Practical course on computing derivatives in code

CRAIG SCHROEDER*, University of California Riverside

Derivatives occur frequently in computer graphics and arise in many different contexts. Gradients and often Hessians of objective functions are required for efficient optimization. Gradients of potential energy are used to compute forces. Constitutive models are frequently formulated from an energy density, which must be differentiated to compute stress. Hessians of potential energy or energy density are needed for implicit integration. As the methods used in computer graphics become more accurate and sophisticated, the complexity of the functions that must be differentiated also increases. The purpose of this course is to show that it is practical to compute derivatives even for functions that may seem impossibly complex. This course provides practical strategies and techniques for planning, computing, testing, debugging, and optimizing routines for computing first and second derivatives of real-world routines. This course will also introduce and explore auto differentiation, which encompasses a variety of techniques for obtaining derivatives automatically. The goal of this course is not to introduce the concept of derivatives, how to use them, or even how to calculate them per se. This is not intended to be a calculus course; we will assume that our audience is familiar with multivariable calculus. Instead, the emphasis is on implementing derivatives of complicated computational procedures in computer programs and actually getting them to work.

CCS Concepts: • **Mathematics of computing → Differential calculus**.

Additional Key Words and Phrases: differentiation, automatic differentiation

ACM Reference Format:
Craig Schroeder. 2019. Practical course on computing derivatives in code. In *Proceedings of SIGGRAPH '19 Courses.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3305366.3328073

1 OVERVIEW

This course focuses on the very practical and occasionally daunting task of implementing derivatives in programs. The starting point is assumed to be a function that takes some arguments (scalars, vectors, or matrices), performs some mathematical calculations on those inputs, and then returns one or more outputs (also scalars, vectors, or matrices). The output is a routine that additionally computes derivatives of the outputs with respect to one or more of the inputs. That is, if the starting routine computes $f(x)$ then the final routine will compute $f(x)$ and $f'(x)$. For some applications, second derivatives are also required, in which case the final routine would compute $f(x)$, $f'(x)$, and $f''(x)$. In principle, any number of derivatives might be required, but in practice generally only one or two are required.

The goal of this course is not to introduce the concept of derivatives, how to use them, or even how to calculate them per se. This is not intended to be a calculus course; we will assume that our audience is familiar with multivariable calculus. Instead, the emphasis is on implementing derivatives of complicated computational procedures in computer programs *and actually getting them to work*. This course emphasizes *practical* strategies for working out and implementing derivatives, *testing* strategies to make sure that the result is correct, tips for making the routines *efficient* without sacrificing *debuggability*, and *numerical robustness* considerations.

SIGGRAPH '19 Courses, July 28 - August 01, 2019, Los Angeles, CA, USA

ACM ISBN 978-1-4503-6307-5/19/07.
https://doi.org/10.1145/3305366.3328073

This course will also introduce *auto differentiation*, which encompasses a variety of techniques for obtaining derivatives automatically. We will explore *forward mode* and *reverse mode* auto differentiation to understand what they are and when each is preferred. Finally, we will explore the benefits and limitations of auto differentiation as a replacement for manually computing derivatives.

2 OUTLINE

- Basics - 30 minutes
  - Motivation
  - What not to do
  - Chain rule
  - Tensor notation
  - Efficiency considerations
- Practical considerations - 25 minutes
  - Testing it
  - Getting it to work
  - Optimizing it
  - Numerical difficulties
  - Implicit differentiation
  - Differentiating the function or the algorithm
- Differentiating matrix factorizations - 15 minutes
  - Differentiating singular values
  - Differentiating eigenvalues
- Automatic differentiation - 20 minutes
  - Compile time
  - Runtime
  - Code generation
  - Differentiate the function
  - When to use automatic differentiatiation
  - Automatic differentiation libraries
- Concluding remarks

3 BASICS

Let $f(x)$ be a function that takes as input a scalar x and returns another scalar. For example, $f(x) = 3x^3 + 2$. The derivative of $f(x)$ tells us how the value of $f(x)$ changes as x changes and is defined as

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

In our example, $f'(x) = 9x^2$. Here the prime (f') denotes the derivative of the function; it is frequently used for functions that depend only on one scalar variable. We will also use other notation such as $\frac{df}{dx}(x)$ or $\frac{d}{dx}f(x)$.

This definition is rarely used to directly calculate derivatives. Instead, it is used to derive rules, which are much simpler to apply. For reference, this is a short table of derivative rules. It is by no means exhaustive, but it includes the rules that we will need to get started. More extensive lists of rules are available readily available in calculus textbooks.

Let $f(x)$ and $g(x)$ be functions of the scalar x. Let a and b be constants. Then

$$\frac{d}{dx}(af(x) + bg(x)) = af'(x) + bg'(x) \qquad \text{differentiation is linear}$$

$$\frac{d}{dx}(f(x)g(x)) = f'(x)g(x) + f(x)g'(x) \qquad \text{product rule}$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2} \qquad \text{quotient rule; } g(x) \neq 0$$

$$\frac{d}{dx}(f(x)^n) = nf(x)^{n-1}f'(x) \qquad \text{power rule; } n \neq 0$$

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x) \qquad \text{chain rule}$$

Derivatives of vectors. Frequently we must compute derivatives of vectors. For example, if $\mathbf{x}(t)$ is a vector representing the position of an object, then $\mathbf{v}(t) = \mathbf{x}'(t)$ is its velocity and $\mathbf{a}(t) = \mathbf{x}''(t)$ is its acceleration. Derivatives of vectors are computed componentwise. Computing derivatives with vectors requires more rules, but these can all be derived from the scalar case by considering the components. For example, if $f(t) = \mathbf{u}(t) \cdot \mathbf{v}(t)$, then $f'(t) = \mathbf{u}'(t) \cdot \mathbf{v}(t) + \mathbf{u}(t) \cdot \mathbf{v}'(t)$; that is, dot product follows the product rule. Cross products also follow the chain rule, though care must be taken not to change the order of the cross product.

Derivatives with respect to vectors. In optimization, if is common for an objective to depend on a vector (either physically or by simply stacking all of the inputs into a vector). For example, I may wish to minimize $f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x}$. In this case, we must compute the derivative of the function with respect to each component of the vector. If the function returns a scalar, then the derivative will be a vector, which we generally call the *gradient* and denote as ∇f or $\nabla f(\mathbf{x})$. If $\mathbf{x}$ is a 3D vector, then

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \end{pmatrix},$$

where $\frac{\partial f}{\partial x_1}$ means the derivative should be taken of f with respect to x_1 (the first component of $\mathbf{x}$) while the other components are treated as constants.

More extensions. We can also consider derivatives of vector-valued functions with respect to vector inputs, in which case the result is a matrix. We use the following convention in this course:

$$\nabla \mathbf{f} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{pmatrix}.$$

Note that the first index indicates which component of $\mathbf{f}$ is being differentiated. The second index indicates which component of the input we are differentiating with respect to. It is also possible to differentiate matrices (also componentwise) or to differentiate other quantities with respect to matrices (also componentwise in a manner analogous to the gradient). For example, differentiating a matrix-valued function by a vector would produce a rank-3 tensor.

3.1 Motivation

Derivatives come up in computer graphics in many different contexts. Many problems are formulated as optimization problems. As a simple case

$$\text{Minimize: } f(\mathbf{x}).$$

Here, $f(\mathbf{x})$ might be a measure of error, where $\mathbf{x}$ is a vector filled with parameters. To optimize this function, we generally like to move from our current configuration $\mathbf{x}$ to a nearby configuration $\mathbf{x} + \delta\mathbf{x}$ that produces a lower error. The gradient ∇f gives us a direction in which we can move to find a smaller error. When the gradient is zero ($\nabla f = \mathbf{0}$), we have found a (locally) optimal solution. Most optimization algorithms use the gradient. More efficient methods also use the *Hessian*, which is the matrix filled with second derivatives, $\mathbf{H} = \frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{x}}$. To use these, we must differentiate our error function $f(\mathbf{x})$ twice. Often this error function is something rather complicated, such as a discrete measure for curvature. In machine learning, optimization procedure is used for training the network.

Derivatives are also ubiquitous in physics and thus physical simulation. One common way to define forces is to instead construct a potential energy function $\phi(\mathbf{x})$, from which the forces can be computed from the gradient as $\mathbf{f} = -\nabla\phi$. Implicit methods frequently involve the force derivatives $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, which involves the Hessian of the potential energy function. Thus, we often must compute first and second derivatives of scalars with respect to vectors.

To give a motivation for the types of energies that must sometimes be differentiated, consider this rather complex example from a graphics paper.

$$\begin{aligned}
d_{min} &= (\mathbf{x}_A - \mathbf{c}) \cdot \frac{(\mathbf{x}_C - \mathbf{x}_A) \times (\mathbf{x}_B - \mathbf{x}_A)}{\|(\mathbf{x}_C - \mathbf{x}_A) \times (\mathbf{x}_B - \mathbf{x}_A)\|} \\
\cot(\alpha) &= \frac{(\mathbf{x}_C - \mathbf{x}_A) \cdot (\mathbf{x}_B - \mathbf{x}_A)}{\|(\mathbf{x}_C - \mathbf{x}_A) \times (\mathbf{x}_B - \mathbf{x}_A)\|} \\
\cot(\beta) &= \frac{(\mathbf{x}_A - \mathbf{x}_B) \cdot (\mathbf{x}_C - \mathbf{x}_B)}{\|(\mathbf{x}_A - \mathbf{x}_B) \times (\mathbf{x}_C - \mathbf{x}_B)\|} \\
\cot(\gamma) &= \frac{(\mathbf{x}_B - \mathbf{x}_C) \cdot (\mathbf{x}_A - \mathbf{x}_C)}{\|(\mathbf{x}_B - \mathbf{x}_C) \times (\mathbf{x}_A - \mathbf{x}_C)\|} \\
E_{Dirichlet} &= d_{min}^{-2}(\cot(\alpha)\|\mathbf{x}_B - \mathbf{x}_C\|^2 + \cot(\beta)\|\mathbf{x}_A - \mathbf{x}_C\|^2 + \cot(\gamma)\|\mathbf{x}_A - \mathbf{x}_B\|^2) \\
E_{area} &= d_{min}^{-2} \frac{\|(\mathbf{x}_A - \mathbf{x}_C) \times (\mathbf{x}_B - \mathbf{x}_C)\|^2}{InputArea} \\
E = E_{combined} &= a \cdot E_{Dirichlet} + b \cdot E_{area}
\end{aligned}$$

Here, two derivatives of E are required with respect to $\mathbf{x}_A$, $\mathbf{x}_B$, and $\mathbf{x}_C$. In the original paper, the authors computed the derivatives in Maple.

