TUTORIALS

An introduction to automatic differentiation

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

Differentiation is one of the fundamental problems in numerical mathematics. The solution of many optimization problems and other applications require knowledge of the gradient, the Jacobian matrix, or the Hessian matrix of a given function.

Automatic differentiation (AD) can compute fast and accurate derivatives of any degree computationally via propagating Taylor series coefficients using the chain rule. AD does not incur any truncation error and would yield exact results if the calculations were done in real arithmetic; in other words the derivatives obtained are accurate to machine precision.

In this tutorial we uncover the details of the AD technology, presenting them in a simple manner. We present basics of AD and its complexity followed by some examples.

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)[1–6] 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[7], ADIFOR for FORTRAN[8] and ADMIT-1 and ADMAT for MATLAB[9,10].

The outline of this tutorial is as follows: In §1, we present some basics of AD. In §2, we present illustrative examples of the working of AD in both forward and reverse modes. In §3, we outline the complexity of AD in the reverse and forward mode. We conclude with a summary and an overview of some extensions to the basic AD technology.

*e-mail: verma@CS.Cornell.EDU

2. Basics of AD

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[1,11].

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

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

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

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

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

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

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 $z_k = f^k_{\text{elem}}(z_i, z_j)$ the intermediate derivative, dz_k/dx, can be propagated in the forward mode as:

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

This chain rule-based computation is done for all the 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 $z_k = f^k_{\text{elem}}(z_i, z_j)$, the derivatives are propagated as:

$$\frac{dy}{dz_i} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{dy}{dz_k} \text{ and } \frac{dy}{dz_j} = \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{dy}{dz_k}.$$

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/du 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

$$\frac{\partial f^k_{\text{elem}}}{\partial z_j}, \frac{\partial f^k_{\text{elem}}}{\partial z_i}$$

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, $F(x)$: $\Re^n \rightarrow \Re^m$, 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 $V \in \Re^{n \times p_1}$, $W \in \Re^{m \times p_2}$, 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 $f(x) = (x + x^2)^2$.

```
function y = f(x)
  z = x*x;
  w = x + z;
  y = w*w;
end
```

Figure 1. A sample function.

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 $y = \text{sqrt}(\sin(A + \log(B + \cos(C + \tanh(D)))))$. The AD tool will internally break it up in the binary form as:

$t1 = \tanh(D)$,

$t2 = \cos(C + t1)$,

$t3 = \log(B + t2)$,

$t4 = \sin(A + t3)$,

$y = \text{sqrt}(t4)$.

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.

```
function (y, ẏ) = f'(x, ẋ)
  z = x*x;
    ż = 2*x*ẋ;
  w = x + z;
    ẇ = ẋ + ż;
  y = w*w;
    ẏ = 2*w*ẇ;
end
```

Figure 2. AD of the sample function.

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

$$z = x*x = 2*2 = 4,$$
$$\dot{z} = 2*x*\dot{x} = 2*2*1 = 4,$$
$$w = x + z = 2 + 4 = 6,$$
$$\dot{w} = \dot{x} + \dot{z} = 1 + 4 = 5,$$
$$y = w*w = 6*6 = 36,$$
$$\dot{y} = 2*w*\dot{w} = 2*6*5 = 60.$$

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

$$\bar{w} = 2 * w * \bar{y} = 2 * 6 * 1 = 12,$$
$$\bar{x} = \bar{w} = 12, \quad \bar{z} = \bar{w} = 12,$$
$$\bar{x} = \bar{x} + 2 * x * \bar{z} = 12 + 2 * 2 * 12 = 60.$$

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 $F(x)$: $\Re^n \rightarrow \Re^m$. Let $\omega(\cdot)$ 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: $(x, V \in \Re^{n \times p_1}) \rightarrow (F(x), JV)$

  Work cost: $\omega(F, JV) = p_1 \cdot \omega(F)$

  Space cost: $S(F, JV) = S(F)$.
- Reverse mode: $(x, W \in \Re^{m \times p_2}) \rightarrow (F(x), J^T W)$

  Work cost: $\omega(F, J^T W) = p_2 \cdot \omega(F)$

  Space cost: $S(F, J^T W) = numIvars(F)$.

numIvars(F) represents the total number of intermediate variables generated in the computation of F. All of these variables need to be saved for the reverse mode operation.

