Abstract

An introduction to both automatic differentiation and object-oriented programming can enrich a numerical analysis course that typically incorporates numerical differentiation and basic MATLAB computation. Automatic differentiation consists of exact algorithms on floating-point arguments. This implementation overloads standard elementary operators and functions in MATLAB with a derivative rule in addition to the function value; for example, $\sin u$ will also compute $(\cos u) * u'$ where u and u' are numerical values. These methods are mostly one-line programs that operate on a class of value-and-derivative objects, providing a simple example of object-oriented programming in MATLAB using the new (as of release 2008a) class definition structure. The resulting powerful tool computes derivative values and multivariable gradients, and is applied to Newton's method for root-finding in both single and multivariable settings. To compute higher-order derivatives of a single variable function, another class of series objects keep Taylor polynomial coefficients up to some order. Overloading multiplication on series objects is a combination (discrete convolution) of coefficients. This idea leads to algorithms for other operations and functions on series objects. A survey of more advanced topics in automatic differentiation includes an introduction to the reverse mode (our implementation is forward mode) and considerations in arbitrary-order multivariable series computation.

Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming

Richard D. Neidinger
Davidson College

October 15, 2009

1 Introduction

Automatic Differentiation (AD) uses exact formulas with floating-point values, not expression strings as in symbolic differentiation. There is no approximation error as in numerical differentiation using difference quotients. AD is a third alternative, also called computational differentiation or algorithmic differentiation. Though the idea often sounds paradoxical at first, students (and instructors) in a first numerical analysis course using MATLAB can quickly come to understand and apply the method. They will appreciate its practical power to supply accurate numerical values of derivatives, gradients, Jacobians, and Taylor polynomials, that can be used in other numerical methods, such as Newton's method.

The key idea is that basic derivative rules from calculus, such as the chain rule, can be implemented in a numerical environment. For example, suppose you evaluate $y = x\sin(x^2)$ at $x = 3$. Your calculating device will (internally) compute the sequence of values on the left of the table below, but it is also possible to have it compute the derivative values on the right.

$x = 3$	$x' = 1$
$y_1 = x^2 = 9$	$y_1' = 2x\,x' = 6$
$y_2 = \sin(y_1) = 0.4121$	$y_2' = \cos(y_1)\,y_1' = -5.46668$
$y = x\,y_2 = 1.2363$	$y' = x'\,y_2 + x\,y_2' = -15.9883$

We want the derivative computations to automatically happen when when evaluating or interpreting $x\sin(x^2)$. An ambitious approach would be a program that reads $x\sin(x^2)$ and produces augmented code that includes all of the statements in the table, or something equivalent. This approach, called *source transformation*, is one of the ways in which AD packages are implemented. The advantage is enabling efficiencies as discussed in Section 7. The disadvantage is that programming becomes as complicated as implementing symbolic differentiation, a task better suited to advanced computer science than to an introductory numerical analysis class. However, there is a simpler alternative.

Object-Oriented Programming (OOP) comes to the rescue by allowing us to define x to be an object that has both the value 3 and the derivative value 1. Then, surprisingly simple programming extends the definition of operators, such as ^, sin and $*$, so that they do both value and derivative computations. The implementation in Section 2 shows how automatic differentiation, OOP and operator overloading can be jointly introduced to a class, in the same time that it would take for one of these topics separately. The marriage of ideas should appeal to those either numerically inclined or more interested in computer science. Students without prior OOP experience get valuable exposure to this approach; those with experience can compare it to MATLAB's new version (introduced with release 2008a) and reinforce concepts. The numerical payoff is seen in the applications of Sections 3 and 4 where the simple code, designed for one derivative value, yields plots, gradients, and Jacobians.

In Section 5, the idea is extended to higher-order derivatives. Objects can carry a value and not only one derivative but a whole list of Taylor polynomial coefficients. Algorithms for the overloaded operations follow from operations on series. To grasp the feasibility, suppose you want $f^{(10)}(0)$, the 10th derivative at zero, of $f(x) = x^2 \exp(-x^2)$. Difference quotient approximation is futile and symbolic differentiation is a hassle. However, manipulating the series for exp, quickly results in $f(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} x^{2n+2}$, so that $n = 4$ gives the coefficient of x^{10}. Thus, $f^{(10)}(0)/10! = 1/4!$ and $f^{(10)}(0) = 151,200$. Although handled symbolically, the powers of x are basically placeholders for a vector of numerical series coefficients. Section 6 explains general algorithms (all based on series multiplication) for operations on series objects. We see how the vector of series coefficients for $-x^2$ about any point can be fed into an overloaded exp function to return series coefficients for $\exp(-x^2)$ about that point.

Unlike symbolic differentiation, the function to be differentiated does not have to be given by a concise expression. In fact, AD is often called *automatic differentiation of programs* since it works just as well when the function is given by a large industrial program code. The goal may be to compute derivatives or sensitivities in order to adjust parameters for optimization. The challenge is to make this large-scale application efficient and robust, leading to an active research area (for example, see conference proceedings [12], [3], [8], [5], and [4]). Our introduction may be used as a bridge from complete unfamiliarity to such research literature and textbooks [15].

Some version of the following sections have been used successfully by the author in many offerings of an undergraduate numerical analysis (methods) course, using anywhere from one to four class periods. A couple of periods suffice to cover first-order ideas and implementation in Sections 2 to 4 and also to gain a working knowledge of higher-order implementation in Section 5. Some course offerings (especially before MATLAB OOP) emphasized the series algorithms in Section 6. Sections 7 and 8 bridge to more advanced topics in automatic differentiation, where implementation is not as simple.

2 Automatic differentiation and OOP basics

From a numerical programming viewpoint, a typical calculus function or expression is program code that runs with numeric `double` input and produces numeric `double` output. The idea of automatic differentiation, AD, is to extend the input to indicate what derivative(s) are desired and to empower the program operations to produce the numeric derivative value(s) as well as the function value. For linear functions, the extension is trivial. For example, if `x=3`, the MATLAB code `2*x+x+7` returns 16. Instead, define `x=[3,1]` to indicate the value of x as well as its derivative 1. Since the constant 7 has derivative 0, the code `2*x+x+[7,0]` will produce the result `[16,3]`, the value and derivative of the linear function $f(x) = 2x + x + 7$ at 3. Of course, non-linear functions require customized operations. For example, `x*x` is not even defined for the vector `x=[3,1]`, but we desire the resulting computation $[3{*}3,\ 1{*}3{+}3{*}1] = [9,6]$ using the product rule in the second coordinate. (The alternative code `x^2` should compute $[3\hat{\ }2,\ 2*3*1] = [9,6]$ by the power rule.) Customized AD operations correspond to the standard calculus derivative rules with built-in chain rule. Table 1 summarizes the approach using the example operations of multiplication and sin, where a is the numerical point at which all function values are desired.

for function		value-and-derivative object	
	Given:	val	der
$f(x) = c$	`c=`	c	0
$f(x) = x$	`x=`	a	1
$u(x)$	`u=`	$u(a)$	$u'(a)$
$v(x)$	`v=`	$v(a)$	$v'(a)$
	Compute:		
$u(x) * v(x)$	`u*v=`	$u(a) * v(a)$	$u'(a) * v(a) + u(a) * v'(a)$
$\sin(u(x))$	`sin(u)=`	$\sin(u(a))$	$\cos(u(a)) * u'(a)$

Table 1: Example value and derivative objects and methods.

