Differentiation of the Cholesky decomposition

Iain Murray

February 2016

Abstract

We review strategies for differentiating matrix-based computations, and derive symbolic and algorithmic update rules for differentiating expressions containing the Cholesky decomposition. We recommend new 'blocked' algorithms, based on differentiating the Cholesky algorithm `DPOTRF` in the `LAPACK` library, which uses 'Level 3' matrix-matrix operations from `BLAS`, and so is cache-friendly and easy to parallelize. For large matrices, the resulting algorithms are the fastest way to compute Cholesky derivatives, and are an order of magnitude faster than the algorithms in common usage. In some computing environments, symbolically-derived updates are faster for small matrices than those based on differentiating Cholesky algorithms. The symbolic and algorithmic approaches can be combined to get the best of both worlds.

1 Introduction

The Cholesky decomposition L of a symmetric positive definite matrix Σ is the unique lower-triangular matrix with positive diagonal elements satisfying $\Sigma = LL^\top$. Alternatively, some library routines compute the upper-triangular decomposition $U = L^\top$. This note compares ways to differentiate the function $L(\Sigma)$, and larger expressions containing the Cholesky decomposition (Section 2). We consider compact symbolic results (Section 3) and longer algorithms (Section 4).

Existing computer code that differentiates expressions containing Cholesky decompositions often uses an algorithmic approach proposed by Smith (1995). This approach results from manually applying the ideas behind 'automatic differentiation' (e.g. Baydin et al., 2015) to a numerical algorithm for the Cholesky decomposition. Experiments by Walter (2011) suggested that — despite conventional wisdom — computing symbolically-derived results is actually faster. However, these experiments were based on differentiating slow algorithms for the Cholesky decomposition. In this note we introduce 'blocked' algorithms for propagating Cholesky derivatives (Section 4), which use cache-friendly and easy-to-parallelize matrix-matrix operations. In our implementations (Appendix A), these are faster than all previously-proposed methods.

2 Computational setup and tasks

This section can be safely skipped by readers familiar with "automatic differentiation", the $\dot{\Sigma}$ notation for "forward-mode sensitivities", and the $\bar{\Sigma}$ notation for "reverse-mode sensitivities" (e.g. Giles, 2008).

We consider a sequence of computations,

$$x \;\rightarrow\; \Sigma \;\rightarrow\; L \;\rightarrow\; f, \tag{1}$$

that starts with an input x, computes an intermediate symmetric positive-definite matrix Σ, its lower-triangular Cholesky decomposition L, and then a final result f. Derivatives of the overall computation $\frac{\partial f}{\partial x}$, can be decomposed into reusable parts with the chain rule. However, there are multiple ways to proceed, some much better than others.

Matrix chain rule: It's tempting to simply write down the chain rule for the overall procedure:

$$\frac{\partial f}{\partial x} = \sum_{i,j \le i} \sum_{k,l \le k} \frac{\partial f}{\partial L_{ij}} \frac{\partial L_{ij}}{\partial \Sigma_{kl}} \frac{\partial \Sigma_{kl}}{\partial x}, \tag{2}$$

where we only sum over the independent elements of symmetric matrix Σ and the occupied lower-triangle of L. We can also rewrite the same chain rule in matrix form,

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial \operatorname{vech}(L)} \frac{\partial \operatorname{vech}(L)}{\partial \operatorname{vech}(\Sigma)} \frac{\partial \operatorname{vech}(\Sigma)}{\partial x}, \tag{3}$$

where the vech operator creates a vector by stacking the lower-triangular columns of a matrix. A derivative $\frac{\partial y}{\partial z}$ is a matrix or vector, with a row for each element of y and a column for each element of z, giving a row vector if y is a scalar, and a column vector if z is a scalar.

The set of all partial derivatives $\left\{\frac{\partial L_{ij}}{\partial \Sigma_{kl}}\right\}$, or equivalently the matrix $\frac{\partial \operatorname{vech}(L)}{\partial \operatorname{vech}(\Sigma)}$, contains $O(N^4)$ values for the Cholesky decomposition of an $N \times N$ matrix. Explicitly computing each of the terms in equations (2) or (3) is inefficient, and simply not practical for large matrices.

We give expressions for these $O(N^4)$ derivatives at the end of Section 3 for completeness, and because they might be useful for analytical study. However, the computational primitives we really need are methods to accumulate the terms in the chain rule moving left (forwards) or right (backwards), without creating enormous matrices. We outline these processes now, adopting the 'automatic differentiation' notation used by Giles (2008) and others.

Forwards-mode accumulation: We start by computing a matrix of sensitivities for the first stage of the computation, with elements $\dot{\Sigma}_{kl} = \frac{\partial \Sigma_{kl}}{\partial x}$. If we applied an infinitesimal perturbation to the input $x \leftarrow x + \mathrm{d}x$, the intermediate matrix would be perturbed by $\mathrm{d}\Sigma = \dot{\Sigma}\mathrm{d}x$. This change would in turn perturb the output of the Cholesky decomposition by $\mathrm{d}L = \dot{L}\mathrm{d}x$, where $\dot{L}_{ij} = \frac{\partial L_{ij}}{\partial x}$. We would like to compute the sensitivities of the Cholesky decomposition, $\dot{L}$, from the sensitivities of the input matrix $\dot{\Sigma}$ and other 'local' quantities (L and/or Σ), without needing to consider where these came from. Finally, we would compute the required result $\dot{f} = \frac{\partial f}{\partial x}$ from L and $\dot{L}$, again without reference to downstream computations (the Cholesky decomposition).

The *forwards-mode* algorithms in this note describe how to compute the reusable function $\dot{L}(L, \dot{\Sigma})$, which propagates the effect of a perturbation forwards through the Cholesky decomposition. The computational cost will have the same scaling with matrix size as the Cholesky decomposition. However, if we want the derivatives with respect to D different inputs to the computation, we must perform the whole forwards propagation D times, each time accumulating sensitivities with respect to a different input x.

Reverse-mode accumulation: We can instead accumulate derivatives by starting at the other end of the computation sequence (1). The effect of perturbing the final stage of the computation is summarized by a matrix with elements $\bar{L}_{ij} = \frac{\partial f}{\partial L_{ij}}$. We need to 'back-propagate' this summary to compute the sensitivity of the output with respect to the downstream matrix, $\bar{\Sigma}_{kl} = \frac{\partial f}{\partial \Sigma_{kl}}$. In turn, this signal is back-propagated to compute $\bar{x} = \frac{\partial f}{\partial x}$, the target of our computation, equal to $\dot{f}$ in the forwards propagation above.

The *reverse-mode* algorithms in this note describe how to construct the reusable function $\bar{\Sigma}(L, \bar{L})$, which propagates the effect of a perturbation in the Cholesky decomposition backwards, to compute the effect of perturbing the original positive definite matrix. Like forwards-mode propagation, the computational cost has the same scaling with matrix size as the Cholesky decomposition. Reverse-mode differentiation or 'back-propagation' has the advantage that $\bar{\Sigma}$ can be reused to compute derivatives with respect to multiple inputs. Indeed if the input x to the sequence of computations (1) is a D-dimensional vector, the cost to obtain all D partial derivatives $\nabla_x f$ scales the same as a single forwards computation of f. For D-dimensional inputs, reverse-mode differentiation scales a factor of D times better than forwards-mode.

