Notes on computation of first and second order partial derivatives for optimization

Niko Brümmer

September 21, 2009

1 Definitions

A function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$, where $\mathbf{y} = f(\mathbf{x})$, or $[y_1, \cdots, y_m]' = f([x_1, \cdots, x_N]')$, has the following first and second order partial derivatives[1]:

Jacobian, an m-by-n matrix, $\mathbf{J} = \left[\frac{\partial y_i}{\partial x_j}\right]$,

Hessians, m different n-by-n matrices, $\mathbf{H}_k = \left[\frac{\partial^2 y_k}{\partial x_i \partial x_j}\right]$.

In general, the Jacobian and Hessian matrices are dependent on $\mathbf{x}$, but we omit this to avoid cluttering notation. We now define three functions in terms of these matrices:

$$J : \mathbb{R}^n \mapsto \mathbb{R}^m, \text{ where} \qquad J(\tilde{\mathbf{x}}) = \mathbf{J}\tilde{\mathbf{x}}, \tag{1}$$

$$G : \mathbb{R}^m \mapsto \mathbb{R}^n, \text{ where} \qquad G(\tilde{\mathbf{y}}) = \mathbf{J}'\tilde{\mathbf{y}}, \tag{2}$$

$$H : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^n, \text{ where} \qquad H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) = \sum_{k=1}^{m} \tilde{y}_k \mathbf{H}_k \tilde{\mathbf{x}}. \tag{3}$$

Since $\mathbf{x}$ is suppressed in the notation, the outputs of these functions are still dependent on $\mathbf{x}$, but also on the new inputs $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$.

In places we *do* need to explicitly refer to the original function input. In this case we use the notation $|_{\mathbf{x}+\alpha\tilde{\mathbf{x}}}$ to denote that the above-defined partial derivatives of f are evaluated at $\mathbf{x}+\alpha\tilde{\mathbf{x}}$, rather than $\mathbf{x}$. We need this notation to express H in terms of the directed derivative of G:

$$H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})|_{\mathbf{x}} = \left.\frac{\partial\, G(\tilde{\mathbf{y}})|_{\mathbf{x}+\alpha\tilde{\mathbf{x}}}}{\partial\alpha}\right|_{\alpha=0} = \lim_{\alpha\to 0} \frac{G(\tilde{\mathbf{y}})|_{\mathbf{x}+\alpha\tilde{\mathbf{x}}}}{\alpha} \tag{4}$$

[1] We tacitly assume we are working with 'nice' non-pathological functions for which these derivatives exist and for which the Hessian matrices are symmetrical.

2 Derivatives for optimization of objective functions

Optimization objective functions have $m = 1$ and $n \geq 1$. If we set $\tilde{y}_1 = 1$, then (2) gives the *gradient*, $\left[\frac{\partial y}{\partial x_i}\right]$, and (3) gives the *Hessian-vector product*, $\mathbf{H}\tilde{\mathbf{x}}$, both of which are usually required by second-order optimization algorithms.

3 Chain rules

3.1 Function composition

Complex objective functions can be constructed by using function composition of simpler functions. Here we give chain rules which show how to compute the derivatives of compositions. Let $g : \mathbb{R}^n \mapsto \mathbb{R}^k$ and $f : \mathbb{R}^k \mapsto \mathbb{R}^m$, which can be composed as $h(\mathbf{x}) = f(g(\mathbf{x}))$. Now, the derivatives of the composition are:

$$J_h(\tilde{\mathbf{x}}) = J_f\Big(J_g(\tilde{\mathbf{x}})\Big), \tag{5}$$

$$G_h(\tilde{\mathbf{y}}) = G_g\Big(G_f(\tilde{\mathbf{y}})\Big), \tag{6}$$

$$H_h(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) = H_g\Big(\tilde{\mathbf{x}}, G_f(\tilde{\mathbf{y}})\Big) + G_g\Big(H_f\left(J_g(\tilde{\mathbf{x}}), \tilde{\mathbf{y}}\right)\Big) \tag{7}$$

To make this notation more familiar it may help to note that when f has a scalar output, then $G_f(1) = \left[\frac{\partial f}{\partial g_i}\right]$ is the gradient of f, and $G_h(1) = \left[\frac{\partial f}{\partial x_i}\right] = \mathbf{J}_g' \left[\frac{\partial f}{\partial g_i}\right]$ is the gradient of the composition. In neural networks this is called *backpropagation* of the gradient.

3.2 Implementation

For efficient implementation of the derivatives of large-scale problems, the following principles can be used:

1. If memory allows, store intermediate results obtained during function value or gradient computations, to be re-used for gradient or Hessian computations.