Another place where difficult derivatives frequently emerge is in constitutive modeling. Here, we define an energy density $\psi(\mathbf{F})$, which is a scalar defined as a function of a matrix (the deformation gradient $\mathbf{F}$). The first derivative yields the first Piola-Kirchoff stress $\mathbf{P} = \frac{\partial \psi}{\partial \mathbf{F}}$. For implicit methods, the second derivative is also required. That is, we need $\frac{\partial \mathbf{P}}{\partial \mathbf{F}}$, which is a rank-4 tensor. In practice, it is often sufficient to compute double-contractions $\frac{\partial \mathbf{P}}{\partial \mathbf{F}} : \mathbf{A}$ with a matrix $\mathbf{A}$. Implementing a routine that takes in a matrix $\mathbf{A}$ and returns this double contraction (also a matrix) is often easier and more efficient than computing and storing a rank-4 tensor. Many commonly-used constitutive models in computer graphics are defined in terms of the singular values of $\mathbf{F}$, which will require us to differentiate the singular values. Although this seems daunting, it is actually quite doable; we will show how to do this later in the course.

3.2 What not to do

When the function to be differentiate seems daunting, there are a few tempting solutions that authors sometimes resort to to avoid the task.

Finite differences. Finite differences are a flexible approach to approximating derivatives on grids. They are one of the most commonly employed and useful discretization techniques for PDEs. They allow one to approximate a derivative by evaluating the function at nearby points and applying the definition of the derivative. For example,

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

There are a number of reasons why this is a *bad idea.* The first of these is that the derivatives are only going to be approximate, with an error that depends on the displacements h that are used. Approximate derivatives may cause numerical optimization routines to fail. For gradients with n components, the function must be evaluated at least $n + 1$ times, which can be quite expensive. A less obvious problem is that the roundoff errors may be very bad if h is small. In this case, the numerator is the difference between two nearly identical quantities, which results in catastrophic cancellation.

Maple/Mathematica. A rather tempting way to obtain the actual derivatives is to use Maple, Mathematica, or another computer algebra system to compute the derivatives. These software packages even have commands to dump out expressions as source code in languages like C, which makes this even more tempting. There are four major problems with this strategy. First, the resulting expressions tend to be very long and require significant manual simplification. The second problem with them is that they tend to not be very robust (division by zero, square root of a negative number). A third problem is that they are often not very efficient, since they do not reuse intermediate computations. In our case, however, the biggest problem with them is that they are neither debuggable nor maintainable. What if you made a mistake while you were simplifying the expressions? How would you ever find the mistake?

Some of these problems can be illustrated on the rather modest example $f(\mathbf{u}, \mathbf{v}) = \left\| \mathbf{u}(\mathbf{u} \cdot \mathbf{v})^2 - \mathbf{v}\|\mathbf{u}\|^3 \right\|^2$. Computing $\mathbf{H} = \frac{\partial^2 f}{\partial \mathbf{u} \partial \mathbf{u}}$ with Maple, asking Maple to simplify, and then generating C code for just the entry H_{11} (with optimization enabled and basic manual simplifications performed) produces

```
t1 = v2 * v2; t4 = v1 * v1; t6 = t1 * t1; t8 = t1 / 10; t9 = v3 * v3; t10 = t9 / 10; t12 = u2 * u2; t13 = t12 * t12;
t19 = u1 * v1; t24 = t12 * u2; t31 = t9 / 5; t33 = u3 * u3; t35 = 3 * t1; t41 = u1 * u1; t42 = t4 * t4; t46 = 3 * t9;
t52 = u3 * v3; t61 = 0.8 * u1 * u3 * v1 * v3; t73 = t33 * t33; t77 = t33 * u3; t88 = t41 * u1; t94 = t41 * t41;
t100 = sqrt(t41 + t12 + t33); t102 = t4 / 5;
H11 = -60 / t100 * (t100 * (t13 * (t4 * (-t1 / 5 - 0.1) - t6 / 30 - t8 - t10) - 0.4 * t24 * v2 * (u3 * (t4 + t1 / 3) * v3
     + (t4 + t1) * t19) + t12 * (t33 * (t4 * (-t1 - t9 - 1) / 5 + t1 * (-t9 - 1) / 5 - t31) - 0.4 * u3 * (t4 + t35) * v3 * t19
     - (t42 + t4 * (6 * t1 + 3) + t35 * t46) * t41 / 5) - 4. / 3 * u2 * (t33 * (0.3 * t4 + t10) + t61 + t4 * t41) * v2 * (t19 + t52)
     + t73 * (t4 * (-t31 - 0.1) - t8 - (t9 + 3) * t9 / 30) - 0.4 * t77 * (t4 + t9) * v3 * t19 - t33 * (t42 + t4 * (6 * t9 + 3) + t35
     + t46) * t41 / 5 - 4. / 3 * v3 * t4 * v1 * u3 * t88 - (t42 + t4 + t1 + t9) * t94 / 2) + (u2 * v2 + t19 + t52) * (t13 * (t102 + t8)
     + 0.8 * t24 * v2 * (t19 + t52 / 4) + t12 * (t33 * (0.4 * t4 + t8 + t10) + t61 + 1.1 * (t4 + 2. / 11 * t1) * t41) + u2 * (t88 * v1
     + 0.4 * t41 * t52 + 0.8 * u1 * v1 * t33 + v3 * t77 / 5) * v2 + t73 * (t102 + t10) + 0.8 * v3 * u1 * v1 * t77 + 1.1 * t33 * t41 * (t4
     + 2. / 11 * t9) + t88 * v3 * u3 * v1 + t4 * t94));
```

Note that this is just the entry H_{11}. There will also be similar expressions for H_{12}, H_{13}, H_{22}, H_{23}, and H_{33}. If I made a mistake in simplifying any of these expressions, how would anyone debug it? I will return to this example later and show how to do this example manually.

3.3 Chain rule

They key to implementing derivatives practically is the chain rule. As a simple example, lets compute the derivative of $f(x) = (x^3 + \sqrt{1 + x^2})^2$. The first step is to break the computation into

very small pieces.

$$a = 1 + x^2 \qquad b = x^3 \qquad c = \sqrt{a} \qquad d = b + c \qquad f = d^2$$

Next, we can compute the derivative of each step using the chain rule.

$$a' = 2x \qquad b' = 3x^2 \qquad c' = \frac{a'}{2c} \qquad d' = b' + c' \qquad f' = 2dd'$$

There is no need to combine all of the pieces together. This is a good form for implementation in code. What is more, it is very easy to see how the derivatives were computed and to see mistakes. If the original expression must change, the derivatives are easy to update accordingly. Of course, we can simplify the expressions when we get them working.

What if we need second derivatives? Surely that would far too complex. But in fact it is quite manageable, since all of the equations we must differentiate are very simple. We just differentiate our formulas for the first derivatives.

$$a'' = 2 \qquad b'' = 6x \qquad c'' = \frac{a''}{2c} - \frac{a'c'}{2a} \qquad d'' = b'' + c'' \qquad f'' = 2(d')^2 + 2dd''$$

Note that I reused a lot of the work I already did for the first derivatives. As a bonus, when computing the second derivatives I basically get the function and first derivatives for free.

Lets now return to our more complex example: $f(\mathbf{u},\mathbf{v}) = \left\|\mathbf{u}(\mathbf{u}\cdot\mathbf{v})^2 - \mathbf{v}\|u\|^3\right\|^2$. This was the example where Maple gave us very complicated results. This example also includes lots of vectors, which will make it more complicated than the very simple scalar example we just did. First, break up the function into small pieces. Smaller is better. Ideally, I want pieces that will stay relatively simple when I compute the second derivatives. If only first derivatives are desired, this is less of a concern.

$$a = \mathbf{u}\cdot\mathbf{v} \qquad b = \|\mathbf{u}\| \qquad c = a^2 \qquad d = b^3 \qquad \mathbf{w} = c\mathbf{u} - d\mathbf{v} \qquad f = \|\mathbf{w}\|^2$$

Next, I take the gradient of each equation in sequence. Note that the gradients of scalars (such as a_u) are vectors, and the gradients of vectors (such as $\mathbf{w}_u$) are actually matrices.

$$a_u = \mathbf{v} \qquad b_u = \frac{\mathbf{u}}{b} \qquad c_u = 2aa_u \qquad d_u = 3b^2b_u \qquad \mathbf{w}_u = c\mathbf{I} + \mathbf{u}c_u^T - \mathbf{v}d_u^T \qquad f_u = 2\mathbf{w}_u^T\mathbf{w}$$

First derivatives were fairly straightforward. Second derivatives will be more difficult, but lets get started with the easier ones.

$$a_{uu} = \mathbf{0} \qquad b_{uu} = \frac{1}{b}(\mathbf{I} - b_u b_u^T) \qquad c_{uu} = 2a_u a_u^T \qquad d_{uu} = 6bb_u b_u^T + 3b^2 b_{uu}$$

Getting the second derivatives on $\mathbf{w}$ is awkward due $\mathbf{w}_{uu}$ being a rank-3 tensor, but using indexing techniques we will introduce later, we are able to eliminate the tensors and get

$$\mathbf{z} = \mathbf{w}\cdot\mathbf{w}_{uu} = (\mathbf{u}\cdot\mathbf{w})c_{uu} + c_u\mathbf{w}^T + \mathbf{w}c_u^T + (\mathbf{v}\cdot\mathbf{w})d_{uu} \qquad f_{uu} = 2\mathbf{z} + 2\mathbf{w}_u^T\mathbf{w}_u$$

Note that unlike in the Maple example, where we only computed the first entry of the Hessian matrix, here we have computed the entire matrix. The resulting expressions are relatively simple and take advantage of existing linear algebra routines. Although these may also seem difficult to debug, we will show later how they can be debugged in a very straightforward way.

