Practical notes on implementing derivatives

Craig Schroeder

May 16, 2016

1 Introduction

For many applications in graphics, one is confronted with the task of solving an optimization problem or solving a nonlinear system of equations. Efficient methods for solving these problems require derivatives to be available. Differentiation in numerical methods also occurs for many other reasons, such as computing forces from potential energy or computing normal and curvature information from an analytic level set. For most authors in the graphics community, the process of working out, implementing, and testing derivative routines does not seem to cause real difficulties. This is not always true, however, and some papers punt on the task in a variety of ways.

One common way of avoiding the task of computing derivatives by hand is to rely on autodifferentiation. Autodifferentiation is a very reasonable and effective solution in some circumstances, which has received a fair amount of attention. This approach can be quite effective for rapid prototyping or for circumstances where computing derivatives does not take a significant fraction of the total runtime. I use autodifferentiation myself for a variety of purposes. There are many contexts in which autodifferentiation may fail to be adequate, such as being too slow or not sufficiently numerically robust. Autodifferentiation also does not work well in some circumstances, such as differentiating quantities depending on a singular value decomposition, a problem that occurs when working with constitutive models. In these cases, it is best to perform the differentiation by hand.

Instead of doing this, some authors avoid the derivatives by using BFGS or L-BFGS instead of Newton's method. Others approximate the derivatives, such as by omitting terms. Other authors choose to compute derivatives using Maple, Mathematica, or other similar software. These practices are all generally bad. BFGS and L-BFGS do not converge as rapidly as Newton's method, especially for large sparse systems. Nor do numerical methods perform as well when derivatives are approximated. Derivatives computed symbolically using a software package tend to have many problems, including being enormously complicated, difficult to maintain, not numerically robust, and inefficient.

In this document, I explore *practical* strategies for implementing derivatives in code. I focus our attention on strategies that are effective even when the derivatives appear hopelessly difficult to work out and implement. As such, many of the suggestions that I make are overkill for simple tasks.

2 Basic ideas

The basic strategy for computing derivatives is fairly straightforward. Consider a simple example like $z = \frac{3x+\sqrt{9x^2-4x\cos x}}{2\cos x}$. The first step is to break the computation into tiny pieces. In this case, one might use

$$a = \cos x \qquad b = \frac{1}{a} \qquad c = xb \qquad d = \frac{3}{2}c \qquad e = d^2 - c \qquad f = \sqrt{e} \qquad z = d + f.$$

Next, one differentiates each equation in turn.

$$a' = -\sin x \qquad b' = -b^2 a' \qquad c' = xb' + b \qquad d' = \frac{3}{2}c' \qquad e' = 2dd' - c' \qquad f' = \frac{e'}{2f} \qquad z' = d' + f'.$$

Next, one would implement each of these equations to compute the final results, z and z'. If second derivatives are required, first break the computations into smaller pieces

$$a' = -\sin x \qquad g = b^2 \qquad b' = -ga' \qquad c' = xb' + b \qquad d' = \frac{3}{2}c'$$
$$e' = 2dd' - c' \qquad h = \frac{1}{f} \qquad f' = \frac{e'h}{2} \qquad z' = d' + f'.$$

Then repeat the differentiation term by term.

$$a'' = -a \qquad g' = 2bb' \qquad b'' = -g'a' - ga'' \qquad c'' = xb'' + 2b' \qquad d'' = \frac{3}{2}c''$$
$$e'' = 2((d')^2 + dd'') - c'' \qquad h' = -h^2 f' \qquad f'' = \frac{e''h + e'h'}{2} \qquad z'' = d'' + f''.$$

The next process is implementing these formulas. Compare this to the results obtained by, for example, cleaning up optimized code generated from Maple.

$$a = \cos(x) \qquad b = 4a - 9x \qquad c = \sqrt{-bx} \qquad d = \sin(x) \qquad e = cd$$
$$f = x^2 \qquad g = a^2 \qquad h = \frac{1}{c} \qquad j = g^2 \qquad k = ga$$
$$m = fx \qquad n = cf \qquad p = f^2 \qquad q = cm \qquad r = ma$$

$$\begin{aligned} s = {} & 108fdg - 8xdk - 54efa + 24exg - 162dr - 12fg + 4fj + 24an \\ & + 81pg + 27gq - 54mk - 12kn + 4j - 162p - 54q + 108r \end{aligned}$$
$$z = \frac{3x + c}{2a} \qquad z' = -\frac{h(2xda - 9fd - 3xe - 3ac - 9xa + 2g)}{2g} \qquad z'' = \frac{hs}{2xbk}$$

3 Debug as you go

One problem that is sometimes encountered when implementing derivatives is that one completes the implementation and finds that they don't actually work. One is then left checking a block of math trying to find a mistake in the derivation or a mistake converting that math into code.

3.1 Numerical derivative test

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 lose 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 $\|\mathbf{z}(\mathbf{x} + \Delta\mathbf{x}) - \mathbf{z}(\mathbf{x} - \Delta\mathbf{x})\|/\delta$, $\|(\nabla\mathbf{z}(\mathbf{x} + \Delta\mathbf{x}) + \nabla\mathbf{z}(\mathbf{x} - \Delta\mathbf{x})\delta\mathbf{x}/\delta$, 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.

3.2 Derivative tests on intermediates

Note that in the original example, there were many intermediate quantities defined: $\{a, b, c, d, e, f, g, h, z\}$. Corresponding to these, each quantity's derivative and most of their second derivatives will be computed. In addition to testing to ensure that z' is the derivative of z and that z'' is the derivative of z', I can also do derivative tests to ensure that these intermediates are consistent with their derivatives. This may be more convenient to do if the derivatives are first implemented in a separate program set up specifically for this task. In this way, each mistake is narrowed down to a very small piece of code, often one or two lines.

Once the code is working and passes derivative tests, it can be copied into the desired code base if necessary. Since a working reference implementation of exactly the same thing is available, including with the same intermediates, debugging will be straightforward.

