Evaluation and approximation of derivatives

April 18, 2012

As a calculus student you learned that computing derivatives is "easier" than computing integrals: to differentiate $f(x)$, just apply the basic rules (chain rule, etc) recursively until done. Integration through antidifferentiation, by contrast, is a zoo of tricks. Worse, not every function has a simple antiderivative: Functions such as e^{x^2} have an easy derivative but *no* closed-form antiderivative. It's clear why numerical integration is needed. But since in principle it's easy to compute the derivative of every differentiable function it's reasonable to ask why numerical approximation of derivatives is ever needed.

1. Your $f(x)$ might be a calculated by a "black box" computer program that's not easily to recast as a differentiable expression. In some cases, you may have no access to the source code: the function f or the algorithm for computing it might be proprietary intellectual property of a company, or a military secret. In the pre-computer days the problem was to differentiate functions whose values are known only as entries in tables; the modern form of that problem is to differentiate functions known only through code that is proprietary, secret, or freely available but simply hard to follow.

2. You may want an approximate representation of the *operation* of differentiation, for use in transformation of a differential equation into a discrete system of algebraic equations.

These two considerations lead us to look at approximation of derivatives. But even in the best case, where you have in hand a mathematical representation of a function whose derivatives are to be computed, it's important to proceed carefully. A derivative computed by hand or by symbolic computation may not be in a form suitable for efficient or stable computation. Even for simple functions, it's no fun to work out derivatives by hand and put them into a computer program, so a procedure to automate the efficient and stable computation of derivatives is handy to have.

So we have two problems: approximation of derivatives or differentiation operators, and also efficient, stable, and preferably automatic evaluation of "exact" derivatives. "Exact" in quotes means the computations are exact in the sense of using no approximations other than the unavoidable floating point arithmetic.

As it's an old topic going back to the days of tables, most introductory textbooks will have a section on approximation of derivatives. Ackleh *et. al* covers that topic well, and also has a section on the more modern and less-often-covered topic of automatic evaluation of

"exact" derivatives. Discussion of derivative approximation and evaluation in multiple dimensions is found in books on nonlinear solvers and optimization (why?) such as Dennis and Schnabel or Nocedal and Wright. Finally, see Griewank, *Evaluating Derivatives*, for a book-length treatment of, you guessed it, evaluating derivatives. There's much more to computing derivatives than most of us would have guessed.

1 Approximating derivatives: finite differences

Upon first encounter with approximate differentiation, the first thought to occur to most of us would be to refer to the definition of derivative,

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

but defer passage to the limit. That is, take h small but not all the way to zero,

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

An error estimate is easy to obtain via Taylor's theorem: assuming $f \in C^2[x_0, x_0 + h]$, then $\exists \xi \in (x_0, x_0 + h)$ such that

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{1}{2} f''(\xi) h^2.$$

Therefore we can approximate the derivative by the formula

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + O(h).$$

To save typing and toner we'll usually write such expressions with the obvious abbreviation

$$f_0' = \frac{f_1 - f_0}{h} + O(h).$$

This is called the two-point forward difference formula, and because the error is $O(h)$ we'll say it's "first order accurate." It's the simplest example of a general method called finite differencing (FD). (In contexts where it's already clear we're talking about finite differencing, the acronym FD will be used for "forward difference.")

For an alternative derivation, construct a Lagrange interpolation to f based on the points x_0 and $x_1 = x_0 + h$,

$$p_1(x) = f_0 \ell_{1,0}(x) + f_1 \ell_{1,1}(x)$$

where p_1 is the first-degree Lagrange interpolant (the best possible with two points) and the ℓs are the Lagrange basis functions. The function and its Lagrange approximation are related by

$$f(x) = p_1(x) + R_1(x) = \frac{x - x_1}{x_0 - x_1} f_0 + \frac{x - x_0}{x_1 - x_0} f_1 + \frac{1}{2} f''(\xi)(x - x_0)(x - x_1).$$