2. Do not explicitly compute and store $\mathbf{J}$ and $\mathbf{H}$, they may be very large. We need only to be able to perform the mappings (1), (2) and (3), for a relatively small number of the inputs $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$.

In a typical second-order optimization iteration, the order of computation will be (i) evaluate the objective function value, (ii) evaluate the gradient, (iii) evaluate $\mathbf{H}\tilde{\mathbf{x}}$ for several different values of $\tilde{\mathbf{x}}$. For a function composition $f(g(\mathbf{x}))$ the chain rules above imply the following order of computation:

1. For the function value compute (i) $\mathbf{y} = g(\mathbf{x})$, then (ii) $f(\mathbf{y})$.
2. For the gradient compute (i) $\tilde{\mathbf{y}} = G_f(\cdot)$, then (ii) $G_g(\tilde{\mathbf{y}})$.
3. For each value of $\tilde{\mathbf{x}}$, compute (i) $\breve{\mathbf{y}} = J_g(\tilde{\mathbf{x}})$ and $\tilde{\mathbf{h}} = H_g(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$, then (ii) $\mathbf{h} = H_f(\breve{\mathbf{y}}, \cdot)$, and finally (iii) $\tilde{\mathbf{h}} + G_g(\mathbf{h})$.

If memory allows, all intermediate values in earlier steps that will be needed in later steps should be stored to avoid re-computation.

For a linear function $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$, the Jacobian is just $\mathbf{A}$. But often a linear mapping $f(\mathbf{x})$ may be defined via some algorithm (for example an FFT) rather than a dense matrix. In this case the algorithm for $f(\mathbf{x})$ is also the algorithm for $J(\mathbf{x})$. The art is then to also implement the algorithm efficiently which computes the transpose of that mapping. See section 5.2 below for the example of the diag() function.

If $\mathbf{A}$ is an explicitly specified large dense matrix, then depending on the platform, the implementation $G(\tilde{\mathbf{y}}) = (\tilde{\mathbf{y}}'\mathbf{A})'$ may be more efficient than $G(\tilde{\mathbf{y}}) = \mathbf{A}'\tilde{\mathbf{y}}$, because in the worst case, transposing $\mathbf{A}$ could involve a non-in-place copying of the large matrix.

As in the case of the Jacobian, it is often possible to find efficient algorithms to analytically compute, or numerically approximate the Hessian mapping (3). We devote section 4 to discuss this. Also see the example below in section 5.1, where we derive an analytical algorithm for (3).

3.3 Functions with multiple inputs

Consider a function of two inputs defined as $g : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^k$. We handle this function in our framework by stacking the inputs into a single vector of size $m + n$. Define $\mathbf{s} = \left[\begin{smallmatrix}\mathbf{x}\\ \mathbf{z}\end{smallmatrix}\right]$ and $\tilde{\mathbf{s}} = \left[\begin{smallmatrix}\tilde{\mathbf{x}}\\ \tilde{\mathbf{z}}\end{smallmatrix}\right]$ and $f(\mathbf{s}) = g(\mathbf{x}, \mathbf{z})$, then the derivatives of f can be expressed in terms of derivatives of g:

$$J_f(\tilde{\mathbf{s}}) = J_{\mathbf{x}}(\tilde{\mathbf{x}}) + J_{\mathbf{z}}(\tilde{\mathbf{z}}) \tag{8}$$

$$G_f(\tilde{\mathbf{y}}) = \begin{bmatrix} G_{\mathbf{x}}(\tilde{\mathbf{y}}) \\ G_{\mathbf{z}}(\tilde{\mathbf{y}}) \end{bmatrix} \tag{9}$$

Here the right-hand sides denotes partial derivatives of g, w.r.t. $\mathbf{x}$ or $\mathbf{z}$ respectively. To obtain H_f, it is best to apply (4) to $G_f(\tilde{\mathbf{y}})$.

4 Computing the Hessian-vector product

Note that it is never necessary to compute the full Hessian matrices, $\mathbf{H}_k$, which grow as the square of the number of input parameters. This can be very large for large-scale optimizations.