3.3 Optimization

Although the approach above will often produce code that is quite efficient, the result can usually be optimized further. Doing this will cause intermediates or their derivatives to no longer be computed, which would make derivative testing on intermediates less effective. However, if the derivatives are first implemented and tested as above, optimizations can be tested by comparing the optimized code directly against the original derivative implementation. Then, optimizations can be made and tested one at a time.

4 Working with vectors, matrices, and tensors

4.1 Index notation

In practice, derivatives are often taken with respect to vectors or matrices, and the quantities being differentiated may involve vectors or matrices. When this happens, the limitations of standard notations quickly becomes evident. Take for example something as simple as $g = \mathbf{u} \cdot \mathbf{v}$. Taking the convention that the second index of $\nabla \mathbf{u}$ is the derivative index and that ∇g should be a column vector, one correct option would be to write $\nabla g = (\nabla \mathbf{u})^T \mathbf{v} + (\nabla \mathbf{v})^T \mathbf{u}$. Many other possibilities exist, such as $\nabla \mathbf{u}\mathbf{v}$, $(\nabla \mathbf{u})\mathbf{v}$, $\nabla \mathbf{u} \cdot \mathbf{v}$, $\nabla \mathbf{u}^T \mathbf{v}$, $\mathbf{v}^T \nabla \mathbf{u}$, and $\mathbf{v} \cdot \nabla \mathbf{u}$. Of these, the first three imply summation on the wrong index. $\nabla \mathbf{u}^T$ is mildly ambiguous with respect to the unlikely interpretation $\nabla(\mathbf{u}^T)$. $\mathbf{v}^T \nabla \mathbf{u}$ should produce a row vector instead of a column vector. The meaning of $\mathbf{v} \cdot \nabla \mathbf{u}$ is somewhat ambiguous, but it is probably best interpreted as $\mathbf{v}^T \nabla \mathbf{u}$, producing a row vector. Once a workable set of conventions is established, this difficulty is overcome.

Next, one runs into things like, for example, $g = \mathbf{u}^T \mathbf{A} \mathbf{B} \mathbf{v}$, where $\mathbf{A}$ and $\mathbf{B}$ are matrices. Now, I am left with the derivatives of $\mathbf{A}$ and $\mathbf{B}$, which produce rank three tensors, and the notation becomes hopeless. There is no established notation for contracting against the middle index.

One common solution to the notation problem is to use index notation. This solution works very well in practice and is the approach that I employ. Then, $g = u_i v_i$ is differentiated to $g_{,r} = u_{i,r} v_i + u_i v_{i,r}$. Note that the quantity u_i implies that $\mathbf{u}$ is a vector, with u_i representing its i component. In this notation, all quantities are scalars (thus the lack of bold typeface), since I am working with components of vectors rather than the vectors themselves. The use of a comma $u_{i,r}$ indicates a partial derivative with respect to a component of the independent variable x_r. Multiple derivatives are similar $u_{i,rs}$. An index that occurs exactly twice in each term is assumed to be summed, a convention referred to as the Einstein summation convention. With these conventions, the problematic example does not cause problems. $g = u_i A_{ij} B_{jk} v_k$ differentiates to $g_{,r} = u_{i,r} A_{ij} B_{jk} v_k + u_i A_{ij,r} B_{jk} v_k + u_i A_{ij} B_{jk,r} v_k + u_i A_{ij} B_{jk} v_{k,r}$. The main difficulties with this notation is that it is more difficult to read and the indices must often be relabeled.

Indexing notation can actually do a lot more than is described here, distinguishing between different types of indices and different types of derivatives. These distinctions are of no consequence for our purposes, so I do not employ them.

A couple special tensors are worth mentioning. δ_{ij} is the identity matrix, and e_{ijk} is the permutation tensor. The latter is typically used for cross products, with $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ being replaced by $w_i = e_{ijk} u_j v_k$. Both tensors are constants, so their derivatives vanish. They have a number of other useful properties, such as $\delta_{ij} = \delta_{ji}$, $\delta_{ij} u_i = u_j$, $e_{ijk} = e_{kij} = e_{jki} = -e_{jik} = -e_{ikj} = -e_{kji}$. Also, note that $x_{i,j} = \delta_{ij}$, where x_i are the independent variables.

I have found that augmenting the derivative notation in a few ways is quite helpful in some cases. The main need is when derivatives are taken with respect to a quantity with more than one index, such as a matrix. For this, I typically group the indices together with parenthesis. For example, derivatives of g and $\mathbf{A}$ with respect to the matrix $\mathbf{F}$ could be written $g_{,(rs)}$ and $A_{ij,(rs)}$. This helps, since $g_{,rs}$ would imply second derivatives with respect to a vector. Partial derivatives with respect to the positions of all particles $\mathbf{x}_p$, indexed as x_{pi}, might be written $g_{,(qr)}$. Note that $x_{pi,(qr)} = \delta_{pq} \delta_{ir}$.

Another convention that has proven to be useful is to designate different sets of indices for different purposes, so that the meaning can be immediately deduced. It is also convenient to set aside a different set of indices when working out derivatives, such as my use of r, s, and q as the variables for derivatives, when I used i, j, and p otherwise. This is convenient since it reduces the need to relabel indices. (The expressions $u_i A_{ij} v_j$ and $u_k A_{ki} v_i$ mean the same thing; the index names for repeated indices do not matter. If I want to multiply the scalars $u_i v_i$ and $u_i A_{ij} v_j$, I must first relabel one of the indices. For example, I could write $u_k v_k u_i A_{ij} v_j$ to avoid the clash.)

4.2 Tensors and code

Coding libraries for graphics often come with some degree of support for small matrices and vectors, along with many of the common operations that are applied to them. These libraries typically do not support

more general tensors. This can cause complications, since differentiation has a tendency to produce such tensors. Dealing with these tensors is generally undesired, even if code support were available for them. One reason for this is that they can be quite expensive to construct and use. A rank-four tensor in 3D contains 81 entries. Each entry must be computed, and addition work must be done on each entry whenever the tensor is used for something. In practice, simplifications can sometimes be done that result in the elimination of tensors, replacing them instead with vector and matrix operations.

