Algorithm 1005: Fortran Subroutines for Reverse Mode Algorithmic Differentiation of BLAS Matrix Operations

KRISTJAN JONASSON, SVEN SIGURDSSON, HORDUR FREYR YNGVASON, PETUR ORRI RAGNARSSON, and PALL MELSTED, University of Iceland

A set of Fortran subroutines for reverse mode algorithmic (or automatic) differentiation of the basic linear algebra subprograms (BLAS) is presented. This is preceded by a description of the mathematical tools used to obtain the formulae of these derivatives, with emphasis on special matrices supported by the BLAS: triangular, symmetric, and band. All single and double precision BLAS derivatives have been implemented, together with the Cholesky factorization from Linear Algebra Package (LAPACK). The subroutines are written in Fortran 2003 with a Fortran 77 interface to allow use from C and C++, as well as dynamic languages such as R, Python, Matlab, and Octave. The subroutines are all implemented by calling BLAS, thereby attaining fast runtime. Timing results show derivative runtimes that are about twice those of the corresponding BLAS, in line with theory. The emphasis is on reverse mode because it is more important for the main application that we have in mind, numerical optimization. Two examples are presented, one dealing with the least squares modeling of groundwater, and the other dealing with the maximum likelihood estimation of the parameters of a vector autoregressive time series. The article contains comprehensive tables of formulae for the BLAS derivatives as well as for several non-BLAS matrix operations commonly used in optimization.

CCS Concepts: • **Software and its engineering** → *Software libraries and repositories*; • **Mathematics of computing** → *Maximum likelihood estimation*; *Continuous optimization*; Mathematical software performance;

Additional Key Words and Phrases: Numerical linear algebra, matrix operations, automatic differentiation, algorithmic differentiation, reverse mode, AD, RMAD

ACM Reference format:
Kristjan Jonasson, Sven Sigurdsson, Hordur Freyr Yngvason, Petur Orri Ragnarsson, and Pall Melsted. 2020. Algorithm 1005: Fortran Subroutines for Reverse Mode Algorithmic Differentiation of BLAS Matrix Operations. *ACM Trans. Math. Softw.* 46, 1, Article 9 (March 2020), 20 pages.
https://doi.org/10.1145/3382191

1 INTRODUCTION

The concept of algorithmic or automatic differentiation (AD) has a long history and can be traced back to Newton and Leibniz. Reverse mode (RM) algorithmic differentiation was being used in the middle of the 20th century. Soon after the advent of computers, programs to automatically differentiate other programs appeared. In 1996, Griewank, Juedes, and Utke published a package for the AD of algorithms written in C and C++ [18]. Various surveys of AD and its history may be found in the literature; see, for example, Refs. [10], [15–17], [20], [23], and [26].

Authors' addresses: K. Jonasson, S. Sigurdsson, H. F. Yngvason, P. Orri Ragnarsson and P. Melsted, Department of Computer Science, University of Iceland, Dunhagi 5, 107 Reykjavik, Iceland; emails: {jonasson, sven, hfy1, por1, pmelsted}@hi.is.

0098-3500/2020/03-ART9
https://doi.org/10.1145/3382191

Computation trees, where a calculation with specified inputs and outputs is considered to be composed of nodes and edges, are a fundamental concept in algorithmic differentiation. Except for the input nodes, each node represents either an intermediate or a final result, and the number of predecessors is the same as the arity of the corresponding operation, most often one or two.

When there are n inputs and m outputs, forward mode AD is, in general, more efficient than reverse mode when $n \ll$ m; and when $n = 1$, the time required to compute the derivative is at most three times the time required for the function value. However, when $n \gg m$, RMAD is faster. When $m = 1$, it requires at most four times the function value time for the derivative [14, 15, 19]. More importantly, reverse mode differentiation will, in general, require more memory than forward mode differentiation, since intermediate results corresponding to the nodes in the computation tree need to be stored. This memory requirement can, however, often be reduced at the cost of some recomputation. Nowadays, as noted for example in Ref. [16], the main issues here are possible delays caused by large data movements to and from memory or external memory devices. In the general case, with many inputs and outputs, the situation is more complex, and the optimal derivative computation involves parsing the computation tree both forward and backward. In fact, the determination of the optimal parsing order has been shown to be NP-complete [25].

The majority of the AD literature has been concerned with the case when node results are scalar; but in some cases, computation trees with vector or matrix valued nodes have been considered, and formulae for matrix differentiation have been invented and reinvented. Smith [27] gives an algorithm for reverse mode differentiation of Cholesky factorization. RM formulae for some matrix operations (including multiplication and inversion) are given in Arun Verma's PhD thesis [28], some follow-up reports, for example [4, 29], as well as the recent book of Coleman and Xu [5]. Efficient implementations of forward mode AD for matrices are discussed in Forth [9] and Weinstein and Rao [32]. Giles [11, 12] provides a good overview of both forward and reverse mode matrix formulae, and includes an algorithm for RM differentiation of the Cholesky factorization, some eigenvalue computations, and singular value decomposition. He also points out that some of the results can be found in the 1948 article by Dwyer and Macphail [8]. In their work on AD, Walter and Lehmann support numerical linear algebra (NLA) functions, such as matrix multiplication, Cholesky factorization, QR factorization, and symmetric eigenvalue decomposition, both forward and reverse mode, in Python with AlgoPy [30].

Some of the articles discussed in the previous paragraph are Matlab related and deal with the automatic differentiation of Matlab programs. But using Matlab is not the only way to do matrix computations. Nowadays, efficient matrix computations are predominantly based on the BLAS and Lapack [7]. Apart from Matlab (and its clones Octave and Scilab), the software packages/systems R, NumPy, Julia, and the Numerical Algorithms Group (NAG) library all use the BLAS and Lapack for dense linear algebra computations. The same holds for many (or most) special purpose programs for statistics, physics, chemistry, atmospheric science, and so on. Thus, in practice, composite linear algebra calculations can often be represented by a computation tree with vector or matrix nodes where the operations are calls to BLAS routines. This view leads to the idea of differentiating BLAS operations using AD.

The BLAS and some of the Lapack routines are usually implemented in a highly optimized library specifically tuned to the computer being used, notably Atlas, GotoBLAS, OpenBLAS, and Math Kernel Library (MKL) [1, 13, 31, 33]. Especially on today's multicore computers, such library routines are frequently several times faster than the corresponding reference routines (which are written in Fortran). In most cases, it is difficult or impossible to obtain the actual source code underlying a given routine, so that applying source code transforming (or operator overloaded) AD packages is not feasible. The alternative, of turning the matrix derivative formulae mentioned

above into subprograms, which can be used as building blocks in the differentiation of a composite calculation, is however quite tractable.

As noted for example in Ref. [30], this also has important implications for the memory issue of RM differentiation mentioned above. Thus, if we have, for example, a matrix node corresponding to the operation $A = BC$, where A, B, and C are all n by n matrices, we only have to store the n^2 elements of A, although the corresponding computation tree of scalar operations has $\sim 2n^3$ scalar nodes.

The contribution of our work is that we provide RM derivatives of the BLAS routines: formulae as well as Fortran subroutines. The use of BLAS routines is ubiquitous in scientific computing, not least in the field of application we mainly have in mind, numerical optimization. Applying algorithmic differentiation techniques to methods that use BLAS routines requires access to derivatives of the respective BLAS routines. Furthermore, several BLAS routines have special characteristics, such as different matrix formats (triangular, symmetric, band), all of which must be taken into account when computing the derivatives. Moreover, by expressing RM derivatives in matrix form, one can apply BLAS routines in the programs for the derivative computation. While it would be feasible to automatically differentiate the reference BLAS source code, our method has two distinct advantages over this approach: firstly, it allows the user to use highly optimized BLAS libraries; secondly, it significantly reduces memory requirements as highlighted in the previous paragraph.

In addition to the BLAS, several non-BLAS operations RM-derivatives are considered. While we opt not to include these in the subroutine package (except for Cholesky factorization), our example programs do demonstrate the use of some of them.

The remaining sections of this article are organized as follows. The next section discusses basic mathematical tools for obtaining formulae for RM matrix derivatives, with emphasis on BLAS operations. Section 3 presents such formulae for the BLAS, and Section 4 for several non-BLAS operations. Section 5 has two examples of how the package can be used. Both examples involve modeling optimization, the first minimizes a sum of squares, and the second evaluates a likelihood function. Before concluding, Section 6 gives an overview of the package contents and reports on portability and running time measurements.