An implementation of these operations could be done in a procedural programming style, but object-oriented programming with operator overloading is much more natural. In standard MATLAB procedural programming, we could actually continue to use a standard 1×2 vector to hold value and derivative and define new functions (m-files) `adtimes` and `adsin` to perform the operations. To compute the value and derivative of $f(x) = x \sin(x^2)$ at 3, one could set `x=[3,1]` and compute `adtimes(x,adsin(adtimes(x,x)))`. In contrast, OOP defines a new class of value-and-derivative objects and then defines methods on such objects. In particular, we can overload the definition of standard operations and functions, such as `*` and `sin`, as shown in Table 1. To compute the value and derivative of $f(x) = x \sin(x^2)$ at 3, create a value-and-derivative object `x` with value $a = 3$ and derivative 1, then compute `x*sin(x*x)`. This allows us to use existing MATLAB code but extend the computations by just

changing the class of input x.

In MATLAB 7.6 release 2008a or 2008b, we define the valder class in a file valder.m. The code in this file is shown below, mimicking the code-folding feature of the MATLAB editor to hide the code of each of the method definitions. In earlier releases of MATLAB, all of the methods would be separate m-files in the directory @valder. For those familiar with object-oriented programming in other languages, the newer form below allows the properties and methods to have different attributes (such as private), as well as (multiple) inheritance from superclass(es), but we won't use these advanced features. A user-defined class does take precedence over built-in classes, so 2*x will look for a valder class method when x is a valder.

```
classdef valder
   properties
      val  %function value
      der  %derivative value or gradient vector
   end
   methods
      function obj = valder(a,b)...
      function vec = double(obj)...
      function h = plus(u,v)...
      function h = uminus(u)...
      function h = minus(u,v)...
      function h = mtimes(u,v)...
      function h = mrdivide(u,v)...
      function h = mpower(u,v)...
      function h = exp(u)...
      function h = log(u)...
      function h = sqrt(u)...
      function h = sin(u)...
      function h = cos(u)...
      function h = tan(u)...
      function h = asin(u)...
      function h = atan(u)...
   end
end
```

Before looking at the simple code behind these methods, here is an example of usage where we begin by calling the method with the class name, known as the class constructor. This creates an instance of the valder object class.

```
>> x=valder(3,1);
>> x*sin(x*x)
ans =
valder
properties:
    val: 1.2364
```

```
    der: -15.9882
list of methods
```

In this command window display, the output words `valder` and `methods` are linked to comments and a listing, respectively. For $f(x) = x * \sin(x * x))$, the symbolic expression for $f'(x)$ is never formed, but the equivalent numerical evaluation results in $f'(3) = -15.9882$.

The code for the valder class constructor shows how each object property is referred to in the same way you would reference a field in a MATLAB structure array. We intend to use the constructor with two input arguments of class double, though the code uses `nargin` (number of input arguments) to allow for creating an empty valder or changing a double into a valder representing a constant function.

```
function obj = valder(a,b)
   %VALDER class constructor; only the bottom case is needed.
   if nargin == 0 %never intended for use.
      obj.val = [];
      obj.der = [];
   elseif nargin == 1 %c=valder(a) for constant w/ derivative 0
      obj.val = a;
      obj.der = 0;
   else
      obj.val = a; %given function value
      obj.der = b; %given derivative value or gradient vector
   end
end
```

The simplicity of the methods cannot be overemphasized. All methods with just one valder argument are all simply *one-line* functions that code the calculus derivative rule, as in the following sin definition.

```
function h = sin(u)
   h = valder(sin(u.val), cos(u.val)*u.der);
end
```

The methods for operations with two arguments are similar, but handle cases where one or both of the arguments are valders. The correct function name for each operation must also be identified. A search for "Overloading Operators" on MATLAB Help will return a page that includes a table of operations and corresponding method names. For simplicity, we choose to focus on scalar operations, such as `*` and `/`, which are associated with matrix operations in MATLAB. Hence, `*` is `mtimes` and `/` is `mrdivide`. The two arguments are often both of class valder, although the same method will be called with one valder and one double (scalar) argument, as in `x+7` or `7*x`. The method could either convert the double to a valder (as when 7 was changed to [7,0] in the initial discussion) or just program the scalar derivative rule (as we choose below). Of course, if neither argument is a valder, then this overloaded method will not be called.

```
function h = mtimes(u,v)
   %VALDER/MTIMES overloads * with at least one valder
   if ~isa(u,'valder') %u is a scalar
      h = valder(u*v.val, u*v.der);
   elseif ~isa(v,'valder') %v is a scalar
      h = valder(v*u.val, v*u.der);
   else
      h = valder(u.val*v.val, u.der*v.val + u.val*v.der);
   end
end
```

The power method explicitly shows the different approaches depending on which argument is variable.

```
function h = mpower(u,v)
   %VALDER/MPOWER overloads ^ with at least one valder
   if ~isa(u,'valder') %u is a scalar
      h = valder(u^v.val, u^v.val*log(u)*v.der);
   elseif ~isa(v,'valder') %v is a scalar
      h = valder(u.val^v, v*u.val^(v-1)*u.der);
   else
      h = exp(v*log(u)); %call overloaded log, * and exp
   end
end
```

It is an easy matter to mimic such method definitions for other elementary functions and operations; the challenge is the plethora of intrinsic functions provided by MATLAB. The fourteen methods (not including valder and double) listed in the valder classdef are enough to handle most typical functions from a calculus course. All files discussed herein can be downloaded from `http://www.davidson.edu/math/neidinger/publicat.html`. Adding functions like `sec` and `cot` are good first exercises for students of this introduction. Many more are listed by the command `help elfun`. To really take advantage of MATLAB, one should overload the vector and matrix operations (such as `.*`) to handle vectors or matrices of valders efficiently. For serious application, a user should consider one of the tools developed for industrial or research work, as inventoried at `http://www.autodiff.org/`. A MATLAB tool very similar to our approach is TOMLAB/MAD described in [10] `http://tomopt.com/tomlab/products/mad/`. With such a robust system, large computer programs can be run in an automatic differentiation environment and produce derivatives (or sensitivities) with respect to selected input variables. For this introduction, our small set of scalar functions will suffice to handle elementary expressions, but the ability to handle functions described by large computer codes is at the heart of most research and application of automatic differentiation [15].