The typical workflow is then to work out the derivatives, manipulate the derivation to eliminate tensors from the computations, implement the results, and then test. If one is lucky, the implementation will pass derivative tests. More likely, one is left carefully examining the derivation and simplifications line by line and comparing to code line by line. If one is fortunate, one my even have help doing this. The need to deal with tensors has prevented me from following the "debug as you go" strategy.

After suffering through this process a few times, I wrote a simple testbed that supports tensors directly from indexing notation, allowing us to perform computations directly off a form that is usable for working out the derivation. The important elements of this setup are that (a) the compile-test cycle is very quick (no more than about a second), (b) it can run derivative tests on quantities of any type, whether they be scalars, vectors, matrices, or even tensors, and (c) the syntax is usable for working out a derivation. The computations themselves are very inefficient, but this is not important. It provides a place to differentiate and simplify expressions while catching mistakes. Once the simplifications are complete, the results can be ported to use the real library structures and operations. Since a working implementation with exactly the same operations is available, it is a very simple matter to test and debug the final implementation. This framework is included in the accompanying code, and examples of its use are provided.

It is important to recognize the limitations of this setup. It works well when the summation convention can be adhered to and more complex operations (matrix inverse, determinant, matrix factorizations) are not directly involved. It remains attractive when complex pieces (e.g., a complex constitutive model) can be abstracted out to a function; the contents of the function can then be replaced by something that fits well in the framework but has sufficiently generic derivatives to still provide a robust tests. For simple tasks, this approach is overkill.

5 Derivatives involving the SVD

5.1 Differentiating functions defined in terms of the SVD

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 $\mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. If $\mathbf{S}$ is symmetric, then $\mathbf{S} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^T$. What is often needed is derivatives like $\frac{\partial \psi}{\partial \mathbf{F}}$, but what is available is $\frac{\partial \hat{\psi}}{\partial \mathbf{\Sigma}}$. 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):

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

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

Each of the second derivative formulas is implemented and demonstrated to pass numerical derivative tests; the files in which this is done are, in table order, `example-svd-dd.cpp`, `example-eig-dd.cpp`, `example-svd-uv.cpp`, `example-svd-uu.cpp`, `example-svd-vv.cpp`, and `example-eig-uu.cpp`. The two rules for first derivatives of scalars are demonstrated alongside the corresponding second derivatives.

Robustness. Note that the quantities a_{ik} and f_{ik} involve dividing by $\sigma_i - \sigma_k$, which is problematic when singular values are nearly equal. These divided differences can and should be computed robustly. I show how this can be done in Section 5.2. In this way, robustness is only an issue when $\sigma_i + \sigma_k = 0$. When the singular values follow the inverting finite element sign convention and the configuration is degenerate or inverted, this is possible. In general, the case $\sigma_i = -\sigma_k \neq 0$ will represent an actual discontinuity, and this division by zero is genuinely unavoidable. I deal with this issue by clamping.

Chain Rule. Sometimes it is the case that a quantity like $\mathbf{A}(\mathbf{B}(\mathbf{F}))$ will be required, where for example $\mathbf{A}(\mathbf{F}) = \mathbf{U}\hat{\mathbf{A}}(\mathbf{\Sigma})\mathbf{V}^T$ and $\mathbf{B}(\mathbf{F}) = \mathbf{U}\hat{\mathbf{B}}(\mathbf{\Sigma})\mathbf{V}^T$. The need to differentiate a construct like this might seem daunting. However, if I let $\hat{\mathbf{C}}(\mathbf{\Sigma}) = \hat{\mathbf{A}}(\hat{\mathbf{B}}(\mathbf{\Sigma}))$, then the process becomes much simpler. I now have $\mathbf{A}(\mathbf{C}) = \mathbf{U}\hat{\mathbf{C}}(\mathbf{\Sigma})\mathbf{V}^T$, so the appropriate rule from the table above may be selected. The functions $\hat{\mathbf{A}}$ and $\hat{\mathbf{B}}$ can be differentiated independently, with $\hat{\mathbf{C}}$ pieced together from the chain rule.

5.2 Robustly computing divided differences

When computing derivatives involving the SVD, I sometimes end up needing expressions like

$$\frac{u_r - u_s}{x_r - x_s}, \tag{1}$$

which will cause numerical problems when $x_r \approx x_s$. Typically, however, these expressions are still finite. I note that these follow a sort of calculus, which allows them to be computed in a straightforward manner. Note that this operation only makes sense with respect to indices, since the difference is performed with respect to an index. In the case of matrices or tensors, it would be performed on any index. It can never be

performed on scalar operations.

Operation	Rule	Note
$a\mathbf{u}$	$a\left(\frac{u_i - u_j}{x_i - x_j}\right)$	
$\mathbf{u} \pm \mathbf{v}$	$\left(\frac{u_i - u_j}{x_i - x_j}\right) \pm \left(\frac{v_i - v_j}{x_i - x_j}\right)$	
$\mathbf{Au}$	$\sum_k \left(\frac{A_{ik} - A_{jk}}{x_i - x_j}\right) u_k$	Products are trivial
$\mathbf{z} = \mathbf{z}(\mathbf{u})$	$\left(\frac{z_i - z_j}{u_i - u_j}\right)\left(\frac{u_i - u_j}{x_i - x_j}\right)$	Chain rule

Note that since divided differences cannot be applied to scalars, rules for operations like $\|\mathbf{u}\|$, $\mathbf{u}\cdot\mathbf{v}$, $\mathrm{tr}(\mathbf{A})$, or $\det(\mathbf{A})$ are unnecessary. The same applies to things like $\sin(a)$ and $\ln(a)$.