(Note that although I've used the symbol ξ in this remainder, it won't have the same value as the ξ in the Taylor remainder.) Now differentiate the approximation,

$$f'(x) = \frac{f_1 - f_0}{x_1 - x_0} + f''(\xi)(2x - x_0 - x_1) + \frac{1}{2}f'''(\xi)\frac{d\xi}{dx}(x - x_0)(x - x_1).$$

The strange term involving $\frac{d\xi}{dx}$ is $O\left(h^2\right)$ compared to $O(h)$ for the more familiar term involving $f''(\xi)$, so we can ignore it. Once again, find

$$f_0' = \frac{f_1 - f_0}{h} + O(h),$$

same as derived via Taylor's theorem.

Right away there's a dark cloud on the horizon: the linear Lagrange approximation is accurate to $O\left(h^2\right)$, but the derivative based on it is accurate only to $O(h)$. This will be a problem in general: the derivative of an approximation will be less accurate than the original approximation.

Notice also that we could have just as well used a *backwards* difference and found

$$f_0' = \frac{f_0 - f_{-1}}{h} + O(h).$$

This distinction will become important when we use derivative approximations in the context of ODE integrators.

1.1 Why not make h really small?

Does the loss of one order of accuracy upon differentiation matter? After all, even with $O(h)$ you can make the error arbitrarily small, right? In exact arithmetic, yes. In floating point, no. An experiment will show the problem. Let's use finite differencing to compute the derivative of $f(x) = \sin(x)$ at $x = 1$. Here are the "exact" values computed to double precision.

$$f(1) = \sin(1) \approx 0.841470984807897$$

$$f'(1) = \cos(1) \approx 0.5403023058681398.$$

The results for a range of steps h are tabulated; the error is in the final column. Reducing h by 10^{-2} reduces the error by 10^{-2} as expected... until around $h \approx 10^{-8}$ the error gets *worse* as h is decreased.

$\log_{10} h$	$\sin(1+h)$	$f'_{\text{FD}} = \frac{\sin(1+h)-\sin(1)}{h}$	$\left\|f'(1) - f'_{\text{FD}}\right\|$
0	0.909297426825682	0.0678264420177852	-0.4724758638503546
2	0.846831844618015	0.536085981011869	-0.00421632485627077
4	0.841525010831038	0.5402602314186211	-0.00004207444951864758
6	0.841471525109782	0.5403018851213304	-4.2074680939308E-7
8	0.84147099021092	0.5403023028982545	-2.969885226633551 E-9
10	0.841470984861927	0.5403022473871033	-5.848103645789138 E-8
12	0.841470984808437	0.5403455460850637	0.00004324021692392321
14	0.841470984807902	0.5440092820663267	0.00370697619818694
16	0.841470984807897	0.	-0.5403023058681398

A look at the tabulated values of $\sin(1+h)$ gives a hint as to the cause: as h gets smaller, $\sin(1+h)$ and $\sin(1)$ agree to increasingly many places, which means their *difference* is accurate to decreasingly many places. With $h = 10^{-14}$, $\sin(1+h)$ and $\sin(1)$ agree to 13 places, leaving only three decimal places with which to represent the difference. Sure enough, the error in the final column shows that at $h = 10^{-14}$, the result is accurate to about three places.

The error estimate we derived did not account for finite precision arithmetic, and apparently this is a problem where finite precision really matters. We need to correct the error estimate for the effects of floating point.'

1.2 Error analysis including roundoff

Looking ahead to higher-order finite difference approximations, let's assume we have a finite-difference formula based on a "stencil" of uniformly-spaced points, accurate to h^p, which we can write generically as

$$f'(x_0) = \frac{1}{h}\sum_{n=-M}^{N} \alpha_n f_n + O(h^p)$$

with the notation $f_n = f(x_0 + nh)$. The simple forward difference formula derived above is a special case with $M = 0$, $N = 1$, $\alpha_0 = -1$, $\alpha_1 = 1$, $p = 1$.

