SIAM J. MATRIX ANAL. APPL.
Vol. 14, No. 2, pp. 353–371, April 1993

003

A SYSTOLIC ARRAY FOR SVD UPDATING*

MARC MOONEN†, PAUL VAN DOOREN‡, AND JOOS VANDEWALLE†

Abstract. In an earlier paper, an approximate SVD updating scheme has been derived as an interlacing of a QR updating on the one hand and a Jacobi-type SVD procedure on the other hand, possibly supplemented with a certain re-orthogonalization scheme. This paper maps this updating algorithm onto a systolic array with $O(n^2)$ parallelism for $O(n^2)$ complexity, resulting in an $O(n^0)$ throughput. Furthermore, it is shown how a square root-free implementation is obtained by combining modified Givens rotations with approximate SVD schemes.

Key words. singular value decomposition, parallel algorithms, recursive least squares

AMS(MOS) subject classifications. 65F15, 65F25

CR classification. G.I.3

1. Introduction. The problem of continuously updating matrix decompositions as new rows are appended frequently occurs in signal processing applications. Typical examples are adaptive beamforming, direction finding, spectral analysis, pattern recognition, etc. [13].

In [12], it has been shown how an *SVD updating* algorithm can be derived by combining QR updating with a Jacobi-type SVD procedure applied to the triangular factor. In each time step an *approximate decomposition* is computed from a previous approximation at a *low computational cost*, namely, $O(n^2)$ operations. This algorithm was shown to be particularly suited for *subspace tracking* problems. The tracking error at each time step is then found to be bounded by the time variation in $O(n)$ time steps, which is sufficiently small for applications with slowly time-varying systems. Furthermore, the updating procedure was proved to be *stable* when supplemented with a certain re-orthogonalization scheme, which is elegantly combined with the updating.

In this paper, we show how this updating algorithm can be mapped onto a *systolic array* with $O(n^2)$ parallelism, resulting in an $O(n^0)$ throughput (similar to the case for mere QR updating; see [5]). Furthermore, it is shown how a square root-free implementation is obtained by combining modified Givens rotations with approximate SVD schemes.

In §2, the updating algorithm is briefly reviewed. A systolic implementation is described in §3 for the easy case, where corrective re-orthogonalizations are left out. In §4, it is shown how to incorporate these re-orthogonalizations. Finally, a square root-free implementation is derived in §5.

2. SVD updating. The *singular value decomposition* (SVD) of a real matrix $A_{m\times n}$ $(m \geq n)$ is a factorization of A into a product of three matrices

$$A_{m\times n} = U_{m\times n} \cdot \Sigma_{n\times n} \cdot V_{n\times n}^T,$$

* Received by the editors November 10, 1989; accepted for publication (in revised form) October 18, 1991. This work was sponsored in part by BRA 3280 project of the European Community.

† ESAT, Katholieke Universiteit Leuven, K. Mercierlaan 94, 3001 Heverlee, Belgium (moonen@esat.kuleuven.ac.be and vandewalle@esat.kuleuven.ac.be). The first author is a senior research assistant with the Belgian N.F.W.O. (National Fund for Scientific Research).

‡ Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1101 West Springfield Avenue, Urbana, Illinois 61801 (vandooren@uicsl.csl.uiuc.edu).

where U has orthonormal columns, V is an orthogonal matrix, and Σ is a diagonal matrix, with the singular values along the diagonal.