3.4 Tensor notation

The last example showed us that there are a couple hurdles we may need to deal with when dealing with vectors, matrices, and derivatives. The first problem is that the vector derivative rules can be quite obscure. Does anyone really remember the rule for $\nabla(\mathbf{u} \cdot \mathbf{v})$? Is it $(\nabla\mathbf{u})\mathbf{v} + (\nabla\mathbf{u})\mathbf{v}$? Or was it $(\nabla\mathbf{u})^T\mathbf{v} + (\nabla\mathbf{u})^T\mathbf{v}$? (It is the latter.) How about the rule for $\nabla \cdot (\mathbf{u} \times \mathbf{v})$? The second problem is that when we have vectors, we tend to end up needing rank-3 tensors. When we are differentiating with respect to matrices, we tend to get rank-4 tensors. Vector notation really is not up to the task here. Indexed tensor notation, though tedious, turns out to work very well here.

The idea behind tensor notation is that we refer to objects by their indices rather than writing them as vectors or matrices directly. Thus, for example, we might write the vector $\mathbf{u}$ as u_i or the matrix $\mathbf{A}$ as A_{ij}. The outer product $\mathbf{A} = \mathbf{u}\mathbf{v}^T$ looks like $A_{ij} = u_i v_j$. The names of the indices are not important. I could just as easily have written $A_{rs} = u_r v_s$. A rank-3 and rank-4 tensors simply have more indices, such as B_{ijk} or C_{ijkl}. They cause us no problems at all.

A dot product looks like $\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$. A very useful observation is that repeated indices in a term are virtually always summed over. This gave rise to the Einstein summation convention, where indices that occur twice in a term are *implicitly* summed. The summation symbol is omitted but understood to be there. Now we can write dot product as $\mathbf{u} \cdot \mathbf{v} = u_i v_i$. Again, the name of the index does not matter. I could write $u_r v_r$ instead. Matrix-vector multiply looks like $v_i = A_{ij}u_j$. Matrix-matrix product looks like $A_{ik} = B_{ij}C_{jk}$. $\mathbf{A}^T\mathbf{u}$ looks like $A_{ji}u_j$. Note the reversed indices on the matrix. Trace is just A_{ii}. The summation convention is elegant indeed.

At this point, we see that the indices are very important. $u_i v_j$ is an outer product matrix, whereas $u_i v_i$ represents a dot product and is a scalar. One more thing to remember is that the u_i is actually the component of the vector, so the $u_i v_i$ is really product of two scalars, with an implied summation sign. As such, we simply use the usual scalar differentiation rules.

Before jumping to differentiation, we need a couple special tensors. δ_{ij} is the identity matrix. It is 1 if $i = j$ and 0 if $i \neq j$. The other matrix is the permutation tensor e_{ijk}, which is only defined in 3D. $e_{123} = e_{312} = e_{231} = 1$, $e_{132} = e_{321} = e_{213} = -1$, and all other entries are zero. This tensor will allow us to do cross products. $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ becomes $w_i = e_{ijk}u_j v_k$. Note that δ_{ij} and e_{ijk} are filled with constants, so their derivatives vanish.

Now we are ready to start doing derivatives. To do this, we need a notation for gradients. This is done with a comma before an index. Thus, the gradient of the scalar a is $a_{,r}$. The gradient of the vector u_i is $u_{i,r}$. The second derivative is $u_{i,rs}$. The vector Laplacian, for example, looks like $u_{i,rr}$. Note that something like $a_{,rs}$ represents the second derivative of a scalar with respect to a vector. (If needed, a notation like $a_{,(rs)}$ could be used to indicate a first derivative with respect to a matrix, but we will not worry about that here.) Observe that $\nabla(\mathbf{u} \cdot \mathbf{v})$ becomes $(u_i v_i)_{,r} = u_{i,r}v_i + u_i v_{i,r}$ simply by applying the scalar product rule and shows the source of the transposes in the vector notation. $\nabla \cdot (\mathbf{u} \times \mathbf{v})$ likewise becomes $(e_{ijk}u_j v_k)_{,r} = e_{ijk}u_{j,r}v_k + e_{ijk}u_j v_{k,r}$ by just using the product rule. One of the beauties of this notation is that there are few rules to remember.

Now lets put this notation to use on something that caused us problems. Lets return to the rank-3 tensor $\mathbf{w}_{uu}$ and the seemingly magical expression for $\mathbf{w} \cdot \mathbf{w}_{uu}$. Recall that $u_{i,s} = \delta_{is}$, since the

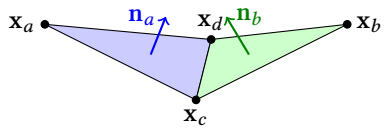

Fig. 1. Two triangles connected by an edge.

derivative is with respect to u_s. Observe that $\delta_{ij}v_j = v_i$, since δ_{ij} is the identity.

$$\begin{aligned}
\mathbf{w}_u &= c\mathbf{I} + \mathbf{u}c_u^T - \mathbf{v}d_u^T \\
w_{i,r} &= c\delta_{ir} + u_i c_{,r} - v_i d_{,r} \\
w_{i,rs} &= c_{,s}\delta_{ir} + u_i c_{,rs} + u_{i,s}c_{,r} - v_i d_{,rs} \\
w_i w_{i,rs} &= w_i c_{,s}\delta_{ir} + w_i u_i c_{,rs} + w_i \delta_{is} c_{,r} - w_i v_i d_{,rs} \\
w_i w_{i,rs} &= w_r c_{,s} + (w_i u_i) c_{,rs} + c_{,r} w_s - (w_i v_i) d_{,rs} \\
\mathbf{w} \cdot \mathbf{w}_{uu} &= \mathbf{w}c_u^T + (\mathbf{w}\cdot\mathbf{u})c_{uu} + c_u\mathbf{w}^T - (\mathbf{w}\cdot\mathbf{v})d_{uu}
\end{aligned}$$

This calculation in tensor notation is relatively straightforward.

Differentiation with respect to many variables. Another consideration with vector notation is the need to indicate which of potentially many variables are being differentiated with respect to. In the previous example, we may actually want f_{uu}, $f_{uv} = f_{vu}$, and f_{vv}. In this case, it is convenient to imagine that $w_{i,rs}$ is referring to differentiation with respect to some components r and s of $\mathbf{u}$ *or* $\mathbf{v}$. In this case, we would not simplify $u_{i,r}$ to δ_{ir} and $v_{i,r}$ to $\mathbf{0}$ while working out the derivation. Instead, when we would implement chain of computations and make three identical copies of it (with different variable names), one copy for f_{uu}, f_{uv}, and f_{vv}. In the f_{uu} and f_{uv} copies of the code, we let $u_{i,r} = \delta_{ir}$. In the f_{vv} copy we let $u_{i,r} = \mathbf{0}$. This will result in a lot of simple optimization opportunities, which can be easily performed (and debugged) once a working version of the derivatives is completed.

Another common case is when we are working our derivatives in unrelated variables. For example, if I need to work out $\frac{\partial \mathbf{u}}{\partial \mathbf{w}} = \frac{\partial \mathbf{u}}{\partial \mathbf{v}}\frac{\partial \mathbf{v}}{\partial \mathbf{w}}$, I will need to distinguish between differentiation in $\mathbf{w}$ and $\mathbf{v}$. One simple way to do this is to reserve different sets of index names for each. For example, I can use r, s for $\mathbf{w}$ and α, β for $\mathbf{v}$. This would allow me to unambiguously write $u_{i,r} = u_{i,\alpha}v_{\alpha,r}$ It is also convenient to designate different indices for different types of indices more generally; for example, I might reserve p, q for particle indices, i, j for grid indices, and then use Greek letters for spatial indices (such as components of vectors or gradient indices).

3.5 Efficiency considerations

Often we are interested in computing partial derivatives with respect to many variables. For example, lets say we want to compute a simple bending energy for two triangles connected by a line segment. (See Figure 1.) A simple bending energy depending on the angle between the triangles is

$$\begin{aligned}
&\mathbf{u} = \mathbf{x}_a - \mathbf{x}_d \qquad \mathbf{v} = \mathbf{x}_b - \mathbf{x}_d \qquad \mathbf{w} = \mathbf{x}_c - \mathbf{x}_d \qquad \mathbf{N}_a = \mathbf{u}\times\mathbf{w} \qquad \mathbf{N}_b = \mathbf{w}\times\mathbf{v} \\
&\mathbf{n}_a = \frac{\mathbf{N}_a}{\|\mathbf{N}_a\|} \qquad \mathbf{n}_b = \frac{\mathbf{N}_b}{\|\mathbf{N}_b\|} \qquad r = \mathbf{n}_a\cdot\mathbf{n}_b \qquad s = \|\mathbf{n}_a\times\mathbf{n}_b\| \qquad \theta = \operatorname{atan2}(s,r)
\end{aligned}$$

Straightforward application of the above approach would involve computing first derivatives and second derivatives in $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d$ for each of the intermediates $\mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{N}_a, \mathbf{N}_b, \mathbf{n}_a, \mathbf{n}_b, r, s, \theta$. Although many of these are zero (for example, $\mathbf{N}_a$ does not depend on $\mathbf{x}_b$), for r, s, and θ we require

the calculation of 4 first derivatives (vectors) and 10 second derivatives (matrices) each. This is inefficient, and we can do better.