3 Univariate application and Newton's method

A student might initially protest that this method only returns the derivative value at one point, but that is exactly what functions do from a computational viewpoint. For example, consider a decaying oscillation

$$f(x) = e^{-\sqrt{x}} \sin(x \ln(1 + x^2))$$

with arguments of exp and sin designed to make it more dramatic by slowing the decay and increasing the frequency, but not too fast. We define a function `fdf.m` that takes a scalar double input and returns a vector of doubles for the function value and derivative.

```
function vec = fdf(a)
x = valder(a,1);
y = exp(-sqrt(x))*sin(x*log(1+x^2));
vec = double(y);
```

The last line calls the overloaded `double` method that is included in the valder classdef.

```
function vec = double(obj)
   %VALDER/DOUBLE Convert valder object to vector of doubles.
   vec = [ obj.val, obj.der ];
end
```

The function expression in the middle line of `fdf` could be replaced by a call to another scalar function or program `f(x)` that was designed without automatic differentiation in mind (provided all operations are overloaded). We can plot our example $f(x)$ and its derivative without coding the formula for $f'(x)$ by command `fplot(@fdf,[0 5 -1 1])`, resulting in Figure 1, after editing the legend.

Automatically computing the derivative overcomes a major disadvantage of Newton's method for root-finding. Just code the expression in `fdf` and call it for a vector of function and derivative value, as in the simplest code below.

```
function root = newtonfdf(a)
delta = 1;
while abs(delta) > .000001
    fvec = fdf(a);
    delta = fvec(1)/fvec(2); %value/derivative
    a = a - delta;
end
root = a;
```

For example, consider the right-most zero in Figure 1.

```
>> newtonfdf(5)
ans =
    4.8871
```

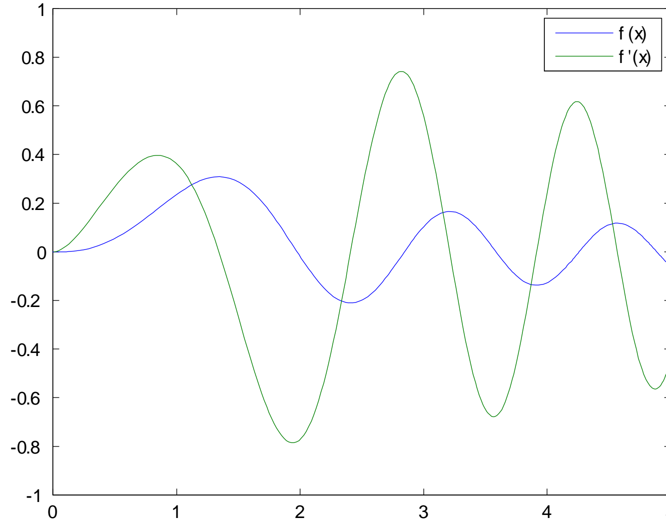


Figure 1: `fplot(@fdf,[0 5 -1 1])`

Newton's method code could, alternatively, take a simple scalar function (or function handle) and create and use valders, such as `x=valder(a,1)` and `y=f(x)`. Since the default attribute for properties is public, the program could then refer to `y.val` and `y.der`, although good OOP style usually means interacting with objects through methods only. Also, as we will see in the multivariable generalization of Newton's method, it is convenient to isolate the automatic differentiation in `fdf`. How `fdf` computes the derivative is irrelevant to the Newton program.

Of course, an introductory course should emphasize that such code is a crude prototype of a root-finding program and not of the quality found in software libraries. Instructors or students should be encouraged to protect against an infinite loop and display iteration values to observe convergence, features omitted here for shear simplicity.

4 Multivariable gradients and Jacobians

Without any changes, our valder class can be used to compute multivariable gradients. We simply use a gradient vector for the derivative property. For example, if `x=valder(3,[1,0])` and `y=valder(5,[0,1])`, then `x*y` will compute the derivative property by `[1,0]*5+3*[0,1]=[5,3]`, the correct gradient at (3,5). Then, `sin(x*y)` will compute the derivative property by `cos(15)*[5,3]`, resulting in the gradient at (3,5). All derivative rules generalize with gradient in place of derivative! So we can define a function of any number of variables and

use valders to compute the gradient.

For example, the horizontal range of a tennis serve can be modeled by the formula

$$f(a, v, h) = \frac{v^2 \cos^2 a}{32} \left(\tan a + \sqrt{\tan^2 a + \frac{64h}{v^2 \cos^2 a}} \right)$$

where a, v, and h are the initial angle, velocity in ft/sec, and height in ft, respectively [13, p. 263]. We prefer measuring the angle by degrees. The corresponding file `fgradf.m` shows how valders can be propagated through a multiple line program.

```
function vec = fgradf(a0,v0,h0)
a = valder(a0,[1 0 0]); %angle in degrees
v = valder(v0,[0 1 0]); %velocity in ft/sec
h = valder(h0,[0 0 1]); %height in ft
rad = a*pi/180;
tana = tan(rad);
vhor = (v*cos(rad))^2;
f = (vhor/32)*(tana + sqrt(tana^2+64*h/vhor)); %horizontal range
vec = double(f);
```

An example run shows that if the ball is hit at a 20° angle at 44 ft/sec from 9 ft off the ground, it travels about 56 feet and all of the partial derivatives are positive, so that increasing any of these will increase the range.

```
>> fgradf(20,44,9)
ans =
   56.0461    1.0717    1.9505    1.4596
```

One can even compute an automatic Jacobian of a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ and use it in a multivariable Newton's method to solve a system of nonlinear equations. For a numerical analysis class, this makes a nice automatic differentiation and MATLAB programming challenge. The author likes to present most of the material up to this point with just routine exercises. Then, students are asked to define a MATLAB function `[F,J]=FJF(A)` that returns the vector function value and Jacobian matrix value. They are asked to imitate `newtonfdf` to create function `newtonFJF` that solves for where the function is zero. The `newtonFJF` is an amazingly simple generalization using the strengths of MATLAB.

```
function root = newtonFJF(A)
delta = 1;
while max(abs(delta)) > .000001
    [F,J] = FJF(A);
    delta = J\F;   %solves the linear system JX=F for X
    A = A - delta;
end
root = A;
```

Function newtonFJF applies to any $F : \mathbb{R}^n \to \mathbb{R}^n$ where vectors in $\mathbb{R}^n$ are stored in $n \times 1$ arrays. In class, code should also display iteration values up to a maximum number of iterations to catch errors and ill-posed problems, again with an instructional goal and not approaching the quality of a software library routine.

