AN ALTERNATIVE NOTATION FOR REPRESENTING DENSE LINEAR ALGEBRA ALGORITHMS *

PAOLO BIENTINESI* AND ROBERT A. VAN DE GEIJN*

Abstract. We present a notation that allows a dense linear algebra algorithm to be represented in a way that is visually recognizable. The primary value of the notation is that it exposes subvectors and submatrices allowing the details of the algorithm to be the focus while hiding the intricate indices related to the arrays in which the vectors and matrices are stored. The applicability of the notation is illustrated through a succession of progressively complex case studies ranging from matrix-vector operations to the chasing of the bulge of the symmetric QR iteration. The notation facilitates comparing and contrasting different algorithms for the same operation as well as similar algorithms for different operations. Finally, we point out how algorithms represented with this notation can be directly translated into high-performance code.

Key words. linear algebra, notation, algorithms, dense matrices, FLAME

AMS subject classifications. 65F05, 65Y10.

1. Introduction.

This paper showcases an alternative notation for representing, visually, algorithms for dense linear algebra operations. This notation has been effective for presenting well-known linear algebra algorithms in undergraduate and graduate courses and for presenting new algorithms in our recent papers [16, 18, 19, 27]. In addition, it has facilitated the systematic and automatic derivation of such algorithms [2, 4, 6, 14, 15, 26]. The strength of the notation is that it avoids indexing and that much of the high-level information that supports the algorithm is captured.

Standard texts in numerical linear algebra, like those written by Golub and Van Loan [13], Demmel [7], Stewart [28], and many others, often develop the theory behind algorithms by partitioning matrices and then discussing how different submatrices are affected, updated, and/or used *without the use of indices for exposing individual entries in matrices.* By contrast, algorithms in these same texts are typically expressed by exposing intricate indices into the arrays that store matrices and vectors. The popularity of Matlab's M-script language [25] for prototyping new algorithms supports and reinforces that view of how algorithms must be expressed: Algorithms are often presented in texts and other papers using "Matlab-like" notation.

The disparity between how one reasons about algorithms and how they are then represented may have a number of possible roots. First, the area of numerical linear algebra developed at a time when typesetting papers and books was expensive. Because of this, it was important to represent algorithms concisely, in as little space as possible. We will see that representing algorithms with explicit indexing has this property. Second, the coding style for implementing the algorithms traditionally exposes explicit indexing. As a result, compilers were written to optimize code that exposed indexing. This then encouraged programmers to continue to write code in the same fashion, since compilers were reasonably good at optimizing such code. Papers and books have continued to present algorithms with exposed indices since it is perceived that this makes it easier for the reader to translate the algorithms to such traditional code. Finally, we have noticed that there are a number of people who reason about matrix algorithms in terms of indices. For them the traditional way of representing

[1]Department of Computer Sciences, The University of Texas, Austin, TX 78712; phone: 512-471-9720, fax: 512-471-8885, {pauldj,rvdg}@cs.utexas.edu.

* This work was supported in part by NSF grants ACR-0305163 and CCF-0342369.

algorithms is more natural.

We will show that if concepts are naturally expressed via the partitioning of matrices (and/or vectors), and are often accompanied by pictures that focus on submatrices being updated and/or used, then algorithms can (and hould) similarly be expressed by tracking partitioned matrices (and/or vectors). Similarly, the statement of theorems and their proofs often involve the partitioning of matrices and vectors. It is these observation that naturally leads to a notation for expressing algorithms that deviates dramatically from the norm and from how algorithms are traditionally represented in code. The same observation should motivate how algorithms are to be coded, for example through Application Programming Interfaces (APIs) that mirror the notation used to express the algorithms [5].

The paper is organized as follows: Section 2 is a motivating example: first, it shows how algorithms for the Cholesky factorization are traditionally derived and represented; then, by contrast, the same algorithms are also displayed by means of a new representation. Sections 3, 4 and 5 are a parade of case studies to which the new notation is applied, starting from simple matrix-vector and matrix-matrix operations (Section 3), passing through the most common matrix factorizations (Section 4), and concluding with the more complicated QR algorithm (Section 5). Section 6 is a short discussion on a coding interface that closely mirrors the newly introduced notation, while Section 7 gives a final summary and comments.

2. A Motivating Example. In this section, the Cholesky factorization is used as a motivating example. We present a very quick review of what this operation entails, the theory that supports it, the classical derivation of one algorithm for computing it, how to attain high performance by casting the computation in terms of operations related to matrix-matrix multiplication, and finally our notation for expressing the algorithm.

2.1. The Cholesky factorization. This operation is only defined for symmetric positive definite matrices:

DEFINITION 2.1. *A matrix $A \in \mathbb{R}^{n\times n}$ is positive definite if and only if for all nonzero vectors $x \in \mathbb{R}^n$ it is the case that $x^T Ax > 0$.*

We will prove the following theorem in Section 2.4:

THEOREM 2.2 (Cholesky Factorization Theorem). *Given a symmetric positive definite (SPD) matrix A there exists a lower triangular matrix L such that $A = LL^T$.*
The lower triangular matrix L is known as the Cholesky factor and LL^T is known as the Cholesky factorization of A. It is unique if the diagonal elements of L are restricted to be positive.

The operation that overwrites the lower triangular part of matrix A with its Cholesky factor will be denoted by $A := \Gamma(A)$, which should be read as "A becomes its Cholesky factor." Typically, only the lower (or upper) triangular part of A is stored, and it is that part that is then overwritten with the result. In this discussion, we will assume that the lower triangular part of A is stored and overwritten.

2.2. Application. The Cholesky factorization is used to solve the linear system $Ax = y$ when A is SPD: Substituting the factors into the equation yields $LL^T x = y$. Letting $z = L^T x$,

$$Ax = L \underbrace{(L^T x)}_{z} = Lz = y.$$

for $j = 1 : n$
  $\alpha_{j,j} := \sqrt{\alpha_{j,j}}$
 	
  for $i = j+1 : n$
    $\alpha_{i,j} := \alpha_{i,j}/\alpha_{j,j}$
  endfor
 	
  for $k = j+1 : n$
    for $i = k : n$
      $\alpha_{i,k} := \alpha_{i,k} - \alpha_{i,j}\alpha_{k,j}$
    endfor
  endfor
endfor
	for $j = 1 : n$
	  $\alpha_{j,j} := \sqrt{\alpha_{j,j}}$
	
	$\Big\}$ $\alpha_{j+1:n,j} := \alpha_{j+1:n,j}/\alpha_{j,j}$
	
	$\Bigg\}$ $\alpha_{j+1:n,j+1:n} :=$
	    $\alpha_{j+1:n,j+1:n} - \text{TRIL}(\alpha_{j+1:n,j}\alpha^T_{j+1:n,j})$
	endfor

FIG. 2.1. *Formulations of the Cholesky factorization that expose indices.*

Thus, z can be computed by solving the triangular system of equations $Lz = y$ and subsequently the desired solution x can be computed by solving the triangular linear system $L^T x = z$.

2.3. Unblocked algorithm. The most common algorithm for computing $A := \Gamma(A)$ can be derived as follows: Consider $A = LL^T$. Partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right). \tag{2.1}$$

Remark 1. *We adopt the commonly used notation where Greek lower case letters refer to scalars, lower case letters refer to (column) vectors, and upper case letters refer to matrices. The $\star$ refers to a part of A that is neither stored nor updated.*

By substituting these partitioned matrices into $A = LL^T$ we find that

$$\left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)\left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)^T = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21}l_{21}^T + L_{22}L_{22}^T \end{array}\right),$$

from which we conclude that

$$\begin{array}{c|c} \lambda_{11} = \sqrt{\alpha_{11}} & \star \\ \hline l_{21} = a_{21}/\lambda_{11} & L_{22} = \Gamma(A_{22} - l_{21}l_{21}^T) \end{array}.$$

These equalities motivate the algorithm

1. Partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. Overwrite $\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}}$.
3. Overwrite $a_{21} := l_{21} = a_{21}/\lambda_{11}$.
4. Overwrite $A_{22} := A_{22} - l_{21}l_{21}^T$ (updating only the lower triangular part of A_{22}).
5. Continue with $A = A_{22}$. (Back to Step 1.)

The algorithm is typically presented in a text using Matlab-like notation as illustrated in Fig. 2.1.

