# Adjoint methods and sensitivity analysis for recurrence relations

Steven G. Johnson

Created October 2007, updated November 9, 2011.

## 1 Introduction

In this note, we derive an adjoint method for sensitivity analysis of the solution of recurrence relations. In particular, we suppose that we have a $M$-component vector $\mathbf{x}$ that is determined by iterating a recurrence relation

$$\mathbf{x}^n = \mathbf{f}(\mathbf{x}^{n-1}, \mathbf{p}, n) \triangleq \mathbf{f}^n$$

for some function $\mathbf{f}$ depending on the previous $\mathbf{x}$,[1] a vector $\mathbf{p}$ of $P$ parameters, and the step index $n$. The initial condition is

$$\mathbf{x}^0 = \mathbf{b}(\mathbf{p})$$

for some given function $\mathbf{b}$ of the parameters. Furthermore, we have some function $g$ of $\mathbf{x}$:

$$g^n \triangleq g(\mathbf{x}^n, \mathbf{p}, n)$$

and we wish to compute the gradient $\frac{dg^N}{d\mathbf{p}}$ of $g^N$, for some $N$, with respect to the parameters $\mathbf{p}$.

## 2 The explicit gradient

The gradient of $g^N$ can be written explicitly as:

$$\frac{dg^N}{d\mathbf{p}} = g^N_{\mathbf{p}} + g^N_{\mathbf{x}} \left( \mathbf{f}^N_{\mathbf{p}} + \mathbf{f}^N_{\mathbf{x}} \left[ \mathbf{f}^{N-1}_{\mathbf{p}} + \mathbf{f}^{N-1}_{\mathbf{x}} \left\{ \mathbf{f}^{N-2}_{\mathbf{p}} + \cdots \right\} \right] \right), \tag{1}$$

where subscripts denote partial derivatives, and should be thought of as row vectors, vs. column vectors $\mathbf{x}$ and $\mathbf{p}$. So, for example, $g^N_{\mathbf{x}}$ is a $1 \times M$ matrix, and $\mathbf{f}^N_{\mathbf{p}}$ is a $M \times P$ matrix. Equation (1) is derived simply by applying the chain rule to $g^N = g(\mathbf{x}^n, \mathbf{p}) = g(f(\mathbf{x}^{n-1}, \mathbf{p}), \mathbf{p}) = g(f(f(\mathbf{x}^{n-2}, \mathbf{p}), \mathbf{p}), \mathbf{p}) = \cdots$.

[1]Note that if $\mathbf{x}^n$ depends on $\mathbf{x}^{n-\ell}$ for $\ell = 1, \ldots, L$, the recurrence can still be cast in terms of $\mathbf{x}^{n-1}$ alone by expanding $\mathbf{x}$ into a vector of length $ML$, in much the same way that an $L$th-order ODE can be converted into $L$ 1st-order ODEs.

The natural way to evaluate eq. (1) might seem to be starting at the innermost parentheses and working outwards, but this is inefficient. Each parenthesized expression is a $M \times P$ matrix that must be multiplied by $\mathbf{f}_{\mathbf{x}}^{n}$, a $M \times M$ matrix, requiring $O(M^2 P)$ time for each multiplication assuming dense matrices. There are $O(N)$ such multiplications, so evaluating the whole expression in this fashion requires $O(NM^2 P)$ time. However, for dense matrices, the evaluation of $g^N$ itself requires $O(NM^2)$ time, which means that the gradient (calculated this way) is as expensive as $O(P)$ evaluations of $g^N$.

Similarly, evaluating gradients by finite-difference approximations or similar numerical tricks requires $O(P)$ evaluations of the function being differentiated (e.g. center-difference approximations require two function evaluations per dimension). So, direct evaluation of the gradient by the above technique, while it may be more accurate than numerical approximations, is not substantially more efficient. This is a problem if $P$ is large.