Componentwise operations also come up sometimes when working in diagonal space, since it is convenient to treat diagonal matrices as vectors. Then, these rules are useful

$$\frac{(u_r v_r) - (u_s v_s)}{x_r - x_s} = v_r\left(\frac{u_r - u_s}{x_r - x_s}\right) + u_s\left(\frac{v_r - v_s}{x_r - x_s}\right) \tag{2}$$

$$\frac{(u_r/v_r) - (u_s/v_s)}{x_r - x_s} = \frac{1}{v_r}\left(\frac{u_r - u_s}{x_r - x_s}\right) - \frac{u_s}{v_r v_s}\left(\frac{v_r - v_s}{x_r - x_s}\right) \tag{3}$$

$$\frac{v_r^{-1} - v_s^{-1}}{x_r - x_s} = -\frac{1}{v_r v_s}\left(\frac{v_r - v_s}{x_r - x_s}\right) \tag{4}$$

$$\frac{u_r^2 - u_s^2}{x_r - x_s} = (u_r + u_s)\left(\frac{u_r - u_s}{x_r - x_s}\right) \tag{5}$$

It is also useful to consider componentwise function applications. After using the chain rule, I am left only to evaluate these expressions robustly (let $w = x - y$ and $p = \frac{x}{y} - 1$)

$$\frac{\sin x - \sin y}{x - y} = \cos y\left(\frac{\sin w}{w}\right) + \sin y\left(\frac{\cos w - 1}{w}\right) \tag{6}$$

$$\frac{\cos x - \cos y}{x - y} = -\sin y\left(\frac{\sin w}{w}\right) + \cos y\left(\frac{\cos w - 1}{w}\right) \tag{7}$$

$$\frac{e^x - e^y}{x - y} = e^y\left(\frac{e^w - 1}{w}\right) \tag{8}$$

$$\frac{\ln x - \ln y}{x - y} = \frac{1}{y}\left(\frac{\ln(p+1)}{p}\right) \tag{9}$$

These four rules then require four primitives, which can be computed robustly using

$$\frac{\sin x}{x} = \begin{cases} 1 & |x| < \epsilon \\ \sin(x)/x & |x| \geq \epsilon \end{cases} \tag{10}$$

$$\frac{\cos x - 1}{x} = -\frac{x}{2}\left(\frac{\sin(x/2)}{x/2}\right)^2 \tag{11}$$

$$\frac{e^x - 1}{x} = \begin{cases} 1 & |x| < \epsilon \\ \mathrm{expm1}(x)/x & |x| \geq \epsilon \end{cases} \tag{12}$$

$$\frac{\ln(x+1)}{x} = \begin{cases} 1 & |x| < \epsilon \\ \mathrm{log1p}(x)/x & |x| \geq \epsilon \end{cases} \tag{13}$$

Here, log1p and expm1 are commonly-available library routines and ϵ is machine epsilon.

It should be noted that divided differences cannot always be robustly computed. It is not, for example, possible to compute divided differences of $\mathbf{u} \times \mathbf{v}$ robustly from the divided differences of $\mathbf{u}$ and $\mathbf{v}$. Indeed, the divided differences in this case need not even be finite.

6 Other tips

6.1 Reusing derivations

Do the derivation in a text file. Typically this would be LaTeX or some sort of testbed. There are certainly faster alternatives, at least at first (working things out on paper, working things out on a blackboard and taking a picture). In the end, text has some advantages. It can be pasted in a technical document to accompany the finally paper, and it can be modified into valid source code for the implementation. Fixing errors and tracing the corrections through a derivation is straightforward. A record of how the derivations were done can be kept with the paper so they are available for reference at a later date.

Another nice attribute of working with text files is that good text editors have powerful and efficient copy-paste and search-replace features. Careful use of these features can help reduce errors. This is especially important when dealing with lots of indices, which are easy to make mistakes with. Each line can begin with a copy and paste of the previous line. Search and replace allows index relabeling to be done quickly and accurately. Search and replace is also useful for making general substitutions, since it ensures that only exact matches are replaced and that the replacements are done without mistakes.

6.2 Computing vs applying derivatives

It is not always necessary to compute derivatives; it may be sufficient to be able to multiply by them. This is the case when using Krylov solvers like conjugate gradient, which only require the ability to multiply by a matrix. This may be a more convenient alternative, especially when the derivatives themselves involve tensors but the application can be done with matrices, vectors, and scalars alone. Application may also be cheaper or take less storage, depending on the circumstances.

6.3 Disabling independent variables

The numerical derivative tests do not test individual components of the derivatives. Rather, the test gives a quick indicator of whether all of the derivatives are correct. This can still be used to test individual blocks, which can be used to very quickly narrow problems down to a small number of terms. Lets say I am differentiating $f(x, y, z)$, computing partials f_x, f_y, f_z, f_{xx}, f_{xy}, f_{xz}, f_{yy}, f_{yz}, and f_{zz}. The derivative test will give pass/fail for the first derivative, and then another pass/fail for first vs second derivatives. If it is found that first derivatives pass but second do not, that leaves a lot of possibilities for where the error might be. If in the test z is set to some fixed value, and the derivatives containing z are set to zero, the derivative test will only be testing f_{xx}, f_{xy}, and f_{yy}. Fixing y and z tests only f_{xx}. In this way, diagonal blocks can be tested individually. When these are verified correct, leaving two variables free allows mixed partials to be tested one at a time.

7 Examples

In this section, I demonstrate these ideas on actual examples. Some of these are taken from graphics papers published in the last ten years where authors avoided the task of manually computing the derivatives (citations are omitted; I doubt the authors would want to be cited for failing to compute derivatives). Other examples are taken from my own papers in which interesting or particularly nontrivial derivatives were required. In each case, code is provided that shows snapshots of the steps I went through in computing the

derivatives. The end result in each case is a routine computing the first and second derivatives and verifying their correctness numerically.

For the purpose of these examples, I use very simple fixed-size matrix and vector classes to ease understanding of the code. The routines themselves are not particularly efficient. Any library with basic operations will do. Trying to implement these things without a reasonably friendly linear algebra framework is not recommended.