Remark 2. *Similar to the* `tril` *function in Matlab, we use* TRIL(B) *to denote the lower triangular part of matrix B.*

2.4. Proof of the Cholesky Factorization Theorem. In this section, we partition A as in (2.1). The following lemmas, which can be found in any standard text, are key to the proof:

LEMMA 2.3. *Let $A \in \mathbb{R}^{n\times n}$ be SPD. Then $\alpha_{11} > 0$.*

Proof: Choose $x = \left(\begin{array}{c} 1 \\ \hline 0 \end{array}\right)$ where 0 represents the vector of $n-1$ zeroes. Then

$$0 < x^T A x = \left(\begin{array}{c} 1 \\ \hline 0 \end{array}\right)^T \left(\begin{array}{c|c} \alpha_{11} & a_{21}^T \\ \hline a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c} 1 \\ \hline 0 \end{array}\right) = \alpha_{11},$$

which proves the lemma. □

LEMMA 2.4. *Let $A \in \mathbb{R}^{n\times n}$ be SPD and $l_{21} = a_{21}/\sqrt{\alpha_{11}}$. Then $A_{22} - l_{21} l_{21}^T$ is SPD.*

Proof: Since A is symmetric so are A_{22} and $A_{22} - l_{21} l_{21}^T$. Let $x_1 \neq 0$ be an arbitrary vector of length $n-1$. Define $x = \left(\begin{array}{c} \chi_0 \\ \hline x_1 \end{array}\right)$ where $\chi_0 = -x_1^T a_{21}/\alpha_{11}$. Then, since $x \neq 0$,

$$\begin{aligned} 0 < x^T A x &= \left(\begin{array}{c} \chi_0 \\ \hline x_1 \end{array}\right)^T \left(\begin{array}{c|c} \alpha_{11} & a_{21}^T \\ \hline a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c} \chi_0 \\ \hline x_1 \end{array}\right) = x_1^T (A_{22} - \frac{a_{21} a_{21}^T}{\alpha_{11}}) x_1 \\ &= x_1^T (A_{22} - l_{21} l_{21}^T) x_1. \end{aligned}$$

We conclude that $A_{22} - l_{21} l_{21}^T$ is SPD. □

Proof: [Cholesky Factorization Theorem] Proof by induction.
Base case: $n = 1$ Clearly the result is true for a 1×1 matrix $A = \alpha_{11}$: In this case, the fact that A is SPD means that $\alpha_{11} > 0$ and its Cholesky factor is then given by $\lambda_{11} = \sqrt{\alpha_{11}}$.
Inductive step: Assume the result is true for SPD matrix $A \in \mathbb{R}^{(n-1)\times(n-1)}$. We will show that it holds for $A \in \mathbb{R}^{n\times n}$. Let $A \in \mathbb{R}^{n\times n}$ be SPD. Partition A and L as in (2.1) and let $\lambda_{11} = \sqrt{\alpha_{11}}$ (which is well-defined by Lemma 2.3), $l_{21} = a_{21}/\lambda_{11}$, and $L_{22} = \Gamma(A_{22} - l_{21} l_{21}^T)$ (which exists thanks to Lemma 2.4 and the induction hypothesis). Then L is the desired Cholesky factor of A.
By the principle of mathematical induction, the theorem holds. □

2.5. Blocked algorithm. In order to attain high performance, the computation is cast in terms of matrix-matrix multiplication by so-called blocked algorithms. For the Cholesky factorization a blocked version of the algorithm can be derived by partitioning

$$A \rightarrow \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L \rightarrow \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right),$$

where A_{11} and L_{11} are $b \times b$. By substituting these partitioned matrices into $A = LL^T$ we find that

$$\left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right)^T = \left(\begin{array}{c|c} L_{11} L_{11}^T & \star \\ \hline L_{21} L_{11}^T & L_{21} L_{21}^T + L_{22} L_{22}^T \end{array}\right).$$

From this we conclude that

$$\begin{array}{c|c} L_{11} = \Gamma(A_{11}) & \star \\ \hline L_{21} = A_{21} L_{11}^{-T} & L_{22} = \Gamma(A_{22} - L_{21} L_{21}^T) \end{array}.$$

$$
\begin{array}{l}
\textbf{for } j = 1 : n \textbf{ in steps of } n_b \\
\quad b := \min(n - j + 1, n_b) \\
\quad \begin{array}{lcl}
A_{j:j+b-1,j:j+b-1} & := & \Gamma(A_{j:j+b-1,j:j+b-1}) \\
A_{j+b:n,j:j+b-1} & := & A_{j+b:n,j:j+b-1} A^{-T}_{j:j+b-1,j:j+b-1} \\
A_{j+b:n,j+b:n} & := & A_{j+b:n,j+b:n} - \text{TRIL}(A_{j+b:n,j:j+b-1} A^{T}_{j+b:n,j:j+b-1})
\end{array} \\
\textbf{endfor}
\end{array}
$$

FIG. 2.2. *Blocked algorithm for computing the Cholesky factorization. Here n_b is the block size used by the algorithm.*

An algorithm is then described by the steps

1. Partition $A \rightarrow \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array} \right)$, where A_{11} is $b \times b$.
2. Overwrite $A_{11} := L_{11} = \Gamma(A_{11})$.
3. Overwrite $A_{21} := L_{21} = A_{21} L_{11}^{-T}$.
4. Overwrite $A_{22} := A_{22} - L_{21} L_{21}^{T}$ (updating only the lower triangular part).
5. Continue with $A = A_{22}$. (Back to Step 1.)

An algorithm that explicitly indexes into the array that stores A is given in Fig. 2.2.

> **Remark 3.** *The Cholesky factorization $A_{11} := L_{11} = \Gamma(A_{11})$ can be computed with the unblocked algorithm or by calling the blocked Cholesky factorization algorithm recursively.*
> *Operations like $L_{21} = A_{21} L_{11}^{-T}$ are computed by solving a linear system with multiple right-hand sides (*TRSM*). See also Section 3.2.*

2.6. Alternative representation. When explaining the above algorithm in a classroom setting, invariably it is accompanied by a picture sequence like the one in Fig. 2.3(left) and the (verbal) explanation:

Beginning of iteration: At some stage of the algorithm (Top of the loop), the computation has moved through the matrix to the point indicated by the thick lines. Notice that we have finished with the parts of the matrix that are in the top-left, top-right (which is not to be touched), and bottom-left quadrants. The bottom-right quadrant has been updated to the point where we only need to perform a Cholesky factorization of it.

Repartition: We now repartition the bottom-right submatrix to expose α_{11}, a_{21}, and A_{22}.

Update: α_{11}, a_{21}, and A_{22} are updated as discussed before.

End of iteration: The thick lines are moved, since we now have completed more of the computation, and only a factorization of A_{22} (which becomes the new bottom-right quadrant) remains to be performed.

Continue: The above steps are repeated until the submatrix A_{BR} is empty.

To motivate our notation, we annotate this progression of pictures as in Fig. 2.3 (right). In those pictures, "T", "B", "L", and "R" stand for "Top", "Bottom", "Left", and "Right", respectively. This then motivates the format of the algorithm in Fig. 2.4 (left). A similar explanation can be given for the blocked algorithm, which is given in Fig. 2.4 (right). In the algorithms, $m(A)$ indicates the number of rows of matrix A.

> **Remark 4.** *The indices in our more stylized presentation of the algorithms are subscripts rather than indices in the conventional sense.*

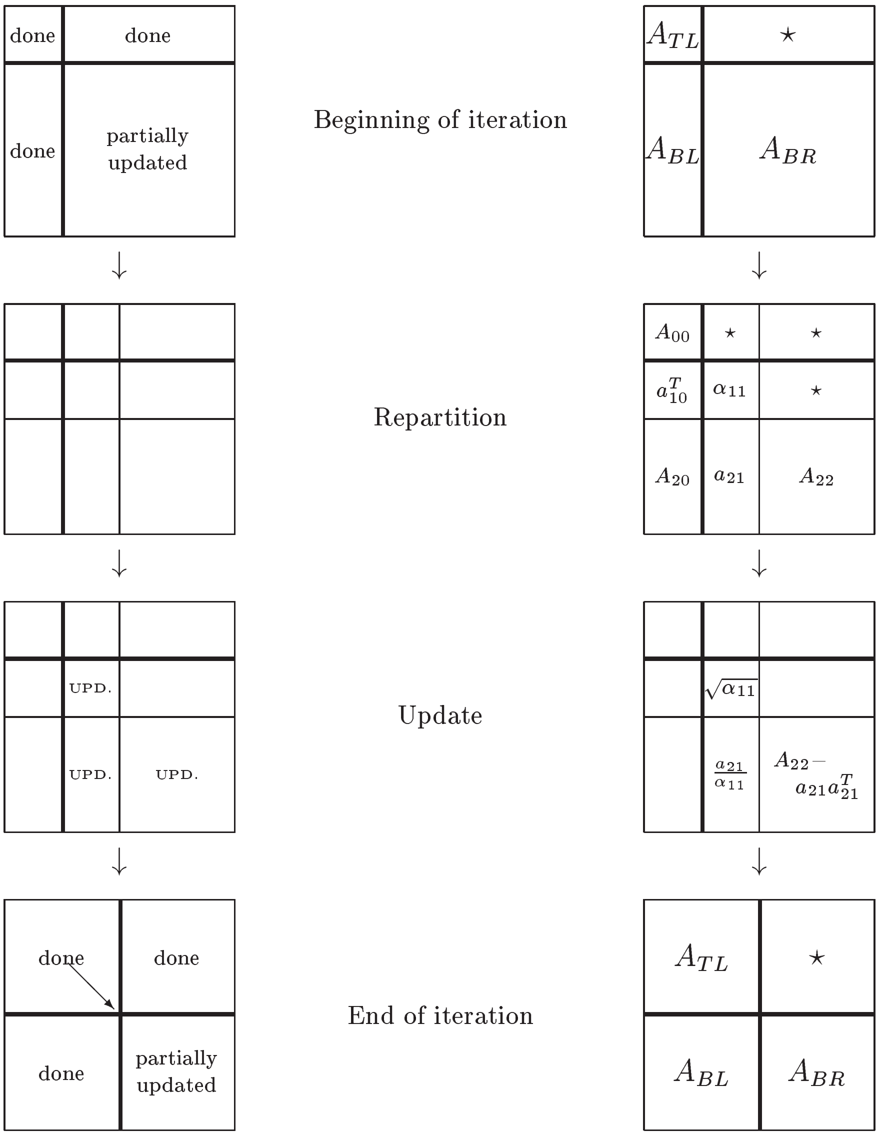

FIG. 2.3. *Left: Progression of pictures that explain Cholesky factorization algorithm. Right: Same pictures, annotated with labels and updates.*