Reverse-mode computations can have greater memory requirements than forwards mode, and are less appealing than forwards-mode if there are more outputs of the computation than inputs.

3 Symbolic differentiation

It is not immediately obvious whether a small, neat symbolic form should exist for the derivatives of some function of a matrix, or whether the forward- and reverse-mode updates are simple to express. For the Cholesky decomposition, the literature primarily advises using algorithmic update rules, derived from the algorithms for numerically evaluating the original function (Smith, 1995; Giles, 2008). However, there are also fairly small algebraic expressions for the derivatives of the Cholesky decomposition, and for forwards- and reverse-mode updates.

Forwards-mode: Särkkä (2013) provides a short derivation of a forwards propagation rule (his Theorem A.1), which we adapt to the notation used here.

An infinitesimal perturbation to the expression $\Sigma = LL^\top$ gives:

$$\mathrm{d}\Sigma = \mathrm{d}L L^\top + L\mathrm{d}L^\top. \tag{4}$$

We wish to re-arrange to get an expression for $\mathrm{d}L$. The trick is to left-multiply by L^{-1} and right-multiply by $L^{-\top}$:

$$L^{-1}\mathrm{d}\Sigma L^{-\top} = L^{-1}\mathrm{d}L + \mathrm{d}L^\top L^{-\top}. \tag{5}$$

The first term on the right-hand side is now lower-triangular. The second term is the transpose of the first, meaning it is upper-triangular and has the same diagonal. We can therefore remove the second term by applying a function Φ to both sides, where Φ takes the lower-triangular part of a matrix and halves its diagonal:

$$\Phi(L^{-1}\mathrm{d}\Sigma L^{-\top}) = L^{-1}\mathrm{d}L, \qquad \text{where } \Phi_{ij}(A) = \begin{cases} A_{ij} & i > j \\ \frac{1}{2}A_{ii} & i = j \\ 0 & i < j. \end{cases} \tag{6}$$

Multiplying both sides by L gives us the perturbation of the Cholesky decomposition:

$$\mathrm{d}L = L\Phi(L^{-1}\mathrm{d}\Sigma L^{-\top}). \tag{7}$$

Substituting the forward-mode sensitivity relationships $\mathrm{d}\Sigma = \dot{\Sigma}\mathrm{d}x$ and $\mathrm{d}L = \dot{L}\mathrm{d}x$ (Section 2), immediately gives a forwards-mode update rule, which is easy to implement:

$$\boxed{\dot{L} = L\Phi(L^{-1}\dot{\Sigma}L^{-\top}).} \tag{8}$$

The input perturbation $\dot{\Sigma}$ must be a symmetric matrix, $\dot{\Sigma}_{kl} = \dot{\Sigma}_{lk} = \frac{\partial \Sigma_{kl}}{\partial x}$, because Σ is assumed to be symmetric for all inputs x.

Reverse-mode: We can also obtain a neat symbolic expression for the reverse mode updates. We substitute (7) into $\mathrm{d}f = \mathrm{Tr}(\bar{L}^\top \mathrm{d}L)$, and with a few lines of manipulation, rearrange it into the form $\mathrm{d}f = \mathrm{Tr}(S^\top \mathrm{d}\Sigma)$. Brewer (1977)'s Theorem 1 then implies that for a symmetric matrix Σ, the symmetric matrix containing reverse mode sensitivities will be:

$$\boxed{\bar{\Sigma} = S + S^\top - \mathrm{diag}(S), \qquad \text{where } S = L^{-\top}\Phi(L^\top \bar{L})L^{-1},} \tag{9}$$

where diag(S) is a diagonal matrix containing the diagonal elements of S, and function Φ is still as defined in (6).

Alternatively, a lower-triangular matrix containing the independent elements of $\bar{\Sigma}$ can be constructed as:

$$\boxed{\mathrm{tril}(\bar{\Sigma}) = \Phi(S + S^\top) = \Phi\big(L^{-\top}(P + P^\top)L^{-1}\big), \qquad \text{where } P = \Phi(L^\top \bar{L}),} \tag{10}$$

with S as in (9), and using function Φ again from (6).

Since first writing this section we have discovered two similar reverse-mode expressions (Walter, 2011; Koerber, 2015). It seems likely that other authors have also independently derived equivalent results, although these update rules do not appear to have seen wide-spread use.

Matrix of derivatives: By choosing the input of interest to be $x = \Sigma_{kl} = \Sigma_{lk}$, and fixing the other elements of Σ, the sensitivity $\dot{\Sigma}$ becomes a matrix of zeros except for ones at $\dot{\Sigma}_{kl} = \dot{\Sigma}_{lk} = 1$. Substituting into (8) gives an expression for all of the partial derivatives of the Cholesky decomposition with respect to any chosen element of the covariance matrix. Some further manipulation, expanding matrix products as sums over indices, gives an explicit expression for any element,

$$\frac{\partial L_{ij}}{\partial \Sigma_{kl}} = \left(\sum_{m>j} L_{im} L^{-1}_{mk} + \tfrac{1}{2} L_{ij} L^{-1}_{jk} \right) L^{-1}_{jl} + (1 - \delta_{kl}) \left(\sum_{m>j} L_{im} L^{-1}_{ml} + \tfrac{1}{2} L_{ij} L^{-1}_{jl} \right) L^{-1}_{jk}. \tag{11}$$

If we compute every (i, j, k, l) element, each one can be evaluated in constant time by keeping running totals of the sums in (11) as we decrement j from N to 1. Explicitly computing every partial derivative therefore costs $\Theta(N^4)$.

These derivatives can be arranged into a matrix, by 'vectorizing' the expression (Magnus and Neudecker, 2007; Minka, 2000; Harmeling, 2013). We use a well-known identity involving the vec operator, which stacks the columns of a matrix into a vector, and the Kronecker product $\otimes$:

$$\operatorname{vec}(ABC) = (C^\top \otimes A) \operatorname{vec}(B). \tag{12}$$

Applying this identity to (7) yields:

$$\operatorname{vec}(\mathrm{d}L) = (I \otimes L) \operatorname{vec}\left(\Phi\left(L^{-1} \mathrm{d}\Sigma L^{-\top} \right) \right). \tag{13}$$

We can remove the function Φ, by introducing a diagonal matrix Z defined such that $Z \operatorname{vec}(A) = \operatorname{vec} \Phi(A)$ for any $N \times N$ matrix A. Applying (12) again gives:

$$\operatorname{vec}(\mathrm{d}L) = (I \otimes L) Z (L^{-1} \otimes L^{-1}) \operatorname{vec}(\mathrm{d}\Sigma). \tag{14}$$

Using the standard *elimination matrix* $\mathcal{L}$, and *duplication matrix* D (Magnus and Neudecker, 1980), we can convert between the vec and vech of a matrix, where $\operatorname{vech}(A)$ is a vector made by stacking the columns of the lower triangle of A.

$$\operatorname{vech}(\mathrm{d}L) = \mathcal{L}(I \otimes L) Z (L^{-1} \otimes L^{-1}) D \operatorname{vech}(\mathrm{d}\Sigma) \quad \Rightarrow \quad \boxed{\frac{\partial \operatorname{vech} L}{\partial \operatorname{vech} \Sigma} = \mathcal{L}(I \otimes L) Z (L^{-1} \otimes L^{-1}) D.} \tag{15}$$

This compact-looking result was stated on *MathOverflow*[1] by pseudonymous user 'pete'. It may be useful for further analytical study, but doesn't immediately help with scalable computation.