7.1 Example A

This example is taken from a recent graphics paper. The original paper used derivatives computed by Maple. Its simplification demonstrates strategies for eliminating tensors from intermediate computations. Many tensors are formed from the differentiation, since I compute the derivatives with respect to a matrix and some intermediates are also matrices. The scalar W is defined in the original paper by

$$\begin{aligned}
\mathbf{d}_1 &= \mathbf{x}_1 - \mathbf{x}_0 \\
\mathbf{d}_2 &= \mathbf{x}_2 - \mathbf{x}_0 \\
\mathbf{D} &= \begin{pmatrix} \mathbf{d}_1 & \mathbf{d}_2 \end{pmatrix} \\
\bar{\mathbf{F}} &= \mathbf{D}\mathbf{R}_X^{-1} \\
\bar{\mathbf{E}} &= \frac{1}{2}(\bar{\mathbf{F}}^T\bar{\mathbf{F}} - \mathbf{I}) \\
W &= A_X\left(\frac{\lambda}{2}(\operatorname{tr}\bar{\mathbf{E}})^2 + \mu\operatorname{tr}(\bar{\mathbf{E}}^2)\right)
\end{aligned}$$

The goal with this one is to compute $\frac{\partial W}{\partial \mathbf{x}_i}$ and $\frac{\partial^2 W}{\partial \mathbf{x}_i \partial \mathbf{x}_j}$, for $i, j \in \{0, 1, 2\}$. Note that

$$\frac{\partial W}{\partial \mathbf{D}} = \begin{pmatrix} \frac{\partial W}{\partial \mathbf{d}_1} & \frac{\partial W}{\partial \mathbf{d}_2} \end{pmatrix} \qquad \frac{\partial W}{\partial \mathbf{x}_0} = -\frac{\partial W}{\partial \mathbf{d}_1} - \frac{\partial W}{\partial \mathbf{d}_2} \qquad \frac{\partial W}{\partial \mathbf{x}_1} = \frac{\partial W}{\partial \mathbf{d}_1} \qquad \frac{\partial W}{\partial \mathbf{x}_2} = \frac{\partial W}{\partial \mathbf{d}_2}.$$

The same idea applies to the second derivatives ($j, k > 0$)

$$\frac{\partial^2 W}{\partial \mathbf{x}_j \partial \mathbf{x}_k} = \frac{\partial^2 W}{\partial \mathbf{d}_j \partial \mathbf{d}_k} \qquad \frac{\partial^2 W}{\partial \mathbf{x}_0 \partial \mathbf{x}_k} = -\frac{\partial^2 W}{\partial \mathbf{d}_1 \partial \mathbf{d}_k} - \frac{\partial^2 W}{\partial \mathbf{d}_2 \partial \mathbf{d}_k} \qquad \frac{\partial^2 W}{\partial \mathbf{x}_0 \partial \mathbf{x}_0} = -\frac{\partial^2 W}{\partial \mathbf{x}_0 \partial \mathbf{x}_1} - \frac{\partial^2 W}{\partial \mathbf{x}_0 \partial \mathbf{x}_2}$$

This gives us the somewhat simpler task of computing $\frac{\partial W}{\partial \mathbf{D}}$. I computed and simplified the derivatives for this one in our testbed (`testbed-a.cpp`) and in the testing framework (`example-a.cpp`). Let $a = A_X$ and $\mathbf{S} = \mathbf{R}_X^{-1}$. Then the first and second derivatives can be computed as

$$\begin{aligned}
\bar{F}_{ij} &= D_{ik}S_{kj} & \bar{E}_{ij} &= (\bar{F}_{ki}\bar{F}_{kj} - \delta_{ij})/2 & B &= \bar{E}_{ii} & W &= a\lambda B^2/2 + a\mu\bar{E}_{ij}\bar{E}_{ji} \\
E_{ki} &= S_{kj}\bar{E}_{ji} & F_{ki} &= S_{kj}\bar{F}_{ij} & G_{ki} &= \bar{F}_{kj}\bar{F}_{ij} & W_{,rs} &= a\lambda F_{sr}B + 2a\mu\bar{F}_{ri}E_{si} \\
H_{sv} &= S_{si}S_{vi} & C_{sv} &= \lambda B H_{sv} + 2\mu S_{sj}E_{vj} & W_{,rsuv} &= a\delta_{ru}C_{sv} + a\lambda F_{sr}F_{vu} + a\mu F_{su}F_{vr} + a\mu H_{sv}G_{ru}
\end{aligned}$$

Note that all operations except for assembling the tensor $W_{,rsuv}$ are strictly scalar, vector, and matrix operations. Recall the relationship between derivatives in $\mathbf{D}$ and derivatives in $\mathbf{d}_k$.

$$\left(\frac{\partial W}{\partial \mathbf{x}_s}\right)_r = W_{,rs} \qquad\qquad \left(\frac{\partial^2 W}{\partial \mathbf{x}_s \partial \mathbf{x}_v}\right)_{ru} = W_{,rsuv}$$

7.2 Example B

This is another example from a paper where derivatives were computed in Maple. This example actually simplifies quite dramatically, though the paper does not mention this. This leads me to wonder whether the author was aware of this. The example is more complex than the previous one, and it demonstrates a different

set of techniques. I differentiate and simplify the expression in one way. I then repeat the differentiation (but not the simplification, which was done the same was as the first time) using a different strategy. The quantity to be differentiated is

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

Here, two derivatives of E are desired with respect to $\mathbf{x}_A$, $\mathbf{x}_B$, and $\mathbf{x}_C$. Some simplification is in order before this one is ready to be differentiated.

$$\mathbf{u} = \mathbf{x}_A - \mathbf{x}_C \qquad \mathbf{v} = \mathbf{x}_B - \mathbf{x}_C \qquad \mathbf{w} = \mathbf{u} - \mathbf{v} \qquad \mathbf{N} = \mathbf{u} \times \mathbf{v} \qquad A = \|\mathbf{N}\| \qquad B = \frac{1}{A} \qquad \mathbf{n} = B\mathbf{N}$$

$$\mathbf{z} = \mathbf{x}_A - \mathbf{c} \qquad d_{min} = \mathbf{z} \cdot \mathbf{n} \qquad g = d_{min}^{-2} \qquad \cot(\alpha) = B\mathbf{u} \cdot \mathbf{w} \qquad \cot(\beta) = -B\mathbf{w} \cdot \mathbf{v} \qquad \cot(\gamma) = B\mathbf{v} \cdot \mathbf{u}$$

$$\begin{aligned}
E_{Dirichlet} &= g(\cot(\alpha)\|\mathbf{v}\|^2 + \cot(\beta)\|\mathbf{u}\|^2 + \cot(\gamma)\|\mathbf{w}\|^2) \\
E_{area} &= g\frac{A^2}{InputArea} \\
E = E_{combined} &= a \cdot E_{Dirichlet} + b \cdot E_{area}
\end{aligned}$$