In finite precision, each term is computed to relative error $\kappa_n O(\epsilon_m)$ where ϵ_m is machine epsilon and κ_n is the condition number for $\alpha_n f_n$: the computed FP $\widetilde{\alpha f}$ and the exact αf are related by

$$\widetilde{\alpha_n f}_n = \alpha_n f_n \cdot (1 + \kappa_n O(\epsilon_m)).$$

The FP derivative approximation $\widetilde{f'_{\text{FD}}}$ will be

$$\widetilde{f'_{\text{FD}}} = \frac{1}{h}\sum_{n=-M}^{N} \alpha_n f_n \cdot (1 + \kappa_n O(\epsilon_n))$$

$$= f'_{\text{FD}} + \frac{1}{h}\sum_{n=-M}^{N} \alpha_n f_n \kappa_n O(\epsilon_n)$$

$$= f' + O\left(h^p\right) + \frac{1}{h}\sum_{n=-M}^{N} \alpha_n f_n \kappa_n O\left(\epsilon_n\right).$$

The usual triangle inequality argument shows that $\exists\, C_1 > 0, C_2 > 0$ such that

$$\left| f' - \widetilde{f'_{\mathrm{FD}}} \right| \leq C_1 h^p + \frac{C_2}{h}\epsilon_m.$$

The constants C_1, C_2 depend on f. Clearly some functions will be differentiated more accurately than others; that's not our concern at the moment. The issue for now is how the error varies with h, and the result is sobering. As $h \to 0$ the error *diverges*! The coefficient of the divergent term is small – $O\left(\epsilon_m\right)$ – so we might hope the "nice" term h^p will dominate for a while, but at some sufficiently small h the error will be dominated by the divergent term. The divergence is due ultimately to the fact that we're dividing the roundoff error by h. Take h small enough, and any finite difference method *must* break down.

How small can h go before roundoff wrecks the calculation? Minimize the error with respect to h, and find

$$pC_1 h_{min}^{p-1} = \frac{C_2}{h^2}\epsilon_m$$

$$h_{min} = \left(\frac{C_2}{pC_1}\epsilon_m\right)^{\frac{1}{p+1}}.$$

The crux is the dependence on ϵ_m, namely, $h_{min} = O\left(\epsilon_m^{\frac{1}{p+1}}\right)$. At this sweet spot, the error is $O\left(\epsilon_m^{\frac{p}{1+p}}\right)$. With $p = 1$ we can reach an error $O\left(\sqrt{\epsilon_m}\right) \approx 10^{-8}$: just about what we saw in the experiment! Only as $p \to \infty$ can the error approach $O\left(\epsilon_m\right)$.

Figure 1 shows errors for the FD approximation to the derivative of $\sin(x)$ at $x_0 = 1$. These results are consistent with our error analysis. As $h \to 0$, all three curves go as h^{-1} (you remember how to measure exponents of power laws from data plotted on log-log scales, right? A power law x^q is a straight line on a log-log plot, with the slope of the line equal to the exponent q). The minima are around the expected values of $h = \epsilon_m^{\frac{1}{2}} \approx 10^{-8}$ for $p = 1$, $h = \epsilon_m^{\frac{1}{3}} \approx 4.6 \times 10^{-6}$ for $p = 2$, and $h = \epsilon_m^{\frac{1}{4}} \approx 10^{-4}$ for $p = 4$. You should verify that the values at the minima are as expected. Again looking at the slopes on the climb out of the valley towards the right, see that sufficiently far above the minimum h, the curves go as h^p, just as expected from the exact-arithmetic analysis.

1.3 Higher-order finite differences

The forward difference formula is $O\left(h\right)$ and so can't do better than $O\left(\sqrt{\epsilon_m}\right)$. Luckily, it's easy to get $O\left(h^2\right)$ and thus $O\left(\epsilon_m^{2/3}\right)$ with no additional cost.