4 Differentiating Cholesky algorithms

We have seen that it is inefficient to compute each term in the chain rule, (2) or (3), applied to a high-level matrix computation. For Cholesky derivatives the cost is $\Theta(N^4)$, compared to $O(N^3)$ for the forward- or reverse-mode updates in (8), (9), or (10). However, evaluating the terms of the chain rule applied to any *low-level* computation — expressed as a series of elementary scalar operations — gives derivatives with the same computational complexity as the original function (e.g. Baydin et al., 2015). Therefore $O(N^3)$ algorithms for the dense Cholesky decomposition can be mechanically converted into $O(N^3)$ forward- and reverse-mode update algorithms, which is called 'automatic differentiation'.

Smith (1995) proposed taking this automatic differentiation approach, although presented hand-derived propagation algorithms that could be easily implemented in any programming environment. Smith also reported applications to sparse matrices, where automatic differentiation inherits the improved complexity of computing the Cholesky decomposition. However, the

1. http://mathoverflow.net/questions/150427/the-derivative-of-the-cholesky-factor#comment450752_167719 — comment from 2014-09-01

algorithms that were considered for dense matrices aren't cache-friendly or easy to parallelize, and will be slow in practice.

Currently-popular numerical packages such as `NumPy`, `Octave`, and `R` (Oliphant, 2006; Eaton et al., 2009; R Core Team, 2012) compute the Cholesky decomposition using the `LAPACK` library (Anderson et al., 1999). `LAPACK` implements *block algorithms* that express computations as cache-friendly, parallelizable 'Level 3 `BLAS`' matrix-matrix operations that are fast on modern architectures. Dongarra et al. (1990) described the Level 3 `BLAS` operations, including an example block implementation of a Cholesky decomposition. For large matrices, we have sometimes found `LAPACK`'s routine to be 50× faster than a C or Fortran implementation of the Cholesky algorithm considered by Smith (1995). Precise timings are machine-dependent, however it's clear that any large dense matrix computations, including derivative computations, should be implemented using blocked algorithms where possible[2].

Block routines, like those in `LAPACK`, ultimately come down to elementary scalar operations inside calls to `BLAS` routines. In principle, automatic differentiation tools could be applied. However, the source code and compilation tools for the optimized `BLAS` routines for a particular machine are not always available to users. Even if they were, automatic differentiation tools would not necessarily create cache-friendly algorithms. For these reasons Walter (2011) used symbolic approaches (Section 3) to provide update rules based on standard matrix-matrix operations.

An alternative approach is to extend the set of elementary routines understood by an automatic differentiation procedure to the operations supported by `BLAS`. We could then pass derivatives through the Cholesky routine implemented by `LAPACK`, treating the best available matrix-matrix routines as black-box functions. Giles (2008) provides an excellent tutorial on deriving forward- and reverse-mode update rules for elementary matrix operations, which we found invaluable for deriving the algorithms that follow[3]. While his results can largely be found in materials already mentioned (Magnus and Neudecker, 2007; Minka, 2000; Harmeling, 2013), Giles emphasised forwards- and reverse-mode update rules, rather than huge objects like (15).

In the end, we didn't follow an automatic differentiation procedure exactly. While we derived derivative propagation rules from the structure of the Cholesky algorithms (unlike Section 3), we still symbolically manipulated some of the results to make the updates neater and in-place. In principle, a sophisticated optimizing compiler for automatic differentiation could do the same.

4.1 Level 2 routines

`LAPACK` also provides 'unblocked' routines, which use 'Level 2' `BLAS` operations (Dongarra et al., 1988a,b) like matrix-vector products. Although a step up from scalar-based algorithms, these are intended for small matrices only, and as helpers for 'Level 3' blocked routines (Section 4.2).

The `LAPACK` routine `DPOTF2` loops over columns of an input matrix A, replacing the lower-triangular part in-place with its Cholesky decomposition. At each iteration, the algorithm uses a row vector $\mathbf{r}$, a diagonal element d, a matrix B, and a column vector $\mathbf{c}$ as follows:

function **level2partition**(A, j)
$\quad \mathbf{r} = A_{j,1:j-1}$
$\quad d = A_{j,j}$
$\quad B = A_{j+1:N,1:j-1}$
$\quad \mathbf{c} = A_{j+1:N,j}$
$\quad$ **return** $\mathbf{r}$, d, B, $\mathbf{c}$

where $A = \begin{pmatrix} \ddots & & & \\ -\,\mathbf{r}\,- & d & & \\ & \vert & \ddots & \\ B & \mathbf{c} & & \ddots \\ & \vert & & \ddots \end{pmatrix}$

2. Historical note: It's entirely reasonable that Smith (1995) did not use blocked algorithms. Primarily, Smith's applications used sparse computations. In any case, blocked algorithms weren't universally adopted until later. For example, `Matlab` didn't incorporate `LAPACK` until 2000, `http://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html`.
3. Ironically, Giles (2008) also considered differentiating the Cholesky decomposition but, like Smith (1995), gave slow scalar-based algorithms.

Here '=' creates a *view* into the matrix A, meaning that in the algorithm below, '$\leftarrow$' assigns results into the corresponding part of matrix A.

function `chol_unblocked`(A)
    # *If at input* $\text{tril}(A) = \text{tril}(\Sigma) = \text{tril}(LL^\top)$*, at output* $\text{tril}(A) = L$.
    for $j = 1$ **to** N:
        $\mathbf{r},\ d,\ B,\ \mathbf{c}$ = `level2partition`$(A,\ j)$
        $d \leftarrow \sqrt{d - \mathbf{r}\mathbf{r}^\top}$
        $\mathbf{c} \leftarrow (\mathbf{c} - B\mathbf{r}^\top)/d$
    return A

The algorithm only inspects and updates the lower-triangular part of the matrix. If the upper-triangular part did not start out filled with zeros, then the user will need to zero out the upper triangle of the final array with the tril function:

$$\text{tril}(A)_{ij} = \begin{cases} A_{ij} & i \geq j \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

In each iteration, $\mathbf{r}$ and B are parts of the Cholesky decomposition that have already been computed, and d and $\mathbf{c}$ are updated in place, from their original settings in A to give another column of the Cholesky decomposition. The matrix-vector multiplication $B\mathbf{r}^\top$ is a Level 2 `BLAS` operation. These multiplications are the main computational cost of this algorithm.