2 DERIVING REVERSE MODE AD MATRIX FORMULAE

2.1 Notation and Terminology

Let the scalar f be the final result of a sequence of operations, let θ be an initial parameter of the computation, and let one of the intermediate quantities in the computation of f be the $m \times n$ matrix $A = (a_{ij})$. The derivative of A with respect to θ is the matrix $A' = (a'_{ij})$ where $a'_{ij} = \partial a_{ij}/\partial\theta$ and the derivative of f with respect to A, referred to as the *adjoint* of A, is denoted by $\overline{A}$. Hence, $\overline{A} = (\overline{a}_{ij})$ where $\overline{a}_{ij} = \partial f/\partial a_{ij}$. Similarly, for vectors, x' and $\overline{x}$ are the derivative and adjoint of x (i.e., $x'_i = \partial x_i/\partial\theta$ and $\overline{x}_i = \partial f/\partial x_i$). Using these definitions, we can write the derivative in a more compact form

$$f' = \sum_{i,j} \frac{\partial f}{\partial a_{ij}} \frac{\partial a_{ij}}{\partial \theta} = \sum_{i,j} \overline{A}_{ij} A'_{ij} = \mathrm{tr}(\overline{A}^T A'). \tag{1}$$

For convenience, three matrix functions are introduced: $\mathrm{tril}(A)$ is the lower triangular matrix with the diagonal and subdiagonal elements of A, $\mathrm{dg}(A)$ is a diagonal matrix with $a_{11}, \ldots, a_{nn}$ on the diagonal, if L is lower triangular, the symmetric matrix with the same lower triangle as L (i.e., the matrix $L + L^T - \mathrm{dg}\, L$) is denoted by $\mathrm{sym}(L)$. In addition, the notation $\mathrm{triu}(A) = (\mathrm{tril}(A^T))^T$ and $\mathrm{trils}(A) = \mathrm{tril}(A) - \mathrm{dg}(A)$ will be used occasionally.

Some care must be exercised in the interpretation of expressions involving such lower triangular matrices. When applying matrix operations, these matrices are treated as full matrices; however,

when considering the adjoint of a matrix, attention is restricted to the adjoints of the non-zero elements on and below the diagonal, i.e., the values that need to be stored.

2.2 Using Trace Identities to Derive Adjoints of Matrices

As described by Giles [12], well-known formulae for the trace of a matrix,

$$\begin{aligned} \mathrm{tr}(A^T) &= \mathrm{tr}(A), \\ \mathrm{tr}(A+B) &= \mathrm{tr}(A) + \mathrm{tr}(B), \\ \mathrm{tr}(AB) &= \mathrm{tr}(BA), \end{aligned} \tag{2}$$

can often be used to derive formulae for the adjoints of matrices from corresponding forward mode derivative formulae in a more compact manner than deriving them from scratch. (Verma [28], however, does not make use of trace formulae in his proofs.)

If f only depends on A and B via $C = g(A, B)$, then, from Equation (1), we see that $f' = \mathrm{tr}(\overline{C}^T C')$. Similarly,

$$f' = \mathrm{tr}(\overline{A}^T A') + \mathrm{tr}(\overline{B}^T B'). \tag{3}$$

The gist of the method is to use a forward mode formula to write C' in terms of A' and B', substitute this into Equation (1), and then use Equation (2) to manipulate the result into the form of Equation (3). Then, $\overline{A}$ and $\overline{B}$ can be read off directly.

Take, for example, $g(A, B) = AB$. Then, $C' = A'B + AB'$ so that

$$f' = \mathrm{tr}(\overline{C}^T (A'B + AB')) = \mathrm{tr}(B\overline{C}^T A') + \mathrm{tr}(\overline{C}^T AB'),$$

and comparing with Equation (3), one obtains $\overline{A} = \overline{C}B^T$ and $\overline{B} = A^T\overline{C}$.

2.3 Deriving Adjoints Using Inverse Operations

Let $y = f(x)$ be an operation for which an adjoint formula is sought, where it may have been necessary to extend x and/or y to make f invertible. Let the adjoint of the inverse operation, $x = f^{-1}(y)$, be given by

$$\overline{y} = g_{x,y}(\overline{x}), \tag{4}$$

and assume that $g_{x,y}$ is also invertible. Then, an adjoint formula for f can often be obtained by solving Equation (4) for $\overline{x}$ as a function of x, y and $\overline{y}$.

For example, consider the operation $C = A^{-1}B$, which can be extended to $(C, D) = (A^{-1}B, A)$, where D is a dummy matrix that is assumed to have no influence on the final result. In this case, $x = (A, B)$ and $y = (C, D)$. The inverse operation is $(A, B) = (D, DC)$ with adjoint

$$(\overline{C}, \overline{D}) = (D^T\overline{B}, \overline{A} + \overline{B}C^T)$$

or, because the final result is independent of D:

$$(\overline{C}, 0) = (D^T\overline{B}, \overline{A} + \overline{B}C^T) \tag{5}$$

Now, Equation (5) can be solved for $\overline{A}$ and $\overline{B}$ to obtain

$$\begin{aligned} \overline{A} &= -A^{-T}\overline{C}C^T \\ \overline{B} &= -A^{-T}\overline{C} \end{aligned}$$

in agreement with Table 7. More examples of formulae derived by this approach may be found in Table 7 and in Sections 4.1 and 4.2.

Table 1. Adjoints for Copy Operations

operation	adjoint
$A = \operatorname{sym} L$	$\overline{L} = \operatorname{tril}(\overline{A} + \overline{A}^T) - dg\overline{A}$
$L = \operatorname{tril} A$	$\overline{A}$ = matrix with lower triangle $\overline{L}$ and zeros above diagonal
$D = \operatorname{dg} A$	$\overline{A}$ = matrix with $\overline{D}$ on the diagonal and zeros elsewhere

2.4 Deriving Adjoints of Symmetric, Band, and Triangular Matrices

Reverse mode derivative formulae for general operations have been studied extensively, but a similar treatment of symmetric, band, and triangular matrix operations is lacking. For their treatment, the following identities are useful, in addition to Equations (1), (2), and (3):

$$\operatorname{tr}(AB^T) = \operatorname{tr}(A^T B) \tag{6}$$

$$\operatorname{tr}(A \operatorname{dg}(B)) = \operatorname{tr}(\operatorname{dg}(A)B) \tag{7}$$

and for lower triangular L:

$$\operatorname{tr}(L^T A) = \operatorname{tr}(L^T \operatorname{tril}(A)). \tag{8}$$

The nonzero structure of the matrices A and/or B in Equations (1), (2), and (3) must be taken into account when applying the trace method. For example, if L is lower triangular, we conclude that

$$\operatorname{tr}(\overline{L}^T L') = \operatorname{tr}(M^T L') \text{ implies } \overline{L} = \operatorname{tril} M \tag{9}$$

(it does not imply that $\overline{L} = M$). Equation (10) in Section 3.2 is an example where these identities are used. They can also be used, together with the results of the previous section, to derive the adjoint formulae for forward substitution given in Table 3.

2.5 Adjoints of Matrices Obtained by Sym, Tril, and Dg

If A is a symmetric matrix, represented by a lower triangular matrix L, i.e., $A = \operatorname{sym} L$, then the adjoint of such a matrix can also be represented by a lower triangular matrix, denoted with $\overline{L}$, since there is precisely one adjoint for each and every one of the original input variables. We note, however, that the adjoint of l_{ij}, $i > j$, need not be the same in the (i, j)-th position of A as it is in the (j, i)-th position. This is, for example, evident in the case when $y = Ax$, where we have $\overline{A} = \overline{y}x^T$. Thus, when $A = \operatorname{sym} L$, we have $\overline{L} = \operatorname{tril}(\overline{A} + \overline{A}^T) - \operatorname{dg} \overline{A}$ as shown in Table 1, simply stating that the adjoint of l_{ij} is the sum of its contributions from the (i, j)-th and the (j, i)-th positions of A. With regard to tril, we note that while the adjoints of some of the strictly upper triangular elements of A may in fact not be zero, their values are irrelevant because they will never be referenced. Thus, it is sufficient and natural to restrict the adjoint of $\overline{A}$ to a lower triangular matrix, as is effectively done in Table 1. A similar argument applies to dg. We finally note that if $A = \operatorname{sym} L$, such that $L = \operatorname{tril} A$ we can equate $\overline{A} = \overline{L}$ in accordance with the formula for tril in Table 1. We choose always to refer to $\overline{L}$ as the adjoint of A.

2.6 Adjoints for Givens Rotations

A basic Givens rotation G satisfies:

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}, \text{ where } G \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a^* \\ 0 \end{pmatrix}$$

The parameters c, s, and a^* are computed from a and b with a sequence of scalar operations, and the corresponding adjoints $\overline{a}$ and $\overline{b}$ are readily obtained with scalar differentiation. The application

of such a rotation involves multiplying a 2 by n matrix X with G, so the associated adjoints are simply those of a matrix multiplication.

A modified Givens rotation H is much more involved, with detailed formulae provided in Ref. [24]. The corresponding adjoints may again be obtained with scalar differentation of these formulae. The application of a modified rotation, and thus its adjoint, are analogous to a basic rotation. For both types of rotation application, the adjoint of X may be expressed in terms of the rotation itself, as shown in Table 5.

3 DERIVATIVES OF BLAS OPERATIONS

3.1 Overview of BLAS Operations

Apart from numbers and vectors, the BLAS routines can handle four different kinds of matrices: general, triangular, symmetric, and Hermitian. In addition, each of these kinds may be banded or not, and the non-banded triangular, symmetric, and Hermitian matrices may be stored in packed form or not. Finally, there are four data types, single and double precision real and complex (denoted with prefix letters S, D, C, and Z).

It is not meaningful to differentiate all BLAS operations. The operation `iamax` does not involve floating operations, and operations involving complex conjugates pose problems because their result is not complex differentiable. The route we have chosen is to consider only operations that satisfy the Cauchy-Riemann equations (and thus have a complex derivative), and discount all operations with conjugate transposes or Hermitian matrices. The operations with meaningful derivatives, apart from `copy` and `swap`, which are trivial, are summarized in Table 2. In addition to the given varieties, real operations can have data types S or D, and complex ones can be C or Z.