Given that we have the SVD of a matrix A, we may need to calculate the SVD of a matrix $\underline{A}$ that is obtained after appending a new row to A.

$$\underline{A} = \begin{bmatrix} A \\ a^T \end{bmatrix} = \underline{U}_{(m+1)\times n} \cdot \underline{\Sigma}_{n\times n} \cdot \underline{V}^T_{n\times n}.$$

In on-line applications, a new updating is often to be performed after each sampling. The data matrix at time step k is then defined in a recursive manner ($k \geq n$)

$$A^{(k)} = \begin{bmatrix} \lambda^{(k)} \cdot A^{(k-1)} \\ a^{(k)^T} \end{bmatrix} = U^{(k)}_{k\times n} \cdot \Sigma^{(k)}_{n\times n} \cdot V^{(k)^T}_{n\times n}.$$

Factor $\lambda^{(k)}$ is a *weighting factor*, and $a^{(k)}$ is the *measurement vector* at time instance k. For the sake of brevity, we consider only the case where $\lambda^{(k)}$ is a constant λ, although everything can easily be recast for the case where it is time varying. Finally, in most cases the $U^{(k)}$ matrices (of growing size!) need not be computed explicitly, and only $V^{(k)}$ and $\Sigma^{(k)}$ are explicitly updated.

An *adaptive algorithm* can be constructed by interlacing a Jacobi-type SVD procedure (Kogbetliantz's algorithm [9], modified for triangular matrices [8], [10]) with repeated QR updates. See [12] for further details.

Initialization

$$V^{(0)} \Leftarrow I_{n\times n}$$
$$R^{(0)} \Leftarrow O_{n\times n}$$

Loop

for $k = 1, \ldots, \infty$

input new measurement vector $a^{(k)}$

$$a^{(k)^T}_\star \Leftarrow a^{(k)^T} \cdot V^{(k-1)}$$
$$R^{(k)}_\lambda \Leftarrow \lambda \cdot R^{(k-1)}$$

QR updating

$$\begin{bmatrix} R^{(k)}_\star \\ 0 \end{bmatrix} \Leftarrow Q^{(k)^T} \cdot \begin{bmatrix} R^{(k)}_\lambda \\ a^{(k)^T}_\star \end{bmatrix}$$
$$R^{(k)} \Leftarrow R^{(k)}_\star$$
$$V^{(k)} \Leftarrow V^{(k-1)}$$

SVD steps

for $i = 1, \ldots, n-1$

$$R^{(k)} \Leftarrow \Theta^{(k)^T}_i \cdot R^{(k)} \cdot \Phi^{(k)}_i$$
$$V^{(k)} \Leftarrow V^{(k)} \cdot \Phi^{(k)}_i$$
$$\{V^{(k)} \Leftarrow T^{(k)}_i \cdot V^{(k)}\}$$

end

end

Matrices $\Theta_i^{(k)}$ and $\Phi_i^{(k)}$ represent plane rotations (ith rotation in time step k) through angles $\theta_i^{(k)}$ and $\phi_i^{(k)}$ in the $(i, i+1)$-plane. The rotation angles $\theta_i^{(k)}$ and $\phi_i^{(k)}$ should be chosen such that the $(i, i+1)$ element in $R^{(k)}$ is zeroed, while $R^{(k)}$ remains in upper triangular form. Each iteration thus amounts to solving a 2×2 SVD on the main diagonal. The updating algorithm then reduces to applying sequences of $n-1$ rotations, where the *pivot index* i repeatedly takes up all the values $i = 1, 2, \ldots, n-1$ (n such sequences constitute a pipelined double sweep [14]), interlaced with QR updates. At each time step, $R^{(k)}$ will be "close" to a (block) diagonal matrix, so that in some sense $V^{(k)}$ is "close" to the exact matrix with right singular vectors; see [12].

Transformations $T_i^{(k)}$ correspond to approximate re-orthogonalizations of row vectors of $V^{(k)}$. These should be included in order to avoid round-off error buildup, if the algorithm is supposed to run for, say, thousands of time steps (see [12]). For the sake of clarity, the re-orthogonalizations are left out for a while, and are dealt with only in §4.

In the sequel, the time index k is often dropped for the sake of conciseness.