Forwards-mode differentiation:

The in-place updates obscure the relationships between parts of the input matrix and its Cholesky decomposition. We could rewrite the updates more explicitly as

$$L_d = \sqrt{\Sigma_d - L_\mathbf{r} L_\mathbf{r}^\top}\,, \tag{17}$$

$$L_\mathbf{c} = (\Sigma_\mathbf{c} - L_B L_\mathbf{r}^\top)/L_d\,. \tag{18}$$

Applying infinitesimal perturbations to these equations gives

$$\begin{aligned} \mathrm{d}L_d &= \frac{1}{2}(\Sigma_d - L_\mathbf{r} L_\mathbf{r}^\top)^{-1/2}(\mathrm{d}\Sigma_d - 2\mathrm{d}L_\mathbf{r} L_\mathbf{r}^\top) \\ &= \frac{1}{L_d}(\mathrm{d}\Sigma_d/2 - \mathrm{d}L_\mathbf{r} L_\mathbf{r}^\top)\,, \end{aligned} \tag{19}$$

$$\begin{aligned} \mathrm{d}L_\mathbf{c} &= (\mathrm{d}\Sigma_\mathbf{c} - \mathrm{d}L_B L_\mathbf{r}^\top - L_B \mathrm{d}L_\mathbf{r}^\top)/L_d - ((\Sigma_\mathbf{c} - L_B L_\mathbf{r}^\top)/L_d^2)\mathrm{d}L_d \\ &= (\mathrm{d}\Sigma_\mathbf{c} - \mathrm{d}L_B L_\mathbf{r}^\top - L_B \mathrm{d}L_\mathbf{r}^\top - L_\mathbf{c}\mathrm{d}L_d)/L_d\,. \end{aligned} \tag{20}$$

We then get update rules for the forward-mode sensitivities by substituting their relationships, $\mathrm{d}\Sigma = \dot{\Sigma}\mathrm{d}x$ and $\mathrm{d}L = \dot{L}\mathrm{d}x$ (Section 2), into the equations above. Mirroring the original algorithm, we can thus convert $\dot{\Sigma}$ to $\dot{L}$ in-place, with the algorithm below:

function `chol_unblocked_fwd`$(L,\ \dot{A})$
    # *If at input* $\text{tril}(\dot{A}) = \text{tril}(\dot{\Sigma})$*, at output* $\text{tril}(\dot{A}) = \dot{L}$*, where* $\Sigma = LL^\top$.
    for $j = 1$ **to** N:
        $\mathbf{r},\ d,\ B,\ \mathbf{c}$ = `level2partition`$(L,\ j)$
        $\dot{\mathbf{r}},\ \dot{d},\ \dot{B},\ \dot{\mathbf{c}}$ = `level2partition`$(\dot{A},\ j)$
        $\dot{d} \leftarrow (\dot{d}/2 - \mathbf{r}\dot{\mathbf{r}}^\top)/d$
        $\dot{\mathbf{c}} \leftarrow (\dot{\mathbf{c}} - \dot{B}\mathbf{r}^\top - B\dot{\mathbf{r}}^\top - \mathbf{c}\dot{d})/d$
    return $\dot{A}$

Alternatively, the Cholesky decomposition and its forward sensitivity can be accumulated in one loop, by placing the updates from this algorithm after the corresponding lines in `chol_unblocked`.

Reverse-mode differentiation:

Reverse mode automatic differentiation traverses an algorithm backwards, reversing the direction of loops and the updates within them. At each step, the effect $\bar{Z}$ of perturbing an output $Z(A, B, C, \dots)$ is 'back-propagated' to compute the effects $(\bar{A}^{(Z)}, \bar{B}^{(Z)}, \bar{C}^{(Z)}, \dots)$ of perturbing the inputs to that step. If the effects of the perturbations are consistent then

$$\mathrm{Tr}(\bar{Z}^\top \mathrm{d}Z) = \mathrm{Tr}(\bar{A}^{(Z)\top} \mathrm{d}A) + \mathrm{Tr}(\bar{B}^{(Z)\top} \mathrm{d}B) + \mathrm{Tr}(\bar{C}^{(Z)\top} \mathrm{d}C) + \dots , \tag{21}$$

and we can find $(\bar{A}^{(Z)}, \bar{B}^{(Z)}, \bar{C}^{(Z)}, \dots)$ by comparing coefficients in this equation. If a quantity A is an input to multiple computations $(X, Y, Z, \dots)$, then we accumulate its total sensitivity,

$$\bar{A} = \bar{A}^{(X)} + \bar{A}^{(Y)} + \bar{A}^{(Z)} + \dots , \tag{22}$$

summarizing the quantity's effect on the final computation, $\bar{A}_{ij} = \frac{\partial f}{\partial A_{ij}}$ (as reviewed in Section 2).

Using the standard identities $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$, $\mathrm{Tr}(A^\top) = \mathrm{Tr}(A)$, and $(AB)^\top = B^\top A^\top$, the perturbations from the final line of the Cholesky algorithm (20) imply:

$$\begin{aligned}\mathrm{Tr}(\bar{L}_{\mathbf{c}}^\top \mathrm{d}L_{\mathbf{c}}) = \mathrm{Tr}((\bar{L}_{\mathbf{c}}/L_d)^\top \mathrm{d}\Sigma_{\mathbf{c}}) - \mathrm{Tr}((\bar{L}_{\mathbf{c}} L_{\mathbf{r}}/L_d)^\top \mathrm{d}L_B) \\ - \mathrm{Tr}((\bar{L}_{\mathbf{c}}^\top L_B/L_d)^\top \mathrm{d}L_{\mathbf{r}}) - \mathrm{Tr}((L_{\mathbf{c}}^\top \bar{L}_{\mathbf{c}}/L_d)^\top \mathrm{d}L_d) .\end{aligned} \tag{23}$$

We thus read off that $\bar{\Sigma}_{\mathbf{c}} = \bar{L}_{\mathbf{c}}/L_d$, where the sensitivities $\bar{L}_{\mathbf{c}}$ include the direct effect on f, provided by the user of the routine, and the knock-on effects that changing this column would have on the columns computed to the right. These knock-on effects should have been accumulated through previous iterations of the reverse propagation algorithm. From this equation, we can also identify the knock-on effects that changing L_d, $L_{\mathbf{r}}$, and L_B would have through changing column $\mathbf{c}$, which should be added on to their existing sensitivities for later.

The perturbation (19) to the other update in the Cholesky algorithm implies:

$$\mathrm{Tr}(\bar{L}_d^\top \mathrm{d}L_d) = \mathrm{Tr}((\bar{L}_d/(2L_d))^\top \mathrm{d}\Sigma_d) - \mathrm{Tr}((\bar{L}_d L_{\mathbf{r}}/L_d)^\top \mathrm{d}L_{\mathbf{r}}) . \tag{24}$$

Comparing coefficients again, we obtain another output of the reverse-mode algorithm, $\bar{\Sigma}_d = \bar{L}_d/(2L_d)$. We also add $\bar{L}_d L_{\mathbf{r}}/L_d$ to the running total for the sensitivity of $L_{\mathbf{r}}$ for later updates.