Some of the BLAS operations are not differentiable everywhere, in particular, `nrm2` when $x = 0$, `asum` when any $x_i = 0$, and `rotg` when $a = b$. This does not have to ruin the usefulness of computing the adjoints: `nrm2` and `asum` are continuous everywhere and differentiable almost everywhere, and for `rotg`, there is an illuminating example in the file test_rot.m in the subroutine package.

3.2 List of BLAS Adjoint Formulae

Tables 3, 4, and 5 provide adjoint formulae for all the BLAS operations of Table 2 except rotmg. In Table 3 (but not in the accompanying software package), operations that add a product times α to an object (such as axpy or ger) have been simplified to have only the product. The same holds for similar operations that simultaneously multiply the object with β (such as gemv or syrk). The formulae in the package are readily deduced using the linearity of the adjoint.

Another feature is that the C operators `+=` and `-=` are used to specify the formulae, to emphasize the fact that if several edges ascend from an intermediate result in the computation tree, then the adjoints resulting from all these edges must be added together. Node K in the computational tree in Figure 1 is a good example of this, as are nodes Σ, A, and X in the computational tree in Figure 3.

All the formulae given in Tables 3, 4, and 5 have been implemented as Fortran subroutines. The subroutine package supports the addition of products of α to objects, as well as their multiplication by β. The package also implements the adjoint for rotmg, which is too involved to show in Table 5, as well as the adjoints of banded matrices, which are not given explicitly in Table 3. In the tables, scalars are denoted by α, β, vectors by x, y, z, general matrices by A, B, C, symmetric matrices by M, lower triangular matrices by L, L_1, permutations by p, and upper triangular matrices by U. Note that triangular matrices can in all cases be lower or upper, even if the table only shows two examples with U. Similarly, symmetric matrices can be kept either below or above the diagonal.

Table 2. BLAS Operations with Meaningful Derivatives

operation	names	varieties
generate Givens rotation	`rotg`, `rotmg`	No complex version.
apply Givens rotation	`rot`, `rotm`	No complex version.
$x \leftarrow \alpha x$	`scal`	In addition to S, D, C, Z, real α and complex x is supported with prefixes CS and ZD.
$y \leftarrow \alpha x + y$	`axpy`	Full complex support.
result $\leftarrow x^T y$	`dot`, `dotu`	The complex version is `dotu`.
result $\leftarrow \|x\|_2$	`nrm2`	No differentiable complex version.
result $\leftarrow \|x\|_1$	`asum`	When x is complex, return $\|\operatorname{Re} x\|_1 + \|\operatorname{Im} x\|_1$.
$y \leftarrow \alpha(\operatorname{op} A)x + \beta y$	`gemv`, `gbmv`, `symv`, `sbmv`, `spmv`	A can be general or symmetric; when A is general, it can be banded or not, and $\operatorname{op} A$ can be A or A^T; when A is symmetric, $\operatorname{op} A = A$ and A can be non-banded, banded, or packed. Complex symmetric A is not supported.
$x \leftarrow (\operatorname{op} A)x$	`trmv`, `tbmv`, `tpmv`, `trsv`, `tbsv`, `tpsv`	A is lower or upper triangular; it can be unit-diagonal or not, and non-banded, banded, or packed; $\operatorname{op} A$ can be A, A^T, A^{-1}, or A^{-T}. Full complex support.
$A \leftarrow \alpha x y^T + A$	`ger`, `geru`	The complex version is `geru`
$A \leftarrow \alpha x x^T + A$	`syr`, `spr`	No complex versions.
$A \leftarrow \alpha(x y^T + y x^T) + A$	`syr2`, `spr2`	No complex versions.
$C \leftarrow \alpha(\operatorname{op} A)(\operatorname{op} B) + \beta C$	`gemm`, `symm`	A and B can both be general and then $\operatorname{op} X$ can be X or X^T; alternatively, one of A and B can be symmetric with $\operatorname{op} X = X$. Full complex support.
$B \leftarrow \alpha(\operatorname{op} A)B$ $B \leftarrow \alpha B(\operatorname{op} A)$	`trmm`, `trsm`	B is general; A is lower or upper triangular; it can be unit diagonal or not; $\operatorname{op} A$ can be A, A^T, A^{-1} or A^{-T}. Full complex support.
$C \leftarrow \alpha A A^T + \beta C$ $C \leftarrow \alpha A^T A + \beta C$	`syrk`	A is general and C symmetric. Full complex support.
$C \leftarrow \alpha(A B^T + B A^T) + \beta C$ $C \leftarrow \alpha(A^T B + B^T A) + \beta C$	`syr2k`	A, B are general and C symmetric. Full complex support.

BLAS handles symmetric matrices by referencing only their lower or only their upper triangle (the other triangle is the transpose of the first one). From the point of view of differentiation, symmetric matrices in BLAS operations should therefore be treated as triangular matrices. Consider the differentiation of the first `syrk` operation in Table 2. This operation is really $L_1 \leftarrow \operatorname{tril}(\alpha A A^T + \beta \operatorname{sym} L)$, where L is the lower triangle of the ingoing C, and L_1 is the lower triangle of the result. Following the trace approach, and making use of Equations (6) and (8),

Table 3. Adjoints of Vector and Matrix Inputs to BLAS Operations

operation	BLAS	formulae	adjoint formulae
scalar times vector	`scal`, `axpy`	$y = \alpha x$	$\overline{x}$ += $\alpha\overline{y}$
inner product	`dot`	$\beta = x^T y$	$\overline{x}$ += $\overline{\beta}y$, $\overline{y}$ += $\overline{\beta}x$
vector 2-norm	`nrm2`	$\beta = \|x\|_2$	$\overline{x}$ += $\overline{\beta}x/\beta$
vector 1-norm	`asum`	$\beta = \|x\|_1$	$\overline{x}_i$ += $\overline{\beta}\,\mathrm{sgn}\,x_i, i = 1, 2, \ldots$
matrix times vector	`gemv`	$y = Ax$	$\overline{A}$ += $\overline{y}x^T$, $\overline{x}$ += $A^T\overline{y}$
transpose times vector	`gemv`	$y = A^T x$	$\overline{A}$ += $x\overline{y}^T$, $\overline{x}$ += $A\overline{y}$
symmetric times vector	`symv`	$y = (\mathrm{sym}\,L)x$	$\overline{L}$ += $\mathrm{tril}(\overline{y}x^T + x\overline{y}^T) - \mathrm{dg}(x\overline{y}^T)$ x += $(\mathrm{sym}\,L)\overline{y}$
lower triangle times vector	`trmv`	$y = Lx$	$\overline{L}$ += $\mathrm{tril}(\overline{y}x^T)$, $\overline{x}$ += $L^T\overline{y}$
upper triangle times vector	`trmv`	$y = Ux$	$\overline{U}$ += $\mathrm{triu}(\overline{y}x^T)$, $\overline{x}$ += $U^T\overline{y}$
forward substitution	`trsv`	$y = L^{-1}x$	$\overline{L}$ −= $\mathrm{tril}(zy^T)$, $\overline{x}$ += z, where $z = L^{-T}\overline{y}$
back substitution	`trsv`	$y = U^{-1}x$	$\overline{U}$ −= $\mathrm{triu}(zy^T)$, $\overline{x}$ += z, where $z = U^{-T}\overline{y}$
outer product	`ger`	$C = xy^T$	$\overline{x}$ += $\overline{C}y$, $\overline{y}$ += $\overline{C}^T x$
symmetric rank 1	`syr`	$L = \mathrm{tril}(xx^T)$	$\overline{x}$ += $(\overline{L} + \overline{L}^T)x$
symmetric rank 2	`syr2`	$L = \mathrm{tril}(xy^T + yx^T)$	$\overline{x}$ += $(\overline{L} + \overline{L}^T)y$ $\overline{y}$ += $(\overline{L} + \overline{L}^T)x$
matrix product	`gemm`	$C = AB$	$\overline{A}$ += $\overline{C}B^T$, $\overline{B}$ += $A^T\overline{C}$
transpose times matrix	`gemm`	$C = A^T B$	$\overline{A}$ += $B\overline{C}^T$, $\overline{B}$ += $A\overline{C}$
lower triangle times matrix	`trmm`	$C = LB$	$\overline{L}$ += $\mathrm{tril}(\overline{C}B^T)$, $\overline{B}$ += $L^T\overline{C}$
forward substitution w. matrix	`trsm`	$C = L^{-1}B$	$\overline{L}$ −= $\mathrm{tril}(MC^T)$, $\overline{B}$ += M, where $M = L^{-T}\overline{C}$
symmetric times general	`symm`	$C = (\mathrm{sym}\,L)B$	$\overline{L}$ += $\mathrm{tril}(B\overline{C}^T + \overline{C}B^T) - \mathrm{dg}(B\overline{C}^T)$, $\overline{B}$ += $(\mathrm{sym}\,L)\overline{C}$
symmetric rank k	`syrk`	$L = \mathrm{tril}(AA^T)$	$\overline{A}$ += $(\overline{L} + \overline{L}^T)A$
symmetric rank k transposed	`syrk`	$L = \mathrm{tril}(A^T A)$	$\overline{A}$ += $A(\overline{L} + \overline{L}^T)$
symmetric rank $2k$	`syr2k`	$L = \mathrm{tril}(AB^T + BA^T)$	$\overline{A}$ += $(\overline{L} + \overline{L}^T)B$ $\overline{B}$ += $(\overline{L} + \overline{L}^T)A$