## 3 The gradient by adjoints

Instead of computing the gradient explicitly (by "forward" differentiation), *adjoint methods* typically allow one to compute gradients with the same cost as evaluating the function roughly twice, regardless of the number $P$ of parameters [1, 2, 3, 4]. A very general technique for constructing adjoint methods involves something similar to Lagrange multipliers, where one adds zero to $g^N$ in a way cleverly chosen to make computing the gradient easier, and in a previous note I derived the adjoint gradient for recurrence relations by this technique, analogous to work by Cao and Petzold on adjoint methods for differential-algebraic equations [2]. However, Gil Strang has pointed out to me that in many cases adjoint methods can be derived much more simply just by parenthesizing the gradient equation in a different way [4], and this turns out to be the case for the recurrence problem above.

The key fact is that, in the gradient equation (1), we are evaluating lots of expressions like $g_{\mathbf{x}}^{N}(\mathbf{f}_{\mathbf{x}}^{N}\mathbf{f}_{\mathbf{p}}^{N-1})$, $g_{\mathbf{x}}^{N}(\mathbf{f}_{\mathbf{x}}^{N}[\mathbf{f}_{\mathbf{x}}^{N-1}\mathbf{f}_{\mathbf{p}}^{N-2}])$, and so on. Parenthesized this way, these expressions require $O(M^2 P)$ operations each, because they involve matrix-matrix multiplications. However, we can parenthesize them a different way, so that they involve only vector-matrix multiplications, in order to reduce the complexity to $O(M^2 + MP)$, which is obviously a huge improvement for large $M$ and $P$. In particular, parenthesize them as $(g_{\mathbf{x}}^{N}\mathbf{f}_{\mathbf{x}}^{N})\mathbf{f}_{\mathbf{p}}^{N-1}$, $[(g_{\mathbf{x}}^{N}\mathbf{f}_{\mathbf{x}}^{N})\mathbf{f}_{\mathbf{x}}^{N-1}]\mathbf{f}_{\mathbf{p}}^{N-2}$ , and so on, involving repeated multiplication of a row vector on the left (starting with $g_{\mathbf{x}}^{N}$) by a matrix $\mathbf{f}_{\mathbf{x}}^{n}$ on the right. This repeated multiplication defines an *adjoint recurrence* relation for a $M$-component column vector $\lambda^n$, recurring backwards from $n = N$ to $n = 0$:

$$\lambda^{n-1} = (\mathbf{f}_{\mathbf{x}}^{n})^T \lambda^n,$$

where $T$ is the transpose, with "initial" condition

$$\lambda^N = \left(g_{\mathbf{x}}^{N}\right)^T.$$

In terms of this adjoint vector (so-called because of the transposes in the expressions above), the gradient becomes:

$$\frac{dg^N}{d\mathbf{p}} = g^N_{\mathbf{p}} + \sum_{n=1}^{N} (\lambda^n)^T \mathbf{f}^n_{\mathbf{p}} + \left(\lambda^0\right)^T \mathbf{b}_{\mathbf{p}}. \tag{2}$$

Consider the computational cost to evaluate the gradient in this way. Evaluating $g^N$ and the $\mathbf{x}^n$ costs $O(NM^2)$ time, assuming dense matrices, and evaluating $\lambda^n$ also takes $O(NM^2)$ time. Finally evaluating equation (2) takes $O(NMP)$ time in the worst case, dominated by the time to evaluate the summation assuming $\mathbf{f}^n_{\mathbf{p}}$ is a dense matrix. So, the total is $O(NM^2 + NMP)$, much better than $O(NM^2P)$ for large $M$ and $P$.

In practice, the situation is likely to be even better than this, because often $\mathbf{f}^n_{\mathbf{p}}$ will be a sparse matrix: each component of $\mathbf{p}$ will appear only for certain components of $\mathbf{x}$ and or for certain steps $n$. In this case the $O(NMP)$ cost will be greatly reduced, e.g. to $O(NM)$ or $O(MP)$ or similar. Then the cost of the gradient will be dominated by the two $O(NM^2)$ recurrences—i.e., as is characteristic of adjoint methods, the cost of finding the gradient will be comparable to the cost of finding the function value twice.

