Algorithmic Differentiation

Math 337 Final Project Handout

Andy Reagan

April 28, 2014

Contents

1 Introduction

The key question that we attempt to answer when modeling real-world phenomena is: how sensitive is the output of model with respect to the input values and parameters? Most of the methods that we have developed in the class rely on finite differences to answer this question, and hence to make accurate predictions. We judge the accuracy of these methods based on how well they approximate the Taylor series of the given function, giving rise to their local truncation error. Accuracy is not the sole desired property of a numerical scheme, but once convergence to the true solution can be achieved, we seek the most efficient and most accurate determination of this solution.

A promising tool to answer the above question, with arbitrary accuracy, support for extended precision arithmetic, and the ability to apply to comlex, large-scale models is Algorithmic Differentiation. My presentation today outlines the basics of Algorithmic Differentation (AD) technology. Some of the literature refers to AD as Automatic Differentiation, but in my experience the differentiation is still far from automatic.

The core functionality of AD relies upon the fact that any computer program can be broken down into a sequence of elementary operations, for which the derivative can be computed easily. One limitation of this approach is that we are restricted to a range of functions that can be broken down this way, although we will see that this is not as limiting in practical implementations. There are many ways to potentially compute derivatives algorithmically, and we will discuss source code transformation, derivative compilers, and operator overloading techniques.

1.1 Taylor Method

Going back to Day 1, we are interesting in solving problems of the form of Eq (0.1) of the notes:

$$y'(x) = f(x, y).$$

Instead of taking the finite difference, we next consider the Taylor series expansion:

$$\begin{aligned} f(x,y) = f(x_0,y_0) &+ \left(\Delta x f_x(x_0,y_0) + \Delta y f_y(x_0,y_0)\right) \\ &+ \frac{1}{2!}\Big((\Delta x)^2 f_{xx}(x_0,y_0) + 2\Delta x \Delta y f_{xy}(x_0,y_0) + (\Delta y)^2 f_{yy}(x_0,y_0)\Big) + \dots \end{aligned}$$

The computation of the forms of f_x, f_y and so forth requires significantly effort from a computer, and will typically be cumbersome. Therefore, evaluating them will be slow. Algorithm differentiation overcomes this difficulty. We will discuss this in more detail in the following section.

2 Algorithmic Differentiation Basics

First I will note other possibilities for computing derivatives and jacobians, namely: symbolic differentiation, hand-coding, and divided differences. In line with the goals of Bischof and Bucker (2000), we seek to evaluate derivatives in a way that has the following properties:

- **Reliability** The computed derivatives should be accurate to the IEEE floating point arithmetic of the computer.
- **Computational Cost** Computing the derivatives is often the main computational cost. The amount of memory and operations should be bounded.
- **Scalability** The approach should give correct results for a 1-line formula as well as a 100,000-line code.
- **Human Effort** A user should not spend much time preparing a code for differentiation, in particular where code changes 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 Bischof and Bucker (2000).

To simplify the discussion, I adopt the notation of **?**. If $a : t \in I \subset \mathbb{R} \to \mathbb{R}$ is a smooth function, denote its normalized n-th derivative to the value

$$a^{[n]}(t) = \frac{1}{n!}a^{(n)}(t).$$

Assume now that $a(t) = F(b(t), c(t))$ and that we know the values of $b^{[j]}(t)$ and $c^{[j]}(t)$, $j = 0, \ldots, n$ for a given t. I will prove the following on the board:

Proposition 2.1. *If the function a and b are of class C^n, and $\alpha \in \{\mathbb{R} - 0\}$, we have*

1. *If $a(t) = b(t) \pm c(t)$, then $a^{[n]}(t) = b^{[t]}(t) \pm c^{[t]}(t)$.*
2. *If $a(t) = b(t)c(t)$, then $a^{[n]}(t) = \sum_{j=0}^{n} b^{[n-j]}(t)c^{[j]}(t)$.*

3. *If* $a(t) = \frac{b(t)}{c(t)}$, *then* $a^{[n]}(t) = \frac{1}{c^{[0]}(t)}\left[b^{[n]}(t) - \sum_{j=1}^{n} c^{[j]}(t)a^{[n-j]}(t)\right]$.

4. *If* $a(t) = b(t)^{\alpha}$, *then* $a^{[n]}(t) = \frac{1}{nb^{[0]}(t)}\sum_{j=0}^{n-1}(n\alpha - j(\alpha+1))\, b^{[n-j]}(t)a^{[j]}(t)$.

5. *If* $a(t) = e^{b(t)}$, *then* $a^{[n]}(t) = \frac{1}{n}\sum_{j=0}^{n-1}(n-j)\, b^{[n-j]}(t)a^{[j]}(t)$.

6. *If* $a(t) = \ln b(t)$, *then* $a^{[n]}(t) = \frac{1}{b^{[0]}(t)}\left[n^{[n]}(t) - \frac{1}{n}\sum_{j=1}^{n-1}(n-j)\, b^{[j]}(t)a^{[n-j]}(t)\right]$.

7. *If* $a(t) = \cos c(t)$ *and* $b(t) = \sin c(t)$, *then*

$$a^{[n]}(t) = -\frac{1}{n}\sum_{j=1}^{n} jb^{[n-j]}(t)c^{[j]}(t), \qquad b^{[n]}(t) = \frac{1}{n}\sum_{j=1}^{n} ja^{[n-j]}(t)c^{[j]}(t).$$

Remark It is possible to derive similar forumulas for other functions, line inverse trigonometric functions.

Corollary 2.2. *The number of arithmetic operations to evaluate the normalized derivation of a function up to order* n *is* $O(n^2)$.

I will prove this on the board as well.

2.1 Short Example

I base my example on that of **?**, and apply the rules of Proposition 2.1 to the Van der Pol equation:

$$\begin{aligned} x' &= y \\ y' &= (1-x^2)y - x \end{aligned}$$

We decompose this function in the following elementary operations:

$$\begin{aligned}
u_1 &= x \\
u_2 &= y \\
u_3 &= u_1 u_1 \\
u_4 &= 1 - u_3 \\
u_5 &= u_2 u_4 \\
u_6 &= u_5 - u_1 \\
x' &= u_2 \\
y' &= u_6
\end{aligned}$$

And applying Proposition 2.1 to the above, we have

$$\begin{aligned}
u_1^{[n]}(t) &= x^{[n]}(t) \\
u_2^{[n]}(t) &= y^{[n]}(t) \\
u_3^{[n]}(t) &= \sum_{i=0}^{n} u_1^{[n-i]}(t) u_1^{[i]}(t) \\
u_4^{[n]}(t) &= 1 - u_3^{[n]}(t) \\
u_5^{[n]}(t) &= \sum_{i=0}^{n} u_2^{[n-i]}(t) u_4^{[i]}(t) \\
u_6^{[n]}(t) &= u_5^{[n]}(t) - u_1^{[n]}(t) \\
x^{[n+1]}(t) &= \frac{1}{n+1} u_2^{[n]}(t) \\
y^{[n+1]}(t) &= \frac{1}{n+1} u_6^{[n]}(t)
\end{aligned}$$

3 Using Taylor 1.4

4 Comparison to Symplectic Methods

5 Conclusions

References

Bischof, C. and H. M. Bucker (2000). Computing derivatives of computer programs. *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition, NIC Series 3*, 315–327.