3. A systolic array for SVD updating. The above SVD updating algorithm—for the time being without re-orthogonalizations—can be mapped elegantly onto a systolic array, by combining systolic implementations for the matrix-vector product, the QR updating, and the SVD. In particular, with $n-1$ SVD iterations after each QR update,[1] an efficient parallel implementation is conceivable with $O(n^2)$ parallelism for $O(n^2)$ complexity. The SVD updating is then performed at a speed comparable to the speed of merely QR updating.

The SVD updating array is similar to the triangular SVD array in [10], where the SVD diagonalization and a preliminary QR factorization are performed on the same array. As for the SVD updating algorithm, the diagonalization process and the QR updating are interlaced, so that the array must be modified accordingly. Also, from the algorithmic description, it follows that the V-matrix should be stored as well. Hence, we have to provide for an additional square array, which furthermore performs the matrix-vector products $a^T \cdot V$. It is shown how the the matrix-vector product, the QR updating, and the SVD can be pipelined perfectly at the cost of little computational overhead. Finally, it is briefly shown how, e.g., a total least squares solution can be generated at each time step, with only very few additional computations.

Figure 1 gives an overview of the array. New data vectors are continuously fed into the left-hand side of the array. The matrix-vector product is computed in the square part, and the resulting vector is passed on to the triangular array that performs the QR updating and the SVD diagonalization. Output vectors are flushed upwards in the triangular array and become available at the right-hand side of the square array. All these operations can be carried out simultaneously, as is detailed next. The correctness of the array has also been verified by software simulation.

We first briefly review the SVD array of [10], and then modify the Gentleman–Kung QR updating array accordingly. Next, we show how to interlace the matrix-vector products, the QR updates, and the SVD process, and additionally generate (total least squares) output vectors.

3.1. SVD array. Figure 2 shows the SVD array of [10]. Processors on the main diagonal perform 2×2 SVDs, annihilating the available off-diagonal elements. Row

[1] If the number of rotations after each QR update is, for instance, halved or doubled, the array can easily be modified accordingly.

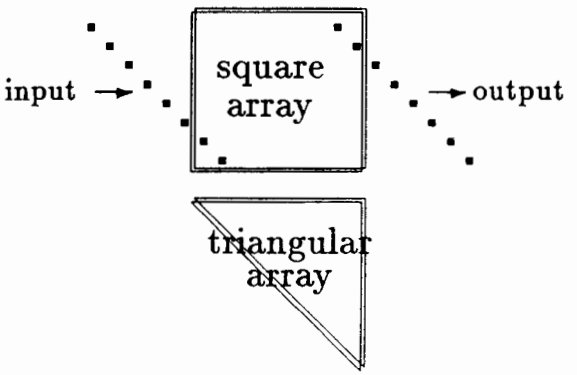


FIG. 1. *Overview SVD updating array*

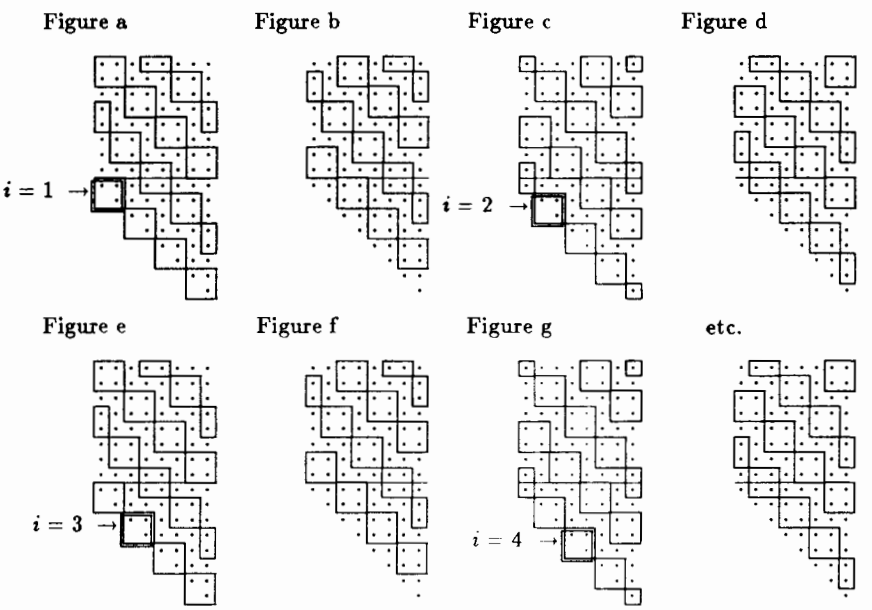


FIG. 2. SVD *array*.