Note that there is, however, at least one drawback of the adjoint method (2) in comparison to the direct method (1): the adjoint method may require more storage. For the direct method, $O(M)$ storage is required for the current $\mathbf{x}^n$ (which can be discarded once $\mathbf{x}^{n+1}$ is computed) and $O(PM)$ storage is required for the $M \times P$ matrix being accumulated, to be multiplied by $g^N_{\mathbf{x}}$ at the end, for $O(PM)$ storage total. In the adjoint method, all of the $\mathbf{x}^n$ must be stored, because they are used in the backwards recurrence for $\lambda^n$ once $\mathbf{x}^N$ is reached, requiring $O(NM)$ storage. [The $\lambda^n$ vectors, on the other hand, can be discarded once $\lambda^{n-1}$ is computed, assuming the summation in eq. (2) is computed on the fly. Only $O(M)$ storage is needed for this summation, assuming $\mathbf{f}^n_{\mathbf{p}}$ can be computed on the fly (or is sparse).] Whether the $O(PM)$ storage for the direct method is better or worse than the $O(NM)$ storage for the adjoint method obviously depends on how $P$ compares to $N$.

## 4 A simple example

Finally, let us consider a simple example of a $M = 2$ linear recurrence:

$$\mathbf{x}^n = A\mathbf{x}^{n-1} + \begin{pmatrix} 0 \\ p_n \end{pmatrix}$$

with an initial condition

$$\mathbf{x}^0 = \mathbf{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and some $2 \times 2$ matrix $A$, e.g.

$$A = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

for $\theta = 0.1$. Here, $P = N$: there are $N$ parameters $p_n$, one per step $n$, acting as "source" terms in the recurrence (which otherwise has oscillating solutions since $A$ is unitary).

Let us also pick a simple function $g$ to differentiate, e.g.

$$g(\mathbf{x}) = (x_2)^2.$$

The adjoint recurrence for $\lambda^n$ is then:

$$\lambda^{n-1} = (\mathbf{f}_\mathbf{x}^n)^T \lambda^n = A^T \lambda^n,$$

with “initial” condition:

$$\lambda^N = (g_\mathbf{x}^N)^T = \begin{pmatrix} 0 \\ 2x_2^N \end{pmatrix}.$$

Notice that this case is rather simple: since our recurrence is linear, the adjoint recurrence does not depend on $\mathbf{x}^n$ except in the initial condition.

The gradient is also greatly simplified because $\mathbf{f}_\mathbf{p}^n$ is sparse: it is a $2 \times N$ matrix of zeros, except for the $n$-th column which is $(0,1)^T$. That means that the gradient (2) becomes:

$$\frac{dg^N}{dp_k} = \lambda_2^k,$$

requiring $O(N)$ work to find the whole gradient.

As a quick test, I implemented this example in GNU Octave (a Matlab clone) and checked it against the numerical center-difference gradient; it only takes a few minutes to implement and is worthwhile to try if you are not clear on how this works. For extra credit, try modifying the recurrence, e.g. to make $\mathbf{f}$ nonlinear in $\mathbf{x}$ and/or $\mathbf{p}$.

## References

[1] R. M. Errico, “What is an adjoint model?,” *Bulletin Am. Meteorological Soc.*, vol. 78, pp. 2577–2591, 1997.

[2] Y. Cao, S. Li, L. Petzold, and R. Serban, “Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution,” *SIAM J. Sci. Comput.*, vol. 24, no. 3, pp. 1076–1089, 2003.

[3] S. G. Johnson, “Notes on adjoint methods for 18.336.” Online at `http://math.mit.edu/ stevenj/18.336/adjoint.pdf`, October 2007.

[4] G. Strang, *Computational Science and Engineering*. Wellesley, MA: Wellesley-Cambridge Press, 2007.