Remark 5. *Clearly Fig. 2.4 does not present the algorithm as concisely as the algorithms given in Figs. 2.1 and 2.2. However, it does capture to a large degree the verbal description of the algorithm mentioned above and therefore, in our opinion, reduces both the effort required to interpret the algorithm and the need for additional explanations. The notation also mirrors that used for the proof in Section 2.4.*

Algorithm: $A := \text{CHOL\_UNB}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**

  Repartition

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline a_{10}^T & \alpha_{11} & \star \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

  where α_{11} is 1×1

$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21}/\alpha_{11}$
$A_{22} := A_{22} - \text{TRIL}(a_{21}a_{21}^T)$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline a_{10}^T & \alpha_{11} & \star \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

endwhile

Algorithm: $A := \text{CHOL\_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

  where A_{11} **is** $b \times b$

$A_{11} := \Gamma(A_{11})$
$A_{21} := A_{21}\,\text{TRIL}(A_{11})^{-T}$
$A_{22} := A_{22} - \text{TRIL}(A_{21}A_{21}^T)$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

FIG. 2.4. *Unblocked and blocked algorithms for computing the Cholesky factorization.*

Remark 6. *The notation in Figs. 2.3 and 2.4 allows the contents of matrix A at the beginning of the iteration to be formally stated:*

$$A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL} & \star \\ \hline L_{BL} & \hat{A}_{BR} - \text{TRIL}(L_{BL}L_{BL}^T) \end{array}\right),$$

where $L_{TL} = \Gamma(\hat{A}_{TL})$, $L_{BL} = \hat{A}_{BL}L_{TL}^{-T}$, and $\hat{A}_{TL}, \hat{A}_{BL}$ and $\hat{A}_{BR}$ denote the original contents of the quadrants A_{TL}, A_{BL} and A_{BR}, respectively.

3. Case Studies, Part I: Simple Operations.

In this section, we discuss a cross-section of algorithms for simple linear algebra operations. These operations are among those supported by the widely used Basic Linear Algebra Subprograms (BLAS) [9, 8].

3.1. Matrix-vector operations.

Matrix vector operations are operations that take a matrix and one or more vectors as operands, and perform $O(n^2)$ operations when the matrix is $n \times n$. Examples include different forms of matrix-vector multiplication (MATVEC: $y := \alpha Ax + \beta y$), rank-1 update (RANK1: $A := \alpha xy + A$), and the solution of a triangular system (TRSV: $y := A^{-1}y$, where A is a triangular matrix). We will focus on this last operation.

Let U be an upper triangular matrix and consider the linear system $Ux = y$. The TRSV operation overwrites y with the solution to this linear system, $y := x = U^{-1}y$. Partitioning

$$U \rightarrow \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right), \quad x \rightarrow \left(\begin{array}{c} \chi_1 \\ \hline x_2 \end{array}\right), \quad \text{and} \quad y \rightarrow \left(\begin{array}{c} \eta_1 \\ \hline y_2 \end{array}\right),$$

Algorithm: $y := \text{TRSV}(U, y)$

Partition
$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
where U_{BR} is 0×0, y_B has 0 elements
while $m(U_{BR}) < m(U)$ **do**

Repartition
$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right), \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \eta_1 \\ \hline y_2 \end{array}\right)$$
where υ_{11} and η_1 are 1×1

Variant 1:	Variant 2:
$\eta_1 := \eta_1 - u_{12}^T y_2$	$\eta_1 := \eta_1 / \upsilon_{11}$
$\eta_1 := \eta_1 / \upsilon_{11}$	$y_0 := y_0 - u_{01} \eta_1$

Continue with
$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right), \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \eta_1 \\ \hline y_2 \end{array}\right)$$
endwhile

Algorithm: $Y := \text{TRSM}(U, Y)$

Partition
$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$, $Y \rightarrow \left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right)$
where U_{BR} is 0×0, Y_B has 0 rows
while $m(U_{BR}) < m(U)$ **do**
Determine block size b
Repartition
$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \quad \left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right) \rightarrow \left(\begin{array}{c} Y_0 \\ \hline Y_1 \\ \hline Y_2 \end{array}\right)$$
where U_{11} **is** $b \times b$, Y_1 **has** b **rows**

Variant 1:	Variant 2:
$Y_1 := Y_1 - U_{12} Y_2$	$Y_1 := U_{11}^{-1} Y_1$
$Y_1 := U_{11}^{-1} Y_1$	$Y_0 := Y_0 - U_{01} Y_1$

Continue with
$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \quad \left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right) \leftarrow \left(\begin{array}{c} Y_0 \\ \hline Y_1 \\ \hline Y_2 \end{array}\right)$$
endwhile

FIG. 3.1. *Algorithms for the solution of a triangular system.* Left: *Single right-hand side.* Right: *Multiple right-hand sides. We note that the operation $Y_1 := U_{11}^{-1} Y_1$ is typically itself implemented as the computation of the solution of the smaller triangular system $U_{11} X_1 = Y_1$, where X_1 overwrites Y_1.*

we have

$$\left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right) \left(\begin{array}{c} \chi_1 \\ \hline x_2 \end{array}\right) = \left(\begin{array}{c} \eta_1 \\ \hline y_2 \end{array}\right), \quad \text{or,} \quad \left\{ \begin{array}{rcl} \upsilon_{11} \chi_1 & = & \eta_1 - u_{12}^T x_2 \\ U_{22} x_2 & = & y_2. \end{array} \right.$$

We conclude that if x_2 has already been computed, overwriting y_2, then χ_1 can be computed by $\chi_1 = (\eta_1 - u_{12}^T y_2)/\upsilon_{11}$, overwriting η_1. The algorithm for this is given in Fig. 3.1(left), Variant 1.

An alternative algorithm can be derived as follows: Partition

$$U \rightarrow \left(\begin{array}{c|c} U_{00} & u_{01} \\ \hline 0 & \upsilon_{11} \end{array}\right), \quad x \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right), \quad \text{and} \quad y \rightarrow \left(\begin{array}{c} y_0 \\ \hline \eta_1 \end{array}\right).$$

Then

$$\left(\begin{array}{c|c} U_{00} & u_{01} \\ \hline 0 & \upsilon_{11} \end{array}\right) \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right) = \left(\begin{array}{c} y_0 \\ \hline \eta_1 \end{array}\right), \quad \text{or,} \quad \left\{ \begin{array}{rcl} U_{00} x_0 + u_{01} \chi_1 & = & y_0 \\ \upsilon_{11} \chi_1 & = & \eta_1, \end{array} \right.$$

which suggests that χ_1 can be computed by solving the second equation, overwriting η_1, after which y_0 can be updated with $y_0 - \eta_1 u_{01}$. The computation continues by solving the smaller triangular system $U_{00} x_0 = y_0$. These observations are represented in the algorithm in Fig. 3.1(left), Variant 2.

As was mentioned for the Cholesky factorization in Remark 2.6, our notation allows us to mathematically describe the contents of vector y before and after execution of the loop-iterations. Let $\hat{y}_T$ and $\hat{y}_B$ to respectively denote the original contents of y_T and y_B, then the algorithms for Variant 1 and 2 maintain in y the contents

$$\left(\frac{y_T}{y_B}\right) = \left(\frac{\hat{y}_T}{x_B}\right) \quad \text{and} \quad \left(\frac{y_T}{y_B}\right) = \left(\frac{\hat{y}_T - U_{TR}x_B}{x_B}\right),$$

respectively, where $x_B = U_{BR}^{-1}\hat{y}_B$.

3.2. Matrix-matrix operations. Matrix-matrix operations are operations that take several matrices as operands and perform $O(n^3)$ operations on $O(n^2)$ data. Examples include different forms of matrix-matrix multiplication (Matmat: $C := \alpha AB + \beta C$) and the solution of a triangular system with multiple right-hand sides (Trsm: $Y := A^{-1}Y$, where A is a triangular matrix). We will focus on this last operation as it is closely related to trsv. We will discuss algorithms that cast most computations in terms of matrix-matrix multiplications, for performance reasons [21].

Let U be an upper triangular matrix and consider the operation, Trsm, that overwrites $Y := X$ under the constraint $UX = \hat{Y}$. Again, $\hat{Y}$ denotes the original contents of matrix Y. Partition, conformally,

$$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right), \quad X \rightarrow \left(\frac{X_T}{X_B}\right), \quad Y \rightarrow \left(\frac{Y_T}{Y_B}\right), \quad \text{and} \quad \hat{Y} \rightarrow \left(\frac{\hat{Y}_T}{\hat{Y}_B}\right).$$

Then

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)\left(\frac{X_T}{X_B}\right) = \left(\frac{\hat{Y}_T}{\hat{Y}_B}\right), \quad \text{or,} \quad \left\{\begin{array}{rcl} U_{TL}X_T & = & \hat{Y}_T - U_{TR}X_B \\ U_{BR}X_B & = & \hat{Y}_B. \end{array}\right.$$

From this we conclude that X_B should be computed before $\hat{Y}_T - U_{TR}X_B$ which in turn must be computed before X_T.