The algorithm below tracks all of these sensitivities, with the updates rearranged to simplify some expressions and to make an algorithm that can update the sensitivities in-place.

function chol_unblocked_rev(L, $\bar{A}$)
    # *If at input* $\mathrm{tril}(\bar{A}) = \bar{L}$, *at output* $\mathrm{tril}(\bar{A}) = \mathrm{tril}(\bar{\Sigma})$, *where* $\Sigma = LL^\top$.
    for $j = N$ **to** 1, **in steps of** -1:
        $\mathbf{r}$, d, B, $\mathbf{c}$ = level2partition(L, j)
        $\bar{\mathbf{r}}$, $\bar{d}$, $\bar{B}$, $\bar{\mathbf{c}}$ = level2partition($\bar{A}$, j)
        $\bar{d} \leftarrow \bar{d} - \mathbf{c}^\top \bar{\mathbf{c}}/d$
        $\begin{bmatrix} \bar{d} \\ \bar{\mathbf{c}} \end{bmatrix} \leftarrow \begin{bmatrix} \bar{d} \\ \bar{\mathbf{c}} \end{bmatrix}/d$
        $\bar{\mathbf{r}} \leftarrow \bar{\mathbf{r}} - \begin{bmatrix} \bar{d} & \bar{\mathbf{c}}^\top \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ B \end{bmatrix}$
        $\bar{B} \leftarrow \bar{B} - \bar{\mathbf{c}}\mathbf{r}$
        $\bar{d} \leftarrow \bar{d}/2$
    return $\bar{A}$

4.2 Level 3 routines

The LAPACK routine DPOTRF also updates the lower-triangular part of an array A in place with its Cholesky decomposition. However, this routine updates blocks at a time, rather than single column vectors, using the following partitions:

function `level3partition`$(A,\ j,\ k)$

$R = A_{j:k,\,1:j-1}$

$D = A_{j:k,\,j:k}$

$B = A_{k+1:N,\,1:j-1}$

$C = A_{k+1:N,\,j:k}$

return $R,\ D,\ B,\ C$

$$\text{where } A = \begin{pmatrix} \ddots & & \\ R & D & \\ B & C & \ddots \end{pmatrix}$$

Only the lower-triangular part of D, the matrix on the diagonal, is referenced. The algorithm below loops over each diagonal block D, updating it and the matrix C below it. Each diagonal block (except possibly the last) is of size $N_b \times N_b$. The optimal block-size N_b depends on the size of the matrix N, and the machine running the code. Implementations of `LAPACK` select the block-size with a routine called `ILAENV`.

function `chol_blocked`$(A,\ N_b)$

\# *If at input* $\text{tril}(A) = \text{tril}(\Sigma) = \text{tril}(LL^\top)$*, at output* $\text{tril}(A) = L$*, for integer* $N_b \geq 1$.

for $j = 1$ **to at most** N **in steps of** N_b:

$k \leftarrow$ `min`$(N,\ j+N_b-1)$

$R,\ D,\ B,\ C =$ `level3partition`$(A,\ j,\ k)$

$D \leftarrow D - \text{tril}(RR^\top)$

$D \leftarrow$ `chol_unblocked`(D)

$C \leftarrow C - BR^\top$

$C \leftarrow C\ \text{tril}(D)^{-\top}$

return A

The computational cost of the blocked algorithm is dominated by Level 3 `BLAS` operations for the matrix-matrix multiplies and for solving a triangular system. The unblocked Level 2 routine from Section 4.1 (`DPOTF2` in `LAPACK`) is also called as a subroutine on a small triangular block. For large matrices it may be worth replacing this unblocked routine with one that performs more Level 3 operations (Gustavson et al., 2013).

Forwards-mode differentiation:

Following the same strategy as for the unblocked case, we obtained the algorithm below. As before, the input sensitivities $\dot{\Sigma}_{ij} = \frac{\partial \Sigma_{ij}}{\partial x}$ can be updated in-place to give $\dot{L}_{ij} = \frac{\partial L_{ij}}{\partial x}$, the sensitivities of the resulting Cholesky decomposition. Again, these updates could be accumulated at the same time as computing the original Cholesky decomposition.

function `chol_blocked_fwd`$(L,\ \dot{A})$

\# *If at input* $\text{tril}(\dot{A}) = \text{tril}(\dot{\Sigma})$*, at output* $\text{tril}(\dot{A}) = \text{tril}(\dot{L})$*, where* $\Sigma = LL^\top$.

for $j = 1$ **to at most** N **in steps of** N_b:

$k \leftarrow$ `min`$(N,\ j+N_b-1)$

$R,\ D,\ B,\ C =$ `level3partition`$(L,\ j,\ k)$

$\dot{R},\ \dot{D},\ \dot{B},\ \dot{C} =$ `level3partition`$(\dot{A},\ j,\ k)$

$\dot{D} \leftarrow \dot{D} - \text{tril}(\dot{R}R^\top + R\dot{R}^\top)$

$\dot{D} \leftarrow$ `chol_unblocked_fwd`$(D,\ \dot{D})$

$\dot{C} \leftarrow \dot{C} - \dot{B}R^\top - B\dot{R}^\top$

$\dot{C} \leftarrow (\dot{C} - C\dot{D}^\top)\, D^{-\top}$

return $\dot{A}$

The unblocked derivative routine is called as a subroutine. Alternatively, `chol_blocked_fwd` could call itself recursively with a smaller block size, we could use the symbolic result (8), or we could differentiate other algorithms (e.g. Gustavson et al., 2013).

Minor detail: The standard `BLAS` operations don't provide a routine to neatly perform the first update for the lower-triangular $\dot{D}$. One option is to wastefully subtract the full matrix

$(\dot{R}R^\top + R\dot{R}^\top)$, then zero out the upper-triangle of $\dot{D}$, meaning that the upper triangle of $\dot{A}$ can't be used for auxiliary storage.

Reverse-mode differentiation:

Again, deriving the reverse-mode algorithm and arranging it into a convenient form was more involved. The strategy is the same as the unblocked case however, and still relatively mechanical.

```
function chol_blocked_rev(L, Ā)
    # If at input tril(Ā) = L̄, at output tril(Ā) = tril(Σ̄), where Σ = LL^⊤.
    for k = N to no less than 1 in steps of −N_b:
        j ← max(1, k−N_b+1)
        R, D, B, C = level3partition(L, j, k)
        R̄, D̄, B̄, C̄ = level3partition(Ā, j, k)
        C̄ ← C̄D^{−1}
        B̄ ← B̄ − C̄R
        D̄ ← D̄ − tril(C̄^⊤C)
        D̄ ← chol_unblocked_rev(D, D̄)
        R̄ ← R̄ − C̄^⊤B − (D̄ + D̄^⊤)R
    return Ā
```

The partitioning into columns is arbitrary, so the reverse-mode algorithm doesn't need to select the same set of blocks as the forwards computation. Here, when the matrix size N isn't a multiple of the block-size N_b, we've put the smaller blocks at the other edge of the matrix.

As in the blocked forwards-mode update, there is a call to the unblocked routine, which can be replaced with alternative algorithms. In the implementation provided (Appendix A) we use the symbolically-derived update (10).

5 Discussion and Future Directions

The matrix operations required by the Cholesky algorithms implemented in LAPACK can be implemented with straightforward calls to BLAS. However, the forwards- and reverse-mode updates we have derived from these algorithms give some expressions where only the triangular part of a matrix product is required. There aren't standard BLAS routines that implement exactly what is required, and our implementations must perform unnecessary computations to exploit the fast libraries available. In future, it would be desirable to have standard fast matrix libraries that offer a set of routines that are closed under the rules for deriving derivative updates.