transformation parameters are passed on to the right, while column transformation parameters are passed on upwards. Off-diagonal processors only apply and propagate these transformations to the next blocks outward. Column transformations are also propagated through the upper square part, containing the V-matrix (V's first row in the top row, etc.).

In this parallel implementation, off-diagonal elements with odd and even row numbers are being zeroed in an alternating fashion (*odd-even ordering*). However, it can easily be verified that an odd–even ordering corresponds to a cyclic-by-row or -column ordering, apart from a different start-up phase [11], [14]. The 2×2 SVDs that are performed in parallel on the main diagonal can indeed be thought of

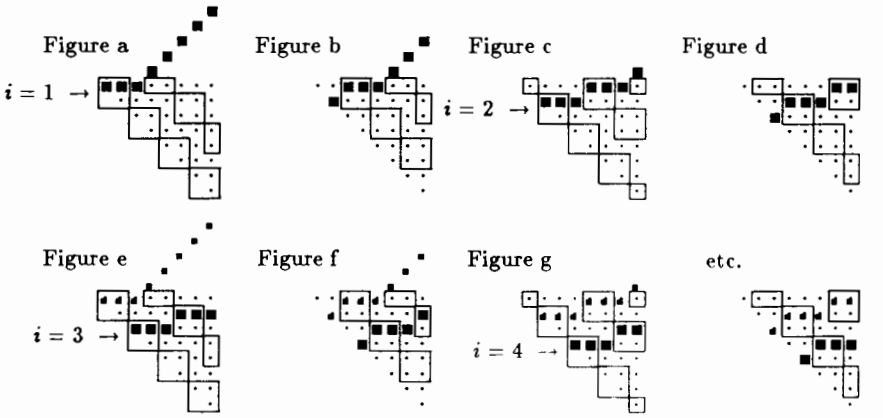


FIG. 3. *Modified Gentleman–Kung array.*

as corresponding to different pipelined sequences of $n-1$ rotations, where in each sequence the pivot index successively takes up the values $i = 1, \ldots, n-1$. A series of n such sequences is known to correspond to a double sweep (pipelined forward + backward) in a cyclic-by-rows ordering. In Fig. 2, one such sequence is indicated with double frames (for $i = 1, \ldots, 7$), starting in Fig. 2(a). In a similar fashion, the next sequence starts off from the top left corner in Fig. 2(e). As pointed out in §2, the QR updatings should be inserted in between two such sequences.

3.2. A modified Gentleman–Kung QR updating array. A QR updating is performed by applying a sequence of orthogonal transformations (*Givens rotations*) [6]. Gentleman and Kung have shown how pipelined sequences of Givens rotations can be implemented on a systolic array (see [5]). This array should now be matched to the SVD array, such that both can be combined.

Figure 3 shows a modified QR updating array. While all operations remain unaltered, the pipelining is somewhat different, so that the data vectors are now propagated through the array in a slightly different manner. The data vectors are fed into the array in a skewed fashion, as indicated, and are propagated downwards while being changed by successive row transformations. On the main diagonal, elementary orthogonal row transformations are generated. Rotation parameters are propagated to the right, while the transformed data vector components are passed on downwards. Note that each 2×2 block combines its first row with the available data vector components and pushes the resulting data vector components one step downwards. The first update starts off in Fig. 3(a) (large, filled boxes), the second in Fig. 3(e) (smaller, filled boxes), etc. Furthermore, each update is seen to correspond to a sequence of rotations where the pivot index takes up the values $i = 1, \ldots, n$. Both the processor's configuration and the pipelining turn out to be the same as for the SVD array.