Two blocked algorithms for computing this operation are given in Fig. 3.1(right). Variant 1 and Variant 2 correspond to the algorithms that maintain in Y the contents

$$\left(\frac{Y_T}{Y_B}\right) = \left(\frac{\hat{Y}_T}{X_B}\right) \quad \text{and} \quad \left(\frac{Y_T}{Y_B}\right) = \left(\frac{\hat{Y}_T - U_{TR}X_B}{X_B}\right),$$

respectively, where $X_B = U_{BR}^{-1}\hat{Y}_B$.

Remark 7. *In [2] we show how algorithms for this and other operations can be systematically derived. The new notation facilitates the methodology presented in that paper.*

4. Case Studies, Part II: Factorization Algorithms. Next, we show how our notation can be used to represent algorithms for factoring matrices. In addition to a more thorough discussion of the Cholesky factorization, we review the LU factorization (with and without pivoting) and the QR factorization via Householder transformations.

4.1. Cholesky factorization, revisited. Let us define the computation of the Cholesky factorization as the overwriting of $A := L = \Gamma(\hat{A})$ where $LL^T = \hat{A}$, the original contents of A. Partitioning the matrices we find that

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)^T =$$

Algorithm: $A := \textsc{Chol\_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**

  Repartition
  $\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline a_{10}^T & \alpha_{11} & \star \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
    where α_{11} is 1×1

  Variant 1:
  $a_{10}^T := a_{10}^T \, \textsc{tril}(A_{00})^{-T}$
  $\alpha_{11} := \sqrt{\alpha_{11} - a_{10}^T a_{10}}$

  Variant 2:
  $\alpha_{11} := \sqrt{\alpha_{11} - a_{10}^T a_{10}}$
  $a_{21} := (a_{21} - A_{20} a_{10}) / \alpha_{11}$

  Variant 3:
  $\alpha_{11} := \sqrt{\alpha_{11}}$
  $a_{21} := a_{21} / \alpha_{11}$
  $A_{22} := A_{22} - \textsc{tril}(a_{21} a_{21}^T)$

  Continue with
  $\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline a_{10}^T & \alpha_{11} & \star \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
endwhile

Algorithm: $A := \textsc{Chol\_blk}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
  Determine block size b
  Repartition
  $\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
    where A_{11} **is** $b \times b$

  Variant 1:
  $A_{10} := A_{10} \, \textsc{tril}(A_{00})^{-T}$
  $A_{11} := \Gamma(A_{11} - \textsc{tril}(A_{10} A_{10}^T))$

  Variant 2:
  $A_{11} := \Gamma(A_{11} - \textsc{tril}(A_{10} A_{10}^T))$
  $A_{21} := (A_{21} - A_{20} A_{10}^T) \, \textsc{tril}(A_{11})^{-T}$

  Variant 3:
  $A_{11} := \Gamma(A_{11})$
  $A_{21} := A_{21} \, \textsc{tril}(A_{11})^{-T}$
  $A_{22} := A_{22} - \textsc{tril}(A_{21} A_{21}^T)$

  Continue with
  $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile

FIG. 4.1. *Unblocked and blocked algorithms for computing the Cholesky factorization.*

$$= \left(\begin{array}{c|c} L_{TL} L_{TL}^T & \star \\ \hline L_{BL} L_{TL}^T & L_{BL} L_{BL}^T - L_{BR} L_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} \hat{A}_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right),$$

so that

$$L_{TL} = \Gamma(\hat{A}_{TL}), \quad L_{BL} = \hat{A}_{BL} L_{TL}^{-T}, \quad \text{and} \quad L_{BR} = \Gamma(\hat{A}_{BR} - L_{BL} L_{BL}^T).$$

From this we note that L_{TL} should be computed before L_{BL}, which in turn should be computed before L_{BR}: Upon completion of the factorization A must contain

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL} & \star \\ \hline L_{BL} & L_{BR} \end{array}\right).$$

Previously, we saw that our notation allows us to identify algorithms by the contents that are maintained in the matrix or vector that is being updated. For the Cholesky factorization three possibilities are identified in Fig. 4.2. The algorithms (both unblocked and blocked) that maintain the indicated contents are given in Fig. 4.1.

Remark 8. *In [3] we show how algorithms for the Cholesky factorization can be systematically derived from the contents that are to be maintained in matrix by the algorithm.*

Variant 1:	Variant 2:
$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & \star \\ \hline L_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 3: $\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & \star \\ \hline L_{BL} & \hat{A}_{BR} - \text{TRIL}(L_{BL}L_{BL}^T) \end{array}\right)$	

FIG. 4.2. *States maintained in matrix A corresponding to the algorithms given in Fig. 4.1 below.*

4.2. LU factorization. Given a matrix A of size $m \times n$ with $m \geq n$, its LU factorization is given by $A = LU$, where L is unit lower trapezoidal and U is upper triangular. This factorization is a reformulation of Gaussian Elimination and it exists under well-known conditions. We use the notation $\{L\backslash U\} = \text{LU}(\hat{A})$ to bring the attention to the fact that the function LU() takes a matrix A and returns two triangular matrices, L and U; these two matrices can be stored overwriting the strictly lower and the upper triangular parts of A respectively. The diagonal of L consists of ones and is not stored. In the following discussion, $\{L\backslash U\}_{TL}$ and $\{L\backslash U\}_{BR}$ are abbreviations for $\{L_{TL}\backslash U_{TL}\}$ and $\{L_{BR}\backslash U_{BR}\}$, respectively.

Five unblocked algorithms have been proposed since Gauss first proposed Gaussian elimination. Partitioning the matrices into quadrants, we find that

$$\begin{aligned}\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) &= \left(\begin{array}{c|c} L_{TL}U_{TL} & L_{TL}U_{TR} \\ \hline L_{BL}U_{TL} & L_{BL}U_{TR} + L_{BR}U_{BR} \end{array}\right) \\ &= \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right),\end{aligned}$$

where $\hat{A}$ denotes the original contents of A. If A is to be overwritten with the final result, this means that upon completion A must contain

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \{L\backslash U\}_{BR} \end{array}\right),$$

where

$$\begin{array}{c|c} \{L\backslash U\}_{TL} = \text{LU}(\hat{A}_{TL}) & U_{TR} = L_{TL}^{-1}\hat{A}_{TR} \\ \hline L_{BL} = \hat{A}_{BL}U_{TL}^{-1} & \{L\backslash U\}_{TL} = \text{LU}(\hat{A}_{BR} - L_{BL}U_{TR}) \end{array}.$$

In Fig. 4.3 we give five states maintained by the algorithms given in Fig. 4.4, which correspond to the five classic unblocked algorithms for computing the LU factorization [28], as well as their blocked counterparts.

Remark 9. *In [14] we show how algorithms for the LU factorization can be systematically derived from the contents that are to be maintained in matrix by the algorithm.*

4.3. LU factorization with partial pivoting. It is well-known that the LU factorization is numerically unstable under general circumstances. In practice, LU factorization with partial pivoting solves these stability problems. For details on this subject we suggest the reader consult any standard text on numerical linear algebra.

Variant 1: $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$	Variant 2: $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 3: $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & \hat{A}_{TR} \\ \hline L_{BL} & \hat{A}_{BR} \end{array}\right)$	Variant 4: $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 5: $\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \hat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$	

FIG. 4.3. *States maintained in matrix A corresponding to the algorithms given in Fig. 4.4 below.*

In this section, we will assume that the reader is familiar with the subject as we introduce new notation for presenting it.

DEFINITION 4.1. *An $n \times n$ matrix P is said to be a* permutation *matrix, or permutation, if, when applied to a vector $x = (\chi_0, \chi_1, \ldots, \chi_{n-1})^{\mathrm{T}}$, it merely rearranges the order of the elements in that vector. Such a permutation can be represented by the vector of integers, $(\pi_0, \pi_1, \ldots, \pi_{n-1})^{\mathrm{T}}$, where $\{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$ is a permutation of the integers $\{0, 1, \ldots, n-1\}$ and the permuted vector Px is given by $(\chi_{\pi_0}, \chi_{\pi_1}, \ldots, \chi_{\pi_{n-1}})^{\mathrm{T}}$.* If P is a permutation matrix then PA rearranges the rows of A exactly as the elements of x are rearranged by Px.