Table 4. Adjoints of Scalar Inputs to BLAS Operations

BLAS	formulae	adjoint formulae
`scal`, `axpy`, $\ldots + \alpha y$	$y = \alpha x$	$\overline{\alpha}$ += $\overline{y}^T x$
`g[eb]mv`[a], `s[ypb]mv`	$y = \alpha Ax$	$\overline{\alpha}$ += $\overline{y}^T Ax$
$\ldots + \alpha C$	$C = \alpha A$	$\overline{\alpha}$ += $\mathrm{vec}(\overline{C})^T\,\mathrm{vec}(A)$
`gemm`, `symm`, `tr[ms]m`	$C = \alpha AB$	$\overline{\alpha}$ += $\mathrm{vec}(\overline{C})^T\,\mathrm{vec}(AB)$
`s[yp]r`	$L = \alpha\,\mathrm{tril}(xx^T)$	$\overline{\alpha}$ += $x^T\overline{L}x$
`s[yp]r2`	$L = \alpha\,\mathrm{tril}(xy^T + yx^T)$	$\overline{\alpha}$ += $x^T\overline{L}y + y^T\overline{L}x$
`syrk`,`syr2k`	$L = \alpha\mathtt{tril}(M)$	$\overline{\alpha}$ += $\mathrm{vech}(\overline{L})^T\,\mathrm{vech}(M)$
`ger`	$C = \alpha xy^T$	$\overline{\alpha}$ += $x^T\overline{C}y$

[a]Notation borrowed from Unix's *sh*; short for "`gbmv` and `gemv`".

Table 5. Adjoints of BLAS Rotation Operations

BLAS[a]	formulae	adjoint formulae
`rotg`	$a^* = \sigma\sqrt{a^2+b^2}$ $c = a/a^*$ $s = b/a^*$ where: $\sigma = \begin{cases} \operatorname{sgn} a \text{ if } \lvert a\rvert > \lvert b\rvert \\ \operatorname{sgn} b \text{ otherwise} \end{cases}$ unless $a^* = 0$, then $c = 1, s = 0$	$\overline{a}$ += $c_1 + cd$ $\overline{b}$ += $s_1 + sd$ where: $c_1 = \overline{c}/a^*$ $s_1 = \overline{s}/a^*$ $d = \overline{a}^* - cc_1 - ss_1$
`rot`	$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = G \begin{pmatrix} x \\ y \end{pmatrix}$ where: $G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$	$\begin{pmatrix} \overline{x} \\ \overline{y} \end{pmatrix}$ += $G^T \begin{pmatrix} \overline{x}^* \\ \overline{y}^* \end{pmatrix}$ $\begin{pmatrix} \overline{c} \\ \overline{s} \end{pmatrix}$ += $G^T \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ where: $\alpha = \overline{x}^* x^{*T} + \overline{y}^* y^{*T}$ $\beta = \overline{x}^* y^{*T} - \overline{y}^* x^{*T}$
`rotm`	$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = H \begin{pmatrix} x \\ y \end{pmatrix}$ where: H is the output matrix of `rotmg`[b]	$\begin{pmatrix} \overline{y} \\ \overline{x} \end{pmatrix}$ += $K \begin{pmatrix} \overline{y}^* \\ \overline{x}^* \end{pmatrix}$ $\overline{H}$ += $A^T H^{-T}$ [c] where:[d] $K = \begin{pmatrix} h_{22} & h_{12} \\ h_{21} & h_{11} \end{pmatrix}$, $A = \begin{pmatrix} x^* \\ y^* \end{pmatrix} \begin{pmatrix} \overline{x}^* \\ \overline{y}^* \end{pmatrix}^T$

[a] See comments in subroutine `srotmg_rmd` for adjoint of `rotmg`, which is too involved to show in this table.
[b] See comments in subroutine `srotmg_rmd` for the definition of H.
[c] Except that $\overline{H}$-entries corresponding to fixed H-entries remain unchanged (see details in comments in subroutine `srotm_rmd`).
[d] K is introduced so that $\overline{x}$ and $\overline{y}$ may be computed with a `rotm-operation`.

we have

$$\begin{aligned} \operatorname{tr}(\overline{A}^T A') + \operatorname{tr}(\overline{L}^T L') &= f' = \operatorname{tr}(\overline{L}_1^T L_1') \\ &= \operatorname{tr}(\overline{L}_1^T \operatorname{tril}(\alpha A' A^T + \alpha A A'^T + \beta \operatorname{sym} L')) \\ &= \alpha \operatorname{tr}(\overline{L}_1^T A' A^T + \overline{L}_1^T A A'^T) + \beta \operatorname{tr}(\overline{L}_1^T L') \\ &= \alpha \operatorname{tr}(A^T (\overline{L}_1^T + \overline{L}_1) A') + \beta \operatorname{tr}(\overline{L}_1^T L'). \end{aligned} \tag{10}$$

Comparing the A' terms on the left-hand and the right-hand sides above, we obtain $\overline{A} = \alpha(\overline{L}_1 + \overline{L}_1^T)A$, and comparison of the L' terms gives $\overline{L} = \beta \overline{L}_1$.

Now, consider the `trmm` operation $B_1 \leftarrow \alpha LB$. Applying the trace method, we obtain $\operatorname{tr}(\overline{L}^T L') = \operatorname{tr}(\alpha B \overline{B}_1^T L')$, and conclude from Equation (7) that $\overline{L} = \operatorname{tril}(\alpha \overline{B}_1 B^T)$. The remaining symmetric and triangular BLAS operations are either special cases of these (`syr` and `trmv`), or else they may be dealt with using similar reasoning (`symv`, `symm`, `trsv`, `trsm`, `syr2`, `syr2k`).

3.3 Composition of Operations and Execution Time

It is not difficult to see that all the adjoint formulae of Table 2 cost about twice as many floating point operations as the BLAS operations being differentiated. This compares favorably with the maximum operation count mentioned in the introduction, that adjoints cost at most four times more than the function value. But operation count does not tell the whole story. As already said, the BLAS operations are usually implemented in highly optimized libraries tuned to the computer

Table 6. Dominant Adjoint-Operations and Their Time Complexity

BLAS-operation	level	dominant operations in adjoint
axpy (n)[a]	1	axpy, dot ($n+n$)
scal (n)	1	scal, dot ($n+n$)
dot (n)	1	two times axpy ($2n$)
nrm2 (n)	1	scal (n)
gemv (mn)	2	ger, gemv ($mn+mn$)
s[yp]mv (n^2)[b]	2	s[yp]r2, s[yp]mv (n^2+n^2)
[gst]bmv (dn)	2	n times[c] axpy, [gst]bmv ($dn+dn$)
t[rpb][ms]v ($n^2/2$)	2	n times[c] axpy, t[rpb][ms]v ($n^2/2+n^2/2$)
ger (mn)	2	two times gemv ($2mn$)
s[yp]r ($n^2/2$)	2	s[yp]mv (n^2)
s[yp]r2 (n^2)	2	two times s[yp]mv ($2n^2$)
gemm (mnk)	3	two times gemm ($2mnk$)
symm (m^2n)	3	syr2k, symm (m^2n+m^2n)
tr[ms]m ($m^2n/2$)	3	n times[c] gemv, tr[ms]m ($m^2n/2+m^2n/2$)
syrk ($m^2n/2$)	3	symm, syrk ($m^2n/2+m^2n/2$)
syr2k (m^2n)	3	two times symm ($2m^2n$).

[a] Time complexity in brackets.
[b] Notation borrowed from Unix's *sh*; short for "symv and spmv"
[c] With level-drop: The adjoint involves n BLAS-operations of the next lower BLAS-level.

being used, and they often run in parallel on multicore computes. BLAS level 2 and level 3 arguments are blocked, and care is taken to access adjacent memory locations in sequence if possible.

For many of the BLAS operations, the corresponding adjoints may be calculated by calling two BLAS operations at the same level. This is, of course, an ideal situation, and one can expect that the execution time, as well as the operation count, will only be doubled compared with the original operation. There are, however, a few operations where the computation of one of the adjoints requires routines from the level below. This holds for all the triangular and band matrix operations; the adjoints for these operations require a level-drop. Table 6 summarizes the dominant operations needed for the adjoints for each BLAS operation, along with their time complexity (the meaning of m, n, k and d is mostly self-explanatory: When both m and n appear, they are the row and column counts of the first (or only) rectangular matrix appearing, and d is the band-width).

It is interesting to see how the adjoint for a BLAS operation may often be computed via different BLAS operations. In many cases, there is a kind of duality: The differentiation of syr2 involves symv and vice versa.

For adjoints that do not require a level-drop, there will be little or no need for optimized implementation because the BLAS operations for the dominant calculation will already be optimized. However, when an adjoint computation cannot be carried out by calling a BLAS routine at the same level, there might be a reason for an optimized (and blocked) version of the rmd-operation. The subroutine package does not attempt to do any such optimization.