Assume $f \in C^3$. Compute $f_{\pm 1} = f\left(x_0 \pm h\right)$ using Taylor's theorem with remainder: there will be two points $\xi_\pm \in \left(x_0, x_0 \pm h\right)$ such that

$$f_{\pm 1} = f\left(x_0\right) \pm h f'\left(x_0\right) + \frac{1}{2} f''\left(x_0\right) h^2 + \frac{1}{6} f^{(3)}\left(\xi_\pm\right) \left(\pm h\right)^3.$$

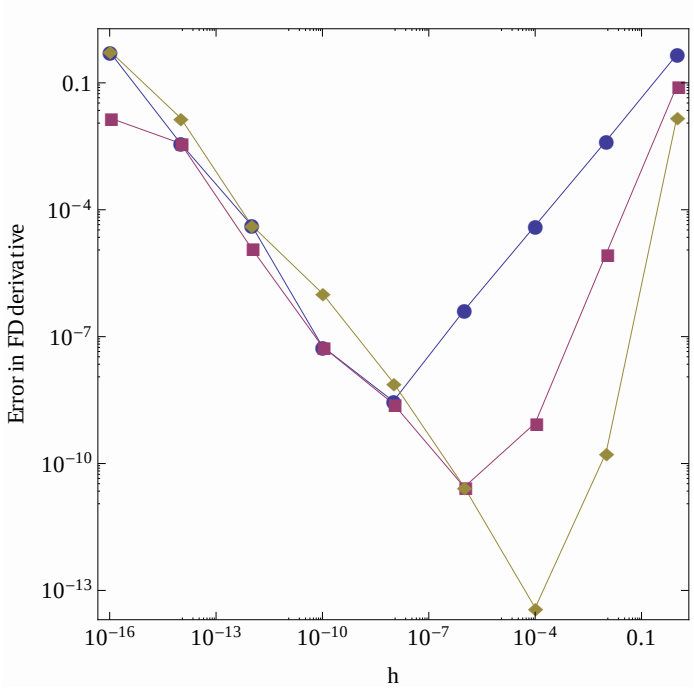

Figure 1: Error in FD differentiation of $f(x) = \sin(x)$ at $x_0 = 1$, as function of h for $p = 1$ (circles), $p = 2$ (squares), $p = 4$ (diamonds).

Subtract, cancelling the even-order terms, leaving

$$f_1 - f_{-1} = 2hf'(x_0) + \frac{1}{6}h^3\left[f^{(3)}(\xi_+) + f^{(3)}(\xi_-)\right].$$

On the space of quadratics, the error vanishes and we have an exact formula. Otherwise, divide by h, rearrange, use triangle, sweep constants and squiggles under the rug of big-O notation, and find

$$f'(x_0) = \frac{f_1 - f_{-1}}{2h} + O\left(h^2\right).$$

This is the *two-point centered difference formula*. You should go through the exercise of deriving it through Lagrange interpolation as well. It is exactly as expensive to compute as the forward difference formula, but is one power of h more accurate (which in turn means that we can push the roundoff-failure limit down from $\sqrt{\epsilon_m}$ to $\epsilon_m^{2/3}$.)

Set up a similar construction with four points, $f_{\pm 1}$ and $f_{\pm 2}$, and juggle Taylor approximations to cancel everything up to h^5, and get a derivative approximation good to $O\left(h^4\right)$. And so on. The derivative will *always* be a degree less accurate than the initial approximation, because the error term *always* gets divided by h.

Notice also that the two-point centered difference approximation is simply the average of the results of the two-point forward difference and two-point backward difference approximations.

Get good at working this sort of calculations; very similar ideas will appear in the setting of ODE integrators.