We will see that when discussing the LU factorization with partial pivoting, a permutation matrix that swaps the first element of a vector with the π-th element of that vector is a fundamental tool. We will denote that matrix by

$$P(\pi) = \begin{cases} I_n & \text{if } \pi = 0 \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & I_{\pi-1} & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{n-\pi-1} \end{pmatrix} & \text{otherwise,} \end{cases}$$

where n is the dimension of the permutation matrix. In the following we will use the notation P_n to indicate that the matrix P is of size n. Let p be a vector of integers satisfying the conditions

$$p = (\pi_0, \ldots, \pi_{k-1})^{\mathrm{T}}, \quad \text{where } 1 \leq k \leq n \text{ and } 0 \leq \pi_i < n - i, \tag{4.1}$$

then $P_n(p)$ will denote the permutation:

$$P_n(p) = \begin{pmatrix} I_{k-1} & 0 \\ 0 & P_{n-k+1}(\pi_{k-1}) \end{pmatrix} \begin{pmatrix} I_{k-2} & 0 \\ 0 & P_{n-k+2}(\pi_{k-2}) \end{pmatrix} \cdots \begin{pmatrix} 1 & 0 \\ 0 & P_{n-1}(\pi_1) \end{pmatrix} P_n(\pi_0).$$

Remark 10. *In the algorithms, the subscript that indicates the matrix dimentions is omitted.*

EXAMPLE 1. *Let $a_0^T, a_1^T, \ldots, a_{n-1}^T$ be the rows of a matrix A. The application of $P(p)$ to A yields a matrix that results from swapping row a_0^T with $a_{\pi_0}^T$, then swapping a_1^T with $a_{\pi_1+1}^T$, a_2^T with $a_{\pi_2+2}^T$, until finally a_{k-1}^T is swapped with $a_{\pi_{k-1}+k-1}^T$.*

Algorithm: $A := \mathrm{LU\_UNB}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $n(A_{TL}) < n(A)$ **do**

  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

  where α_{11} is 1×1

Variant 1:
$a_{01} := L_{00}^{-1} a_{01}$
$a_{10}^T := a_{10}^T U_{00}^{-1}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$

Variant 2:
$a_{10}^T := a_{10}^T U_{00}^{-1}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$

Variant 3:
$a_{01} := L_{00}^{-1} a_{01}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{21} := (a_{21} - A_{20}\ a_{01})/\alpha_{11}$

Variant 4:
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{21} := (a_{21} - A_{20}\ a_{01})/\alpha_{11}$
$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$

Variant 5:
$a_{21} := a_{21}/\alpha_{11}$
$A_{22} := A_{22} - a_{21} a_{12}^T$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

endwhile

Algorithm: $A := \mathrm{LU\_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $n(A_{TL}) < n(A)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

  where A_{11} **is** $b \times b$

Variant 1:
$A_{01} := L_{00}^{-1} A_{01}$
$A_{10} := A_{10} U_{00}^{-1}$
$A_{11} := \mathrm{LU}(A_{11} - A_{10} A_{01})$

Variant 2:
$A_{10} := A_{10} U_{00}^{-1}$
$A_{11} := \mathrm{LU}(A_{11} - A_{10} A_{01})$
$A_{12} := L_{11}^{-1}(A_{12} - A_{10} A_{02})$

Variant 3:
$A_{01} := L_{00}^{-1} A_{01}$
$A_{11} := \mathrm{LU}(A_{11} - A_{10} A_{01})$
$A_{21} := (A_{21} - A_{20} A_{01}) U_{11}^{-1}$

Variant 4:
$A_{11} := \mathrm{LU}(A_{11} - A_{10} A_{01})$
$A_{21} := (A_{21} - A_{20} A_{01}) U_{11}^{-1}$
$A_{12} := L_{11}^{-1}(A_{12} - A_{10} A_{02})$

Variant 5:
$A_{11} := \mathrm{LU}(A_{11})$
$A_{21} := A_{21} U_{11}^{-1}$
$A_{12} := L_{11}^{-1} A_{12}$
$A_{22} := A_{22} - A_{21} A_{12}$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

FIG. 4.4. *Unblocked algorithm for computing the LU factorization. In this figure, $n(A)$ is a function that yields the number of columns of matrix A and L_{ii} and U_{ii} denote the unit lower triangular matrix and upper triangular matrix stored over A_{ii}, respectively.*

Remark 11. *For those familiar with how pivot information is stored in LINPACK and LAPACK, notice that those packages store the vector of pivot information* $(\pi_0 + 1, \pi_1 + 2, \ldots, \pi_{k-1} + k)^{\mathrm{T}}$.

Algorithm: $[A,p] := \text{LU}_{\text{PIV\_UNB}}(\ A\ )$

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right),\ p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0, p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

where α_{11} and π_1 are scalars

Variant 3 (with pivoting):

$a_{01} := L_{00}^{-1} a_{01}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{21} := a_{21} - A_{20} a_{01}$
$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \text{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$a_{21} := a_{21}/\alpha_{11}$
$\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right)$

Variant 4 (with pivoting):

$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{21} := a_{21} - A_{20} a_{01}$
$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$
$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \text{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$a_{21} := a_{21}/\alpha_{11}$
$\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right)$

Variant 5 (with pivoting):

$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \text{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$a_{21} := a_{21}/\alpha_{11}$
$\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right)$
$A_{22} := A_{22} - a_{21} a_{12}^T$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Algorithm: $[A,p] := \text{LU}_{\text{PIV\_BLK}}(\ A\ )$

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right),\ p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0, p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11} **is** $b \times b$ **,** p_1 **is** $b \times 1$

Variant 3 (with pivoting):

$A_{01} := L_{00}^{-1} A_{01}$
$A_{11} := A_{11} - A_{10} A_{01}$
$A_{21} := A_{21} - A_{20} A_{01}$
$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU}_{\text{PIV}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$

Variant 4 (with pivoting):

$A_{11} := A_{11} - A_{10} A_{01}$
$A_{21} := A_{21} - A_{20} A_{01}$
$A_{12} := A_{12} - A_{10} A_{02}$
$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU}_{\text{PIV}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$
$A_{12} := L_{11}^{-1} A_{12}$

Variant 5 (with pivoting):

$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU}_{\text{PIV}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$
$A_{12} := L_{11}^{-1} A_{12}$
$A_{22} := A_{22} - A_{21} A_{12}$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

FIG. 4.5. *Unblocked and blocked algorithms for LU factorization with partial pivoting. Here the matrix is pivoted like LAPACK does, so that $P(p)A = LU$ upon completion. In this figure, L_{ii} denotes the unit lower triangular matrix stored over A_{ii}.*

Having introduced our notation for permutation matrices, we can now define the LU factorization with partial pivoting: Given an $m \times n$ matrix A, with $m \geq n$, we wish to compute a) a vector p of n integers which satisfies the conditions (4.1), b) a unit lower trapezoidal matrix L, and c) an upper triangular matrix U so that $P(p)A = LU$. An algorithm for computing this operation is typically represented by

$$[A, p] := \text{LU}\textsc{piv}(A),$$

where upon completion A has been overwritten by $\{L \backslash U\}$.

Using our notation, unblocked and blocked algorithms for computing LUpiv are given in Fig. 4.5. Three variants are given, corresponding to the Variants 3–5 in Fig. 4.4 to which pivoting has been added. The function $[x, \pi] = Pivot(x)$ determines the index of the element in vector x that has the maximal absolute value and swaps the first element of x with the maximal (in absolute value) element.

4.4. Computing an orthonormal basis. The goal of the Gram-Schmidt process is to compute an orthonormal basis for the space spanned by the columns of a given $m \times n$ matrix A (for $m \geq n$) with n linearly independent columns. The vectors that constitute the orthonormal basis become the columns of an $m \times n$ matrix Q. A particular case of this problem can be stated as computing $[Q, R] = QR(A)$, where R is an $n \times n$ upper triangular matrix so that $A = QR$. In this case, Q has the property that the first k columns of Q span the space spanned by the first k columns of A, for $k = 1, \ldots, n$.

Algorithms for computing Q and R can be described by partitioning A, Q, and R as

$$A \rightarrow \left(A_L \,\big|\, A_R \right), \quad Q \rightarrow \left(Q_L \,\big|\, Q_R \right), \quad \text{and} \quad R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right),$$

where A_L and Q_L have k columns and R_{TL} is $k \times k$. Then

$$\left(A_L \,\big|\, A_R \right) = \left(Q_L \,\big|\, Q_R \right) \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right).$$

Classical Gram-Schmidt. Assume that Q_L and R_{TL} have already been computed. In other words, an orthonormal basis for A_L has already been computed.