We test with FJF defined for the system of equations used throughout Chapter 10 (Nonlinear Systems) of [6]:

$$\begin{aligned} 3x - \cos(yz) - \frac{1}{2} &= 0 \\ x^2 - 81(y+0.1)^2 + \sin(z) + 1.06 &= 0 \\ e^{-x\,y} + 20z + \frac{10\pi - 3}{3} &= 0. \end{aligned}$$

Coding is very simple with careful attention to converting between valders, vectors and matrices.

```
function [F, J] = FJF(A)
x = valder(A(1),[1 0 0]);
y = valder(A(2),[0 1 0]);
z = valder(A(3),[0 0 1]);
f1 = 3*x-cos(y*z)-1/2;
f2 = x^2 -81*(y+0.1)^2+sin(z)+1.06;
f3 = exp(-x*y)+20*z+(10*pi-3)/3;
values = [double(f1); double(f2); double(f3)];
F = values(:,1);
J = values(:,2:4);
```

The example run

```
>> newtonFJF([.1;.1;-.1])
ans =
   0.500000000000000
   0.000000000000000
  -0.523598775598299
```

gives full accuracy ($z = -\pi/6$) after five iterations, or eight iterations starting from [1;1;-1]. There is also another solution lurking near the same x and z with y near -0.2.

5 Higher-order AD and Taylor series

Just to show the fascinating power of the simple valder objects, try nesting valders for a single-variable function. For a third order derivative, we could nest to three levels as in the following.

```
>> x=valder(valder(valder(3,1),1),1);
>> f=x*sin(x*x);
```

```
>> f.der.der.der
ans =
  495.9280
```

The result is $f'''(3)$ for $f(x) = x\sin(x^2)$. The second derivative is found under `f.der.der.val`, but also under `f.val.der.der` and even `f.der.val.der`. Computations are duplicated for each equivalent location, just as the derivative 1 is duplicated in the definition of `x`. The data structure can be thought of as a binary tree with a val and der branch at each level, requiring 2^n locations for computing $n+1$ distinct values. In the definition of `x`, we see that a der branch may be replaced by scalar double if all higher der branches would have value zero. Our simple valder class definition cannot quite handle higher derivatives of multivariable functions, because our methods do not correctly handle vectors of valders. With this extension, higher-order partial derivatives could be computed successfully [22]. Students may find this success fascinating but be confused by how it is accomplished. However, the scheme is probably not worth further clarification due to its clear inefficiency.

Let's focus on a new class of objects for higher-order automatic differentiation of function of one variable. For a more efficient scheme, we use objects that carry only $n+1$ distinct values for differentiation through order n. We use truncated Taylor series coefficients instead of derivative values in the data structure, since the corresponding overloaded operations are simpler, as we will see in the next Section. For any function u analytic at a point a,

$$u(x) = u_0 + u_1(x-a) + u_2(x-a)^2 + \cdots + u_n(x-a)^n + \cdots$$

and we store numerical values $[u_0, u_1, u_2, \ldots, u_n]$. Each $u_k = u^{(k)}(a)/k!$, although we will directly compute u_k and use $u^{(k)}(a) = k!\,u_k$ to find the derivative if desired. The finite number of coefficients might more appropriately be called a polynomial, but we call it a `series` object since operations on truncated series properly return truncated series through the same order, whereas true polynomial multiplication would increase the order. After studying the valder class, a parallel plan for the series class should be a clear, with a constructor and overloaded methods for elementary operations and functions. We choose to isolate u_0 in the val property and keep the vector $[u_1, u_2, \ldots, u_n]$ in the coef property. The method `double` will reunite them into $[u_0, u_1, u_2, \ldots, u_n]$.

```
classdef series
   properties
      val  %function value (constant term)
      coef %vector of Taylor coefficients, linear to highest term
   end
   methods
      function obj = series(a,der,order)...
      function vector = double(obj))...
      function h = plus(u,v)...
      ...
```

The class constructor will be used to create the series object for a variable, specifying the point a about which the series will be computed, the derivative 1, and the order of the desired truncated series. The constructor fills in zeros for nonlinear terms, since higher-order derivatives of the variable are zero. Then the overloaded methods take over as in the following example.

```
>> x=series(3,1,6);
>> x*sin(x*x)
ans =
series
properties:
     val: 1.2364
    coef: [-15.9882 -30.4546 82.6547 145.6740 -85.9661 -257.6078]
list of methods
```

For $f(x) = x\sin(x^2)$, we find $f^{(6)}(3) = (6!)(-257.6078) = -185,477.6$. Although convergent series coefficients must eventually go to zero, it often takes a while, so that derivatives are very large and series coefficients more reasonable to work with numerically. The following example of a horrendous (to differentiate by hand) function will return series coefficients in addition to the function value. Figure 2 shows the resulting plot of the function and the first, second, and third series coefficients (scaled derivatives).

```
function vec = fseries3(a)
x = series(a,1,3);
y = cos(x)*sqrt(exp(-x*atan(x/2)+log(1+x^2)/(1+x^4)));
vec = double(y);
```

6 Algorithms for operations on series

Based on discussion of valder objects, students usually have a good sense of the program structure and usage for series objects, but can be fascinated by the existence and derivation of concise formulas for operations on series, that make all of this possible. Therefore, we show only a few examples of code but focus on the derivation of algorithms. As we will see, multiplication of series is the key. Most of the resulting formulas for operations on series could be found in [15] or earlier sources. The corresponding code for all of the series methods can be found at `http://www.davidson.edu/math/neidinger/publicat.html`.