3.3. Matrix-vector product. The matrix-vector product $a^T \cdot V$ can be combined with the SVD steps, as depicted in Figs. 4(a)–(g). The data vectors a^T are fed into the array in a skewed fashion, as indicated, and are propagated to the right, *in between two rotation fronts corresponding to the* SVD *diagonalization* (*frames*). Each processor receives a-components from its left neighbor, and intermediate results from its lower neighbor. The intermediate results are then updated and passed on to the upper neighbor, while the a-component is passed on to the right. The resulting

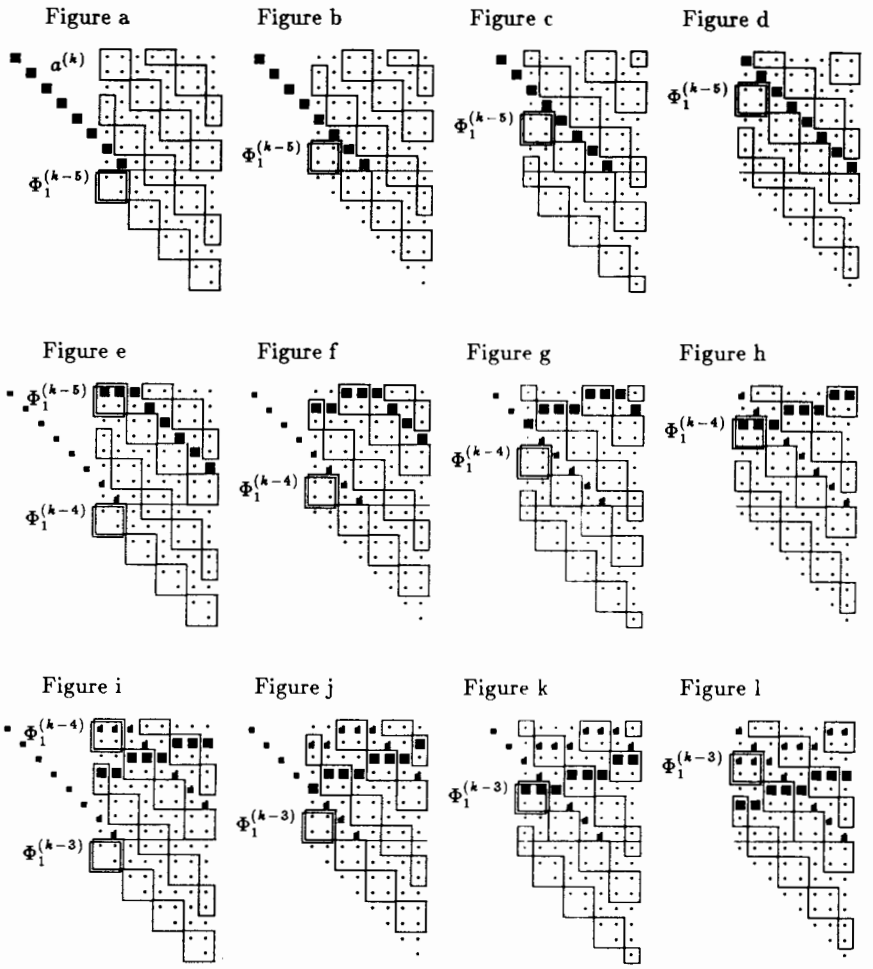


FIG. 4. SVD *updating array*.

matrix-vector product becomes available at the top end of the square array.

It should be stressed that a consistent matrix-vector product $a \cdot V$ *can only* be formed in between two SVD rotation fronts. That is a restriction, and it is worthwhile analyzing its implications.