Repartition, exposing columns to be updated,

$$\left(A_L \,\big|\, A_R \right) \rightarrow \left(A_0 \,\big|\, a_1 \,|\, A_2 \right), \quad \left(Q_L \,\big|\, Q_R \right) \rightarrow \left(Q_0 \,\big|\, q_1 \,|\, Q_2 \right), \quad \text{and}$$

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right).$$

Then

$$\begin{aligned} \left(A_0 \,\big|\, a_1 \,|\, A_2 \right) &= \left(Q_0 \,\big|\, q_1 \,|\, Q_2 \right) \left(\begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right) \\ &= \left(Q_0 R_{00} \,\big|\, Q_0 r_{01} + \rho_{11} q_1 \,|\, Q_0 R_{02} + q_1 r_{12}^T + Q_2 R_{22} \right). \end{aligned}$$

CGS:	MGS:
$\left(A_L \,\Big\|\, A_R \right) = \left(Q_L \,\Big\|\, \hat{A}_R \right)$	$\left(A_L \,\Big\|\, A_R \right) = \left(Q_L \,\Big\|\, \hat{A}_R - Q_L Q_L^T \hat{A}_R \right)$

FIG. 4.6. *States maintained in matrix A corresponding to the algorithms given in Fig. 4.7 below.*

Algorithm: $[A, R] := \mathrm{QR}(A)$

Partition $A \rightarrow \left(A_L \,\big\|\, A_R \right)$, $R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right)$
  where A_L has 0 columns, R_{TL} is 0×0

while $n(A_L) < n(A)$ **do**
  Repartition

$$\left(A_L \,\big\|\, A_R \right) \rightarrow \left(A_0 \,\big\|\, a_1 \,\big|\, A_2 \right), \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$$

  where a_1 has 1 column, ρ_{11} is a scalar

CGS:	MGS:
$r_{01} := A_0^T a_1$	$\rho_{11} := \|a_1\|_2$
$a_1 := a_1 - A_0 r_{01}$	$a_1 := a_1 / \rho_{11}$
$\rho_{11} := \|a_1\|_2$	$r_{12}^T := a_1^T A_2$
$a_1 := a_1 / \rho_{11}$	$A_2 := A_2 - a_1 r_{12}^T$

  Continue with

$$\left(A_L \,\big\|\, A_R \right) \leftarrow \left(A_0 \,\big|\, a_1 \,\big\|\, A_2 \right), \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$$

endwhile

FIG. 4.7. *Orthogonalization via Gram-Schmidt and Modified Gram-Schmidt.*

Now, Q_0 and R_{00} have already been computed and we wish to compute the next column of Q and R. From $a_1 = Q_0 r_{01} + \rho_{11} q_1$, we deduce that $Q_0^T a_1 = r_{01}$ and $q_1 = (a_1 - Q_0 r_{01})/\rho_{11}$. Since q_1 must have unit length, we find that the following steps compute q_1:

Classical Gram-Schmidt (CGS) algorithm

$r_{01} := Q_0^T a_1$
$q_1 := a_1 - Q_0 r_{01}$ (= the component of a_1 orthogonal to the columns of Q_0)
$\rho_{11} := \|q_1\|_2$
$q_1 := q_1 / \rho_{11}$ (normalization).

This procedure is summarized in Fig. 4.7, where the process overwrites A with the orthogonal matrix Q.

Modified Gram-Schmidt. The CGS algorithm is notoriously numerically unstable. More stable is the Modified Gram-Schmidt (MGS) algorithm. In this variant, every time a new column of Q, q_1, is computed, the remaining columns of A are updated by subtracting out the component of those columns that lies in the direction of q_1. Thus, at the beginning of a given step, a_1 and A_2 are already orthogonal to the columns of Q_0, which have overwritten A_0. The next column, q_1 is then computed by normalizing the current contents of a_1. Finally, the columns of A_2 are updated by subtracting out the component in the direction of q_1: $A_2 := A_2 - q_1 r_{12}^T$, where $r_{12}^T = q_1^T A_2$. The MGS algorithm is also given in Fig. 4.7, and in Fig. 4.6 we show the states maintained in matrix A by the algorithms CGS and MGS.

4.5. QR factorization via Householder transformations. In this section, we introduce one unblocked algorithm for computing the QR factorization via Householder transformations. Enough background is given so that someone already familiar with this operation will be able to recognize how our notation supports it.

Householder transformations. A Householder transformation (often called a *reflector*) is a matrix of the form $H = I - \beta v v^T$, where $v \neq 0$ and $\beta = 2/v^T v$. Notice that $H = H^T$ and $H^T H = H^{-1} H = I$. In other words, H is orthogonal, symmetric, and equals its own inverse.

Now, let $x \neq 0$ and partition $x \rightarrow \left(\frac{\chi_1}{x_2} \right)$. If v is chosen as $v = \left(\frac{\chi_1 \pm \|x\|_2}{x_2} \right)$, then $(I - \beta v v^T)x = \left(\frac{\mp\|x\|_2}{0} \right)$ holds. In other words, given a vector x, there exists a Householder transformation with the property that zeroes out the elements below the first element while preserving the norm of the vector[1]. The $\pm$ is picked to equal the sign of χ_1, for numerical stability reasons.

The discussion so far can be found in a typical numerical linear algebra text. We will now discuss how the vector that defines the Householder transformation can be chosen in a more practical way. Given the vector x, partitioned as before, we will define the Householder transformation instead as $I - \frac{1}{\tau} u u^T$, where $u = \left(\frac{1}{u_2} \right)$ and $\tau = u^T u/2$. Clearly this is simply a Householder transformation in disguise: The vector v in the Householder transformation can be scaled arbitrarily by scaling β correspondingly, and u is derived from v by dividing it by the first element of v. Thus, given a vector x, partitioned as before, u_2 and τ now must satisfy

$$\left(I - \frac{1}{\tau} \left(\frac{1}{u_2} \right) \left(\frac{1}{u_2} \right)^T \right) \left(\frac{\chi_1}{x_2} \right) = \left(\frac{\rho}{0} \right).$$

The following formulae compute u_2, τ, and ρ:

$$\begin{aligned} u_2 &:= \begin{cases} 0 & \text{if } x = 0 \\ x_2/(\chi_1 \pm \|x\|_2) & \text{otherwise} \end{cases} \\ \tau &:= 2/u^T u = 2/(1 + u_2^T u_2) \\ \rho &:= \mp\|x\|_2. \end{aligned}$$

We introduce the notation $\left[\left(\frac{\rho}{u_2} \right), \tau \right] := h(x)$ as the function that computes the vector u_2, and scalars ρ and τ from the input vector x.

Let A be an $m \times n$ with $m \geq n$. We will now show how to compute the QR factorization $A \rightarrow QR$, which, through a sequence of Householder transformations, eventually zeroes out all elements of matrix A below the diagonal.

In the first iteration, we partition

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array} \right).$$

[1]If $x = 0$, then β is ill-defined. In that case, we will take $v = \left(\frac{1}{0} \right)$ and $\beta = 2$, although some implementations take $v = 0$ and set $\beta = 0$.

Algorithm: $[A, t] := \mathrm{QR}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $s \rightarrow \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right)$

where A_{TL} is 0×0, t_T has 0 elements

while $n(A_{BR}) \neq 0$ **do**

Determine block size *blank*

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \rightarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right)$$

where α_{11} **and** τ_1 **are scalars**

$$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \tau_1\right] := \left[\left(\begin{array}{c} \eta \\ \hline u_{21} \end{array}\right), \tau_1\right] = h\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$w_{12}^T := (a_{12}^T + u_{21}^T A_{22})/\tau_1$

$a_{12}^T := a_{12}^T - w_{12}^T$

$A_{22} := A_{22} - u_{21} w_{12}^T$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \leftarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right)$$

endwhile

FIG. 4.8. *Unblocked QR factorization algorithm.*

Let

$$\left[\left(\begin{array}{c} \rho_{11} \\ \hline u_{21} \end{array}\right), \tau_1\right] = h\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

be the Householder transform computed from the first column of A. Then applying this Householder transform to A yields

$$\begin{aligned} \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) &:= \left(I - \frac{1}{\tau}\left(\begin{array}{c} 1 \\ \hline u_2 \end{array}\right)\left(\begin{array}{c} 1 \\ \hline u_2 \end{array}\right)^T\right)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \\ &= \left(\begin{array}{c|c} \rho_{11} & a_{12}^T - w_{12}^T \\ \hline 0 & A_{22} - u_{21} w_{12}^T \end{array}\right), \end{aligned}$$

where $w_{12}^T = (a_{12}^T + u_{21}^T A_{22})/\tau_1$. Computation of a full QR factorization of A proceeds with the updated matrix A_{22}. The complete unblocked algorithm is given in Fig. 4.8.

Remark 12. *We refer the interested reader to [16] for a more complete treatment of the QR factorization that presents unblocked and blocked algorithms as well as extensions that support out-of-core computation of a QR factorization. How to aggregate Householder transformations in support of a blocked algorithm is discussed in [18].*

5. Case Studies, Part III: The QR Algorithm. For the expert reader, we now discuss a much more advanced algorithm: The QR algorithm [11, 22], for computing the eigenvalues of a symmetric tridiagonal matrix T. We will assume that the reader is already intimately familiar with that algorithm. A good reference is [13].