Modes of differentiation. Lets say that we have a calculation where x is the input, y is the output, and $a = a(x)$, $b = b(a)$, and $c = c(b)$ are intermediates with the result computed as $y = y(c)$. Then, our strategy was to compute

$$\frac{\partial a}{\partial x} = \frac{\partial a}{\partial x} \qquad \frac{\partial b}{\partial x} = \frac{\partial b}{\partial a}\frac{\partial a}{\partial x} \qquad \frac{\partial c}{\partial x} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial x} \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial x}$$

This is called *forward mode* differentiation. But we did not need to do it this way. Here is another way to compute the same thing.

$$\frac{\partial y}{\partial c} = \frac{\partial y}{\partial c} \qquad \frac{\partial y}{\partial b} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial b} \qquad \frac{\partial y}{\partial a} = \frac{\partial y}{\partial b}\frac{\partial b}{\partial a} \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial x}$$

This is called *reverse mode* differentiation [Bartholomew-Biggs et al. 2000]. (For those familiar with machine learning techniques, the technique of backpropagation of errors is really an application of reverse mode differentiation [Baydin et al. 2018].)

To see that these modes can have different computational cost, consider two different cases. In the first case, x is a vector but a, b, c, y are scalars. The forward mode calculation requires three scalar-vector multiplies, since all of the derivatives are vector quantities. In the reverse case, $\frac{\partial y}{\partial x}$ is a vector and the rest are scalars, so the resulting computations are cheaper. If on the other hand y is a vector but the rest are scalars, the conclusion is the opposite. All of the reverse mode derivatives are vector quantities but only one for the forward mode.

Consider that $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{y}$ are all vectors. Then, the same calculation looks like

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \frac{\partial \mathbf{y}}{\partial \mathbf{c}}\frac{\partial \mathbf{c}}{\partial \mathbf{b}}\frac{\partial \mathbf{b}}{\partial \mathbf{a}}\frac{\partial \mathbf{a}}{\partial \mathbf{x}}$$

The solution is obtained as the product of four matrices. (Each of the derivatives is the partial derivative of a vector with respect to a vector, which results in a matrix.) We are free to perform the matrix-matrix multiplications in any order we like, and we can choose the order that is cheapest. For example, if the dimensions of $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{y}$ are $4, 3, 20, 2, 5$ then the matrices have dimensions 5×2, 2×20, 20×3, and 3×4. The optimal order is then

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \frac{\partial \mathbf{y}}{\partial \mathbf{c}}\left(\left(\frac{\partial \mathbf{c}}{\partial \mathbf{b}}\frac{\partial \mathbf{b}}{\partial \mathbf{a}}\right)\frac{\partial \mathbf{a}}{\partial \mathbf{x}}\right).$$

We call this *mixed mode*. By contrast, forward mode performs the multiplications from right to left. Reverse mode performs them left to right. Note that $\mathbf{c}$ and $\mathbf{a}$ both small sets of intermediate variables from which the result can be computed. I will refer to small sets of intermediates like this as *bottlenecks*. Note that the optimal ordering saves the bottlenecks states for last. Although this is not guaranteed to be the case (the task of determining the optimal ordering is called the optimal Jacobian accumulation problem; it is NP-complete [Naumann 2008]), it is a very useful heuristic.

Let us return to the triangle bending angle example, whose calculation dependencies are shown in Figure 2. We can think of the intermediates in this example as being the vectors

$$\mathbf{x} = \begin{pmatrix}\mathbf{x}_a\\ \mathbf{x}_b\\ \mathbf{x}_c\\ \mathbf{x}_d\end{pmatrix} \qquad \mathbf{z} = \begin{pmatrix}\mathbf{u}\\ \mathbf{v}\\ \mathbf{w}\end{pmatrix} \qquad \mathbf{N} = \begin{pmatrix}\mathbf{N}_a\\ \mathbf{N}_b\end{pmatrix} \qquad \mathbf{n} = \begin{pmatrix}\mathbf{n}_a\\ \mathbf{n}_b\end{pmatrix} \qquad \mathbf{p} = \begin{pmatrix}r\\ s\end{pmatrix} \qquad \theta = \theta$$

with dimensions 12, 9, 6, 6, 2, and 1. Note that these matrices are not fully dense. For example, the 6×6 matrix $\frac{\partial \mathbf{n}}{\partial \mathbf{N}}$ is composed of two 3×3 blocks on the diagonal, since $\mathbf{n}_a$ does not depend

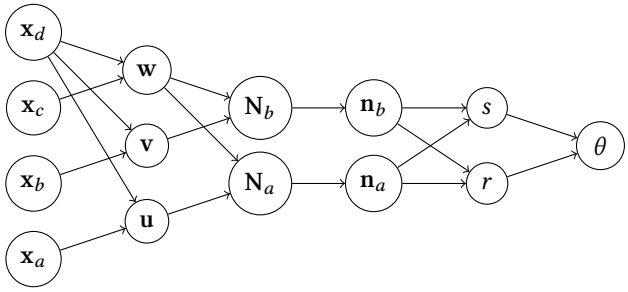

Fig. 2. Computational dependencies for angle between two triangles.

on $\mathbf{N}_b$ and $\mathbf{n}_b$ does not depend on $\mathbf{N}_a$. The matrix $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ contains few nonzeros, all of which are ± 1. Nevertheless, this suggests that the optimal order here is reverse mode. That is, we should be computing

$$\frac{\partial \theta}{\partial \mathbf{p}} \quad \frac{\partial \theta}{\partial \mathbf{n}} \quad \frac{\partial \theta}{\partial \mathbf{N}} \quad \frac{\partial \theta}{\partial \mathbf{z}} \quad \frac{\partial \theta}{\partial \mathbf{x}} \quad \frac{\partial^2 \theta}{\partial \mathbf{p} \partial \mathbf{p}} \quad \frac{\partial^2 \theta}{\partial \mathbf{n} \partial \mathbf{n}} \quad \frac{\partial^2 \theta}{\partial \mathbf{N} \partial \mathbf{N}} \quad \frac{\partial^2 \theta}{\partial \mathbf{z} \partial \mathbf{z}} \quad \frac{\partial^2 \theta}{\partial \mathbf{x} \partial \mathbf{x}}$$

Since θ is actually just a scalar, the first derivatives are all vectors and the second derivatives are all matrices. This suggests that we do not need any tensors, but this may not actually be the case. For example, we will need to deal with $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{N} \partial \mathbf{N}}$, which is a rank-3 tensor. For explicitness, the calculation rules take the form

$$\frac{\partial \theta}{\partial \mathbf{n}} = \frac{\partial \theta}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{n}} \qquad \frac{\partial^2 \theta}{\partial \mathbf{n} \partial \mathbf{n}} = \left(\frac{\partial \mathbf{p}}{\partial \mathbf{n}} \right)^T \frac{\partial^2 \theta}{\partial \mathbf{p} \partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{n}} + \frac{\partial \theta}{\partial \mathbf{p}} \frac{\partial^2 \mathbf{p}}{\partial \mathbf{n} \partial \mathbf{n}}$$

Using i, j for indices into $\mathbf{n}$ and r, s for indices into $\mathbf{p}$, we can write this in index notation as

$$\theta_{,i} = \theta_{,r} p_{r,i} \qquad \theta_{,ij} = \theta_{,rs} p_{r,i} p_{s,j} + \theta_{,r} p_{r,ij}$$

Note the pattern of summation on the rank-3 tensors, which is less clear in the vector notation. If θ was instead a vector (indexed by k), then

$$\theta_{k,i} = \theta_{k,r} p_{r,i} \qquad \theta_{k,ij} = \theta_{k,rs} p_{r,i} p_{s,j} + \theta_{k,r} p_{r,ij}$$

4 PRACTICAL CONSIDERATIONS

At this point, the basic ideas for computing derivatives are all here. The next problem will be actually getting them to work.

4.1 Testing it

As with most things, code cannot be assumed to be working until it has been tested and shown to be working. Derivatives are no different. Observing that Newton's method for optimization converged does not mean the derivatives are correct, and fixing the derivatives may accelerate the convergence. Thus, we need a practical way to test the derivatives. One obvious way to do this is to compute the derivatives in another way (such as Maple/Mathematica or finite differences) and then compare them to see if they match. This works, but there is a much more convenient way to do this.

One nice property of the approach advocated above, namely breaking the computation into tiny pieces, is that the implementation can be tested in tiny pieces. The tool for doing this is something I will refer to as the derivative test. The test will take different forms depending on the type of

object being considered (scalar, vector, matrix, etc.) In its basic form, I compute both z and z'. I can test to see if this is right by using the definition of a derivative.

$$\frac{z(x+\Delta x)-z(x)}{\Delta x}-z'(x)=O(\Delta x)$$

There are a couple ways of using this. One way is to do a refinement test to verify that convergence is observed. This is in practice more than is required. It is often sufficient to verify that the error is *small*. When doing so, this in not the best formula to use. Better is to do a central difference and central average.

$$\frac{z(x+\Delta x)-z(x-\Delta x)}{2\Delta x}-\frac{z'(x+\Delta x)+z'(x-\Delta x)}{2}=O(\Delta x^2)$$

There is now a much bigger separation between a passing score $O(\Delta x^2)$ and a failing score $O(1)$. When using this formula, it is best to do the test using $\Delta x \approx \epsilon^{1/3}$, where ϵ is machine precision. This is because one factor of Δx is lost due to cancelation in the central difference, leaving two factors of Δx left to distinguish a passing and failing score. This test should always be done in double precision, in which case I typically choose Δx randomly in the interval $[-\delta,\delta]$, where $\delta=10^{-6}$. Note that an average is used for the derivative rather than a midpoint approximation, since this way all computations are done at two locations; this is convenient since the value will often be computed along with the derivatives.