The operations on series can be conceptually understood by considering series expansions of analytic functions, such as

$$\begin{aligned} u(x) &= u_0 + u_1(x-a) + u_2(x-a)^2 + \cdots + u_n(x-a)^n + \cdots, \\ v(x) &= v_0 + v_1(x-a) + v_2(x-a)^2 + \cdots + v_n(x-a)^n + \cdots, \text{ and} \\ h(x) &= h_0 + h_1(x-a) + h_2(x-a)^2 + \cdots + h_n(x-a)^n + \cdots. \end{aligned} \quad (1)$$

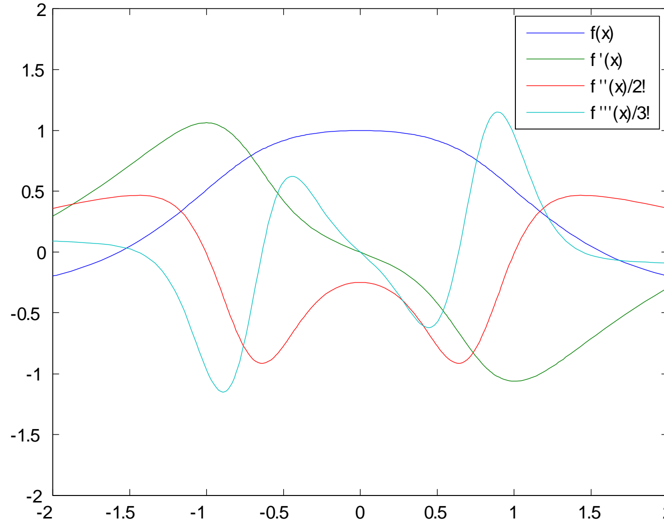


Figure 2: `fplot(@fseries3,[-2 2 -2 2])`

When $u(x)$ and $v(x)$ are multiplied, the product will result in a term of order k exactly when the subscripts (and powers) sum to k. Thus,

$$\begin{aligned} \text{if } h(x) &= u(x)\, v(x), \\ \text{then } h_k &= [u_0, u_1, \ldots, u_{k-1}, u_k]\, [v_k, v_{k-1}, \ldots, v_1, v_0]^T \end{aligned} \tag{2}$$

using matrix multiplication to perform a dot product of numerical vectors where one is reversed, an operation known as *discrete convolution*. This **multiplication theorem** (2) is at the heart of all operations and elementary functions on series, except trivial addition and scalar multiplication. The vector formulation is easy to work with in theoretical manipulations below and is natural in the MATLAB environment where matrix operations are highly efficient. To multiply series objects `u` and `v` to produce `h`, MATLAB code loops through the vector operation

```
h.coef(k) = uvec(1:k+1) * vvec(k+1:-1:1)';
```

where `uvec = [u.val, u.coef]` and `vvec = [v.val, v.coef]` are the series coefficient vectors, with index offset so that `uvec(1)` holds u_0. About half the methods directly use the coefficient vector `u.coef` that omits the constant term, so that `u.coef(k)` conveniently holds u_k. In summation notation, (2) becomes $h_k = \sum_{i=0}^{k} u_i v_{k-i}$, which can be compared to Leibniz's rule for higher-order derivative products $h^{(k)}(a) = \sum_{i=0}^{k} \binom{k}{i} u^{(i)}(a)\, v^{(k-i)}(a)$. The absence of the binomial coefficient is what makes series coefficient computation preferable

(in efficiency and stability) to carrying vectors of derivative values. However, Leibniz's rule would enable a similar implementation of derivative objects that directly carry vectors of higher-order derivative values [21].

A little algebraic manipulation yields convolution formulas for division of series objects or square root of a series object. If $h(x) = u(x)/v(x)$, then $u(x) = h(x)\,v(x)$, so by (2),

$$\begin{aligned} u_k &= [h_0, h_1, \ldots, h_{k-1}, h_k]\,[v_k, v_{k-1}, \ldots, v_1, v_0]^T, \\ u_k &= [h_0, h_1, \ldots, h_{k-1}]\,[v_k, v_{k-1}, \ldots, v_1]^T + h_k v_0, \text{ and} \\ h_k &= \frac{1}{v_0}(u_k - [h_0, h_1, \ldots, h_{k-1}]\,[v_k, v_{k-1}, \ldots, v_1]^T). \end{aligned}$$

If $h(x) = \sqrt{u(x)}$, then apply (2) to $u(x) = h(x)\,h(x)$ and similarly solve for h_k. Deriving and/or implementing the square root method is a good exercise for students, following a discussion of division.

For a standard transcendental function f, we seek the series coefficients for $f(g(x))$ when g coefficients are known. For each f, we find a differential equation of the simple form $h'(x) = u'(x)\,v(x)$ or a pair of such equations, where h, u, v can be f, g, and/or an auxiliary function. For example, $f(x) = \ln(g(x))$ satisfies $g'(x) = f'(x)\,g(x)$. Thus, (2) will yield an expression that can be solved for f_k. It is helpful to establish the pattern as the **DE Theorem**, a corollary of the multiplication theorem: for $k \geq 1$,

$$\begin{aligned} \text{if } h'(x) &= u'(x)\,v(x), \\ \text{then } h_k &= \frac{1}{k}[1u_1, 2u_2, \ldots, (k-1)\,u_{k-1}, k\,u_k]\,[v_{k-1}, v_{k-2}, \ldots, v_1, v_0]^T \end{aligned} \quad (3)$$

where subscripts come from the original series (1). The series coefficients for $u'(x)$ can be clearly seen in (3). Here, the dot product gives p_{k-1} where $p(x) = h'(x)$. Thus, $h_k = h^{(k)}(a)/k! = p^{(k-1)}(a)/k! = p_{k-1}/k$. Each overloaded method on series objects implements some form of (3).

For example, if $h(x) = \exp(u(x))$, then $h'(x) = u'(x)\,h(x)$ and (3) is applied directly to form a recursive formula with h_i in place of v_i. This formula is implemented in the following MATLAB code for the overloaded method.

```
function h = exp(u)
   d = length(u.coef);
   up = (1:d).*u.coef; %vector of coefficients for u'(x)
   hvec = [exp(u.val), zeros(1,d)]; %hvec(k+1) holds h_k coef
   for k = 1:d
       hvec(k+1) = (up(1:k) * hvec(k:-1:1)') / k;
   end
   h = series(hvec(1), hvec(2:d+1));
end
```

Observe that the actual scalar transcendental function is only evaluated once; the rest of the computation is arithmetic. This remarkable fact is true for all the

transcendental functions. The last line shows how the series class constructor with two arguments will simply assign val and coef properties. The formula and method for $h(x) = \ln(u(x))$ is now a nice exercise, where the roles of h, u, v in (3) are completely re-identified as $u'(x) = h'(x)\, u(x)$. This combines (3) and manipulation as in division.