For each iteration of this algorithm, the following steps are executed:

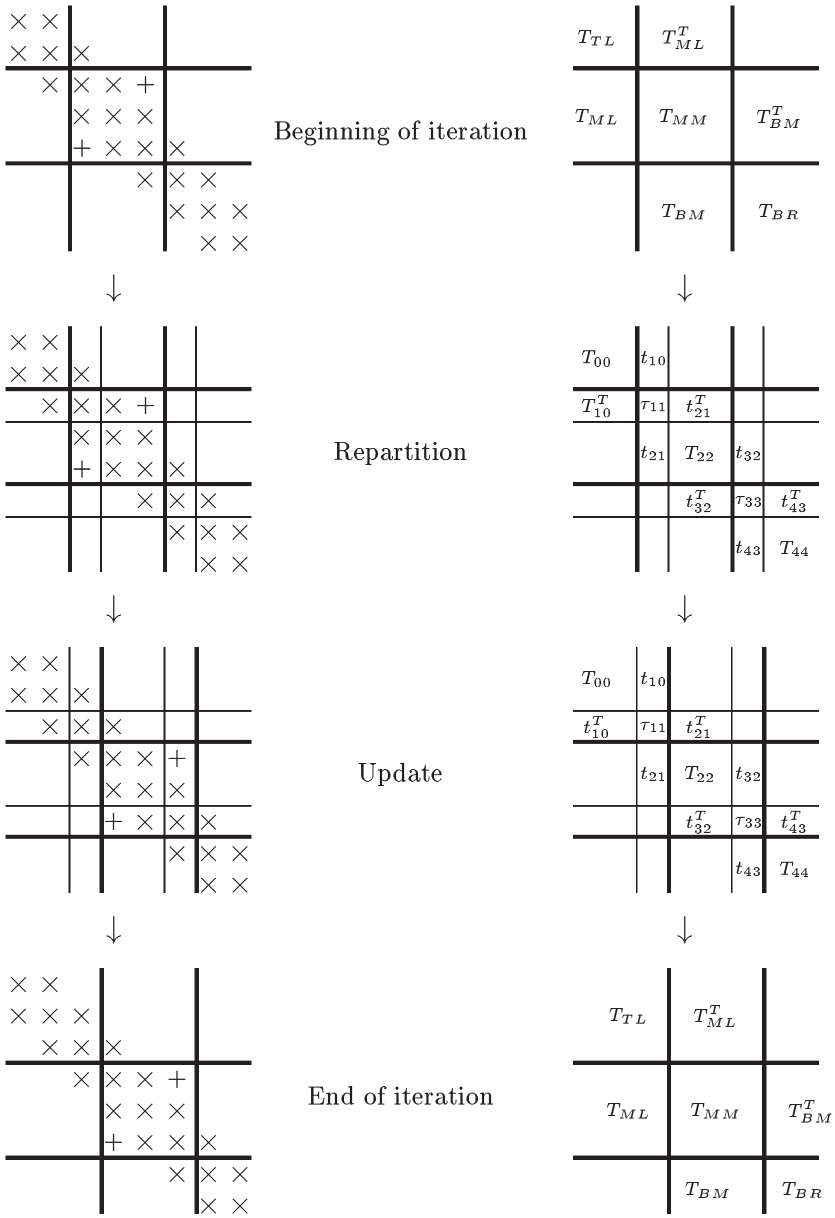

FIG. 5.1. *One step of "chasing the bulge" in the implicitly shifted symmetric QR algorithm.*

- Determine a shift ρ (approximate eigenvalue of T).
- Determine a first Givens' rotation G that annihilates the first element of the first subdiagonal of matrix $T - \rho I$.
- Update $T := G^T T G$. This creates a nonzero in the first element of the second subdiagonal.
- Determine Givens' rotations that "chase the bulge" to make matrix t again tridiagonal.

It is this last step that we discuss in this section.

Algorithm: $T := \textsc{ChaseBulge}(T)$

Partition $T \rightarrow \left(\begin{array}{c|c|c} T_{TL} & \star & \star \\ \hline T_{ML} & T_{MM} & \star \\ \hline 0 & T_{BM} & T_{BR} \end{array}\right)$
where T_{TL} is 0×0 and T_{MM} is 3×3

while $m(T_{BR}) > 0$ **do**

Repartition

$$\left(\begin{array}{c|c|c} T_{TL} & \star & 0 \\ \hline T_{ML} & T_{MM} & \star \\ \hline 0 & T_{BM} & T_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} T_{00} & \star & 0 & 0 & 0 \\ \hline t_{10}^T & \tau_{11} & \star & 0 & 0 \\ \hline 0 & t_{21} & T_{22} & \star & 0 \\ \hline 0 & 0 & t_{32}^T & \tau_{33} & \star \\ \hline 0 & 0 & 0 & t_{43} & T_{44} \end{array}\right)$$

where τ_{11} and τ_{33} are scalars
(during final step, τ_{33} is 0×0)

Compute (c, s) s.t. $G_{c,s}^T t_{21} = \begin{pmatrix} \tau_{21} \\ 0 \end{pmatrix}$, and **Assign** $t_{21} = \begin{pmatrix} \tau_{21} \\ 0 \end{pmatrix}$

$T_{22} = G_{c,s}^T T_{22} G_{c,s}$

$t_{32}^T = t_{32}^T G_{c,s}$ (not performed during final step)

Continue with

$$\left(\begin{array}{c|c|c} T_{TL} & \star & 0 \\ \hline T_{ML} & T_{MM} & \star \\ \hline 0 & T_{BM} & T_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} T_{00} & \star & 0 & 0 & 0 \\ \hline t_{10}^T & \tau_{11} & \star & 0 & 0 \\ \hline 0 & t_{21} & T_{22} & \star & 0 \\ \hline 0 & 0 & t_{32}^T & \tau_{33} & \star \\ \hline 0 & 0 & 0 & t_{43} & T_{44} \end{array}\right)$$

endwhile

FIG. 5.2. *Chasing the bulge.*

A typical intermediate stage in the chasing of the bulge is illustrated in Fig. 5.1. We are now using more lines to track progress through the matrix: An initial 3×3 partitioning becomes a 5×5 partitioning, which then becomes again a 3×3 partitioning to set up the next iteration.

This example gives us the opportunity to ask the question why in the other examples matrices were partitioned into a 2×2 partitioning and repartitioned to a 3×3 partitioning. The answer is that a 2×2 partitioning is needed to track the four quadrants. To be able to identify the submatrices that are moved from one side of the thick line to the other, the two 2×2 partitionings are superimposed, creating the familiar 3×3 partitioning.

In order to identify the bulge, a 3×3 partitioning is needed at the top of the loop, as well as at the bottom of the loop, as illustrated in the left-most and right-most pictures in Fig. 5.1. By superimposing these two pictures, the 5×5 partitioning appears, which now identifies submatrices that are to be moved across the thick line boundaries. The resulting algorithm for chasing the bulge is now given in Fig. 5.2.

6. Representing Algorithms in Code. The notation we presented does not closely resemble how algorithms have traditionally been represented in code. In this section we show that it is possible to easily define an API for a target language that allows the code to closely mirror the algorithm.

The translation of the algorithm to traditional Matlab code still introduces a complicated indexing, as is shown in Fig. 6.1 for the blocked algorithm of Fig. 2.4. In order to overcome the indexing, we suggest coding algorithms in the style illustrated

```
n = size(A,1);
for j=1:nb:n
  b = min( n-j+1, nb );
  A(j:j+b-1,j:j+b-1) = Chol(A(j:j+b-1,j:j+b-1));
  A(j+b:n,j:j+b-1)   = A(j+b:n,j:j+b-1) / tril( A(j:j+b-1,j:j+b-1);
  A(j+b:n,j+b:n)     = A(j+b:n,j+b:n) - ...
                       tril(A(j+b:n,j:j+b-1)*A(j+b:n,j+b-1)');
end
```

FIG. 6.1. *Traditional Matlab code for blocked Cholesky factorization (Variant 3).*

```
function [ A_out ] = Chol_blk( A, nb_alg )
  [ ATL, ATR, ...
    ABL, ABR ] = FLA_Part_2x2( A, ...
                               0, 0, 'FLA_TL' );
  while ( size( ATL, 1 ) < size( A, 1 ) )
    b = min( size( ABR, 1 ), nb_alg );

    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                                               ABL, ABR, ...
                                               b, b, 'FLA_BR' );
    %------------------------------------------------------------%
    A11 = Chol_unb( A11 );
    A21 = A21 / tril( A11 )';
    A22 = A22 - tril( A21 * A21' );
    %------------------------------------------------------------%
    [ ATL, ATR, ...
      ABL, ABR ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                             A10, A11, A12, ...
                                             A20, A21, A22, ...
                                             'FLA_TL' );
  end
  A_out = [ ATL, ATR
            ABL, ABR ];
return
```

FIG. 6.2. *FLAME@lab (Matlab) code for a blocked Cholesky factorization (Variant 3).*

in Fig. 6.2. For details on this API and similar APIs for other programming languages, we refer the reader to [5].

The Linear Algebra Package (LAPACK) [1] is a widely used package that supports a broad set of dense linear algebra operations. It casts most computation in terms of calls to the Basic Linear Algebra Subprograms (BLAS) [24, 9, 8]. A code segment from the LAPACK routine for the blocked LU factorization with pivoting is given in Fig. 6.3. Clearly, there is the opportunity for the introduction of indexing errors. Indeed, we challenge the reader to find the errors that we purposely introduced.

