast solution to the optimization problem by potentially exploiting the sparsity (if there is sparsity in Jacobian or Hessian matrices, they can be computed efficiently, see ref. 2) or problem structure for a practical and painless solution of the application at hand.

ACKNOWLEDGEMENTS. This work was partially supported by the Cornell Theory Center which receives funding from Cornell University, New York State, the National Center for Research Resources at the National Institutes of Health, the Department of Defense Modernization Program, the United States Department of Agriculture, and members of the Corporate Partnership Program and the National Science Foundation under the grant DMS-9704509.