4 DERIVATIVES OF NON-BLAS OPERATIONS

The Verma and Giles formulae lists [11, 28] include many operations that are not part of BLAS. Some are operations contained in Lapack, but some are normally not used directly in practical computations, such as matrix inverses and determinants. These are nevertheless interesting theoretically. We have constructed our own list of reverse mode derivatives of interesting and/or

Table 7. Adjoint Formulae for a Few Lapack and Other Non-BLAS Matrix Operations

operation	lapack	formulae	adjoint formulae
solve linear equations	`getrs`	$y = A^{-1}x$	$\overline{A} \mathrel{-}= zy^T$, $\overline{x} \mathrel{+}= z$, where $z = A^{-T}\overline{y}$
solve linear matrix equations	`getrs`	$C = A^{-1}B$	$\overline{A} \mathrel{-}= MC^T$, $\overline{B} \mathrel{+}= M$, where $M = A^{-T}\overline{C}$
solve symmetric equations	`potrs`	$y = (\mathrm{sym}\, L)^{-1}x$	$\overline{L} \mathrel{-}= \mathrm{tril}(zy^T + yz^T) - \mathrm{dg}(yz^T)$, $\overline{x} \mathrel{+}= z$, where $z = (\mathrm{sym}\, L)^{-1}\overline{y}$
LU factorization	`getrf`	$[L, U, P] = \mathrm{plu}(A)$	$\overline{A} \mathrel{+}= P^T\hat{A}$, where $\hat{A}$ is the solution to the system $\overline{U} = \mathrm{triu}(L^T\hat{A})$, $\overline{L} = \mathrm{trils}(\hat{A}U^T)$
Cholesky factorization	`potrf`	$L_1 = \mathrm{chol}(\mathrm{sym}\, L)$	$\overline{L} \mathrel{+}= \overline{L}_0$, the solution to $\overline{L}_1 = \mathrm{tril}((\overline{L}_0 + \overline{L}_0^T)L_1)$
sum of squares		$\beta = x^Tx$	$\overline{x} \mathrel{+}= 2\overline{\beta}x$
quadratic matrix term		$C = A^TMA$	$\overline{A} \mathrel{+}= MA\overline{C}^T + M^TA\overline{C}$, $\overline{M} \mathrel{+}= A\overline{C}A^T$
quadratic matrix term		$C = AMA^T$	$\overline{A} \mathrel{+}= \overline{C}AM^T + \overline{C}^TAM$, $\overline{M} \mathrel{+}= A^T\overline{C}A$
quadratic scalar term		$\beta = x^TMx$	$\overline{x} \mathrel{+}= \overline{\beta}(M + M^T)x$, $\overline{M} \mathrel{+}= \overline{\beta}xx^T$
quadratic inverse matrix term		$C = A^TM^{-1}A$	$\overline{A} \mathrel{+}= M^{-1}A\overline{C}^T + M^{-T}A\overline{C}$, $\overline{M} \mathrel{-}= M^{-T}A\overline{C}A^TM^{-T}$
quadratic inverse scalar term		$\beta = x^TM^{-1}x$	$\overline{x} \mathrel{+}= \overline{\beta}(y + z)$, $\overline{M} \mathrel{-}= \overline{\beta}zy^T$, where $y = M^{-1}x$ and $z = M^{-T}x$
vector sum		$z = x + y$	$\overline{x} \mathrel{+}= \overline{z}$, $\overline{y} \mathrel{+}= \overline{z}$
matrix sum		$C = A + B$	$\overline{A} \mathrel{+}= \overline{C}$, $\overline{B} \mathrel{+}= \overline{C}$
determinant		$\beta = \det A$	$\overline{A} \mathrel{+}= \beta\overline{\beta}A^{-T}$
matrix inverse		$C = A^{-1}$	$\overline{A} \mathrel{-}= C^T\overline{C}C^T$
symmetric inverse		$L_1 = \mathrm{tril}(\mathrm{sym}\, L)^{-1}$	$\overline{L} \mathrel{-}= 2\,\mathrm{tril}(\hat{L}) - \mathrm{dg}(\hat{L})$ where $\hat{L} = \frac{1}{2}S(\overline{L}_1 + \overline{L}_1^T)S$, $S = \mathrm{sym}\, L_1$
discrete Lyapunov equation		X solves $X - AXA^T = M$	$\overline{M} \mathrel{+}= S$, $\overline{A} \mathrel{+}= SAX^T + S^TAX$ where S solves $S - A^TSA = \overline{X}$

important non-BLAS operations, given in Table 7. Explanations related to some of the formulae are in order and follow below.

4.1 Solution to Linear Equations and Matrix Factorization

Formulae for the adjoints for solutions to general linear equations (Lapack `getrs`) may be found in Refs. [11] and [28] and are derived by the trace method. Here, we note that these formulae, including the symmetric version (Lapack `potrs`), may be deduced directly from the formulae for the BLAS operations `gemv`, `gemm`, and `symv`, respectively, given in Table 3 by the inversion argument as shown in Section 2.3.

A remarkable property of these adjoint formulae is that if the factors A or $\mathrm{sym}\, L$ are kept, then the adjoint calculation is only an $O(n^2)$ operation, compared with $O(n^3)$ for the original equation solution.

The formulae for the LU and Cholesky factorizations are deduced in the same way by the inverse operation argument from the formulae given for the BLAS operations `gemm` and `syrk` in Table 3. With regard to the LU-factorization, we first note that the elements of the permutation matrix, P, and the diagonal elements of the lower triangular matrix, L, are fixed and thus have zero adjoints. Secondly, we note that an efficient procedure for calculating $\hat{A}$ is to alternately calculate the columns in the upper triangular part of $\hat{A}$ from right to left, and the rows in the strictly lower

triangular part of $\hat{A}$ from bottom to top. Each column is computed by equating the corresponding elements in $\overline{U} = L^T\hat{A}$ from bottom to top and each row by equating the corresponding elements in $\overline{L} = \hat{A}U^T$ from right to left.

Giles [11] derives an algorithm for the adjoint of Cholesky factorization, starting with a row-oriented algorithm for calculating the Cholesky factor L_1 from the symmetric matrix $A = \operatorname{sym} L$, deducing from it an algorithm for calculating the derivative L_1', and finally obtaining an algorithm for calculating $\overline{A}$ from L_1 and $\overline{L}_1$ in reverse row order. This algorithm is, however, also readily deduced simply by equating both sides of the equation $\overline{L}_1 = \operatorname{tril}((\overline{L} + \overline{L}^T)L_1)$ row by row, bottom up, and from right to left.

The subroutine package contains two implementations of Cholesky factorization adjoint, both of which are different from the one just described: A Fortran implementation based on a recursive Cholesky factorization method, and a Matlab/Octave implementation based on a column oriented method. Details may be found in comments in the respective program files.

4.2 Quadratic Terms and Discrete Lyapunov Equations

Formulae for the adjoints for solutions to the quadratic matrix terms may again be found in Refs. [11] and [28], derived in a fairly straightforward way by the trace method (as are the adjoint formulae for the determinant and matrix inverse). Here, we just note that the formula for the quadratic inverse matrix term follows directly from the formulae for the quadratic matrix term and the quadratic inverse. The corresponding quadratic scalar terms can be deduced directly by the trace method or as special cases of the quadratic matrix terms.

Finally, note that in the special common case when M is a symmetric matrix, $M = \operatorname{sym}(L)$, one can conclude from Table 1 that $\overline{L} = 2\operatorname{tril}\overline{M} - \operatorname{dg}\overline{M}$, where the expression for $\overline{M}$ is that given in Table 7. The formula for the symmetric matrix inverse is similarly deduced from the formula for the matrix inverse and the results in Table 1. Previous work does not seem to deal specifically with these symmetric cases.

The formula for the adjoint of the solution to the discrete Lyapunov equation may be obtained with the inverse method of Section 2.3. Let $x = (A, M)$ and $y = (X, D)$ where D is a dummy matrix with zero adjoint. The inverse operation is

$$\begin{aligned} A &= D \\ M &= X - DXD^T, \end{aligned}$$

where X satisfies $X - AXA^T = M$. Using the second result on quadratic matrix terms in Table 7, the adjoint Equation (4) becomes:

$$\begin{aligned} \overline{D} &= \overline{A} - \overline{M}AX^T - \overline{M}^T AX \\ \overline{X} &= \overline{M} - D^T\overline{M}D \end{aligned}$$

and using $\overline{D} = 0$ and $D = A$, the last results in Table 7 follow.

Alternatively, the formula can be derived from trace identities, using the fact that if the spectral radius of A is less than 1, then the solution can be expressed as $X = \sum_{k=0}^{\infty} A^k M (A^T)^k$.