The form of $E_{Dirichlet}$ is rather curious. Ignoring the factor of gB common to all of the terms, I have a degree four polynomial in the components of $\mathbf{u}$ and $\mathbf{v}$. The expression must also be very highly symmetrical with respect to these, since this value should be invariant with respect to permuting the vertices of the triangle. That suggests that this polynomial may have a rather simple form. Investigating further leads to the substantial simplification

$$\begin{aligned}
E_{Dirichlet} &= g(\cot(\alpha)\|\mathbf{v}\|^2 + \cot(\beta)\|\mathbf{u}\|^2 + \cot(\gamma)\|\mathbf{w}\|^2) \\
&= g(B\mathbf{u} \cdot \mathbf{w}\|\mathbf{v}\|^2 - B\mathbf{w} \cdot \mathbf{v}\|\mathbf{u}\|^2 + B\mathbf{v} \cdot \mathbf{u}\|\mathbf{w}\|^2) \\
&= 2gB\|\mathbf{N}\|^2 \\
&= 2gA \\
E &= a \cdot E_{Dirichlet} + b \cdot E_{area} \\
&= 2agA + bg\frac{A^2}{InputArea} \\
&= 2agA + cgA^2 \qquad c = \frac{b}{InputArea} \\
&= \frac{2aA^3 + cA^4}{(\mathbf{z} \cdot \mathbf{N})^2}
\end{aligned}$$

The derivatives can now be computed. Observe that E depends on $\mathbf{x}_A$, $\mathbf{x}_B$, and $\mathbf{x}_C$ only through $\mathbf{z}$ and $\mathbf{N}$, noting that $A = \|\mathbf{N}\|$. This suggests that a compact approach would be to first compute $\mathbf{E}_N = \frac{\partial E}{\partial \mathbf{N}}$ and $\mathbf{E}_z = \frac{\partial E}{\partial \mathbf{z}}$, following by the chain rule. First, the energy E is computed (using $\mathbf{z}$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{N}$, A from above)

$$d = \mathbf{z} \cdot \mathbf{N} \qquad g = d^{-2} \qquad C = (2aA + fA^2)A^2 \qquad E = gC$$

Next, I compute the first derivatives.

$$\mathbf{P}_A = -\mathbf{v}^* \quad \mathbf{P}_B = \mathbf{u}^* \quad \mathbf{P}_C = (\mathbf{v} - \mathbf{u})^* \qquad h = 6aA + 4fA^2 \qquad \mathbf{E}_N = -2g^2 dC\mathbf{z} + gh\mathbf{N} \quad \mathbf{E}_z = -2g^2 dC\mathbf{N}$$

$$q_A = 1 \qquad q_B = 0 \qquad q_C = 0 \qquad \frac{\partial E}{\partial \mathbf{x}_v} = \mathbf{P}_v^T \mathbf{E}_N + q_v \mathbf{E}_z$$

where subscript v runs over the vertices $\{A, B, C\}$. The quantity $\mathbf{u}^*$ is the cross product matrix, chosen so that $\mathbf{u}^*\mathbf{v} = \mathbf{u} \times \mathbf{v}$ for any vector $\mathbf{v}$. Finally, the second derivatives are

$$m = 2g^2(4gd^2 - 1) \quad \mathbf{p} = mC\mathbf{z} - 2g^2 dh\mathbf{N} \quad \mathbf{E}_{NN} = \mathbf{p}\mathbf{z}^T + g((6a/A + 8f)\mathbf{N} - 2gdh\mathbf{z})\mathbf{N}^T + gh\mathbf{I} \quad \mathbf{E}_{zz} = mC\mathbf{N}\mathbf{N}^T$$

$$\mathbf{E}_{Nz} = \mathbf{p}\mathbf{N}^T - 2g^2 dC\mathbf{I} \qquad \frac{\partial^2 E}{\partial \mathbf{x}_u \partial \mathbf{x}_v} = \mathbf{P}_u^T \mathbf{E}_{NN} \mathbf{P}_v + q_v \mathbf{P}_u^T \mathbf{E}_{Nz} + q_u \mathbf{E}_{Nz}^T \mathbf{P}_v + q_u q_v \mathbf{E}_{zz} - r_{uv} \mathbf{E}_N^*$$

where subscripts u, v run over the vertices $\{A, B, C\}$. The coefficients r_{uv} are skew symmetric ($r_{uv} = -r_{vu}$, $r_{uu} = 0$) with $r_{AB} = r_{BC} = r_{CA} = 1$. As with the first example, I computed and simplified the derivatives for this one in our testbed (`testbed-b.cpp`) and in the testing framework (`example-b.cpp`). Note that only scalars, vectors, and matrices were used.

7.2.1 Change of variables

Notice how in this example the energy depends on positions $\mathbf{x}_A$, $\mathbf{x}_B$, and $\mathbf{x}_C$ only through the two quantities $\mathbf{z} = \mathbf{x}_A - \mathbf{c}$ and $\mathbf{N} = (\mathbf{x}_A - \mathbf{x}_C) \times (\mathbf{x}_B - \mathbf{x}_C)$. By computing partial derivatives of energy with respect to these two intermediates $(\mathbf{E}_N, \mathbf{E}_z, \mathbf{E}_{NN}, \mathbf{E}_{Nz}, \mathbf{E}_{zz})$, the desired partial derivatives can then be computed using the chain rule. This is a divide and conquer strategy; the complexities of the energy are decoupled from the complications relating to the cross product. This strategy is often helpful when there is a sort of bottleneck, where the dependence on the original variables can be expressed in terms of a relatively small number of expressions. In addition to simplifying the derivatives, it also factors out computations that would otherwise be redundant. Both parts can be debugged independently, since they can both be tested with numerical derivative tests.