The trigonometric functions require twice the computation, since they satisfy a pair of simple differential equations. For $s(x) = \sin(u(x))$ and $c(x) = \cos(u(x))$, we have $s'(x) = u'(x)\, c(x)$ and $c'(x) = -u'(x)\, s(x)$, resulting in the pair of convolutions

$$\begin{aligned} s_k &= \frac{1}{k}[1u_1, 2u_2, \ldots, (k-1)\, u_{k-1}, k\, u_k]\, [c_{k-1}, c_{k-2}, \ldots, c_1, c_0]^T \\ c_k &= \frac{-1}{k}[1u_1, 2u_2, \ldots, (k-1)\, u_{k-1}, k\, u_k]\, [s_{k-1}, s_{k-2}, \ldots, s_1, s_0]^T \end{aligned}$$

where s_k and c_k are series coefficients for $s(x)$ and $c(x)$, respectively. Thus, an overloaded method for `sin` must compute the series coefficients for both sin and cos. The calling code could be made more efficient by calling `[s,c]=sincos(u)` if both are useful, though the power of AD is that such foresight is not necessary. Other trigonometric functions require a bit more creativity in the pair of differential equations, as in the following table, along with (3) and algebraic manipulation.

For	Define	Then	and
$h(x) = \tan(u(x))$	$v(x) = 1 + (h(x))^2$	$h'(x) = u'(x)\, v(x)$	$v'(x) = 2h'(x)\, h(x)$
$h(x) = \arcsin(u(x))$	$v(x) = \sqrt{1 - (u(x))^2}$	$u'(x) = h'(x)\, v(x)$	$v'(x) = -h'(x)\, u(x)$
$h(x) = \arctan(u(x))$	$v(x) = 1 + (u(x))^2$	$u'(x) = h'(x)\, v(x)$	$v'(x) = 2u'(x)\, u(x)$

The power operation on series `u^v` can be implemented by `exp(v*log(u))`. For a series object `v`, `exp(v*log(u))` works fine and, if `u` is a scalar, only uses one convolution in the call to `exp`. If both are series objects, there are three overloaded calls, each using one convolution. However, if `v` is a scalar (and an integer, in particular), then `log` unnecessarily restricts to positive base values. This can be overcome by a recursive formula. Let $h(x) = u(x)^r$ where r is a scalar. Then $h'(x)\, u(x) = r\, u'(x)\, h(x)$, and (3) may be applied to both sides of the equality. Solving for h_k results in a formula with two convolutions, roughly equivalent to the work to exp and log.

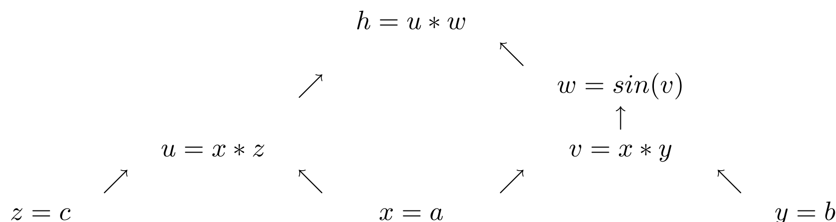
These algorithms can be applied to ODEs, to find the unknown series solutions. Up to this point, we have considered functions of the form $y = f(x)$ where f is composed of elementary functions and operations. When `x` is a series object for the variable x at x_0, the overloaded calls in `f(x)` will perform loops over convolutions, as in the exp code above, to compute the series for y about x_0. Now, consider an ODE of the form $y' = f(y)$ with initial condition (x_0, y_0). For induction, suppose some series coefficients for y are known, say $[y_0, y_1, \ldots, y_k]$. If these coefficients are used in a series object `y`, then `f(y)` will find the series for y' through order k. Of course, this is wasteful because all coefficients were already known except the last one which gives a new value for y_{k+1}. Instead of

using the series class, we should use the convolution formulas from this section in one loop given by the differential equation. For example, if $y' = y * y$, then, by (2), $(y')_k = [y_0, y_1, \ldots, y_k][y_k, y_{k-1}, \ldots, y_0]^T$. Since $(y')_k = (k+1)y_{k+1}$, we find a recurrence relation for y_{k+1}. This method has been used to successfully implement very high-order Taylor series solution methods for ODEs and DAEs, with stepsize approaching the radius of convergence [7], [9].

7 Introduction to the reverse mode of AD

We now turn from implementation to directions for further study for students interested in pursuing AD beyond this introduction. This section considers an alternative method of derivative evaluation, which is more efficient for a large number of independent variables. Returning to the multivariate setting, we implemented the *forward mode* of automatic differentiation by carrying gradients forward through program execution. We now use one simple example to compare the *reverse mode*, which students will quickly encounter in any study of AD literature [15, p. 7].

Consider the evaluation of $h(x, y, z) = (x*z)*\sin(x*y)$ as represented in the following *computational graph*, which has a node for each intermediate operation value and a directed edge to where that value is used (forming a directed acyclic graph).



Forward mode works in the direction of the directed edges computing the full gradient, in addition to the value, at each node. Reverse mode also begins with an evaluation pass in this direction, but the operation at each node computes only the partial derivative of that node operation with respect to the immediate arguments; creating a value corresponding to each edge. Assuming a, b, and c are numerical values, these partials are also numerical values from the most basic derivative rules. The multivariable chain rule, just as taught in multivariable calculus [1, p. 968], corresponds to tracing paths in the reverse direction of the arrows multiplying edge values along the way, from the top node to each of the bottom nodes in our display. Let's follow the computations for this example, first in forward mode then in reverse mode.

Forward mode performs the following sequence of computations, where we explicitly display the gradient vectors inside each valder operation as described

in Sections 2 and 4.

$$\begin{array}{cc}
x = a & \nabla x = [1, 0, 0] \\
y = b & \nabla y = [0, 1, 0] \\
z = c & \nabla z = [0, 0, 1] \\
u = ac & \nabla u = c[1, 0, 0] + a[0, 0, 1] = [c, 0, a] \\
v = ab & \nabla v = b[1, 0, 0] + a[0, 1, 0] = [b, a, 0] \\
w = \sin(v) & \nabla w = \cos(v)[b, a, 0] = [b\cos(v), a\cos(v), 0] \\
h = uw & \nabla h = w[c, 0, a] + u[b\cos(v), a\cos(v), 0]
\end{array}$$

Of course, all entries would be numerical values, but we can see the correct symbolic result $\partial h/\partial x = c\sin(ab) + abc\cos(ab)$. To see the potential inefficiency, imagine these were just 3 of 100 independent variables. Carrying all those zeros in the gradient would be wasteful. It is not just a matter of sparsity, since a program could first sum all the variables for a full gradient (since direction $[1, 1, 1, ..., 1]$ may be of particular interest), but propagating it through several subsequent operations is expensive. In general, let n be the number of independent (input) variables. The cost of the forward mode of AD is very roughly $(n + 1)$ times the cost of the function (program) evaluation, since each step computes n entries in addition to the value. In contrast, the cost of the reverse mode is a constant multiple of the function cost, not depending on n (though it does depend on the number of dependent output variables which is one in our example).