One special case of the reverse mode is gradient evaluation, where we can use $W = 1$ (a scalar) to compute the gradient $\nabla f(x) \equiv J^T$ at a cost proportional to $\omega(F)$; the constant in front can be shown to be about 5 in practice. This is also known as the **cheap gradient result**. However, this does require space proportional to *numIvars*(F) which can be prohibitive. Note however that the forward mode costs $n \cdot \omega(F)$.

Hence in the reverse mode

$$\omega(f, \nabla f) = 5 \cdot \omega(f), \quad S(f, \nabla f) = numIvars(f).$$

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[1].

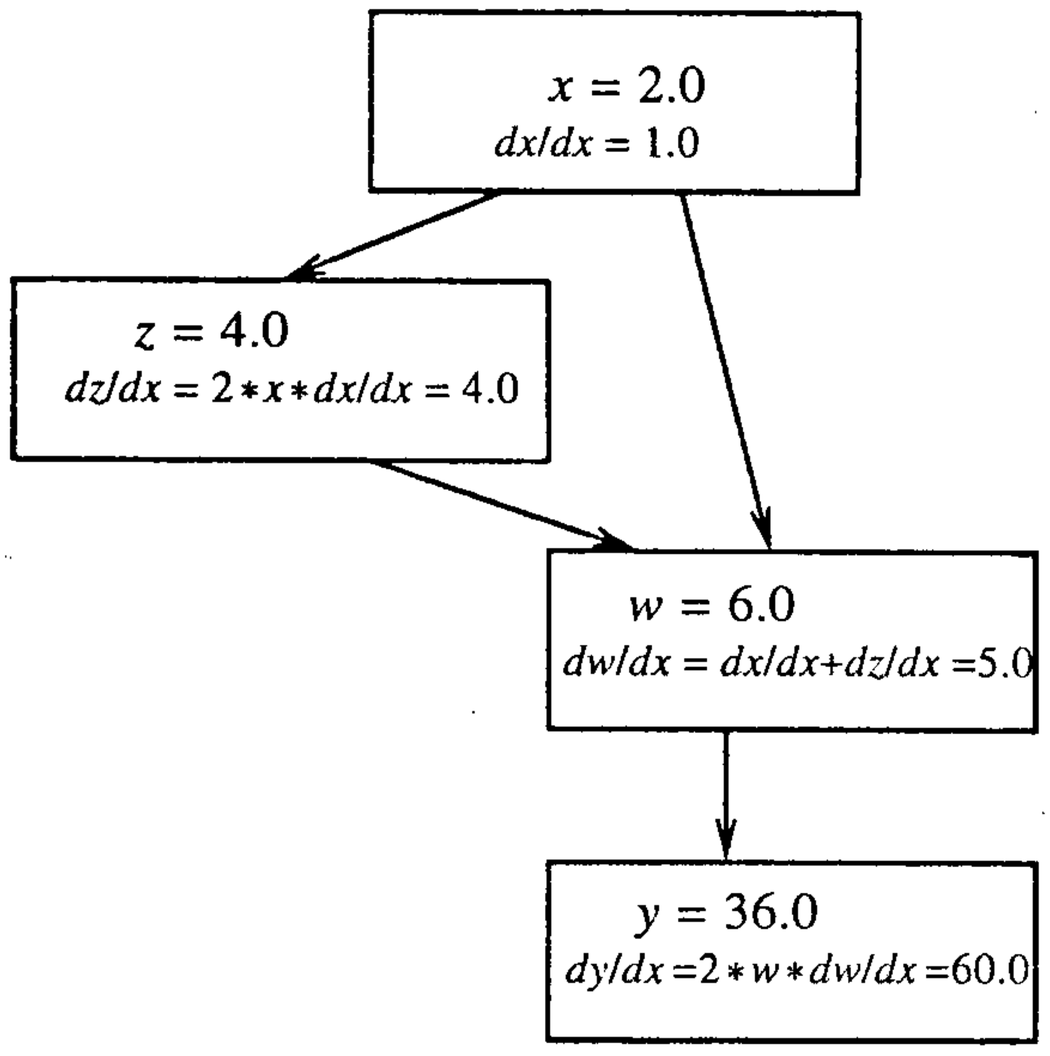

Figure 3. Flowchart corresponding to the sample function in Figure 1.