—First, the propagation of the SVD rotation fronts dictates the direction in which a matrix-vector product can be formed. The resulting vector $a_\star$ thus inevitably becomes available at the top end of the square array, while it should be fed into the triangular array at the bottom for the subsequent QR update. The $a_\star$-components therefore have to be reflected at the top end and propagated downwards, towards the triangular array (Figs. 4(e)–(p)). The downward propagation of an $a_\star$-vector is then carried out in exactly the same manner as the propagation in the modified Gentleman–Kung array (see also Fig. 3).

—Second, the V-matrix that is used for computing $a^T \cdot V$ is in fact some older

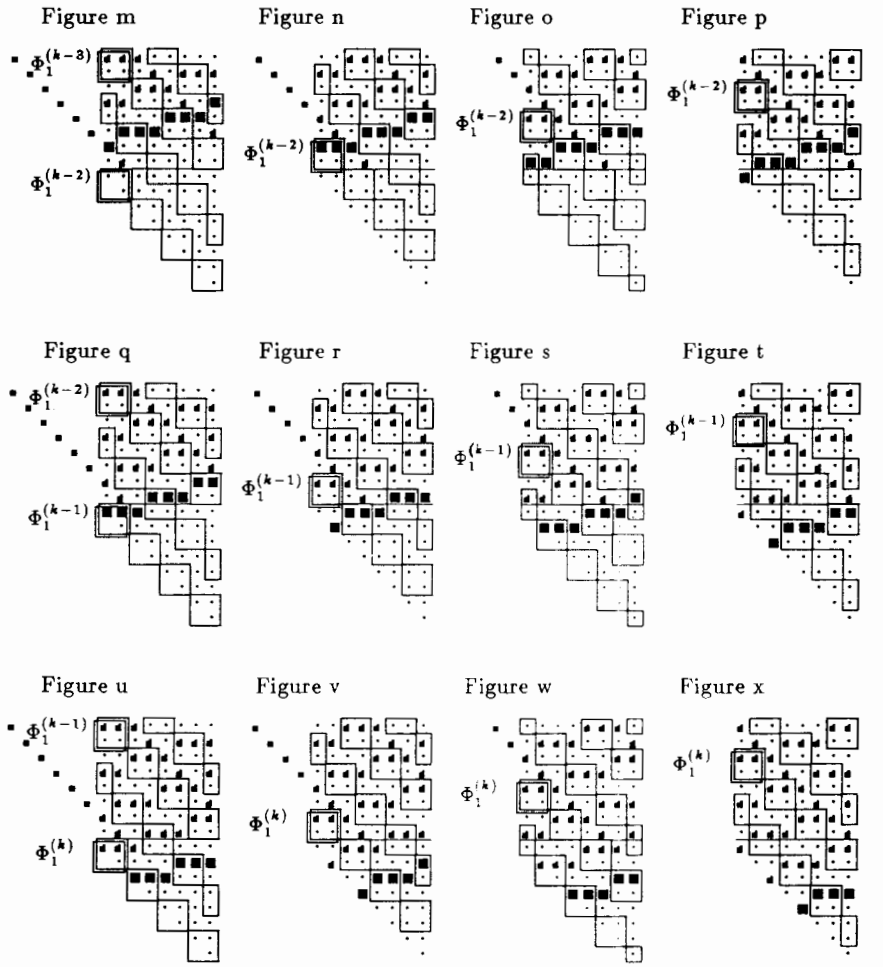


FIG. 4 (*continued*).

version of V, which we term $V^{(\natural)}$.[2] For a specific input vector $a^{(k)}$, this $V^{(\natural)}$ equals $V^{(k-1)}$ up to a number of column transformations, such that

$$V^{(k-1)} = V^{(\natural)} \cdot \Phi^{(\natural)},$$

where $\Phi^{(\natural)}$ denotes the accumulated column transformations. In order to obtain $a_\star^{(k)}$, it is necessary to apply $\Phi^{(\natural)}$ to the computed matrix-vector product

$$a_\star^{(k)} = a^{(k)^T} \cdot V^{(k-1)} = \underbrace{a^{(k)^T} \cdot V^{(\natural)}}_{a_\star^{(\natural)^T}} \cdot \Phi^{(\natural)}.$$