5 EXAMPLES

5.1 Example 1: Least Squares Estimation of Parameters in a Groundwater Model

The level of the groundwater, u, within a region can be simulated by a Poisson equation, $\nabla \cdot (k\nabla u) = g$, where k denotes the permeability of the ground formation, typically assumed to be constant within given subregions, and g is an input/output function, accounting, for example,

ALGORITHM 1: Sum of squares for a groundwater model

Input: parameters, k (m-vector)
Data: A_i ($n \times n$), $i = 0, 1, \ldots, m$, f_0 (n-vector), F ($n \times m$), C ($s \times n$), d (s-vector)
Output: sum of squares, θ, and parameter adjoint, $\bar{k}$ (m-vector; $\bar{k} = \nabla_k \theta$)

1 $B = A_0$
2 **for** $j = 1, \ldots, m$ **do**
3 $\quad B = B + k_j A_j$ ▷ axpy $(k_j, \mathsf{vec}(A_j), \mathsf{vec}(B))$
4 **end**
5 $g = Fk + f_0$ ▷ gemv (F, k, f_0)
6 Solve the linear system $Bu = g$ for u
7 $r = Cu - d$ ▷ gemv $(C, u, -d)$
8 $\theta = r^T r$ ▷ $\theta =$ dot (r, r)

▷ **Compute adjoint:**
9 $\bar{\theta} = 1$ ▷ influence of θ on θ
10 $\bar{r} = \mathsf{ddot\_rmd}_r(r, \bar{\theta})$
11 $\bar{u} = \mathsf{dgemv\_rmd}_u(C, u, \bar{r})$
12 Solve the linear system $B^T z = \bar{u}$ for z
13 $\bar{g} = z$ ▷ linear system adjoint for g
14 $\bar{B} = -zu^T$ ▷ linear system adjoint for B
15 $\bar{k} = \mathsf{dgemv\_rmd}_k(F, k, \bar{g})$
16 **for** $j = m, \ldots, 1$ **do**
17 $\quad \bar{k}_j$ += dot $(\mathsf{vec}(\bar{B}), \mathsf{vec}(A_j))$ ▷ daxpy adjoint for k_j
18 **end**

for rainfall. In addition, there are some specified boundary conditions, for example, for a specific water level or for a noflow boundary. The k-values are often estimated by minimizing the sum of squared deviations between measured and simulated water levels at given points within the region. Thus, one is interested in the influence of the k-values on this sum, in other words, their adjoint. After discretization by a finite difference scheme, this task reduces to the following problem:

Find k that minimizes θ, defined by:

$$\theta = r^T r \text{ with } r = Cu - d \text{ and } u = A(k)^{-1} f(k), \tag{11}$$

where

$$A(k) = A_0 + \sum_{j=1}^{m} k_j A_j \text{ and } f(k) = f_0 + \sum_{j=1}^{m} k_j f_j;$$

and m is the number of subregions. Some more details of how these equations arise may be found in comments in the file run_groundwater.m in the matlab subdirectory, discussed below.

An algorithm to evaluate θ along with the derivative of θ with respect to k (the adjoint of k) is needed, for use in a continuous optimization method to minimize θ. The first part of Algorithm 1 is obtained directly from Equation (11), where the f_i for $i \geq 1$ have been placed in a matrix F. There are three BLAS-operations (lines 5, 7, and 8), and the fourth operation, the scalar-times-matrix on line 3, can be turned into daxpy by stacking the matrix columns using vec, as indicated in the associated comment. The computation tree for the case $m = 2$ is shown in Figure 1 for further clarification.

The derivatives of gemv and dot are readily obtained with the RMD-subroutines of the current article, and the second part of Algorithm 1 shows the resulting operations on lines 10, 11, and 15.

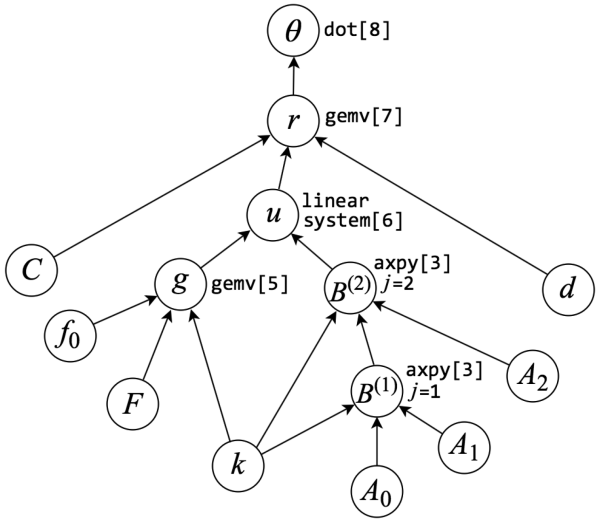

Fig. 1. Computational tree for Algorithm 1 for the case $m = 2$. Numbers in square brackets refer to lines in the algorithm, and superscripts (1) and (2) denote subsequent values of B computed on line 3.

Note how these RMD-operations are in reverse order. The required adjoint for axpy on line 3 is of the scalar k_j, and the RMD-routines do not provide this. Instead, one may use the result of Table 3, seen on line 17. The linear system solution on line 6 could be solved with the LAPACK-routines dgetrf and dgetrs. Using the adjoint formula for getrs in Table 7, the needed adjoints can be obtained by solving the linear system on line 12; for this, the dgetrf factorization of B can be reused.

Observe that not all adjoints for individual operations need to be computed to get the final result, $\bar{k}$. For line 7, only $\bar{u}$ is needed; for the linear system on line 6, one needs $\bar{g}$ and $\bar{B}$; only $\bar{k}$ is needed for line 5; and, finall, y for line 3, $\bar{k}_j$ must be updated (for all j). This last RMD-operation is the only one where += is needed; the remaining adjoints can all be assigned. This fact can be seen by inspecting the computation tree in Figure 1—k is the only variable with more than one outgoing arrow, and the $k \to g$ adjoint is computed before the other k-adjoints.

Because only a subset of the adjoints are required, the derivative is cheaper than indicated in Section 3.3, where it is stated that the operation count of RMD-adjoints is normally about double that of the corresponding BLAS operations. Actually, the derivative in Algorithm 1 only costs a fraction of the function value if the linear system on line 6 is solved with a matrix factorization that is reused on line 12. The factorization will cost about $n^3/3$ flops and the solution on line 12 about n^2 flops. For all the other operations, the number of flops of the RMD-operations is about the same as of the original operation.

The Fortran statements in Figure 2 show an implementation of the algorithm using the BLAS-RMD package of the current article (see Section 6). Pay special attention to the fact that the adjoints must be initialized to zero (on line 17) because most RMD-operations update (i.e., add to) the adjoints. Note also how the *sel*-parameter is used to suppress computation of adjoints that are not needed; on line 19, only ui is computed and only ki on line 24. These statements are in the subroutine demo2_likely in the demo directory of the package.

Another implementation of the algorithm is in run_groundwater and groundw_sumsq.m in the package's Matlab directory. These programs are compatible with both Matlab and Octave. The Matlab/Octave programs call the minimization function fminunc (built into Octave and in Matlab's

```
1  ! COMPUTE SUM-OF-SQUARES
2  B = A0;
3  do j=1,m
4    call daxpy(n*n, k(j), A(1,1,j), 1, B, 1)         ! B = B + k(j)*A(:,:,j);
5  enddo
6  g = f0;
7  call dgemv('n', n, m, 1d0, F, n, k, 1, 1d0, g, 1)  ! g := F*k + g
8  call dgetrf(n, n, B, n, ipiv, info)            ! linsys: PLU-factorize B
9  if (info /= 0) stop 'singular matrix'          ! linsys
10 u = g                                          ! linsys
11 call dgetrs('n', n, 1, B, n, ipiv, u, n, info) ! linsys: Solve B*u = g
12 r = d;
13 call dgemv('n', s, n, 1d0, C, s, u, 1, -1d0, r, 1) ! r := C*u - d
14 theta = ddot(s, r, 1, r, 1);
15
16 ! COMPUTE INFLUENCE OF k
17 thetai = 1; ri = 0; ui = 0; Bi = 0; ki = 0
18 call ddot_rmd(s, r, 1, r, 1, thetai, ri, ri, '11')
19 call dgemv_rmd('n', s, n, 1d0, C, s, u, 1, -1d0, 1, dummy, ui, ri, '010')
20 z = ui
21 call dgetrs('t', n, 1, B, n, ipiv, z, n, info)  ! linsys_rmd: Solve B'*z = ui
22 gi = z                                          ! linsys_rmd
23 call dger(n, n, -1d0, z, 1, u, 1, Bi, n)        ! linsys_rmd: Bi := -z*u'
24 call dgemv_rmd('n', n, m, 1d0, F, n, k, 1, 1d0, 1, dummy, ki, gi, '010')
25 do j=m,1,-1
26   ki(j) = ki(j) + ddot(n*n, Bi, 1, A(1,1,j), 1)
27 enddo
```

Fig. 2. Computational part of a Fortran implementation of Algorithm 1.

optimization toolbox) to determine the parameters of a simple two-parameter groundwater model, explained in comments at the beginning of `run_groundwater`.