1.3.1 Centered versus one-sided differences

When you can use them, centered difference stencils – nodes placed symmetrically about the evaluation point – give the best bang for the buck. But you can't always use them, in which case one-sided stencils are used. For example, a second-order accurate forward difference formula is

$$f'(x_0) = \frac{1}{h}\left[-\frac{3}{2}f_0 + 2f_1 - \frac{1}{2}f_2\right] + O\left(h^2\right).$$

which requires three function evaluations instead of two for the centered second-order formula. You should derive this using both Taylor's theorem and Lagrange interpolation. Cases where one-sided stencils are needed are (1) when working close enough to a singularity that a centered stencil would hit the singularity, and (2) when approximating differential operators near a boundary.

1.4 Finite difference approximations to differential operators

We've focused on FD as a means to compute derivatives, but at least as important is approximation to differential operators in the context of solving differential equations.

1.5 Some concluding comments

The standard FD formulas are constructed to be exact for polynomials up to a specified degree. As with integration, you may sometimes want to work with a different function space, especially when near a singularity. If you know a function has a logarithmic singularity, you can construct a FD formula that's exact for logarithms times quadratics or quartics, for instance. This can be useful in solving differential equations where the boundary conditions specify that the solution diverges in a particular way.

2 Approximating derivatives: derivatives of approximations

Interpolation, polynomial or otherwise, is but one way to approximate functions. A general way to think about approximation of derivatives is: *approximate the function, then differentiate the approximation*. If ϕ_n is a basis and a_n are coefficients computed so as to approximate f in your favorite way, then the approximation and remainder take the general form

$$\begin{aligned} f(x) &= \sum_{n=0}^{N} a_n \phi_n(x) + R_N(x) \\ &= f_N(x) + R_N(x). \end{aligned}$$

Then

$$f'(x) = \sum_{n=0}^{N} a_n \phi_n'(x) + R_N'(x).$$

When $\{\phi_n\}$ is the Lagrange basis $\{\ell_{N,n}\}$with evenly spaced nodes and coefficients a_n determined by interpolation, this leads to the standard FD formulas. As another example, suppose we've approximated a periodic function by approximation in the Fourier basis. I'll ignore the error terms at first. The $2N+1$ term trigonometric polynomial is

$$f(x) \approx \sum_{n=-N}^{N} a_n e^{-inx}$$

and so the approximation to f' is the trigonometric polynomial

$$f'(x) \approx -\sum_{n=-N}^{N} ina_n e^{-inx}.$$

The high-frequency terms (*i.e.*, those with $|n|$ large) are amplified by differentiation: differentiation acts to "roughen" a function. The error analysis is difficult in general, but if $|a_n| \to 0$ rapidly then the error is dominated by the first term omitted, which varies as $e^{-i(N+1)x}$. The error in the derivative is then $(N+1)$ times larger than the error in the original approximation. Just as with FD, differentiation loses accuracy relative to an initial approximation. This argument is independent of *how* the trigonometric polynomial was obtained; the coefficients a_n might have been obtained by trigonometric interpolation, or by best L^2 approximation (in which case they are the Fourier coefficients) or by best L^∞ approximation, or whatever. No matter; the derivative comes out worse than the initial approximation.

A similar argument goes through with, for instance, approximation in Chebyshev polynomials.

A practical consequence of this is that if you know you're going to need to differentiate your approximation, *use the accuracy requirements on the derivative to choose the accuracy to which you approximate the function.*

3 "Exact" differentiation

Mathematica tells me the derivative of

$$f(x) = \left(x^2 - 2x\right) e^{x^3}$$

is

$$f'(x) = e^{x^3}(2x-2) + 3e^{x^3}\left(x^2-2x\right)x^2.$$

This is easy enough to type into MATLAB:

```
function [f,df]=stupid(x)
  f=(x.^2 + 2.0*x).* exp(x.^3);
  df=exp(x.^3).*(2.0*x-2) + 3.0*exp(x.^3).*(x.^2-2.0*x).*x.^2;
```