Reverse mode will consist of a forward and reverse pass through the computational graph. The forward pass computes the immediate partial derivatives.

$$\begin{array}{ccc}
x = a & & \\
y = b & & \\
z = c & & \\
u = ac & u_x = c & u_z = a \\
v = ab & v_x = b & v_y = a \\
w = \sin(v) & w_v = \cos(v) & \\
h = wu & h_u = w & h_w = u
\end{array}$$

The reverse pass will trace backwards through the computational graph, accumulating products of these immediate partial derivatives associated with edges. The accumulation at each node is defined to be an *adjoint variable*, denoted with a bar over the variable. At a node with one directed edge going out, say at v with arrow pointing up to w, we assume $\overline{w} = \frac{\partial h}{\partial w}$ has been computed and now compute $\overline{v} = \overline{w}\, w_v = \frac{\partial h}{\partial w}\frac{\partial w}{\partial v} = \partial h/\partial v$ by the chain rule. In the case of multiple edges going out, sum the products for each edge as in $\overline{x} = \overline{u}\, u_x + \overline{v}\, v_x = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial h}{\partial v}\frac{\partial v}{\partial x} = \frac{\partial h}{\partial x}$. Here is the entire sequence of reverse

pass computations.

$$\begin{aligned}
\overline{h} &= 1 \\
\overline{w} &= \overline{h}\, h_w \\
\overline{v} &= \overline{w}\, w_v \\
\overline{u} &= \overline{h}\, h_u \\
\overline{z} &= \overline{u}\, u_z \\
\overline{y} &= \overline{v}\, v_y \\
\overline{x} &= \overline{u}\, u_x + \overline{v}\, v_x
\end{aligned}$$

The last expression unravels to the chain rule $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial h}{\partial w}\frac{\partial w}{\partial v}\frac{\partial v}{\partial x}$ and correctly computes $wc + u\cos(v)b = c\sin(ab) + abc\cos(ab)$.

Observe that the total number of assignments for both passes is about 3 times the 7 assignments in the first column of the function evaluation. A very thorough analysis of all time costs associated with the reverse method in general shows that the cost is always less than 4 times the cost of just evaluation [15, p. 44, 85]. For our example with $n = 3$, this doesn't make a difference but for large n it can be worthwhile (e.g. [16]), especially when the number of output variables is much smaller than the number of input variables.

The implementation of the reverse mode can be accomplished through operator overloading in the forward pass, but the operators must write the immediate partial derivatives and corresponding operations to be done on the reverse pass. Exactly where and how this is written can vary, from a block of program code, to storage on a *tape*, or entries in a sparse extended Jacobian matrix. A classic AD approach is a pre-compiler that performs source transformation, i.e. it reads program codes and produces a larger code that lists assignments such as displayed above (e.g. ADIFOR tool example in [23]). For large programs, the size of this code and associated memory allocation can be huge. However, since the values and instructions are needed in reverse order of their generation, these can be recorded on a *tape* that is a last-in-first-out (LIFO) data structure that can contain a large amount of data [15, p. 64]. To numerical analysis students, all this code generation may seem far from a numerical method. This may be countered with the following numerical matrix formulation of the reverse mode.

Just as intermediate values are associated with variable names, they can be associated with a row (and the same column) of large matrix. We form an extended Jacobian matrix which only stores the immediate partial derivatives. In the forward pass, when a row variable has an immediate dependence on a column variable, save its partial derivative in that location. For our example $h(x, y, z)$, we form the 7×7 matrix L below, where each blank entry is a zero, and an identity matrix has been subtracted to form the diagonal (as explained

in [11]).

	x	y	z	u	v	w	h
x	-1						
y		-1					
z			-1				
u	c		a	-1			
v	b	a			-1		
w					$cos(v)$	-1	
h				w		u	-1

Observe that $[\overline{x}, \overline{y}, \overline{z}, \overline{u}, \overline{v}, \overline{w}, \overline{h}]\, L = [0,0,0,0,0,0,-1]$ is a collection of the multivariable chain rules. For example, the first column of this matrix multiplication becomes $\overline{x}(-1) + \overline{u}\, u_x + \overline{v}\, v_x = 0$ which means $\overline{x} = \frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial h}{\partial v}\frac{\partial v}{\partial x}$. The fifth column is $\overline{v}(-1) + \overline{w}\, w_v = 0$, or $\frac{\partial h}{\partial v} = \frac{\partial h}{\partial w}\frac{\partial w}{\partial v}$. Finally, the last equation gives $\overline{h} = 1$, which begins the process of solving for the adjoint variables. Simple back substitution will solve this triangular system, while exactly performing the steps of the reverse pass described above! In general, this method always forms and solves a huge but very simple numerical system with lots of structure to exploit [15, p. 188]. Only the sparse entries below the diagonal need actually be stored. Still, a huge program can make it impractical to work directly with the matrix. Nevertheless, implementing the reverse method can be thought of as an efficient solve of a simple numerical linear system. Albeit, finding the intermediate variables and locations that define the matrix is equivalent to forming the adjacency matrix of the directed graph.

8 Multivariable higher-order methods

Since we've covered gradients of multivariable functions and series of univariate functions, another natural direction for further study is higher-order derivatives or series of multivariable functions. We identify some issues, ideas, and references for two different approaches.

In principle, the forward-mode series idea extends naturally to multivariable functions, but there are practical challenges in dealing with the potentially massive number of terms in a non-rectangular multi-dimensional structure. For analytic $f : \mathbb{R}^n \to \mathbb{R}$ at point $\mathbf{a}$, there are $\binom{n+d}{d}$ unique partial derivatives of order $\leq d$ (and hence Taylor coefficients of degree $\leq d$). These are naturally indexed by the number of derivatives $(j_1, j_2, \ldots, j_n)$ in each of the n variables, but $j_1 + j_2 + \ldots + j_n \leq d$ fills only a small (hyper-tetrahedral) corner of an n-dimensional array. An `mvseries` class for multivariable series objects could be created to hold a data structure for such a corner of Taylor coefficients. Operations on mvseries can be derived from Multiplication and DE Theorems analogous to (2) and (3) [19]. However, the convolutions use rectangular subarrays taken from the corners. Addressing these parts of the non-rectangular n-dimensional data structures (usually implemented in a linear array) becomes a focus for successful implementation of such a method [2] [20].