This strategy could have been repeated. For example, E only depends on $\mathbf{z}$ and $\mathbf{N}$ through $d = \mathbf{z} \cdot \mathbf{N}$ and $A^2 = \|\mathbf{N}\|^2$. This is nice because now the derivatives are functions of scalars only, which are often easier and less expensive to work with.

I can pursue this strategy further, considering the energy to depend, in turn, on (g, A^2), (d, A^2), $(d, \mathbf{N})$, $(\mathbf{z}, \mathbf{N})$, $(\mathbf{z}, \mathbf{u}, \mathbf{v})$, and finally $(\mathbf{x}_A, \mathbf{x}_B, \mathbf{x}_C)$. When this is done, I get the following computations, using $q = A^2$ for convenience:

Pair	Chain rule	New derivatives
1	$C_q = 3aA + 2fq \quad C_{qq} = \frac{3a}{2A} + 2f$	$E_g = C \quad E_q = gC_q \quad E_{qg} = C_q \quad E_{qq} = gC_{qq}$
2	$g_d = -2g^2 d \quad g_{dd} = 6g^2$	$E_d = E_g g_d \quad E_{dd} = E_g g_{dd} \quad E_{qd} = E_{qg} g_d$
3	$\mathbf{q}_N = 2\mathbf{N} \quad \mathbf{q}_{NN} = 2\mathbf{I}$	$\hat{\mathbf{E}}_N = E_q \mathbf{q}_N \quad \hat{\mathbf{E}}_{NN} = E_{qq}\mathbf{q}_N\mathbf{q}_N^T + E_q\mathbf{q}_{NN} \quad \mathbf{E}_{Nd} = E_{qd}\mathbf{q}_N$
4	$\mathbf{d}_z = \mathbf{N} \quad \mathbf{d}_N = \mathbf{z} \quad \mathbf{d}_{Nz} = \mathbf{I}$	$\mathbf{E}_z = E_d\mathbf{d}_z \quad \mathbf{E}_N = \hat{\mathbf{E}}_N + E_d\mathbf{d}_N \quad \mathbf{E}_{zz} = E_{dd}\mathbf{d}_z\mathbf{d}_z^T$ $\mathbf{E}_{Nz} = (\mathbf{E}_{Nd} + E_{dd}\mathbf{d}_N)\mathbf{d}_z^T + E_d\mathbf{d}_{Nz}$ $\mathbf{E}_{NN} = \hat{\mathbf{E}}_{NN} + \mathbf{d}_N\mathbf{E}_{Nd}^T + (\mathbf{E}_{Nd} + E_{dd}\mathbf{d}_N)\mathbf{d}_N^T$
5	$\mathbf{N}_u = -\mathbf{v}^* \quad \mathbf{N}_v = \mathbf{u}^*$	$\mathbf{E}_u = \mathbf{N}_u^T\mathbf{E}_N \quad \mathbf{E}_v = \mathbf{N}_v^T\mathbf{E}_N \quad \mathbf{E}_{uu} = \mathbf{N}_u^T\mathbf{E}_{NN}\mathbf{N}_u$ $\mathbf{E}_{uv} = \mathbf{N}_u^T\mathbf{E}_{NN}\mathbf{N}_v - \mathbf{E}_N^* \quad \mathbf{E}_{vv} = \mathbf{N}_v^T\mathbf{E}_{NN}\mathbf{N}_v$ $\mathbf{E}_{uz} = \mathbf{N}_u^T\mathbf{E}_{Nz} \quad \mathbf{E}_{vz} = \mathbf{N}_v^T\mathbf{E}_{Nz}$
6	$\mathbf{z}_A = \mathbf{u}_A = \mathbf{v}_B = \mathbf{I}$ $\mathbf{u}_C = \mathbf{v}_C = -\mathbf{I}$	$\mathbf{E}_A = \mathbf{E}_z + \mathbf{E}_u \quad \mathbf{E}_B = \mathbf{E}_v \quad \mathbf{E}_C = -\mathbf{E}_u - \mathbf{E}_v$ $\mathbf{E}_{AA} = \mathbf{E}_{zz} + \mathbf{E}_{uu} + \mathbf{E}_{uz}^T + \mathbf{E}_{uz} \quad \mathbf{E}_{AB} = \mathbf{E}_{vz}^T + \mathbf{E}_{uv}$ $\mathbf{E}_{AC} = -\mathbf{E}_{vz}^T - \mathbf{E}_{uu} - \mathbf{E}_{uz}^T - \mathbf{E}_{uv} \quad \mathbf{E}_{BB} = \mathbf{E}_{vv}$ $\mathbf{E}_{BC} = -\mathbf{E}_{uv}^T - \mathbf{E}_{vv} \quad \mathbf{E}_{CC} = \mathbf{E}_{uu} + \mathbf{E}_{uv} + \mathbf{E}_{uv}^T + \mathbf{E}_{vv}$

This computational sequence is, of course, ripe for simplification and optimization. This sequence was worked out directly in the testing framework (`example-b2.cpp`). Snapshots captured along the way reveal how the intermediate numerical derivative tests were done. Note that partial derivatives also depend on what is being held fixed; The need to distinguish $\hat{\mathbf{E}}_N$ and $\hat{\mathbf{E}}_{NN}$ from $\mathbf{E}_N$ and $\mathbf{E}_{NN}$ reflects this.