The automatic differentiation tools that have proved popular in machine learning differentiate high-level array-based code. As a result, these tools don't have access to the source code of the Cholesky decomposition, and need to be told how to differentiate it. Theano (Bastien et al., 2012; Bergstra et al., 2010), the first tool to be widely-adopted in machine learning, and AutoGrad (Maclaurin et al., 2015) use the algorithm by Smith (1995). TensorFlow (Abadi et al., 2015) in its first release can't differentiate expressions containing a Cholesky decomposition, but a fork (Hensman and de G. Matthews, 2016) also uses the algorithm by Smith (1995), as previously implemented by The GPy authors (2015).

The approaches in this note will be an order of magnitude faster for large matrices than the codes that are in current wide-spread use. Some illustrative timings are given at the end of the code listing (Appendix A). As the algorithms are only a few lines long, they could be ported to a variety of settings without introducing any large dependencies. The simple symbolic expressions (Section 3) could be differentiated using most existing matrix-based tools. Currently AutoGrad can't repeatedly differentiate the Cholesky decomposition because of the in-place updates in the (Smith, 1995) algorithm.

The 'Level 3' blocked algorithms (Section 4.2) are the fastest forwards- and reverse-mode update rules for large matrices. However, these require helper routines to perform the updates on small triangular blocks. In high-level languages (`Matlab`, `Octave`, `Python`), the 'Level 2' routines — similar to the algorithms that automatic differentiation would provide — are slow, and we recommend using the symbolic updates (Section 3) for the small matrices instead.

It should be relatively easy to provide similar derivative routines for many standard matrix functions, starting with the rest of the routines in `LAPACK`. However, it would save a lot of work to have automatic tools to help make these routines. Although there are a wide-variety of tools for automatic differentiation, we are unaware of practical tools that can currently create algorithms as neat and accessible as those made by hand for this note.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. White paper, Google Research, 2015. Software available from `http://tensorflow.org`. TensorFlow is a trademark of Google Inc.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' guide*, volume 9. SIAM, 1999. `http://www.netlib.org/lapack/`.

F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey, 2015.

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

J. W. Brewer. The gradient with respect to a symmetric matrix. *IEEE Transactions on Automatic Control*, 22(2):265–267, 1977.

J. J. Dongarra, J. Ducroz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988a.

J. J. Dongarra, J. Ducroz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1988b.

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

J. W. Eaton, D. Bateman, and S. Hauberg. *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2009. URL `http://www.gnu.org/software/octave/doc/interpreter`. ISBN 1441413006.

M. B. Giles. An extended collection of matrix derivative results for forward and reverse mode automatic differentiation, 2008.

F. G. Gustavson, J. Waśniewski, J. J. Dongarra, J. R. Herrero, and J. Langou. Level-3 Cholesky factorization routines improve performance of many Cholesky algorithms. *ACM Transactions on Mathematical Software*, 39(2):9:1–9:10, 2013.

S. Harmeling. Matrix differential calculus cheat sheet. Technical Report Blue Note 142, Max Planck Institute for Intelligent Systems, 2013. `http://people.tuebingen.mpg.de/harmeling/bn142.pdf`.

J. Hensman and A. G. de G. Matthews. GPFlow, 2016. As of February 2016, https://github.com/GPflow/GPflow.

P. Koerber. Adjoint algorithmic differentiation and the derivative of the Cholesky decomposition, 2015. Preprint, available at SSRN: http://dx.doi.org/10.2139/ssrn.2703893.

D. Maclaurin, D. Duvenaud, M. Johnson, and R. P. Adams. Autograd: Reverse-mode differentiation of native Python, 2015. Version 1.1.3, http://github.com/HIPS/autograd and https://pypi.python.org/pypi/autograd/.

J. R. Magnus and H. Neudecker. The elimination matrix: some lemmas and applications. *SIAM Journal on Algebraic and Discrete Methods*, 1(4):422–449, 1980.

J. R. Magnus and H. Neudecker. *Matrix differential calculus with application in statistics and econometrics*. 3rd edition, 2007. Available from http://www.janmagnus.nl/misc/mdc2007-3rdedition and older editions from Wiley.

T. Minka. Old and new matrix algebra useful for statistics, 2000. MIT Media Lab note (1997; revised 12/00), http://research.microsoft.com/en-us/um/people/minka/papers/matrix/.

T. E. Oliphant. *Guide to NumPy*. Provo, UT, 2006.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL http://www.R-project.org/. ISBN 3-900051-07-0.

S. Särkkä. *Bayesian filtering and smoothing*. Cambridge University Press., 2013.

S. P. Smith. Differentiation of the Cholesky algorithm. *Journal of Computational and Graphical Statistics*, 4 (2):134–147, 1995.

The GPy authors. GPy: A Gaussian process framework in Python, 2015. Version 0.8.8, http://github.com/SheffieldML/GPy.

S. F. Walter. *Structured higher-order algorithmic differentiation in the forward and reverse mode with application in optimum experimental design*. PhD thesis, Humboldt-Universität zu Berlin, 2011.

A Illustrative Python code

Equations (11) and (15) were checked numerically using `Octave/Matlab` code, not provided here.

The rest of the equations and algorithms in this note are illustrated below using `Python` code that closely follows the equations and pseudo-code. There are differences due to the note using `Matlab`/Fortran-style ranges, which are one-based and inclusive, e.g. $1{:}3 = [1,2,3]$. In contrast, `Python` uses zero-based, half-open ranges, e.g. $0{:}3 = {:}3 = [0,1,2]$. The code is also available as `pseudocode_port.py` in the source tar-ball for this paper, available from arXiv.