When $\mathbf{x}$ is a vector, the perturbation $\delta\mathbf{x}$ is also a vector, whose entries are chosen uniformly in the range $[-\delta,\delta]$. The test must be altered somewhat, since division is no longer possible. Note that the factor of 2 can be omitted.

$$z(\mathbf{x}+\Delta\mathbf{x})-z(\mathbf{x}-\Delta\mathbf{x})-(\nabla z(\mathbf{x}+\Delta\mathbf{x})+\nabla z(\mathbf{x}-\Delta\mathbf{x}))\cdot\delta\mathbf{x}=O(\delta^3)$$

In practice, it is more convenient to keep failing scores at $O(1)$, so one would use

$$\frac{z(\mathbf{x}+\Delta\mathbf{x})-z(\mathbf{x}-\Delta\mathbf{x})-(\nabla z(\mathbf{x}+\Delta\mathbf{x})+\nabla z(\mathbf{x}-\Delta\mathbf{x}))\cdot\delta\mathbf{x}}{\delta}=O(\delta^2)$$

If $\mathbf{z}(\mathbf{x})$ is a vector, then $\nabla\mathbf{z}(\mathbf{x})$ is a matrix, and the test can be performed as

$$\frac{\|\mathbf{z}(\mathbf{x}+\Delta\mathbf{x})-\mathbf{z}(\mathbf{x}-\Delta\mathbf{x})-(\nabla\mathbf{z}(\mathbf{x}+\Delta\mathbf{x})+\nabla\mathbf{z}(\mathbf{x}-\Delta\mathbf{x}))\delta\mathbf{x}\|}{\delta}=O(\delta^2)$$

Any norm can be used; L^2 or L^∞ are reasonable choices. Other cases, such as where $\mathbf{x}$ or $\mathbf{z}$ are a matrix are treated similarly. Note that if $\mathbf{x}$ is a matrix, one would design the test so that it behaved the same as if $\mathbf{x}$ were instead flattened into a vector.

There are some other practical tips that can be used to improve the test. One of these is to print out $\|\mathbf{z}(\mathbf{x}+\Delta\mathbf{x})-\mathbf{z}(\mathbf{x}-\Delta\mathbf{x})\|/\delta$, $\|(\nabla\mathbf{z}(\mathbf{x}+\Delta\mathbf{x})+\nabla\mathbf{z}(\mathbf{x}-\Delta\mathbf{x}))\delta\mathbf{x}/2\|/\delta$, and the relative error, which is obtained by dividing the above error by the maximum of these quantities, while preventing the denominator from being zero. This improves the test for a number of reasons. One is that it makes the test invariant with respect to the absolute scale of $\mathbf{z}$. Another is that it gives an idea of the size of the quantities being computed; if they are very tiny, then they may actually be roundoff approximations of zero, in which case the relative error will be huge, but the test should still be considered as a pass. One more benefit is that the mistake of having the answer off by a simple factor (usually being off by a sign) will be very obvious from the output.

Testing second derivatives is done in the same way; in this case, $\mathbf{z}$ will be the first derivatives, and $\nabla\mathbf{z}$ will be the Hessian.

Reverse mode vs forward mode. Derivative testing seems more natural in forward mode. The inputs to the function are being differentiated with respect to throughout the calculations, which makes it easy to do the derivative tests. But in fact, the technique works well in reverse mode or even in arbitrary orders. All that is required to test $\frac{\partial y}{\partial x}$ is the availability of two slightly different values for x, along with corresponding y and $\frac{\partial y}{\partial x}$. These will be available even if x and y are intermediates (unless of course x does not depend on the inputs).

Pass or fail. Sometimes it can be difficult to distinguish between a passing result and a failing result. A clever trick to help distinguish what is passing and what is a failing result that just happens to be small in magnitude is to introduce an error deliberately. For example, multiply the derivatives by 2. The result should obviously be failing, and the magnitude of the output will give a sense for how large failing results should be. It is also a good idea to avoid testing derivatives near configurations where the derivatives are not well-behaved, such as near a divide-by-zero configuration. It is also important to make sure that exactly the same code path is taken in both configurations; the derivative tests may fail if a collision is detected in one configuration but not the other. Or if different numbers of conjugate gradient iterations are performed.

4.2 Getting it to work

The key to getting things to work is to perform derivative tests on the intermediates directly. I don't need to implement the entire thing, and then run derivative tests on f vs f_u or f_u vs f_{uu}. Instead, I test a_u, b_u, c_u, d_u, $\mathbf{w}_u$, and f_u in sequence (or indeed run the derivative test on all of them at the same time). At each step, I know that the inputs are being computed correctly (they were already tested). The first result that is failing will be a very tiny expression that is very easily fixed on inspection. Thus, I can fix the derivatives in small digestible pieces. The second derivatives are the same way. I can test a_{uu}, b_{uu}, c_{uu}, d_{uu}, and f_{uu} in sequence. In this case, however, we lack a simple way to test $\mathbf{w}_{uu}$. A simple solution to this is to implement rank-3 tensors as a type and compute $\mathbf{w}_{uu}$ directly, verifying that it is correct, then testing $\mathbf{z}$ against $\mathbf{w} \cdot \mathbf{w}_{uu}$.

Although these things are easiest to write and test in a separate standalone setting, these tests can often be performed even from within a complex system. All that is needed is to arrange for the routines that compute the derivatives to be called with two slightly different inputs. The first set of values can be stored in global variables until the second call is made, at which point the test can be performed and the results printed.

4.3 Optimizing it

Optimizing derivatives should start before any code is written. It is often very helpful to simplify the function we are differentiating before actually computing the derivatives. Differentiating simple expressions can make the process far easier. It will be a lot easier to start with simplified expressions than to optimize it afterwards with the derivatives. Next, we can break the computation into lots of tiny intermediate steps and work out the dependency graph (as we did in Figure 2). This will make it easier to identify *bottlenecks* in the computation, and it can also help us to choose an ordering for our derivative calculations. A good compromise is to identify bottlenecks and then compute the derivatives from one bottleneck to the next using forward differentiation. Then, combine the remaining derivatives at the end.

The straightforward implementation is often not the most efficient. For example, derivatives might be constant or even zero. We might end up multiplying by identity matrices. Once everything is working, we can be more flexible about intermediates. The result can still be tested by running the numerical derivative test on the final result or by comparing directly against the unoptimized

but working version. The strategy is to perform optimizations in small steps, testing the results after each small change. In this way, we can have routines that work and are efficient, too.

4.4 Numerical difficulties

Numerical difficulties tend to occur when working with derivatives. The types of difficulties that may arise can be illustrated even with simple function $y = \|\mathbf{x}\|$. Letting $\mathbf{u} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$, we have $\frac{\partial y}{\partial \mathbf{x}} = \mathbf{u}$ and $\frac{\partial^2 y}{\partial \mathbf{x} \partial \mathbf{x}} = \frac{1}{y}(\mathbf{I} - \mathbf{u}\mathbf{u}^T)$.

Computing y when $\mathbf{x} \approx \mathbf{0}$ may lead to underflow, since w^2 may underflow to zero in floating point even when w can be represented. Similarly, overflow may occur, where w^2 overflows to infinity even though w can be represented. We will ignore these difficulties in this course. Overflow can be avoided by choosing units that keep numbers on the order of one. Even with this, stiffnesses may be 10^6, and time step sizes may be 10^{-5}, but calculations on numbers at this scale are unlikely to overflow. Similarly, if a number like 10^{-30} is computed, it is likely zero with roundoff error. In this case, underflowing to zero may be fine.

The calculation of $\frac{\partial y}{\partial \mathbf{x}} = \mathbf{u} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$ brings up more serious issues. The most obvious is that if $\mathbf{x} = \mathbf{0}$, we will divide by zero. Although it may seem tempting to let $\mathbf{u} = \mathbf{0}$ in this case, that is a bad idea. It breaks the invariant that $\|\mathbf{u}\| = 1$. Instead, it is better to select something like $\mathbf{u} = \langle 1, 0, 0 \rangle$. In this way, one may imagine that the input was perturbed due to roundoff error to $\langle \epsilon, 0, 0 \rangle$ and normalized correctly. The next complication is that the values of $\mathbf{u}$ vary wildly when $\mathbf{x} \approx \mathbf{0}$. This difficulty is *mathematical*, in the sense that the difficulty exists analytically and is not merely an artifact of floating point or the way in which it is computed. Difficulties of this nature must be lived with; we can do nothing to fix them since they aren't broken. With care, we can ensure that all components of $\mathbf{u}$ are no larger than 1 in magnitude and that $\|\mathbf{u}\| \approx 1$. In spite of these difficulties, we will say that $\mathbf{u}$ can be (and should be) robustly computed.

The calculation of $\frac{\partial^2 y}{\partial \mathbf{x} \partial \mathbf{x}}$ presents the first major problems. With the robust computation of $\mathbf{u}$, the computation of $\mathbf{I} - \mathbf{u}\mathbf{u}^T$ poses no problems. However, scaling it by $\frac{1}{y}$ may result in an unbounded result. This difficulty is mathematical, in the sense that $\frac{\partial^2 y}{\partial \mathbf{x} \partial \mathbf{x}}$ is really unbounded in the neighborhood of zero. In this case, one may choose to calculate the derivative as infinity or clamp the magnitude of the derivatives to avoid overflow to infinity. Avoiding the infinity may be preferable, since some calculations with infinity may lead to NaNs. Calculating $\frac{1}{y}(\mathbf{I} - \mathbf{u}\mathbf{u}^T)$ is preferable to $\frac{1}{y^3}(\mathbf{I}\|\mathbf{x}\|^2 - \mathbf{x}\mathbf{x}^T)$, since in the first case at least two factors of y have been handled robustly.