5.2 Example 2: Maximum Likelihood for Vector-Autoregressive Time Series

When estimating the parameters of the vector-autoregressive one-term (VAR(1)) time series

$$x_t = Ax_{t-1} + \varepsilon_t \quad (x, \varepsilon\ r\text{-vectors}, A\ r \times r \text{ matrix})$$
$$\Sigma = \mathrm{Cov}(\varepsilon_t, \varepsilon_t)$$

from given data $x_t, t = 0, 1, \ldots, n$ one seeks to maximize the log-likelihood value

$$\begin{aligned} l(A, \Sigma) &= -\tfrac{1}{2}(nr \log 2\pi + \log\det\Omega + w^T\Omega^{-1}w) \\ &= -\tfrac{1}{2}\left(nr \log 2\pi + \log\det S_0 + (n-1)\log\det\Sigma + w_1^T\Omega^{-1}w_1 + \sum_{t=2}^{n} w_t^T\Sigma^{-1}w_t\right) \\ &= -\tfrac{1}{2}\left(nrlog2\pi + 2\sum_{t=0}^{n}(\log l_{S,ii} + (n-1)\log l_{\Sigma,ii}) + \sum_{t=1}^{n} u_t^T u_t\right); \end{aligned}$$

thus, the adjoint of A and Σ on l is sought. Here,

$$\Omega = \begin{bmatrix} S_0 & & & \\ & \Sigma & & \\ & & \ddots & \\ & & & \Sigma \end{bmatrix} \quad rn \times rn \text{ block diagonal matrix,}$$

where $S_0 = \mathrm{Cov}(x_t, x_t)$, $w = (w_1^T, \ldots, w_n^T)^T$ is an rn-vector, $w_1 = x_1$, and $w_t = x_t - Ax_{t-1}, t = 2, \ldots, n$. Moreover, $L_S = (l_{S,ij})_{r\times r}$ and $L_\Sigma = (l_{\Sigma,ij})_{r\times r}$ are the Cholesky-factors of S_0 and

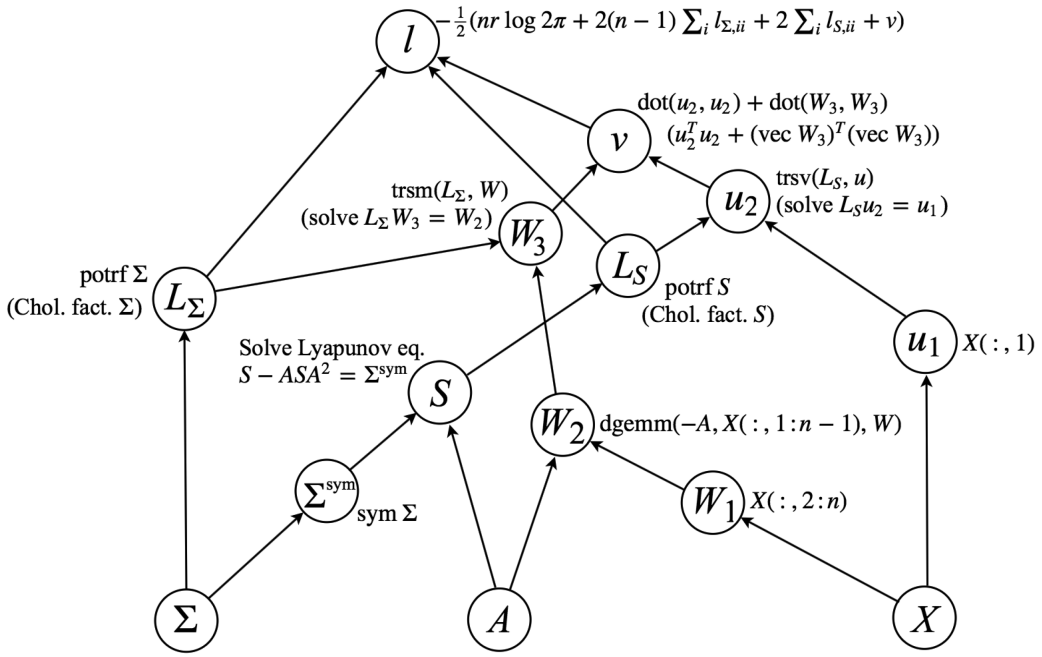

Fig. 3. Computational tree for Example 2. Subscripts $(W_1, W_2, W_3; u_1, u_2)$ denote items that are updated (i.e., overwritten) by the corresponding BLAS operations. The initial quantities are the lower triangle of Σ (covariances), A (coefficients), and X (time series data), and the final quantity is the likelihood l.

Σ respectively, and $u_1 = L_S^{-1} w_1$, $u_t = L_\Sigma^{-1} w_t$, $t = 2, \ldots, n$. Furthermore, $S_0 = AS_1^T = \Sigma$, where $S_1 = \mathrm{Cov}(x_t, x_{t-1}) = AS_0$. Hence, S_0 is the solution to the discrete Lyapunov equation

$$S_0 - AS_0A^T = \Sigma.$$

Figure 3 shows the computation tree of these formulae. Their Fortran implementation is in the demo directory in `demo2_driver.f90` and `demo2_likely.f90`, and a program to solve the discrete Lyapunov equation is in `dlyap.f90`. Another implementation, for Matlab/Octave is in `demo_var.m`, `var1_likelihood.m` and `dlyap1.m`. In these programs, care is taken not to overwrite Σ and S when computing their Cholesky factors because they are needed for the adjoints. Further details are not given in this article, but the reader is encouraged to study these programs, and compare with Figure 3.

The demo directory also contains a shell script to measure the runtime of Example 2. For such timing to be realistic, it is important to solve the discrete Lyapunov equation efficiently, and to do that, one can use the Fortran program SB03MD from the Subroutine Library in Control Theory (SLICOT) package [3]. This program is GNU-licensed and not included, but a shell script to download it from github is provided. We return to the timing of Example 2 in Section 6.3.

6 THE FORTRAN SUBROUTINE PACKAGE

6.1 Overview

The present software package consists of a set of 52 Fortran subroutines that implement the adjoint formulae given in Tables 3–5, as well as the adjoint of potrf from LAPACK. The package is accompanied by a reference manual and a user manual. The reference manual has an introduction, which describes the package design and explains its use, followed by detailed documentation for

Table 8. Timing of Individual Operations

Computer, CPU, Speed	BLAS library	N. of threads	Matrix size	Time of BLAS operation (BLAS-time, ps/flop)				Time of RMD-operation (multiple of BLAS-time)			
				dgemv	dgemm	dsyrk	dtrsm	dgemv	dgemm	dsyrk	dtrsm
Xeon, E5-2640, 2.6 GHz	MKL	1	200	212	47	55	61	2.7	2,0	1.9	4.3
Xeon, E5-2650, 2.0 GHz	MKL	1	20	1467	490	892	667	2,0	1.8	1.4	2.3
Xeon, E5-2650, 2.0 GHz	**MKL**	**1**	**200**	**944**	**116**	**153**	**132**	**2.3**	**1.9**	**1.6**	**3.6**
Xeon, E5-2650, 2.0 GHz	MKL	1	2000	623	97	107	107	2.9	2,0	1.9	8.2
Xeon, E5-2650, 2.0 GHz	MKL	4	200	823	42	51	61	1.9	1.8	2.3	5.5
Xeon, E5-2650, 2.0 GHz	MKL	16	200	978	30	38	53	2.4	1.8	3.1	7.9
Xeon, E5-2650, 2.0 GHz	MKL	16	2000	400	10	9	18	3.7	2,0	3,0	2.8
Xeon, E5-2650, 2.0 GHz	OpenBLAS	1	200	291	110	126	177	2.8	2,0	1.8	2.7
Xeon, E5-2650, 2.0 GHz	Ref. BLAS	1	200	634	621	680	1194	2.8	2.9	2,0	1.5
Mac, Core-i5, 2.9 GHz	Accelerate	1	200	235	35	48	63	1.7	2,0	1.8	5,0
Mac, Core-i5, 2.9 GHz	MKL	1	200	156	45	52	57	2.5	2,0	2,0	3.6
Mac, Core-i5, 2.9 GHz	OpenBLAS	1	200	775	68	145	127	11.6	1.9	1.6	3.2
Mac, Core-i5, 2.9 GHz	OpenBLAS	1	2000	430	44	52	62	4.5	2.2	2.1	9.3
Ubuntu, Core-i5, 3.4 GHz	MKL	1	200	215	76	101	84	2.6	2,0	1.7	3.7

All runtimes were obtained with the `timermd` program, compiled with `gfortran -O3`. Operations were applied repeatedly on several matrices spread over memory; therefore, the results include time for reading from main memory into CPU-cache. See main text for discussion.

each subroutine. The user manual describes the directory structure of the package, its installation and compilation, as well as the execution of the packages's example, testing and timing programs. Additional accompanying files include a document generation script, a precision conversion script, as well as a few Matlab programs for comparison. Also included are selected parts of two packages of the first author, for displaying matrices [22] and for time series modeling [21]. The package is referred to as BLAS-RMD in its documentation, and individual subroutines as RMD-routines.

6.2 Portability and Standard Compliance

The subroutines are written in Fortran 2003, and have been executed using Intel's ifort, the GNU gfortran, and NAG's nagfor compilers. Results of running gfortran with switches that check standard compliance are in the evidence directory, as well as results from nagfor with maximum warnings and runtime checks. The same directory contains log-files of successful runs with gfortran and ifort on three different platforms (Ubuntu, CentOS, and macOS) with four different BLAS versions (OpenBLAS [2], Apple's Accelerate, Intel's MKL [1], and the Netlib reference BLAS [6]).

6.3 Timing

A timing program, `timermd.f90`, accompanies the package, in a folder named `time`. Once made, a summary of usage is obtained by running it without parameters. By default, the program times four operations, the level 2 BLAS `dgemv` and the level 3 BLAS `dgemm`, `dsyrk`, and `dtrsm`. The last one is an example of an operation involving a level drop for the RMD-routine; cf. Table 6.