Development of alternative implementations in multiple programming languages is on-going. At the time of writing, Fortran code with `Matlab/Octave` and `Python` bindings, and pure `Matlab` code is available at `https://github.com/imurray/chol-rev`. The Fortran code is mainly useful for smaller matrices, as for large matrices, the time spent inside `BLAS` routines dominates, regardless of the language used. The code repository also contains a demonstration of pushing derivatives through a whole computation (the log-likelihood of the hyperparameters of a Gaussian process).

```
# Demonstration code for Cholesky differentiation
# Iain Murray, February 2016

# These routines need Python >=3.5 and NumPy >= 1.10 for matrix
# multiplication with the infix operator "@". For earlier Python/NumPy,
# replace all uses of "@" with the np.dot() function.

# Tested with Python 3.5.0, NumPy 1.10.4, and SciPy 0.17.0, with MKL
# from Anaconda's distribution with a quad core i5-3470 CPU @ 3.20GHz.

import numpy as np
from numpy import tril
from scipy.linalg import solve_triangular as _solve_triangular

# There are operations that are not performed in place but could be by
# splitting up the operations, and/or using lower-level code. In this
# version of the code, I've instead tried to keep the Python syntax
# close to the illustrative pseudo-code.

# Where the pseudo code contains inverses of triangular matrices, it's
# commonly understood that the matrix product of the inverse with the
# adjacent term should be found by solving the resulting linear system of
# equations. The code below contains some commented-out lines with a
# straightforward rewriting of the pseudocode using inv(), followed by
# equivalent but more efficient lines calling a linear solver (_st()
# defined below). To get an inv function for testing we could do:
#from numpy.linalg import inv
# I'd have to call the underlying LAPACK routines myself to solve these
# systems in-place, as the matrix tranposes haven't worked out to match
# what the SciPy routine can do in-place.

def _st(A, b, trans=0):
    """
    solve triangular system "tril(A) @ x = b", returning x

    if trans==1, solve "tril(A).T @ x = b" instead.
    """
    if b.size == 0:
        return b
    else:
        return _solve_triangular(A, b, trans=trans, lower=True)
```

```
def Phi(A):
    """Return lower-triangle of matrix and halve the diagonal"""
    A = tril(A)
    A[np.diag_indices_from(A)] *= 0.5
    return A

def chol_symbolic_fwd(L, Sigma_dot):
    """
    Forwards-mode differentiation through the Cholesky decomposition

    This version uses a "one-line" symbolic expression to return L_dot
    where "_dot" means sensitivities in forwards-mode differentiation,
    and Sigma = L @ L.T.
    """
    # invL = inv(L)
    # return L @ Phi(invL @ Sigma_dot @ invL.T)
    return L @ Phi(_st(L, _st(L, Sigma_dot.T).T))

def chol_symbolic_rev(L, Lbar):
    """
    Reverse-mode differentiation through the Cholesky decomposition

    This version uses a short symbolic expression to return
    tril(Sigma_bar) where "_bar" means sensitivities in reverse-mode
    differentiation, and Sigma = L @ L.T.
    """
    P = Phi(L.T @ Lbar)
    #invL = inv(L)
    #return Phi(invL.T @ (P + P.T) @ invL)
    return Phi(_st(L, _st(L, (P + P.T), 1).T, 1))

def level2partition(A, j):
    """Return views into A used by the unblocked algorithms"""
    # diagonal element d is A[j,j]
    # we access [j, j:j+1] to get a view instead of a copy.
    rr = A[j, :j]      # row
    dd = A[j, j:j+1]   # scalar on diagonal   /      \
    B = A[j+1:, :j]    # Block in corner      | r d  |
    cc = A[j+1:, j]    # column               \ B c  /
    return rr, dd, B, cc

def chol_unblocked(A, inplace=False):
    """
    Cholesky decomposition, mirroring LAPACK's DPOTF2

    Intended to illustrate the algorithm only. Use a Cholesky routine
    from numpy or scipy instead.
    """
    if not inplace:
        A = A.copy()
    for j in range(A.shape[0]):
        rr, dd, B, cc = level2partition(A, j)
        dd[:] = np.sqrt(dd - rr@rr)
        cc[:] = (cc - B@rr) / dd
    return A
```

```
def chol_unblocked_fwd(L, Adot, inplace=False):
    """
    Forwards-mode differentiation through the Cholesky decomposition

    Obtain L_dot from Sigma_dot, where "_dot" means sensitivities in
    forwards-mode differentiation, and Sigma = L @ L.T.

    This version uses an unblocked algorithm to update sensitivities
    Adot in place. tril(Adot) should start containing Sigma_dot, and
    will end containing the L_dot. The upper triangular part of Adot
    is untouched, so take tril(Adot) at the end if triu(Adot,1) did
    not start out filled with zeros.

    If inplace=False, a copy of Adot is modified instead of the
    original. The Abar that was modified is returned.
    """
    if not inplace:
        Adot = Adot.copy()
    for j in range(L.shape[0]):
        rr, dd, B, cc = level2partition(L, j)
        rdot, ddot, Bdot, cdot = level2partition(Adot, j)
        ddot[:] = (ddot/2 - rr@rdot) / dd
        cdot[:] = (cdot - Bdot@rr - B@rdot - cc*ddot) / dd
    return Adot

def chol_unblocked_rev(L, Abar, inplace=False):
    """
    Reverse-mode differentiation through the Cholesky decomposition

    Obtain tril(Sigma_bar) from L_bar, where "_bar" means sensitivities
    in reverse-mode differentiation, and Sigma = L @ L.T.

    This version uses an unblocked algorithm to update sensitivities
    Abar in place. tril(Abar) should start containing L_bar, and will
    end containing the tril(Sigma_bar). The upper triangular part of
    Adot is untouched, so take tril(Abar) at the end if triu(Abar,1)
    did not start out filled with zeros. Alternatively, (tril(Abar) +
    tril(Abar).T) will give the symmetric, redundant matrix of
    sensitivities.

    If inplace=False, a copy of Abar is modified instead of the
    original. The Abar that was modified is returned.
    """
    if not inplace:
        Abar = Abar.copy()
    for j in range(L.shape[0] - 1, -1, -1): # N-1,N-2,...,1,0
        rr, dd, B, cc = level2partition(L, j)
        rbar, dbar, Bbar, cbar = level2partition(Abar, j)
        dbar -= cc @ cbar / dd
        dbar /= dd # / These two lines could be
        cbar /= dd # \ done in one operation
        rbar -= dbar*rr  # / These two lines could be done
        rbar -= cbar @ B # \ with one matrix multiply
        Bbar -= cbar[:,None] @ rr[None,:]
        dbar /= 2
    return Abar
```

```
def level3partition(A, j, k):
    """Return views into A used by the blocked algorithms"""
    # Top left corner of diagonal block is [j,j]
    # Block size is NB = (k-j)
    R = A[j:k, :j]     # Row block                      /      \
    D = A[j:k, j:k]    # triangular block on Diagonal   |      |
    B = A[k:, :j]      # Big corner block               | R D  |
    C = A[k:, j:k]     # Column block                   \ B C  /
    return R, D, B, C

def chol_blocked(A, NB=256, inplace=False):
    """Cholesky decomposition, mirroring LAPACK's DPOTRF

    Intended to illustrate the algorithm only. Use a Cholesky routine
    from numpy or scipy instead."""
    if not inplace:
        A = A.copy()
    for j in range(0, A.shape[0], NB):
        k = min(N, j + NB)
        R, D, B, C = level3partition(A, j, k)
        D -= tril(R @ R.T)
        chol_unblocked(D, inplace=True)
        C -= B @ R.T
        #C[:] = C @ inv(tril(D)).T
        C[:] = _st(D, C.T).T
    return A

def chol_blocked_fwd(L, Adot, NB=256, inplace=False):
    """
    Forwards-mode differentiation through the Cholesky decomposition

    Obtain L_dot from Sigma_dot, where "_dot" means sensitivities in
    forwards-mode differentiation, and Sigma = L @ L.T.