When handling numerical difficulties, it is helpful to first determine whether the difficulty is mathematical or numerical. If the derivatives are analytically bounded, then it should be possible to arrange the calculations in a way that avoids computing unbounded quantities. For example, consider the function $y = \cos \|\mathbf{x}\|$. Broken up into pieces, we have $w = \|\mathbf{x}\|$ and $y = \cos w$. Then,

$$w = \|\mathbf{x}\| \qquad \mathbf{u} = \frac{\mathbf{x}}{w} \qquad w' = \mathbf{u} \cdot \mathbf{x}' \qquad \mathbf{u}' = \frac{1}{w}(\mathbf{x}' - \mathbf{u}w') \qquad w'' = \mathbf{u}' \cdot \mathbf{x}' + \mathbf{u} \cdot \mathbf{x}''$$

$$y = \cos w \qquad y' = -(\sin w)w' \qquad y'' = -(\cos w)(w')^2 - (\sin w)w''$$

In this case, we see two divisions by w, which could be zero or nearly zero. The computation of $\mathbf{u}$, as noted above, can be done robustly. The computation of $\mathbf{u}'$ may be unbounded. This in turn may cause w'' to be unbounded. But we see that w'' only comes into our calculation of y'' multiplied by $\sin w$, which goes to zero as w'' diverges. This suggests delaying the division by w with the

assignments $\mathbf{p} = w\mathbf{u}' = \mathbf{x}' - \mathbf{u}w'$ and $q = ww'' = \mathbf{p} \cdot \mathbf{x}' + \mathbf{x} \cdot \mathbf{x}''$. The new computation is

$$w = \|\mathbf{x}\| \qquad \mathbf{u} = \frac{\mathbf{x}}{w} \qquad w' = \mathbf{u} \cdot \mathbf{x}' \qquad \mathbf{p} = \mathbf{x}' - \mathbf{u}w' \qquad q = \mathbf{p} \cdot \mathbf{x}' + \mathbf{x} \cdot \mathbf{x}''$$

$$y = \cos w \qquad y' = -(\sin w)w' \qquad y'' = -(\cos w)(w')^2 - \frac{\sin w}{w}q$$

This computation is robust, since $\frac{\sin w}{w}$ can be computed robustly. These difficulties are best handled *after* the derivatives already work in the general case. They can be fixed in small steps as we do for optimizations.

4.5 Implicit differentiation

Occasionally, a the quantity that must be differentiated cannot be computed using arithmetic or elementary functions. For example, the quantity of interest may be the root of a polynomial of degree three of higher. (Cubic and quartic polynomials can be solved in closed form, but the formulas are not numerically robust. It is generally better to solve these iteratively. Whether the appropriate root is computed iteratively or in closed form, the derivatives should always be computed implicitly.) In this case, it is often necessary to employ implicit differentiation.

Implicit differentiation is used when some quantity y is defined implicitly in terms of some other quantity x as $f(x, y) = 0$. Differentiating both sides of the equation with respect to x yields $f_x(x, y) + f_y(x, y)y' = 0$, which can be solved for y'. The second derivative is $(f_{yy}y' + 2f_{xy})y' + f_{xx} + f_y y'' = 0$. From these, y' and y'' can be computed as

$$y' = -\frac{f_x}{f_y} \qquad y'' = -\frac{(f_{yy}y' + 2f_{xy})y' + f_{xx}}{f_y}.$$

Note that numerical difficulties may occur if $f_y \approx 0$. It is convenient that implicit differentiation is quite inexpensive. The calculation of y may be relatively expensive and may involve iterative root-finding techniques. By contrast, the calculations of y' and y'' only involve basic arithmetic.

The same procedure can be applied with vector quantities. For example, if $\mathbf{x}$ and $\mathbf{y}$ are vectors, then $\mathbf{y}$ may be defined by $\mathbf{f}(\mathbf{x}, \mathbf{y}) = \mathbf{0}$. The function $\mathbf{f}$ and unknown $\mathbf{y}$ should have the same dimensions, since we need the same number of equations as unknowns. The quantity that in the scalar case was f_{yy} will become a rank-three tensor in the vector case, so I will use indexed tensor notation. I will need to distinguish between f_x and f_y, which I will do by using $f_{i,r}$ and $f_{i,[s]}$ to represent the derivatives $\frac{\partial f_i}{\partial x_r}$ and $\frac{\partial f_i}{\partial y_s}$ (using square brackets to indicate that the derivative is with respect to the second argument). Let M_{ij} be the inverse of $f_{i,[k]}$ so that $M_{ji}f_{i,[k]} = \delta_{jk}$.

$$\begin{aligned}
0 &= f_i \\
0 &= f_{i,r} + f_{i,[k]}y_{k,r} \qquad \implies y_{j,r} = -M_{ji}f_{i,r} \\
0 &= f_{i,rs} + f_{i,r[k]}y_{k,s} + f_{i,[k]s}y_{k,r} + f_{i,[k][m]}y_{k,r}y_{m,s} + f_{i,[k]}y_{k,rs} \\
y_{j,rs} &= -M_{ji}(f_{i,rs} + f_{i,r[k]}y_{k,s} + f_{i,[k]s}y_{k,r} + f_{i,[k][m]}y_{k,r}y_{m,s})
\end{aligned}$$

As in the scalar case, numerical difficulties may occur if M_{ij} is not invertible.

Implicit differentiation is also a useful way to derive differentiation rules. For example, differentiating $\|\mathbf{x}\|^2 = \mathbf{x} \cdot \mathbf{x}$ yields $2\|\mathbf{x}\|\frac{d}{dt}\|\mathbf{x}\| = 2\mathbf{x} \cdot \mathbf{x}'$, which can be solved to recover the rule $\frac{d}{dt}\|\mathbf{x}\| = \frac{\mathbf{x}}{\|\mathbf{x}\|} \cdot \mathbf{x}'$.

4.6 Differentiating the function or the algorithm

There is a subtle but very important distinction to be made between differentiating the *result* of a function and differentiating the *algorithm* used to compute that function. For example, consider the

example $\mathbf{A} = \mathbf{F}^{-1}$, where the matrix $\mathbf{A}(t)$ is defined as a function of the time-varying matrix $\mathbf{F}(t)$. One way to compute compute $\mathbf{A}'$ from $\mathbf{F}'$ would be to write out the algorithm used to compute the inverse (for example, Gaussian elimination) as a sequence of simple arithmetic computations and then differentiate those computations to construct the result. In this case, the derivative can be obtained in closed form using implicit differentiation (by differentiating $\mathbf{AF} = \mathbf{I}$ to $\mathbf{A}'\mathbf{F}+\mathbf{AF}' = \mathbf{0}$ and solving for $\mathbf{F}' = -\mathbf{FA}'\mathbf{F}$.) When possible, one should always differentiate the function. In particular, the function and the numerical routine used to compute it may have very different derivatives. This is especially true when the algorithm must make choices (such as pivoting for Gaussian elimination) or when the algorithm is iterative (such as for computing the roots of a polynomial). In other cases, especially when the output is not otherwise uniquely determined (such as the QR decomposition), differentiating the algorithm may be necessary.

5 DIFFERENTIATING MATRIX FACTORIZATIONS

In constitutive modeling, two matrix factorizations occur frequently enough to merit special consideration: the symmetric eigenvalue decomposition $\mathbf{S} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$ and the singular value decomposition $\mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. We will be interested in differentiating quantities defined in terms of the eigenvalues $\mathbf{\Lambda}$ or singular values $\mathbf{\Sigma}$ with respect to the original matrix $\mathbf{S}$ or $\mathbf{F}$. We will present the formulas for reference and explain how to use them, but we will not derive them (See [Stomakhin et al. 2012] and the technical document that accompanies it).

5.1 Differentiating singular values

When dealing with hyperelastic isotropic constitutive models, it is not uncommon to define the energy density $\psi(\mathbf{F})$ in terms of the singular values, $\hat{\psi}(\mathbf{\Sigma})$, where $\mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. What is often needed is derivatives like $\frac{\partial \psi}{\partial \mathbf{F}}$, but what is available is $\frac{\partial \hat{\psi}}{\partial \mathbf{\Sigma}}$. For example, the popular corotated constitutive model is conveniently defined in terms of singular values as

$$\hat{\psi}(\sigma_1, \sigma_2, \sigma_3) = \mu((\sigma_1 - 1)^2 + (\sigma_2 - 1)^2 + (\sigma_3 - 1)^2) + \frac{\lambda}{2}(\sigma_1 + \sigma_2 + \sigma_3 - 3)^2.$$

The derivatives $\frac{\partial \hat{\psi}}{\partial \mathbf{\Sigma}}$ are easy to compute; this function is a degree-two polynomial.

Depending on how the function relates to its diagonal space version, the full derivatives can often be conveniently obtained directly from the diagonal space version. The table below summarizes many of these. Note that for the table below, the computations in the right are done assuming the diagonal matrices are treated as vectors. The following intermediates are used (no summation is implied):

$$a_{ik} = \frac{\hat{\psi}_{,i} - \hat{\psi}_{,k}}{\sigma_i - \sigma_k} \quad b_{ik} = \frac{\hat{\psi}_{,i} + \hat{\psi}_{,k}}{\sigma_i + \sigma_k} \quad f_{ik} = \frac{\hat{A}_i - \hat{A}_k}{\sigma_i - \sigma_k} \quad g_{ik} = \frac{\hat{A}_i + \hat{A}_k}{\sigma_i + \sigma_k} \quad h_{ik} = \frac{\hat{A}_i - \hat{A}_k}{\sigma_i + \sigma_k}$$

In each case, we compute the desired derivatives in three steps. (1) Compute the intermediates. (2) Compute the diagonal space version of the desired result ($\hat{T}_{ij}$ or $\hat{T}_{ijkl}$) using the fourth column of the table. (3) Transform the diagonal space quantity into regular space to get the desired result (T_{ij} or T_{ijkl}) using the third column of the table.