The product $\mathbf{H}_k\tilde{\mathbf{x}}$ can often be computed analytically and efficiently—see section 5 for examples. But it is also well suited to numerical approximation, provided a function which computes the gradient is available. Then a finite-difference approximation to (4) can be used. This can be implemented, with some suitably small $\epsilon > 0$, as

$$H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) \approx \frac{G(\tilde{\mathbf{y}})|_{\mathbf{x}+\epsilon\tilde{\mathbf{x}}} - G(\tilde{\mathbf{y}})|_{\mathbf{x}}}{\epsilon}, \tag{10}$$

or more accurately as

$$H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) \approx \frac{G(\tilde{\mathbf{y}})|_{\mathbf{x}+\epsilon\tilde{\mathbf{x}}} - G(\tilde{\mathbf{y}})|_{\mathbf{x}-\epsilon\tilde{\mathbf{x}}}}{2\epsilon}, \text{ or} \tag{11}$$

or if complex arithmetic is available, as

$$H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) \approx \frac{\operatorname{imag} G(\tilde{\mathbf{y}})|_{\mathbf{x}+i\epsilon\tilde{\mathbf{x}}}}{\epsilon}. \tag{12}$$

The last method is called *complex step differentiation*. It has the advantages that:

- It is easier to choose a suitable value for ϵ, because it is accurate for a wider range of values than the real step methods.
- In particular, ϵ can be made very small, which can then give very accurate approximation.

but, it has the following pitfalls and disadvantages:

- It uses complex arithmetic, which uses more memory. (It also uses more CPU per function evaluation, but probably not more than two real function evaluations, which the above symmetric real step method also does.)
- It cannot be applied to calculations which already involve complex arithmetic. In particular, you cannot differentiate twice with this method. This is why we need an implementation of the gradient.
- MATLAB's transpose is a problem, because the usual operator $'$ does a complex conjugate transpose. Code that will be differentiated with the complex step method should instead transpose a matrix $\mathbf{X}$ as $\mathbf{X}.'$, where the sign of the imaginary part is not changed.

5 Matrix differentiation

For functions using matrix multiplications, inverses, determinants etc., matrix differential calculus[2] can provide solutions to computing the required derivatives.

To understand the following, you need at least some previous exposure to matrix differential calculus. We show how to compute the required derivatives by way of a few examples. We use the notation of Magnus and Neudecker, where d denotes *differential* and D denotes *partial derivatives*.

5.1 Example 1

Let f be a function which maps a matrix $\mathbf{X}$ to a matrix $\mathbf{Y}$, thus:

$$\mathbf{Y} = \mathbf{X}'\mathbf{K}\mathbf{X} \tag{13}$$

where $\mathbf{K}$ is a constant matrix of appropriate size. We define f in terms of the vectorized representations of the matrices:

$$f : \mathrm{vec}(\mathbf{X}) \mapsto \mathrm{vec}(\mathbf{X}'\mathbf{K}\mathbf{X}) \tag{14}$$

Now we need to compute the above-defined mappings J, G and H:

5.1.1 Jacobian

J is the easiest, via straight-forward computation of differentials: To apply the Jacobian mapping, $J(\tilde{\mathbf{x}})$ do:

1. Reshape input to matrix: $\tilde{\mathbf{x}} \mapsto \mathsf{d}\mathbf{X}$
2. Compute differentials: $\mathsf{d}\mathbf{Y} = \mathsf{d}\mathbf{X}'\mathbf{K}\mathbf{X} + \mathbf{X}'\mathbf{K}\mathsf{d}\mathbf{X}$
3. Output: $\mathrm{vec}(\mathsf{d}\mathbf{Y})$

5.1.2 Gradient

To compute the transpose Jacobian mapping, we need to consider how partial derivatives backpropagate from some arbitrary scalar-valued outer function g in the composition $g(\mathbf{Y}) = g(\mathbf{X}'\mathbf{K}\mathbf{X})$. Let the matrix of partial derivatives

[2]See Tom Minka's introductory tutorial, 'Old and new matrix algebra useful for statistics', available here `http://research.microsoft.com/en-us/um/people/minka/papers/matrix/minka-matrix.pdf`. For a more complete treatment, see the textbook by Magnus and Neudecker, *Matrix Differential Calculus with Applications in Statistics and Econometrics*.