This is ugly, and I'd need to test it to make sure I've not made any typos. It's also inefficient: the exponential is being evaluated three times, x is being cubed three times and squared three times, and so on. Better would be a simplified form

```
function [f, df]=notCompletelyStupid(x)
  f = x.*(x+2.0).*exp(x.^3);
  df = exp(x.^3).*(x.*(3.0*(x-2.0).*x.^2+2)-2);
```

and better still

```
function [f, df]=maybeSomewhatSmart(x)
  xSq = x.*x;
  xCub= x.*xSq;
  tmp = exp(xCub);
  f = x.*(x+2.0).*tmp;
  df = tmp.*(x.*(3.0*(x-2.0).*xSq+2)-2);
```

In the final version, the exponential is being computed only once (which certainly helps) and the square is being reused in the computation of the cube (which may or may not help, because additional storage is needed). To really do it right requires a good understanding of MATLAB's memory management. The final version has also pretty much obscured what's actually being computed, and there's almost certainly a bug somewhere[1]. Things only get worse with complicated functions: derivatives are simple but often messy, and except for simple cases (polynomials) and lucky cancellations, they usually have more terms than the initial expression. This expansion in complexity is particularly serious because summing many terms with varying magnitude and/or alternate signs can lead to catastrophic round-off error.

3.1 Automating derivatives

Definition 1. The *dual numbers* $\mathbb{D}$ are vectors in $\mathbb{R}^2$ with additional properties. As vectors, they inherit the usual rules for vector addition and scalar-vector multiplication. We'll usually write dual numbers as ordered pairs. The additional properties are

1. $(f,u) \times (g,v) = (fg, fv+gu)$
2. $(f,u) \div (g,v) = \left(\frac{f}{g}, \frac{gu-fv}{g^2}\right)$
3. For every differentiable $f : \mathbb{R} \to \mathbb{R}$, there is a corresponding $F : \mathbb{D} \to \mathbb{D}$ defined by $F((x,u)) = (f(x), f'(x)u)$.
4. A real variable x has the corresponding dual variable $(x,1)$.

The dual numbers are sometimes called the Rall numbers in honor of their inventor Louis Rall.

Once one gets used to the idea, no notational distinction is ordinarily made between a function f and its extension F on the duals; the symbol f would be used for both. On the assumption that you're not yet used to the idea, I'll distinguish the two through capitalization.

[1] After typing enerything in, I looked over the code and detected one bug. To reinforce the point that it's easy to make such mistakes, I deliberately did not fix it.There are probably others.

Theorem 2. *The constant function $f(x) = \alpha$ has dual correspondent $F((x,u)) = (\alpha, 0)$.*

Proof. Immediate from property 3. □

Theorem 3. *The identity map on the reals, $f(x) = x$, has as its dual correspondent the identity map on the duals, $F((x,u)) = (x,u)$*

Proof. Immediate from property 3. □

Theorem 4. *(Dual chain rule) Let f and g be real-valued and differentiable. Then their composition $f \circ g$ has dual correspondent $F \circ G$*

$$(F \circ G)((x,u)) = \left((f \circ g)(x), f'(g)\, g'(x)\, u\right)$$

Proof. Direct calculation using property 3. First find the dual of g,

$$G((x,u)) = \left(g(x), g'(x)\, u\right)$$

then

$$F(G((x,u))) = \left(f(g(x)), f'(g)\, g'(x)\, u\right).$$

□

Example 5. An example is in order. Let's compute x^2 in dual arithmetic. The first step is to write the real variable x as a dual variable $(x,1)$. Then

$$(x,1) \times (x,1) = \left(x^2, x + x\right)$$

$$= \left(x^2, 2x\right).$$

Example 6. Compute e^{x^2} in dual arithmetic. We already know that $(x,1)^2 = \left(x^2, 2x\right)$. Use the dual chain rule to compute

$$e^{\left(x^2, 2x\right)} = \left(e^{x^2}, 2xe^{x^2}\right).$$

By now you understand what's going on: the algebra of dual numbers embodies the rules of differential calculus, so that if you replace calculation of a real $f(x)$ with calculation of its dual $F((x,1))$, the real-valued result $f(x)$ goes into the first slot, its derivative $f'(x)$ into the second. If you want to evaluatef', just compute F on the duals and read out the second slot.