Expression	Want	Relation to diagonal	Nonzero entries (no summation)
$\psi(\mathbf{F}) = \hat{\psi}(\Sigma)$	$\mathbf{T} = \dfrac{\partial \psi}{\partial \mathbf{F}}$	$T_{ij} = \hat{T}_{mn} U_{im} V_{jn}$	$\hat{T}_{ii} = \hat{\psi}_{,i}$
$\psi(\mathbf{F}) = \hat{\psi}(\Sigma)$	$\mathbf{T} = \dfrac{\partial^2 \psi}{\partial \mathbf{F}^2}$	$T_{ijkl} = \hat{T}_{mnrs} U_{im} V_{jn} U_{kr} V_{ls}$	$\hat{T}_{iikk} = \hat{\psi}_{,ik}$ $\hat{T}_{ikik} = \dfrac{a_{ik} + b_{ik}}{2}, i \neq k$ $\hat{T}_{ikki} = \dfrac{a_{ik} - b_{ik}}{2}, i \neq k$
$\mathbf{A}(\mathbf{F}) = \mathbf{U}\hat{\mathbf{A}}(\Sigma)\mathbf{V}^T$	$\mathbf{T} = \dfrac{\partial \mathbf{A}}{\partial \mathbf{F}}$	$T_{ijkl} = \hat{T}_{mnrs} U_{im} V_{jn} U_{kr} V_{ls}$	$\hat{T}_{iikk} = \hat{A}_{i,k}$ $\hat{T}_{ikik} = \dfrac{f_{ik} + g_{ik}}{2}, i \neq k$ $\hat{T}_{ikki} = \dfrac{f_{ik} - g_{ik}}{2}, i \neq k$
$\mathbf{A}(\mathbf{F}) = \mathbf{U}\hat{\mathbf{A}}(\Sigma)\mathbf{U}^T$	$\mathbf{T} = \dfrac{\partial \mathbf{A}}{\partial \mathbf{F}}$	$T_{ijkl} = \hat{T}_{mnrs} U_{im} U_{jn} U_{kr} V_{ls}$	$\hat{T}_{iikk} = \hat{A}_{i,k}$ $\hat{T}_{ikik} = \dfrac{f_{ik} - h_{ik}}{2}, i \neq k$ $\hat{T}_{ikki} = \dfrac{f_{ik} + h_{ik}}{2}, i \neq k$
$\mathbf{A}(\mathbf{F}) = \mathbf{V}\hat{\mathbf{A}}(\Sigma)\mathbf{V}^T$	$\mathbf{T} = \dfrac{\partial \mathbf{A}}{\partial \mathbf{F}}$	$T_{ijkl} = \hat{T}_{mnrs} V_{im} V_{jn} U_{kr} V_{ls}$	$\hat{T}_{iikk} = \hat{A}_{i,k}$ $\hat{T}_{ikik} = \dfrac{f_{ik} + h_{ik}}{2}, i \neq k$ $\hat{T}_{ikki} = \dfrac{f_{ik} - h_{ik}}{2}, i \neq k$

The table has two types of entries. The first two rows have a scalar function defined in terms of the eigenvalues $\psi(\mathbf{F}) = \hat{\psi}(\Sigma)$. We want to compute the derivatives $\frac{\partial \psi}{\partial \mathbf{F}}$ (a matrix) and $\frac{\partial^2 \psi}{\partial \mathbf{F}^2}$ (a rank-four tensor).

The last three rows consider quantities of the form $\mathbf{U}\hat{\mathbf{A}}\mathbf{V}^T$, $\mathbf{U}\hat{\mathbf{A}}\mathbf{U}^T$, or $\mathbf{V}\hat{\mathbf{A}}\mathbf{V}^T$. This is a generalization of the Hessian rule, and derivatives of this form are sometimes encountered. Here, $\hat{\mathbf{A}}(\Sigma)$ is a diagonal matrix. We will sometimes treat the diagonal matrix $\hat{\mathbf{A}}$ as the vector containing its diagonal entries (such as when we write $\hat{A}_k$). The use of quantities as vectors or matrices can be deduced from context.

Note that there are many other variations that could we written down, such as $\mathbf{A}(\mathbf{F}) = \mathbf{U}\hat{\mathbf{A}}(\Sigma)\mathbf{U}$ or $\mathbf{A}(\mathbf{F}) = \hat{\mathbf{A}}(\Sigma)\mathbf{V}^T$. These forms do not have physically meaningful transformation properties. As such, they are very unlikely to be encountered, and their derivatives are unlikely to take on nice forms.

5.2 Differentiating eigenvalues

Let $\mathbf{S}$ be a symmetric matrix and $\mathbf{S} = \mathbf{U}\Lambda\mathbf{U}^{-1}$ its eigenvalue decomposition. We can always compute $\mathbf{U}$ to be an orthogonal matrix, and we will assume this has been done. Then, $\mathbf{U}^{-1} = \mathbf{U}^T$. This gives us $\mathbf{S} = \mathbf{U}\Lambda\mathbf{U}^T$. We can define a scalar function $\psi(\mathbf{S})$ in terms of the singular values as $\hat{\psi}(\Lambda)$. The eigenvalue decomposition case can be treated as a special case of the singular value decomposition, and the differentiation formulas are similar. The intermediates we use are

$$a_{ik} = \frac{\hat{\psi}_{,i} - \hat{\psi}_{,k}}{\lambda_i - \lambda_k} \quad f_{ik} = \frac{\hat{A}_i - \hat{A}_k}{\lambda_i - \lambda_k}.$$

The corresponding derivative rules are given by the table below. The rules are used in the same way as the rules for the singular value decomposition.

Expression	Want	Relation to diagonal	Nonzero entries (no summation)
$\psi(\mathbf{S}) = \hat{\psi}(\mathbf{\Lambda})$	$\mathbf{T} = \dfrac{\partial \psi}{\partial \mathbf{S}}$	$T_{ij} = \hat{T}_{mn} U_{im} U_{jn}$	$\hat{T}_{ii} = \hat{\psi}_{,i}$
$\psi(\mathbf{S}) = \hat{\psi}(\mathbf{\Lambda})$	$\mathbf{T} = \dfrac{\partial^2 \psi}{\partial \mathbf{S}^2}$	$T_{ijkl} = \hat{T}_{mnrs} U_{im} U_{jn} U_{kr} U_{ls}$	$\hat{T}_{iikk} = \hat{\psi}_{,ik}$ $\hat{T}_{ikik} = \hat{T}_{ikki} = a_{ik}, i \neq k$
$\mathbf{A}(\mathbf{S}) = \mathbf{U}\hat{\mathbf{A}}(\mathbf{\Lambda})\mathbf{U}^T$	$\mathbf{T} = \dfrac{\partial \mathbf{A}}{\partial \mathbf{S}}$	$T_{ijkl} = \hat{T}_{mnrs} U_{im} U_{jn} U_{kr} U_{ls}$	$\hat{T}_{iikk} = \hat{A}_{i,k}$ $\hat{T}_{ikik} = \hat{T}_{ikki} = f_{ik}, i \neq k$

6 AUTOMATIC DIFFERENTIATION

Although much of this course has focused on manually writing code for calculating derivatives, there are many tools available to compute them automatically. These techniques are called automatic differentiation.

Automatic differentiation strategies differ in the *mode* of differentiation (usually forward mode or reverse mode) and the *time* when the derivatives are computed (code generation, compile time, or runtime). We have already seen forward mode and reverse mode differentiation; we explored these when choosing an effient strategy for calculating derivatives. Depending on when and how automatic differentiation is performed, the same tradeoffs may be available. We begin by exploring options for when automatic differentiation can be performed.

6.1 Compile time

One particularly simple way to implement automatic differentiation is using operator overloading. This strategy is particularly appealing in languages like C++ which have native support operator overloading. This is done by creating a custom structure type (`Diff_TT` below; I am using the two letters to indicate the type of quantity that is being computed and what type of object it is being differentiated with respect to) that contains the value being computed as well as its derivatives. For example,

```
struct Diff_TT
{
  double x, dx;

  Diff_TT() {}
  Diff_TT(double y) : x(y), dx(0) {}
  Diff_TT(double y, double dy) : x(y), dx(dy) {}
};

Diff_TT operator+ (const Diff_TT& a, const Diff_TT& b)
{
  Diff_TT r = {a.x + b.x, a.dx + b.dx};
  return r;
}

Diff_TT operator* (const Diff_TT& a, const Diff_TT& b)
{
```

```
  Diff_TT r = {a.x * b.x, a.dx * b.x + a.x * b.dx};
  return r;
}

Diff_TT sin(const Diff_TT& a)
{
  Diff_TT r = {sin(a.x), cos(a.x) * a.dx};
  return r;
}

// and so on ...
```

This class might be used to implement a routine to calculate a function like $f(x) = (x^2 + 2x + 3)\sin(x + 1)$ and its derivative.

```
void Compute_Derivatives(double x, double & f, double & df)
{
  Diff_TT y = {x, 1};
  Diff_TT w = (y * y + y * 2 + 3) * sin(y + 1);
  f = w.x;
  df = w.dx;
}
```

This strategy is intuitive, easy to implement, and easy to use. The code that must be written often looks nearly the same as the code that would have been written to just compute the function alone. Once the result is computed, the derivatives are available as well. With inlining, all of the calculations are available for compiler optimization. For example, the code above results in addition of zero or multiplication by zero or one in several places. These routines can be improved by creating overloads for mixtures of scalars and derivative types which avoid generating these unnecessary operations.