of g w.r.t. the elements of $\mathbf{Y}$ be denoted[3] as $\mathsf{D}\mathbf{Y} = [\frac{\partial g}{\partial y_{ij}}]$. The chain rule for differentials now gives[4]: $\mathsf{d}g(\mathbf{Y}) = \mathrm{trace}(\mathsf{d}\mathbf{Y}'\mathsf{D}\mathbf{Y})$. The trace allows[5] for re-arrangement of matrix products, to find $\mathsf{D}\mathbf{X} = [\frac{\partial g}{\partial x_{ij}}]$. Note that we re-use the above expansion of $\mathsf{d}\mathbf{Y}$ in the form $\mathsf{d}\mathbf{Y}' = \mathsf{d}\mathbf{X}'\mathbf{K}'\mathbf{X} + \mathbf{X}'\mathbf{K}'\mathsf{d}\mathbf{X}$:

$$\begin{aligned}
\mathsf{d}g(\mathbf{Y}) &= \mathrm{trace}(\mathsf{d}\mathbf{Y}'\mathsf{D}\mathbf{Y}) \\
&= \mathrm{trace}(\mathsf{d}\mathbf{X}'\mathbf{K}'\mathbf{X}\mathsf{D}\mathbf{Y}) + \mathrm{trace}(\mathbf{X}'\mathbf{K}'\mathsf{d}\mathbf{X}\mathsf{D}\mathbf{Y}) \\
&= \mathrm{trace}(\mathsf{d}\mathbf{X}'\mathbf{K}'\mathbf{X}\mathsf{D}\mathbf{Y}) + \mathrm{trace}(\mathsf{D}\mathbf{Y}'\mathsf{d}\mathbf{X}'\mathbf{K}\mathbf{X}) \\
&= \mathrm{trace}(\mathsf{d}\mathbf{X}'\mathbf{K}'\mathbf{X}\mathsf{D}\mathbf{Y}) + \mathrm{trace}(\mathsf{d}\mathbf{X}'\mathbf{K}\mathbf{X}\mathsf{D}\mathbf{Y}') \\
&= \mathrm{trace}(\mathsf{d}\mathbf{X}'(\mathbf{K}'\mathbf{X}\mathsf{D}\mathbf{Y} + \mathbf{K}\mathbf{X}\mathsf{D}\mathbf{Y}')) \\
&= \mathrm{trace}(\mathsf{d}\mathbf{X}'\mathsf{D}\mathbf{X})
\end{aligned} \tag{15}$$

where the desired result is now:

$$\mathsf{D}\mathbf{X} = \mathbf{K}'\mathbf{X}\mathsf{D}\mathbf{Y} + \mathbf{K}\mathbf{X}\mathsf{D}\mathbf{Y}' \tag{16}$$

Finally, to implement the mapping $G(\tilde{\mathbf{y}})$, where $\tilde{\mathbf{y}} = \mathrm{vec}(\mathsf{D}\mathbf{Y})$, we do:

1. Reshape $\tilde{\mathbf{y}}$ to matrix to recover $\mathsf{D}\mathbf{Y}$.
2. Compute the partial derivatives, $\mathsf{D}\mathbf{X}$, using (16).
3. Output: $\mathrm{vec}(\mathsf{D}\mathbf{X})$.

5.1.3 Hessian product

To compute $H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$, we re-use the above-derived expression for $\mathsf{D}\mathbf{X}$, which already expresses the dependency on $\tilde{\mathbf{y}}$. If we let $\tilde{\mathbf{x}} = \mathrm{vec}(\tilde{\mathbf{X}})$, then $H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) = \mathbf{H}\,\mathrm{vec}(\tilde{\mathbf{X}})$, where $\mathbf{H}$ is the Hessian of the composition $g(\mathbf{X}'\mathbf{K}\mathbf{X})$. In this case we can find the Hessian product by exact evaluation of (4), as follows:

1. Reshape $\tilde{\mathbf{x}}$ to recover $\tilde{\mathbf{X}}$.
2. In (16), replace $\mathbf{X} \to \mathbf{X} + \alpha\tilde{\mathbf{X}}$, which makes $\mathsf{D}\mathbf{X}$ a function of the scalar α.
3. Differentiate, then set $\alpha = 0$: $\frac{\partial}{\partial\alpha}\mathsf{D}\mathbf{X}\big|_{\alpha=0} = \mathbf{K}'\tilde{\mathbf{X}}\mathsf{D}\mathbf{Y} + \mathbf{K}\tilde{\mathbf{X}}\mathsf{D}\mathbf{Y}'$.
4. Output: $\mathrm{vec}(\frac{\partial}{\partial\alpha}\mathsf{D}\mathbf{X}\big|_{\alpha=0})$.

[3]Our definition of D is *different* from that in Magnus and Neudecker, who use D to denote the *transpose* of the matrix of partial derivatives. We change the orientation to be more convenient to our purposes here.

[4]Note: $\mathrm{trace}(\mathsf{d}\mathbf{Y}'\mathsf{D}\mathbf{Y}) = \sum_{i,j} \mathsf{d}y_{ij}\frac{\partial g}{\partial y_{ij}}$.

[5]You can transpose or rotate the argument of the trace, without changing its value.