The `timermd` program was run on several combinations of BLAS versions, number of threads, matrix sizes, and platforms, and the results are summarized in Table 8, with more details in the subdirectory `time/reports`. Since it is not the purpose of this article to do a comprehensive timing study, not all combinations were timed. Instead, a base combination was chosen, involving MKL, with one thread, 200 by 200 matrices, on a 2.0-GHz Xeon Linux workstation (bold line in the table). The other timed combinations changed one of these four at a time, with a few exceptions.

Table 9. Runtimes for Example 2

		Computation time (s)			
		1 thread		8 threads	
r	n	likelihood	adjoint	likelihood	adjoint
1000	100000	20	66	7	26
1000	500000	97	308	27	94
5000	100000	819	2223	356	837

The model is an r-dimensional VAR(1) model of a time series of length n. It involves a symmetric $r \times r$ parameter matrix Σ, an $r \times n$ parameter matrix A, and an $r \times n$ data matrix X. Its computation involves five different BLAS operations. The table shows the runtime (in seconds) of computing one function value and one adjoint value on a 2 GHz Linux computer using MKL.

The first thing to notice in Table 8 is that for level 3 BLAS with no level-drop (`dgemm` and `dsyrk`), the time for the RMD operations was on average very close to twice that of the corresponding BLAS; in fact, the average factor for all the shown combinations is 2.02. For the level 2 `dgemv`, there was more difference, 3-fold on average, and with the level-drop of `dtrsm`, the difference was 4.5-fold on average. Similar factors for the level 2 `dtrmv` and `dgbmv`, which drop to level 1 for the RMD, were measured (a somewhat smaller factor was obtained; data not shown).

Comparing BLAS versions on the Linux workstation, MKL was about six times faster than the reference BLAS for level 3; in general, the performance of OpenBLAS was similar to that of MKL. For the level 2 `dgemv`, the story was somewhat different, with OpenBLAS fastest and MKL slowest. Apple's Accelerate performed similarly to MKL on the Macintosh, but OpenBLAS was considerably slower. The largest RMD factor in the table, 11.6, occurred with `dgemv`/OpenBLAS/Mac and $n = 200$. That combination with $n = 2{,}000$ was therefore also run, giving somewhat better performance.

The comparison with different matrix sizes (lines 2–4 in the table) did not reveal anything unexpected. However, the speedup with increased number of threads was rather limited. For level 3 and $n = 200$ going from 1 thread to 4, an average level 3 speedup factor of 2.6 was given, and for 16 threads 3.3. To see if there is improvement for larger matrices, 16 threads with $n = 2{,}000$ was also timed, and improvement was indeed confirmed with an average speedup factor of 8.4.

For a somewhat more realistic setting, we have also measured the runtimes of Example 2 for a selection of parameters on the 2.0 GHz Linux workstation using gfortran and MKL. The results are shown in Table 9. The observed runtimes show that it would indeed be realistic to use numerical optimization, such as the Polak-Ribiere method, to estimate the parameters of these models. We note that the adjoint takes more than twice the time of the function value, which is expected, since for three of the five level 3 BLAS operations in the example, the adjoint involves a drop to level 2. The same fact may explain why the full power of the eight threads is not attained.

7 CONCLUSIONS

This article has presented a systematic overview of formulae for reverse mode AD of matrix operations, with special emphasis on the elementary BLAS operations, which have been implemented in the accompanying subroutine package. By expressing these formulae in matrix terms, one can make use of the BLAS operations themselves when implementing the derivatives into programs, thus ensuring effective running times. This is borne out by the timing study and examples, which show that the gradients of functions involving chiefly matrix operations can be evaluated in about

twice the time of a function evaluation by such programs. In order to achieve this, it has also been important to pay attention to the special forms of matrices taken care of in the BLAS.

The awareness of the potential benefits of automatic differentiation in the numerical and statistical software community is clearly increasing. The possibility of using computation trees with matrix nodes (instead of scalar) is also appearing more often in the AD literature. Nevertheless, current numerical optimization software often uses expensive numerical gradients by default, costing n or $2n$ function evaluations. There are several scenarios in which our package can significantly aid in improving such optimization software and its use. Firstly, scientific programmers can use BLAS-RMD manually in the vein of the examples of Section 5. It could also be incorporated into software that would generate derivative code automatically from programs that call BLAS. Yet another possibility is to use BLAS-RMD for reference when software for automatic differentiation of matrix operations is being constructed.

ACKNOWLEDGMENT

We thank an anonymous referee for a very thorough and helpful review.

REFERENCES

[1] 2017. Intel Math Kernel Library. Retrieved October 13, 2017 from https://software.intel.com/en-us/mkl.
[2] 2017. OpenBLAS: An optimized BLAS library. Retrieved December 4, 2017 from http://www.openblas.net/.
[3] Ai Barraud. 1977. A numerical algorithm to solve $A^T XA - X = Q$. *IEEE Transactions on Automatic Control* 22, 5 (1977), 883–885.
[4] Thomas F. Coleman and Arun Verma. 1998. ADMAT: An automatic differentiation toolbox for MATLAB. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing, SIAM, Philadelphia, PA*, Vol. 2.
[5] Thomas F. Coleman and Wei Xu. 2016. *Automatic Differentiation in MATLAB Using ADMAT with Applications*. SIAM.
[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.
[7] Jack J. Dongarra and David W. Walker. 1995. Software libraries for linear algebra computations on high performance computers. *SIAM Review* 37, 2 (1995), 151–180.
[8] Paul S. Dwyer and M. S. MacPhail. 1948. Symbolic matrix derivatives. *The Annals of Mathematical Statistics* 19, 4 (1948), 517–534.
[9] Shaun A. Forth. 2006. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)* 32, 2 (2006), 195–222.
[10] Ralf Giering and Thomas Kaminski. 1998. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software (TOMS)* 24, 4 (1998), 437–474.
[11] M. B. Giles. 2008. *An Extended Collection of Matrix Derivative Results for Forward and Reverse Mode Automatic Differentiation*. Technical Report. Oxford University Computing Laboratory.
[12] Mike B. Giles. 2008. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke (Eds.). Springer, Berlin, 35–44.
[13] Kazushige Goto and Robert van de Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)* 35, 1 (2008), 4.
[14] Andreas Griewank. 1989. On automatic differentiation. In *Mathematical Programming*, Masao Iri and K. Tanabe (Eds.). Kluwer Academic Publishers, Dordrecht, 83–108.
[15] Andreas Griewank. 2003. A mathematical view of automatic differentiation. In *Acta Numerica*. Vol. 12. Cambridge University Press, Cambridge, 321–398.
[16] Andreas Griewank. 2012. Who invented the reverse mode of differentiation? *Documenta Mathematica, Extra Volume ISMP* (2012), 389–400.
[17] Andreas Griewank. 2015. Automatic differentiation, Mark R. Dennis, Paul Glendinning, Paul A. Martin, Fadil Santosa, and Jared Tanner (Eds.). Princeton University Press, 749–752.
[18] Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software* 22, 2 (1996), 131–167.
[19] Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2nd ed.). Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA.

[20] Masao Iri. 1991. History of automatic differentiation and rounding error estimation. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Andreas Griewank and George F. Corliss (Eds.). SIAM, Philadelphia, 1–16.
[21] Kristjan Jonasson. 2008. Algorithm 878: Exact VARMA likelihood and its gradient for complete and incomplete data with Matlab. *ACM Transactions on Mathematical Software (TOMS)* 35, 1 (2008), 6:1–6:11.
[22] Kristjan Jonasson. 2009. Algorithm 892: DISPMODULE, a Fortran 95 module for pretty-printing matrices. *ACM Transactions on Mathematical Software (TOMS)* 36, 1 (2009), 6:1–6:7.
[23] Gershon Kedem. 1980. Automatic differentiation of computer programs. *ACM Trans. Math. Software* 6, 2 (June 1980), 150–165.
[24] Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.
[25] Uwe Naumann. 2008. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming* 112, 2 (2008), 427–441.
[26] Uwe Naumann. 2012. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* Siam, Philadelphia, PA.
[27] S. P. Smith. 1995. Differentiation of the Cholesky algorithm. *Journal of Computational and Graphical Statistics* 4, 2 (1995), 134–147.
[28] Arun Verma. 1998. *Structured Automatic Differentiation.* Ph.D. Dissertation. Department of Computer Science, Cornell University, Ithaca, NY.
[29] Arun Verma. 1999. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, M. E. Henderson, C. R. Anderson, and S. L. Lyons (Eds.). SIAM, Philadelphia, 174–183.
[30] Sebastian F. Walter and Lutz Lehmann. 2013. Algorithmic differentiation in Python with AlgoPy. *Journal of Computational Science* 4, 5 (2013), 334–344.
[31] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM, 25:1–25:12.
[32] Matthew J. Weinstein and Anil V. Rao. 2017. Algorithm 984: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in matlab using source transformation via operator overloading. *ACM Transactions on Mathematical Software (TOMS)* 44, 2 (2017), 21:1–21:25.
[33] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27, 1 (2001), 3–35.

Received December 2017; revised August 2019; accepted August 2019