This strategy is also rather flexible. Second derivatives can be handled by storing second derivatives in the structure and adding suitable differentiation rules to the overloaded operators and functions. Similar structures `Diff_VT` and `Diff_MT` can be defined for vectors and matrices being differentiated by a scalar. One may then provide overloads for routines like matrix-vector multiplication and dot product. Differentiation with respect to vectors and matrices can be handled by storing derivatives of the correct type. For example, we might create types `Diff_TV` and `Diff_VV` for storing the derivative of a scalar or a vector with respect to another vector. These structures might look like the following.

```
template<int n>
struct diff_TV
{
  double x;
  fixed_size_vector<n> dx;
};

template<int m,int n>
struct Diff_VV
```

```
{
  fixed_size_vector<m> x;
  fixed_size_matrix<m,n> dx;
};

template<int m,int n>
Diff_TV<n> dot(const Diff_VV<m,n>& a, const Diff_VV<m,n>& b)
{
  Diff_TV<n> r = {dot(a.x, b.x), a.dx.t() * b.x + b.dx.t() * a.x};
  return r;
}
```

Observe that the size of dx scales with the number of inputs that must be differentiated with respect to. The cost of computing those entries will grow with the storage. When Hessians (ddx) are required, the storage and computational costs grow even quicker. The convenience and efficiency of this strategy break down when computing derivatives with respect to multiple different inputs. Note also that this strategy is mostly restricted to forward mode differentiation.

Reverse mode differentiation is possible in a compile-time implementation through the use of expression templates in C++. The technique is significantly more complex than the simple strategy described above and largely amounts to recording the operations that were used to compute a quantity as part of the *type* of that quantity (e.g., for $z = 3x^2 + \cos y$, the type of z might be something like `Add<Scale<Square<Var<0> > >, Cos<Var<1> > >`). Special member functions in each of these helper classes are then used to recurse back through the operations to perform the reverse mode gradient calculations. We do not pursue this strategy further in this course. Instead, we refer the interested reader to [Hogan 2014].

6.2 Runtime

An alternative strategy for automatically computing derivatives is to do so at runtime. As code is executed (E.g., `z = x * y + x * 2 + 3`), a list of operations can be recorded dynamically (with $v_0 = x, v_1 = y$ and the result $z = t_3$):

out	op	in0	in1
t_0	$*$	v_0	v_1
t_1	$*$	v_0	2
t_2	$+$	t_0	t_1
t_3	$+$	t_2	3

Rather than defining overloaded operators to compute derivatives on the fly, the operators can instead add an entry to a list indicating the inputs and operation that should be performed. This creates a dynamic representation of the function that can be manipulated by the program. The value or derivatives can be computed from this representation. Since the entire function is available before derivatives are computed, it is possible to calculate them using forward mode or reverse mode.

To do forward mode, one simply walks the list. Since derivatives are required with respect to two inputs in this example ($v_0 = x$ and $v_1 = y$), we would store the value t_k and both partial derivatives $\frac{\partial t_k}{\partial v_0}$ and $\frac{\partial t_k}{\partial v_1}$ at each step in the calculation in much the same way that we did in the compile-time approach.

It is also straightforward to compute the derivatives in reverse mode by walking the list of operations backwards. First, the intermediates values t_0, t_1, t_2, t_3 are computed as usual. Then, in a

reverse pass, we compute derivatives. For the t_3 operation, we compute $\frac{\partial t_3}{\partial t_2}$. For the t_2 operation, we compute $\frac{\partial t_3}{\partial t_0}$ and $\frac{\partial t_3}{\partial t_1}$. For the t_1 operation, we compute $\frac{\partial t_3}{\partial v_0}$. Finally, for the t_0 operation we compute $\frac{\partial t_3}{\partial v_1}$ and *modify* $\frac{\partial t_3}{\partial v_0}$. Unlike the forward mode implementation, this reverse mode is very efficient for the case when a scalar must be differentiated with respect to many inputs (gradient of a scalar). In fact, it scales at the same rate as the function value computation itself.

The dynamic approach to automatic differentiation has a number of advantages. One of these is the ability to compute derivatives in reverse mode. It can also be used when the number of inputs is not known at compile time.

A major disadvantage of the dynamic approach is its runtime cost. Building up and evaluating the dynamic representation creates overhead. Further, these operations cannot be inlined or optimized by the compiler. Indeed, calculating the function and derivatives from the stored representation amounts to interpreting the code rather than compiling it.

There are some variations on the dynamic approach that are worth mentioning. There are some circumstances where one may want to dynamically parse code and interpret that code anyway. This occurs when, for example, you would like to be able to specify analytical expressions in input files. In this case, the dynamic representation is already available, and derivatives can be computed directly from this representation.

6.3 Code generation

Another option for automatic differentiation is to use code generation. A program takes as input code to compute some function and outputs a new routine to compute the function and its derivatives. This strategy is in many ways similar to the dynamic strategy. Instead of actually performing the calculations, however, the program instead emits code to perform the calculations. This code can then be copied to a source file, compiled, and optimized as with any other code. The program may also emit code that uses linear algebra routines like matrix-vector multiplication or vector addition to produce more compact and readable code. The resulting code may also be used as a starting point for manual optimization.

The code generation strategy should be distinguished from the Maple/Mathematica approach that we discouraged earlier. A good automatic differentiation package based on code generation will emit code that is designed to be efficient to execute. Unlike symbolic tools, it will not produce a tangled mess by writing the code as a monolithic expression and then trying to simplify it. The linear algebra representation tends to naturally capture redundancies in the computations. Symbolic tools tend to lose these natural intermediates.

Since the code generation is done only once while the program is written (and perhaps repeated if the code is changed), one can afford to spend more computational resources computing and optimizing the derivatives. This includes choosing between forward mode, reverse mode, or even a mixed mode. The program can use dynamic programming to optimize the order in which to evaluate the derivatives. The program can also identify and perform optimizations that take advantage of the special structure of some intermediate values. For example, a derivative may be a scalar multiple of the identity matrix (the derivative of the expression $2\mathbf{x}+1$ with respect to $\mathbf{x}$ is $2\mathbf{I}$), which would allow a subsequent matrix-matrix multiply to be replaced with a more efficient scalar-matrix multiply.

6.4 Differentiate the function

When using automatic differentiatiation, it is important to keep in mind that some functions should be differentiated by differentiating their implementation and others should be differentiated directly. Especially for functions whose definitions are templated to compile for `float` or `double`, it may be

possible to instantiate the functions on a custom type like our `Diff_TT` or `Diff_TV`. This would cause us to differentiate the implementation of the function. Often, this will be the desired behavior. In some cases, however, it is better to create a special overloaded version of the function to compute the derivatives in a way that is specific to the mathematical properties of the function. Simple math functions (trigonometry, exponentials, logarithms, and other special functions like the gamma function or Bessel functions) should be differentiated using appropriate calculus rules. Routines for solving equations should be differentiated using implicit differentiation.

6.5 When to use automatic differentiatiation

There are many times when it is appropriate to automate the process of computing derivatives. **Prototyping.** Such tools are particularly useful for prototying, since they allow one to quickly implement, test, and evaluate many different options before selecting the the one the performs best. Only the option that is selected needs to be differented manually. **Debugging.** Some styles of automatic differentiation give you a working implementation of derivatives, which can be optimized by hand or used to debug a manually-derived version. **Infrequently executed code.** Some derivatives are used only infrequently and do not contribute significantly to the overall runtime of a system. It may not be worthwhile to invest time and resources to manually differentiate and optimize such routines.

The performance cost of automatic differentiation can be quite significant. My experiences with the compiled approach, even with sigificant effort to be efficient, is that I should expect to pay a factor of two performance cost. This depends greatly on the particular function being differentiated and the number of inputs and outputs. With the code generation strategy, this may be reduced somewhat. The dynamic strategy will likely be quite a bit slower.

One should also keep in mind that automatic differentiation will not make adjustments to the way quantities are computed in order to resolve numerical difficulties. This may lead to Infs, NaNs, or inaccurate derivatives. One one hand, these difficulties may only rarely case problems. On the other hand, that also makes them harder to find with testing.

6.6 Automatic differentiation libraries

Many libraries are available for automatic differentiation. These libraries vary in license, programming language, modes supported (forward, reverse, often both), the number of derivatives they can compute (gradients, Hessians, or unlimited), efficiency, and ease of use. Those interested in using an automatic differentiation library are referred to http://www.autodiff.org/. This site maintains an extensive list of existing software tools and libraries. It also provides a reading list for those interested in learning more about the techniques involved or those interested in writing their own customized library.

7 CONCLUDING REMARKS

Differentiating computer routines may seem daunting, especially when the routines in questions are complex. Further, this task is often delegated to students or interns, who may lack many of the skills that have been accumulated by their more senior researchers. The main message of this course is that manually calculating derivatives is often quite feasible even for the very complex functions that occur in practical applications. These derivatives can be implemented, numerically tested, and debugged in small manageable pieces. Optimization and robustness concerns can be addressed and tested incrementally once the derivatives work. Derivative testing routines are also useful to keep around for later debugging purposes, even after the implementation has been thoroughly tested, since this can help when debugging more subtle problems like inefficient solver convergence.

In some cases, automatic differentiation is a convenient alternative to manual derivatives. This is especially useful for prototyping, since is allows new ideas to me implemented and evaluated more rapidly. If derivative computations occupy a significant fraction of overall runtime, these derivatives should be replaced with manually-computed and optimized derivatives when a final prototype has been selected.

ACKNOWLEDGMENTS

Much of the advice contained within this document was developed through trial and error over the course of many research projects. For this, I would like to thank the many coauthors who went through these challenges with me. I would also like to thank my postdoctoral advisor Joseph Teran for hosting a research environment where I could develop and share these ideas with others.

REFERENCES

Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. 2000. Automatic differentiation of algorithms. *J. Comput. Appl. Math.* 124, 1-2 (2000), 171–190.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research* 18 (2018), 1–43.

Robin J Hogan. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software (TOMS)* 40, 4 (2014), 26.

Uwe Naumann. 2008. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming* 112, 2 (2008), 427–441.

A. Stomakhin, R. Howes, C. Schroeder, and J.M. Teran. 2012. Energetically Consistent Invertible Elasticity. In *Proc. Symp. Comp. Anim.* 25–32.