5.2 Example 2

We now modify the function from the previous example to instead return only the diagonal of the product matrix. We shall use the notation $\mathrm{diag}(\mathbf{X})$ to denote the function which maps a square matrix $\mathbf{X}$ to its the diagonal, represented as a column vector. This function is a linear transformation. The transpose (not inverse) of this linear transformation we denote as $\mathrm{Diag}(\mathbf{x})$ and it returns a square diagonal matrix, with $\mathbf{x}$ on the diagonal and zeros elsewhere.

We now redefine f to be a function which maps a matrix $\mathbf{X}$ to a vector $\mathbf{y}$, thus:

$$\mathbf{y} = \mathrm{diag}(\mathbf{X}'\mathbf{K}\mathbf{X}) \tag{17}$$

where $\mathbf{K}$ is a constant matrix of appropriate size. We define f in terms of the vectorized representation of the input matrix:

$$f : \mathrm{vec}(\mathbf{X}) \mapsto \mathrm{diag}(\mathbf{X}'\mathbf{K}\mathbf{X}) \tag{18}$$

Now we need to compute the above-defined mappings J, G and H:

5.2.1 Jacobian

To compute $J(\tilde{\mathbf{x}})$ for the modified function imposes no extra difficulty. Since diag() is linear it is also its own Jacobian transform, so that $\mathsf{d}\,\mathrm{diag}(\mathbf{Y}) = \mathrm{diag}(\mathsf{d}\mathbf{Y})$. This gives the recipe:

1. Reshape input to matrix: $\tilde{\mathbf{x}} \mapsto \mathsf{d}\mathbf{X}$
2. Compute differentials: $\mathsf{d}\mathbf{y} = \mathrm{diag}(\mathsf{d}(\mathbf{X}'\mathbf{K}\mathbf{X})) = \mathrm{diag}(\mathsf{d}\mathbf{X}'\mathbf{K}\mathbf{X}+\mathbf{X}'\mathbf{K}\mathsf{d}\mathbf{X})$
3. Output: $\mathsf{d}\mathbf{y}$

5.2.2 Gradient

To implement $G(\tilde{\mathbf{y}})$, the quickest route[6] is to use the fact that Diag() is the transpose of diag(). Now the chain rule requires to first apply Diag() and then proceed as before:

1. $\tilde{\mathbf{Y}} = \mathrm{Diag}(\tilde{\mathbf{y}})$
2. In (16), replace $\mathsf{D}\mathbf{Y} \to \tilde{\mathbf{Y}}$ to get: $\mathsf{D}\mathbf{X} = (\mathbf{K}' + \mathbf{K})\mathbf{X}\tilde{\mathbf{Y}}$.
3. Output: $\mathrm{vec}(\mathsf{D}\mathbf{X})$

[6] Alternatively, one can proceed again with the full recipe given in the previous example and use the fact that $\mathbf{y}'\,\mathrm{diag}(\mathbf{M}) = \mathrm{trace}(\mathrm{Diag}(\mathbf{y})\mathbf{M})$ and manipulate the trace as before.

5.2.3 Hessian product

To compute the Hessian, we differentiate the gradient in the same way as before. This gives:

$$H(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) = \text{vec}((\mathbf{K}' + \mathbf{K})\tilde{\mathbf{X}}\tilde{\mathbf{Y}}). \tag{19}$$

5.3 Example 3

Let

$$f : \text{vec}(\mathbf{X}) \mapsto \text{vec}(\mathbf{X}^{-1}) \tag{20}$$

5.3.1 Jacobian

If $\mathbf{Y} = \mathbf{X}^{-1}$, then

$$\mathsf{d}\mathbf{Y} = -\mathbf{X}^{-1}\mathsf{d}\mathbf{X}\mathbf{X}^{-1} \tag{21}$$

For some scalar-valued function $g(Y)$, let $\mathsf{D}\mathbf{Y} = [\frac{\partial g}{\partial y_{ij}}]$. Then

$$\begin{aligned} \mathsf{d}g(\mathbf{Y}) &= \text{trace}(\mathsf{d}\mathbf{Y}'\mathsf{D}\mathbf{Y}) \\ &= \text{trace}(-(\mathbf{X}^{-1})'\mathsf{d}\mathbf{X}'(\mathbf{X}^{-1})'\mathsf{D}\mathbf{Y}) \\ &= \text{trace}(-\mathsf{d}\mathbf{X}'(\mathbf{X}^{-1})'\mathsf{D}\mathbf{Y}(\mathbf{X}^{-1})') \end{aligned} \tag{22}$$