In Fig. 6.4 we show an implementation of the same algorithm using the FLAME/C API [14]. We believe that our API naturally captures the algorithm as presented using our notation, thereby reducing the opportunity for the introduction of inadvertent coding errors. A similar API for FORTRAN has also been defined. The translation

```
      DO 20 J = 1, MIN( M, N ), NB
         JB = MIN( MIN( M, N )-J+1, NB )
*
*        Factor diagonal and subdiagonal blocks and test for exact
*        singularity.
*
         CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ), IINFO )
*
*        Adjust INFO and the pivot indices.
*
         IF( INFO.EQ.0 .AND. IINFO.GT.0 )
    $       INFO = IINFO + J - 1
         DO 10 I = J, MIN( M, J+JB+1 )
            IPIV( I ) = J - 1 + IPIV( I )
   10    CONTINUE
*
*        Apply interchanges to columns 1:J-1.
*
         CALL DLASWP( J-1, A, LDA, J, J+JB+1, IPIV, 1 )
*
         IF( J+JB.LE.N ) THEN
*
*           Apply interchanges to columns J+JB:N.
*
            CALL DLASWP( N-J-JB-1, A( 1, J+JB ), LDA, J, J+JB+1, IPIV, 1 )
*
*           Compute block row of U.
*
            CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
    $                   N-J-JB-1, ONE, A( J, J ), LDA, A( J, J+JB ), LDA )
            IF( J+JB.LE.M ) THEN
*
*              Update trailing submatrix.
*
               CALL DGEMM( 'No transpose', 'No transpose', M-J-JB-1,
    $                      N-J-JB-1, JB, -ONE, A( J+JB, J ), LDA,
    $                      A( J, J+JB ), LDA, ONE, A( J+JB, J+JB ), LDA )
            END IF
         END IF
   20 CONTINUE
```

FIG. 6.3. *LAPACK code for blocked LU factorization (Variant 5 with pivoting).*

of algorithm to code is essentially equivalent to the application of simple rewrite rules to that algorithm. Indeed, an API has been defined even for G, the graphical programming language that underlies LabView [20].

One word on performance: For blocked algorithms the additional cost of hiding intricate indices in code via the appropriate APIs is amortized over enough computation that it does not adversely affect performance. Often times the overhead is instead noticeable for unblocked algorithms. However, the code as presented in Figures 6.2 and 6.4 passes to a compiler a lot more high-level information than does code that exposes intricate indices. Compilers like those pursued by the Broadway project [17] could enable the generation of highly optimized implementations from algorithms expressed at this level of abstraction.

A recent topic of research relates to the storage of matrices by blocks rather than the traditional row- or column-major orderings [10, 12, 29]. Our notation and the new

```
while (b = min(min(FLA_Obj_length( ABR ), FLA_Obj_width( ABR )), nb_alg) )
{
  FLA_Repart_2x1_to_3x1( ipivT,                 &ipiv0,
                      /* ***** */            /* ***** */
                                               &ipiv1,
                         ipivB,                &ipiv2,
                         b, /* length ipiv1 split from */ FLA_BOTTOM );

  FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,      &A00, /**/ &A01, &A02,
                      /* ************* */ /* ******************** */
                              /**/           &A10, /**/ &A11, &A12,
                         ABL, /**/ ABR,      &A20, /**/ &A21, &A22,
                         b, /* by */ b, /* A11 split from */ FLA_BR );

  /* ******************************************************************** */

  FLA_LU_unb( A11,
              A21, ipiv1 );
  FLA_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, ipiv1, A10,
                                                       A20 );
  FLA_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, ipiv1, A12,
                                                       A22 );
  FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
            ONE, A11, A12 );
  FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22 );

  /* ******************************************************************** */

  FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, A01, /**/ A02,
                                  /**/            A10, A11, /**/ A12,
                          /* ************** */ /* ***************** */
                            &ABL, /**/ &ABR,      A20, A21, /**/ A22,
                            /* with A11 added to submatrix */ FLA_TL );

  FLA_Cont_with_3x1_to_2x1( &ipivT,                  ipiv0,
                                                     ipiv1,
                          /* ***** */            /* ***** */
                            &ipivB,                  ipiv2,
                            /* with ipiv1 added to */ FLA_TOP );
}
```

FIG. 6.4. *FLAME code for blocked LU factorization (Variant 5 with pivoting).*

APIs allow the very complex indexing associated with such storage to be completely hidden from the programmer. An alternative (and in our view better) solution to this problem is to allow matrices to have entries that themselves are matrices, which can also be accommodated by our notation and APIs [5].

7. Conclusion. The contribution of the presented notation is that it allows algorithms to be expressed at the level of abstraction of the underlying theory. Our notation embraces the notion of subvectors and submatrices while hiding the physical arrays that store them.

The manner in which compution sweeps through vectors and matrices is often consistent across different algorithms for computing the same operation. It is how subvectors and aubmatrices are updated that differs. This important insight is typically obscured by algorithms that explicitly expose indices. Conversely, our notation highlights this commonality, and thus allows different algorithmic variants to be easily compared and contrasted. Likewise commonalities and differences between algorithms for different operations can be exhibited.

The presentation of multiple variants for the same operation leads to some obvious questions. How can one find all loop-based algorithmic variants for a given linear algebra operation? Are there more variants than we have presented? Are all variants numerically stable? What are the performance benefits of one variant over another? What is the relationship between loop-based algorithms and recursive algorithms? Indices add a level of complexity to the problem that obscures the mathematical statement of these questions and makes it harder to answer them. Our recent research presents evidence that the proposed notation facilitates the formal statement of these questions as well as their answers. In [2] we present a systematic approach for deriving loop-based algorithms. We show the methodology to be sufficiently systematic to enable mechanical (automatic) generation of algorithms in [4], while initial results regarding numerical stability are given in [6]. A comparative study of the performance benefits for a family of algorithms is provided in [3]. A final comment on the typesetting of algorithms: we have developed a set of simple commands for typing algorithms using our notation in LaTeX [23]. These commands have been used by undergraduate and graduate students with no previous exposure to LaTeX in classes that we have taught.

More information. Visit `http://www.cs.utexas.edu/users/flame/`.

Acknowledgments. We would like to thank Peter Brune, Margaret Myers, Monika Petera, Enrique Quintana-Ortí, Field Van Zee, and Petko Yanev for their comments on drafts of this paper.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' guide (third ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn, *The science of deriving dense linear algebra algorithms*, ACM Trans. Math. Soft., 31 (2005), pp. 1–26.

[3] P. Bientinesi, B. Gunter, and R. van de Geijn, *High performace and parallel inversion of a symmetric positive definite matrix*, SIAM J. Sci. Comput. submitted.

[4] P. Bientinesi, S. Kolos, and R. van de Geijn, *Automatic derivation of linear algebra algorithms with application to control theory*, in Proceedings of PARA'04 State-of-the-Art in Scientific Computing, June 20-23 2004. To appear.

[5] P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn, *Representing linear algebra algorithms in code: The FLAME APIs*, ACM Trans. Math. Soft., 31 (2005), pp. 27–59.

[6] P. Bientinesi and R. van de Geijn, *Formal correctness and stability of dense linear algebra algorithms*, in Proceedings of the 17th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation, July 11-16 2005. To appear.

[7] J. W. Demmel, *Applied numerical linear algebra*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[9] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.

[10] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström, *Recursive blocked algorithms and hybrid data structures for dense matrix library software*, SIAM Review, 46 (2004), pp. 3–45.

[11] G. J. F. Francis, *The QR transformation, parts I and II*, Computer J., 4 (1961-62), pp. 265–271,332–345.
[12] J. D. Frens and D. S. Wise, *Auto-blocking matrix-multiplication or tracking blas3 performance with source code*, in PPoPP97, June 1997.
[13] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 3nd ed., 1996.
[14] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *FLAME: Formal linear algebra methods environment*, ACM Transactions on Mathematical Software, 27 (2001), pp. 422–455.
[15] J. A. Gunnels and R. A. van de Geijn, *Formal methods for high-performance linear algebra libraries*, in The Architecture of Scientific Software, R. F. Boisvert and P. T. P. Tang, eds., Kluwer Academic Press, 2001, pp. 193–210.
[16] B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Trans. Math. Soft., (2005).
[17] S. Z. Guyer and C. Lin, *Broadway: A Software Architecture for Scientific Computing*, Kluwer Academic Press, October 2000, pp. 175–192.
[18] T. Joffrain, T. M. Low, E. S. Quintana-Ortí, R. van de Geijn, and F. V. Zee, *On accumulating Householder transformations*, TOMS. in revision.
[19] T. Joffrain, E. S. Quintana-Ortí, and R. A. van de Geijn, *Rapid development of high-performance out-of-core solvers for electromagnetics*, in Proceedings of PARA'04 State-of-the-Art in Scientific Computing, June 20-23 2004. To appear.
[20] G. W. Johnson, R. Jennings, and R. Jennigns, *LabVIEW Graphical Programming*, McGraw-Hill.
[21] B. Kågström, P. Ling, and C. V. Loan, *GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark*, ACM Trans. Math. Soft., 24 (1998), pp. 268–302.
[22] V. N. Kublanovskaya, *On some algorithms for the solution of the complete eigenvalue problem*, Zh. Vych. Mat., 1 (1961), pp. 555–570.
[23] L. Lamport, *LaTeX: A Document Preparation System*, Addison-Wesley, Reading, MA, 1986.
[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308–323.
[25] C. Moler, J. Little, and S. Bangert, *Pro-Matlab, User's Guide*, The Mathworks, Inc., 1987.
[26] E. S. Quintana-Ortí and R. A. van de Geijn, *Formal derivation of algorithms: The triangular Sylvester equation*, ACM Transactions on Mathematical Software, 29 (2003), pp. 218–243.
[27] G. Quintana-Ortí and R. van de Geijn, *Improving the performance of reduction to Hessenberg form*, ACM Trans. Math. Soft. Accepted for publication.
[28] G. W. Stewart, *Introduction to Matrix Computations*, Academic Press, Orlando, Florida, 1973.
[29] V. Valsalam and A. Skjellum, *A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels.* Concurrency and Computation: Practice and Experience, 2002.