function $(\bar{x}) = \bar{f}(x, \bar{y})$

$z = x * x$; (1)

$w = x + z$; (2)

$y = w * w$; (3)

// Reverse the flow for derivative computation

$\bar{w} = 2 * w * \bar{y}$; (from 3)

$\bar{x} = \bar{w}$; $\bar{z} = \bar{w}$; (from 2)

$\bar{x} = \bar{x} + 2 * x * \bar{z}$; (from 1)

end

Figure 4. AD in the sample function.

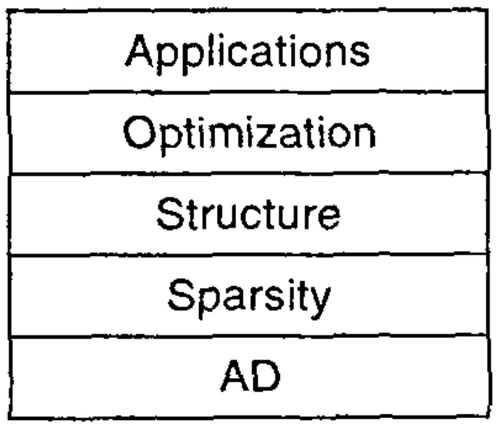

Figure 5. Layered view.

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)[3,4].

For example, consider a simple MATLAB computation of inverting a 200×200 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 nónlinear 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[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[3–5].

AD technology has been applied to a variety of applications, in particular some recent work has been in the computational finance area[12], seismic inversion[13], and shape optimization[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 subproblem. 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.

1. Griewank, A., in *Complexity in Nonlinear Optimization* (ed. Pardalos, P.), World Scientific, Singapore, 1993, pp. 128–161.
2. Coleman, T. F. and Verma, A., *SIAM J. Sci. Comput.*, 1998, **19**, 1210–1233.
3. Coleman, T. F. and Verma, A., in *Computational Differentiation: Techniques, Applications and Tools* (eds Berz, M., Bischof, C., Corliss, G. and Griewank, A.), SIAM, Philadelphia, 1966, pp. 149–159.
4. Coleman, T. F. and Verma, A., Proceedings of 96th International Conference on Nonlinear Programming (ed. Yuan, Y.-X.), Kluwer Academic Publishers, Boston, 1996, pp. 55–72.
5. Coleman, T. F., Santosa, F. and Verma, A., in *Computational Methods for Optimal Design and Control* (eds Borggaard, J., Burns, J., Cliff, E. and Schreek, S.), Birkhauser, 1977, pp. 113–126.
6. Verma, A., Ph D thesis, Department of Computer Science, Cornell University, NY, 1988.
7. Griewank, A. Juedes, D. and Utke, J., *ACM Trans. Math. Software*, 1966, **22**, 131–167.
8. Bischof, C. H., Carle, A., Corliss, G. F., Griewank, A. and Hovland, P., *Sci. Program.*, 1992, **1**, 11–29.
9. Coleman, T. F. and Verma, A., *ACM TOMS*, 1998 (submitted).
10. Coleman, T. F. and Verma, A., Proceedings of SIAM workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Philadelphia, 1998.
11. Griewank, A., in *Large Scale Numerical Optimization* (eds Coleman, T. F., Li, Y.), SIAM, Philadelphia, 1990, pp. 115–137.
12. Coleman, T. F., Li, Y. and Verma, A., *J. Comput. Finan.*, 1999.
13. Coleman, T. F., Santosa, F. and Verma, A., *J. Comput. Phys.*, 1999; Also appeared as IMA preprint series #1579, June 1998, University of Minnesota and Cornell Theory Center, CTC9803.
14. Borggaard, J. and Verma, A., *SIAM J. Sci. Comput.*, 1999 (submitted).

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 Defence 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.