    This version uses a blocked algorithm to update sensitivities Adot
    in place. tril(Adot) should start containing Sigma_dot, and will
    end containing the L_dot. Take tril() of the answer if
    triu(Adot,1) did not start out filled with zeros. Unlike the
    unblocked routine, if the upper triangular part of Adot started
    with non-zero values, some of these will be overwritten.

    If inplace=False, a copy of Adot is modified instead of the
    original. The Abar that was modified is returned.
    """
    if not inplace:
        Adot = Adot.copy()
    for j in range(0, L.shape[0], NB):
        k = min(N, j + NB)
        R, D, B, C = level3partition(L, j, k)
        Rdot, Ddot, Bdot, Cdot = level3partition(Adot, j, k)
        Ddot[:] = tril(Ddot) - tril(Rdot @ R.T + R@Rdot.T)
        #chol_unblocked_fwd(D, Ddot, inplace=True) # slow in Python
        Ddot[:] = chol_symbolic_fwd(D, Ddot + tril(Ddot, -1).T)
        Cdot -= (Bdot@R.T + B@Rdot.T)
        #Cdot[:] = (Cdot - C@Ddot.T) @ inv(tril(D)).T
        Cdot[:] = _st(D, Cdot.T - Ddot@C.T).T
    return Adot
```

```
214 def chol_blocked_rev(L, Abar, NB=256, inplace=False):
215     """
216     Reverse-mode differentiation through the Cholesky decomposition
217 
218     Obtain tril(Sigma_bar) from L_bar, where "_bar" means sensitivities
219     in reverse-mode differentiation, and Sigma = L @ L.T.
220 
221     This version uses a blocked algorithm to update sensitivities Abar
222     in place. tril(Abar) should start containing L_bar, and will end
223     containing the tril(Sigma_bar). Take tril(Abar) at the end if
224     triu(Abar,1) did not start out filled with zeros. Alternatively,
225     (tril(Abar) + tril(Abar).T) will give the symmetric, redundant
226     matrix of sensitivities.
227 
228     Unlike the unblocked routine, if the upper triangular part of Abar
229     started with non-zero values, some of these will be overwritten.
230 
231     If inplace=False, a copy of Abar is modified instead of the
232     original. The Abar that was modified is returned.
233     """
234     if not inplace:
235         Abar = Abar.copy()
236     for k in range(L.shape[0], -1, -NB):
237         j = max(0, k - NB)
238         R, D, B, C = level3partition(L, j, k)
239         Rbar, Dbar, Bbar, Cbar = level3partition(Abar, j, k)
240         #Cbar[:] = Cbar @ inv(tril(D))
241         Cbar[:] = _st(D, Cbar.T, trans=1).T
242         Bbar -= Cbar @ R
243         Dbar[:] = tril(Dbar) - tril(Cbar.T @ C)
244         #chol_unblocked_rev(D, Dbar, inplace=True) # slow in Python
245         Dbar[:] = chol_symbolic_rev(D, Dbar)
246         Rbar -= (Cbar.T @ B + (Dbar + Dbar.T) @ R)
247     return Abar
248 
249 # Testing code follows
250 
251 def _trace_dot(A, B):
252     """_trace_dot(A, B) = trace(A @ B) = A.ravel() @ B. ravel()"""
253     return A.ravel() @ B.ravel()
254 
255 def _testme(N):
256     """Exercise each function using NxN matrices"""
257     import scipy as sp
258     from time import time
259     if N > 1:
260         Sigma = np.cov(sp.randn(N, 2*N))
261         Sigma_dot = np.cov(sp.randn(N, 2*N))
262     elif N == 1:
263         Sigma = np.array([[sp.rand()]])
264         Sigma_dot = np.array([[sp.rand()]])
265     else:
266         assert(False)
267     tic = time()
268     L = np.linalg.cholesky(Sigma)
269     toc = time() - tic
270     print('Running np.linalg.cholesky:')
```

```
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    L_ub = tril(chol_unblocked(Sigma))
    toc = time() - tic
    print('Unblocked chol works: %r'
            % np.all(np.isclose(L, L_ub)))
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    L_bl = tril(chol_blocked(Sigma))
    toc = time() - tic
    print('Blocked chol works: %r'
            % np.all(np.isclose(L, L_bl)))
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    Ldot = chol_symbolic_fwd(L, Sigma_dot)
    toc = time() - tic
    hh = 1e-5
    L2 = np.linalg.cholesky(Sigma + Sigma_dot*hh/2)
    L1 = np.linalg.cholesky(Sigma - Sigma_dot*hh/2)
    Ldot_fd = (L2 - L1) / hh
    print('Symbolic chol_fwd works: %r'
            % np.all(np.isclose(Ldot, Ldot_fd)))
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    Ldot_ub = tril(chol_unblocked_fwd(L, Sigma_dot))
    toc = time() - tic
    print('Unblocked chol_fwd works: %r'
            % np.all(np.isclose(Ldot, Ldot_ub)))
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    Ldot_bl = tril(chol_blocked_fwd(L, Sigma_dot))
    toc = time() - tic
    print('Blocked chol_fwd works: %r'
            % np.all(np.isclose(Ldot, Ldot_bl)))
    print('   Time taken: %0.4f s' % toc)
    Lbar = tril(sp.randn(N, N))
    tic = time()
    Sigma_bar = chol_symbolic_rev(L, Lbar)
    toc = time() - tic
    Delta1 = _trace_dot(Lbar, Ldot)
    Delta2 = _trace_dot(Sigma_bar, Sigma_dot)
    print('Symbolic chol_rev works: %r'
            % np.all(np.isclose(Delta1, Delta2)))
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    Sigma_bar_ub = chol_unblocked_rev(L, Lbar)
    toc = time() - tic
    Delta3 = _trace_dot(Sigma_bar_ub, Sigma_dot)
    print('Unblocked chol_rev works: %r'
            % np.all(np.isclose(Delta1, Delta3)))
    print('   Time taken: %0.4f s' % toc)
    tic = time()
    Sigma_bar_bl = chol_blocked_rev(L, Lbar)
    toc = time() - tic
    Delta4 = _trace_dot(Sigma_bar_bl, Sigma_dot)
    print('Blocked chol_rev works: %r'
            % np.all(np.isclose(Delta1, Delta4)))
```

```
328         print('   Time taken: %0.4f s' % toc)
329
330 if __name__ == '__main__':
331         import sys
332         if len(sys.argv) > 1:
333             N = int(sys.argv[1])
334         else:
335             N = 500
336         _testme(N)
337
338         # Example output for N = 500
339         # ---------------------------
340         # Running np.linalg.cholesky:
341         #     Time taken: 0.0036 s
342         # Unblocked chol works: True
343         #     Time taken: 0.0319 s
344         # Blocked chol works: True
345         #     Time taken: 0.0356 s
346         # Symbolic chol_fwd works: True
347         #     Time taken: 0.0143 s
348         # Unblocked chol_fwd works: True
349         #     Time taken: 0.0592 s
350         # Blocked chol_fwd works: True
351         #     Time taken: 0.0112 s
352         # Symbolic chol_rev works: True
353         #     Time taken: 0.0165 s
354         # Unblocked chol_rev works: True
355         #     Time taken: 0.1069 s
356         # Blocked chol_rev works: True
357         #     Time taken: 0.0093 s
358
359         # Example output for N = 4000
360         # ----------------------------
361         # Running np.linalg.cholesky:
362         #     Time taken: 0.4020 s
363         # Unblocked chol works: True
364         #     Time taken: 25.8296 s
365         # Blocked chol works: True
366         #     Time taken: 0.7566 s
367         # Symbolic chol_fwd works: True
368         #     Time taken: 3.9871 s
369         # Unblocked chol_fwd works: True
370         #     Time taken: 51.6754 s
371         # Blocked chol_fwd works: True
372         #     Time taken: 1.2495 s
373         # Symbolic chol_rev works: True
374         #     Time taken: 4.1324 s
375         # Unblocked chol_rev works: True
376         #     Time taken: 96.3179 s
377         # Blocked chol_rev works: True
378         #     Time taken: 1.2938 s
379
380         # Times are machine and configuration dependent. On the same test
381         # machine, my Level 3 Fortran implementation is only ~10% faster
382         # for N=4000, although can be a lot faster for small matrices.
383         # A Level 2 Fortran implementation isn't as bad as the Python
384         # version, but is still >15x slower than the blocked Python code.
```