A different approach focuses on using univariate series computations (as in Sections 5 and 6) on functions of the form $f(\mathbf{a} + t\mathbf{v})$, thus computing univariate directional derivatives for a variety of fixed directions $\mathbf{v}$. Enough of these can be combined to construct the entire corner of multivariable Taylor series coefficients. Two different schemes for choosing directions $\mathbf{v}$ and constructing the corner values are found in [14] and [17], where these methods are shown to be efficient compared with the approach of the previous paragraph. In [17], the construction is literally a multivariable interpolation problem solved using a divided-difference algorithm, analogous to the (univariate) Newton interpolating polynomial in introductory numerical analysis. Even if this is not pursued, the power of our univariate `series` class can be seen. In some applications, the directional results themselves may be more useful than the corner of multivariable Taylor series coefficients [15].

9 Conclusion

Experience has shown that students in a first numerical analysis (methods) course appreciate an introduction to automatic differentiation. The MATLAB implementation is surprisingly simple and powerful, as seen in each one-line method that overloads a standard elementary function to include a derivative rule. There is a bit of overall structure to the class definition file for valder and/or series, but this introduces or reinforces object-oriented programming in a naturally motivated setting. With the advent of MATLAB 2008a, the OOP style is more consistent with other familiar programming environments, which makes this a good time to add AD to numerical courses already using MATLAB.

This introduction busts the common myth in numerical methods that "accurate derivatives are hard to find without hand or symbolic algebra manipulation." The implication for other topics is significant. In this setting, Newton's method only requires evaluation of the function, since the overloaded operations return the floating-point derivative value as well. Multivariable Newton's method for a system of equations is a satisfying culmination, especially if `A\b` has been discussed as an efficient linear solver. Here, AD replaces the tedious task of computing the Jacobian. The busted myth would go on to say "so higher-order derivatives become impractical." On the contrary, students can easily use the series class to automatically compute series coefficients. The end of Section 6 discussed how the algorithms can be used in a high-order Taylor series method for solving ODEs.

Numerical analysis classes should also consider the stability and efficiency of AD and the numerical or symbolic alternatives. Difference quotient approximations are efficient but are typically shown to have significant error in the first derivative (with choice of step-size offering a trade-off between approximation error and loss of precision) and progressively more error for second and third order derivatives. In contrast, AD and evaluation of symbolic derivatives are more like Gaussian elimination, since these are exact algorithms without any approximation error but performed with floating-point arithmetic. These

methods typically have a high degree of precision that may be worth the software overhead. To test stability look at high-order derivatives, where difference quotients are out of the running. A past study compared series algorithms to symbolic derivatives in Mathematica 3.0 [18] (reconfirmed in 7.0). A symbolic high-order (say 10 to 50) derivative can become unwieldy for some expressions that grow on each step. It's clear that AD does not require the space necessary to store huge symbolic expressions. As expected, the time to compute series coefficients using AD is about the same order of magnitude as just evaluating the resulting huge symbolic expression. Creating the symbolic expression and simplifying it are both a couple of orders of magnitude more costly in time. The surprise is that series algorithms remain stable in situations where symbolic derivative evaluation is unstable, and the opposite has not been observed. For example, high-order derivatives of $\exp(-x^4)$ result in a polynomial factor with huge integer coefficients that alternate in sign. Evaluation at a positive point caused the digits of precision to degrade from 14 to 2 as the order increased from 10 to 50, while the numerical series computation of the same value kept at least 12 digits of precision. Though not a thorough analysis, these considerations are enough to show that AD can be excellent choice of method.

In MATLAB, automatic differentiation plays to a strength in numerical vector computation and fills a weakness in derivative computation without the Symbolic Math Toolbox. More generally, automatic differentiation adds a whole new perspective on derivatives in numerical methods.

References

[1] H. Anton, I. Bivens, and S. Davis, *Calculus Early Transcendentals*, 8th ed., John Wiley, Hoboken, NJ, 2005.

[2] M. Berz, *Algorithms for Higher Derivatives in Many Variables with Applications to Beam Physics*, in [12] (1991), pp. 147-156.

[3] M. Berz, C. Bischof, G. Corliss, A. Griewank, eds., *Computational Differentiation: Techniques, Applications and Tools*, SIAM, Philadelphia, 1996.

[4] C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, eds., *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering 64, Springer, Berlin, 2008.

[5] H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris, eds., *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering 50, Springer, New York, 2005.

[6] R. L. Burden and J. D. Faires, *Numerical Analysis*, 8th ed., Thomson Brooks/Cole, Belmont, CA, 2005.

[7] Y. F. Chang and G. Corliss, *ATOMFT: Solving ODEs and DAEs Using Taylor Series*, Computers & Mathematics with Applications, 28 (1994), pp. 209-233.

[8] G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann, eds., *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Springer, New York, 2002.

[9] H. Flanders, *Application of AD to a Family of Periodic Functions*, in [8] (2002), pp. 319-326.

[10] S. A. Forth, *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB*, ACM Transactions on Mathematical Software, 32 (2006), pp. 195–222.

[11] S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid, *Jacobian Code Generated by Source Transformation and Vertex Elimination can be as Efficient as Hand-Coding*, ACM Transactions on Mathematical Software, 30 (2004), pp. 266-299.

[12] A. Griewank and G. F. Corliss, eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.

[13] R. N. Greenwell, N. P. Ritchey and M. L. Lial, *Calculus with Applications for the Life Sciences*, Addison Wesley, Boston, 2003.

[14] A. Griewank, J. Utke, and A. Walther, *Evaluating Higher Derivative Tensors by Forward Propagation of Univariate Taylor Series*, Mathematics of Computation, 69 (2000), pp. 1117-1130.

[15] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., SIAM, Philadelphia, 2008.

[16] G. Haase, U. Langer, E. Linder, and W. Mühlhuber, *Optimal Sizing of Industrial Structural Mechanics Problems Using AD*, in [8] (2002), pp. 181-188.

[17] R. D. Neidinger, *Directions for Computing Truncated Multivariate Taylor Series*, Mathematics of Computation, 74 (2004), pp. 321-340.

[18] R. D. Neidinger, *Series as a Computational Differentiation Tool*, Mathematica in Education and Research, 9 (2000), pp. 5-14.

[19] R. D. Neidinger, *Computing Multivariable Taylor Series to Arbitrary Order*, ACM SIGAPL APL Quote Quad, 25 (1995), pp. 134-144.

[20] R. D. Neidinger, *An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order*, ACM Transactions on Mathematical Software, 18 (1992), pp. 159-173.

[21] R. D. Neidinger, *Automatic Differentiation and APL*, College Math. J., 20 (1989), pp. 238-251.

[22] M. Padulo, S. A. Forth and M. D. Guenov, *Robust Aircraft Conceptual Design Using Automatic Differentiation in Matlab*, in [4] (2008), pp. 271–280.

[23] L. B. Rall and G. F. Corliss, *Introduction to Automatic Differentiation*, in [3] (1996), pp. 1-18.