7.3 Example C

The following example is from one of my papers. In this case, there are a number of components that make the derivatives difficult. The first is that tensors cannot be avoided; the first derivatives depend on the second derivatives of a provided energy function, so second derivatives will inevitably require fourth derivatives. The other striking thing that makes this example challenging is that one of the quantities (s) must be computed as the solution to a cubic equation. Another complication is that the provided energy function is evaluated at a location that depends on the solution to the cubic. Differentiation of the result ψ is with respect to $\mathbf{x}$, and the quantities $\mathbf{r}$ and a are constant. In particular, $\mathbf{r}$ is the all-ones vector, and $a \in (0, 1)$ is a parameter.

$$\mathbf{u} = \frac{\mathbf{x} - \mathbf{r}}{\|\mathbf{x} - \mathbf{r}\|} \qquad \mathbf{q} = \mathbf{r} + (\mathbf{x} - \mathbf{r})s \qquad h = (\mathbf{x} - \mathbf{q}) \cdot \mathbf{u} \qquad a = \prod_\alpha q_\alpha = \prod_\alpha (r_\alpha + (x_\alpha - r_\alpha)s)$$

$$\phi = \Psi(\mathbf{q}) \qquad g_i = \Psi_{,i}(\mathbf{q}) \qquad H_{ij} = \Psi_{,ij}(\mathbf{q}) \qquad \psi = \phi + h\mathbf{g} \cdot \mathbf{u} + \frac{1}{2}h^2\mathbf{u}^T\mathbf{H}\mathbf{u}$$

The original derivatives I worked out for the energy were very complicated. Although very carefully worked out with two sets of eyes, debugging the results was quite difficult. I have reworked the derivatives from scratch, using the techniques and tools outlined in this document. The result is much simpler. Let $\mathbf{v} = \mathbf{x} - \mathbf{r}$ and solve the cubic to get s. Then,

$$t = 1 - s \quad w = \frac{t^2}{2} \qquad \mathbf{q} = \mathbf{r} + s\mathbf{v} \qquad \mathbf{M} = \begin{pmatrix} 0 & \mathbf{q}_z & \mathbf{q}_y \\ \mathbf{q}_z & 0 & \mathbf{q}_x \\ \mathbf{q}_y & \mathbf{q}_x & 0 \end{pmatrix} \quad \mathbf{y} = \mathbf{M}\mathbf{q} \qquad \mathbf{z} = \mathbf{M}\mathbf{v} \quad \mathbf{T}_c = \mathbf{y}s$$

$$T_s = \mathbf{y} \cdot \mathbf{v} \quad T_{ss} = 2\mathbf{z} \cdot \mathbf{v} \quad \mathbf{T}_{cs} = \mathbf{y} + 2s\mathbf{z} \quad \mathbf{T}_{cc} = 2s^2\mathbf{M} \qquad \mathbf{s}_x = -\frac{\mathbf{T}_c}{T_s} \quad \mathbf{q}_x = \mathbf{v}\mathbf{s}_x^T + s\mathbf{I}$$

$$\mathbf{s}_{xx} = -\frac{T_{ss}\mathbf{s}_x\mathbf{s}_x^T + \mathbf{T}_{cs}\mathbf{s}_x^T + \mathbf{s}_x\mathbf{T}_{cs}^T\mathbf{T}_{cc}}{Ts} \qquad \mathbf{A} = \mathbf{v}\mathbf{s}_x^T - \frac{3}{2}t\mathbf{I}$$

Now, I need to evaluate the energy and its first four derivatives, contracting with $\mathbf{v}$ immediately for the third and fourth derivatives to avoid dealing with tensors in the code. $Z^{00} = \phi(\mathbf{q})$, $Z_i^{10} = \phi_{,i}(\mathbf{q})$, $Z_{ij}^{20} = \phi_{,ij}(\mathbf{q})$,

$Z^{31}_{ij} = \phi_{,ijk}(\mathbf{q})v_k$, $Z^{42}_{ij} = \phi_{,ijkm}(\mathbf{q})v_k v_m$. Further contractions with $\mathbf{v}$ are useful and computed as needed while computing the final energy ψ and derivatives $\boldsymbol{\psi}_x$ and $\boldsymbol{\psi}_{xx}$.

$$Z^{11} = \mathbf{Z}^{10} \cdot \mathbf{v} \qquad \mathbf{Z}^{21} = \mathbf{Z}^{20}\mathbf{v} \qquad Z^{22} = \mathbf{Z}^{21} \cdot \mathbf{v} \qquad \psi = Z^{00} + tZ^{11} + wZ^{22}$$
$$\mathbf{Z}^{32} = \mathbf{Z}^{31}\mathbf{v} \qquad Z^{33} = \mathbf{Z}^{32} \cdot \mathbf{v} \qquad \boldsymbol{\psi}_x = \mathbf{Z}^{10} + t\mathbf{Z}^{21} + w\mathbf{q}_x^T\mathbf{Z}^{32}$$
$$\boldsymbol{\psi}_{xx} = \mathbf{Z}^{20} - t\mathbf{A}^T\mathbf{Z}^{31}\mathbf{A} + \left(\frac{5}{2}w + 1\right)t\mathbf{Z}^{31} + wZ^{33}\mathbf{s}_{xx} + w\mathbf{q}_x^T\mathbf{Z}^{42}\mathbf{q}_x$$

The simplification of the original is dramatic and does not involve square roots. One significant piece of this, not noticed in the original derivation, is that $h\mathbf{u} = t\mathbf{v}$. The only uses for h and $\mathbf{u}$ occur in the combination $h\mathbf{u}$; this substitution is very beneficial, since $\mathbf{v}$ has very simple derivatives and $t = 1 - s$ is a scalar. These simplifications, and many others, were possible because of the ability to make small speculative simplifications without the risk of introducing mistakes.

In the case of this example, I am differentiating an energy in diagonal space, so in addition to two derivatives I also need divided differences.

$$\frac{\psi_{,i} - \psi_{,j}}{x_i - x_j} = \left(\frac{Z^{10}_i - Z^{10}_j}{x_i - x_j}\right) + t\left(\frac{Z^{21}_i - Z^{21}_j}{x_i - x_j}\right) + \frac{2s^2wZ^{33}}{T_s}M_{ij} + sw\left(\frac{Z^{32}_i - Z^{32}_j}{x_i - x_j}\right)$$

The remaining divided differences are of contractions of energy density derivatives and should be computed along with energy density derivatives. Derivatives were worked out and simplified in the testbed (`testbed-c.cpp`). These were then transferred to the testing framework, debugging by matching the computations from the testbed. Divided differences were then computed directly in the testing framework (`example-c.cpp`).