What a roundabout way to compute a derivative! By hand, you'd never organize a calculation this way. But on a computer, once you have a library of routines for dual arithmetic (and the exponentials and logs and so on implemented in duals) all you need to do is replace real calculations with dual calculations. Even that sounds daunting, because it appears you need to rewrite your whole code for the duals. But you don't. There are automated tools to do the conversion for code in most of the popular languages; google "automatic differentiation" (AD) for up-to-date information. I'll call out specifically a few ways to do this:

- In a language such as C++ or Python that supports user-defined data types and "overloaded" operators on them (meaning that you can define `+-*/` for your data types), implement a dual number data type and define appropriate operators for them (as well as the usual elementary functions). If you know these languages this won't seem too hard; however, it is tricky to do efficiently. Try it yourself for fun and learning, but use one of the professional implementations (such as Sacado from Sandia National Laboratories) for high-performance work.
- When using a lower-level language such as C or Fortran, *use a compiler that replaces all FP operations by the corresponding operations on dual numbers.* Such compilers have been written for you by people who know what they're doing. The ADIFOR and ADIC compilers from Argonne National Laboratory and the ADOL/C compiler from Germany can work with your favorite Fortran or C compiler to translate your ordinary code to code that acts on dual numbers.
- MATLAB now has AD add-ons. I've not used them, and can't testify to their ease of use or efficiency.

The point is that if you have some complicated $f(x)$ for which you need exact derivatives, don't do the differentiation by hand (or symbolically) and enter it into your code. Instead, write code to evaluate f, but then tell the computer to do the calculation in dual arithmetic. A good implementation of dual numbers will automatically do optimizations such as caching expensive calculations (such as e^{x^3} in the example) for reuse without recomputation.

Dual numbers are the most convenient way to introduce automatic differentiation, but they're only the tip of the iceberg. The most obvious next step is to extend to D dimensions (in which case a dual number becomes a $D+1$-dimensional vector with rules for vector calculus). Less obvious is to do something similar, but using implicit differentiation; that's called backwards mode automatic differentiation, and is usually more efficient for multivariable problems. Another alternative useful in problems where the same operations must be evaluated at many points is to build and store an expression graph, and walk the graph doing batches of function and derivative evaluations. See the book by Griewank, *Evaluating Derivatives*, for more information. I'll also shamelessly plug a paper by Kevin Long, Rob Kirby, and Bart van Bloemen Waanders that describes how to take these ideas even further, applying automatic differentiation to automate the mapping between high-level problem specifiers and low-level discretizations in the context of high-performance PDE simulations: you write a PDE in high-level language, and AD is used to work out (and then perform) the lower-level operations needed to approximate a solution.

3.1.1 Embedded calculations beyond differentiation

The basic idea of dual numbers is to define extended number systems that "shadow" a primary calculation (a function evaluation) with rules for a subsidiary calculation (*e.g.*, a derivative). Once you think about it, the possibilities are endless. You can shadow the primary calculation with any useful secondary calculation that can be expressed in terms of propagation through the elementary operations. For example, in a calculation where the inputs are random variables with uncertainties due to experimental errors or modeling errors, you can do propagation of uncertainties by keeping a running estimate of the PDF for the

result (this is possible in theory, but is not easy in practice!). An ingeniously simple idea due to Mike Heroux is to shadow each step of a calculation with the identical step done in a different floating point system; at the end of the calculation, the difference between the two results gives an estimate of the effect of roundoff error. Rigorous bounds on roundoff error can be computed by shadowing a calculation with interval arithmetic (see Ackleh *et al* for more about this). You can even combine all of these ideas, as outlined in a recent paper by Pawlowski, Phipps, and Salinger.

The possibilities for efficiently embedding "shadow" calculations is a hot research topic in mathematics and computer science. The problem of computing the humble derivative has taken us from a simple finite difference method known to Newton up to today's cutting-edge computational math.