These additional transformations represent a computational overhead, which is the penalty for pipelining the matrix-vector products with the SVD steps on the same

[2] One can check that it is not possible to substitute a specific time index for the "$\natural$."

array. Notice, however, that at the same time, the throughput improves greatly. Waiting until $V^{(k-1)}$ is formed completely before calculating the matrix-vector product would induce $O(n)$ time lags and likewise result in an $O(n^{-1})$ throughput. With the additional computations, the throughput is $O(n^0)$.

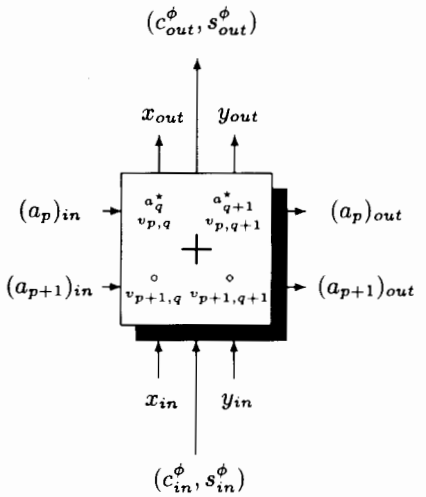
Let us now focus on the transformations in $\Phi^{(k)}$ and the way these can be processed. One of these transformations is, e.g., $\Phi_1^{(k-5)}$ (see §2 for notation), which is computed on the main diagonal in Fig. 4(a) (double frame). While propagating downwards, the $a_\star^{(k)^T}$-vector crosses the upgoing rotation $\Phi_1^{(k-5)}$ in Fig. 4(e). At this point, this transformation can straightforwardly be applied to the available $a_\star^{(k)^T}$-components. Similarly, one can verify that $\Phi_1^{(k-4)}$, $\Phi_1^{(k-3)}$, $\Phi_1^{(k-2)}$, and $\Phi_1^{(k-1)}$ are applied in Figs. 4(h), 4(k), 4(n), and 4(q), respectively. The transformations in the other columns can be traced similarly. In conclusion, each frame in the square array now corresponds to a column transformation that is applied to a 2×2 block of the V-matrix *and* to the two available components of an $a_\star^{(k)}$-vector. These components are propagated one step downwards next. A complete description for a 2×2 block in the V-matrix is presented in Display 1. Notation is slightly modified for conciseness, and $\circ$ and $\diamond$ represent memory cells that are filled by the updated elements of the $a_\star^{(k)}$-vector.

By the end of Fig. 4(p), the first $a_\star$-vector leaves the square array in a form directly amenable to the (modified Gentleman–Kung) triangular array.

3.4. Interlaced QR updating and SVD diagonalization. Finally, the modified Gentleman–Kung array and the triangular SVD array are easily combined (Fig. 4(q)–(x)). In each frame, column and row transformations corresponding to the SVD diagonalization are performed first (see also Fig. 2), while in a second step, only row transformations are performed corresponding to the modified QR updating (affecting the $a_\star$-components and the upper part of the 2×2 -blocks (see also Fig. 3). Again, column transformations in the first step should be applied to the $a_\star$-components as well. Boundary cells and internal cells are described in Displays 2 and 3.

Without disturbing the array operations, it is possible to output particular singular vectors (e.g., total least squares solutions [15]) at regular time intervals. This is easily done by performing matrix-vector multiplications $V \cdot t$, where t is a vector with all its components equal to zero, except for one component equal to 1, and which is generated on the main diagonal. The t-vector is propagated upwards to the square array, where the matrix-vector product $V \cdot t$ is performed, which singles out the appropriate right singular vector. While t is propagated upwards, intermediate results are propagated to the right, such that the resulting vector becomes available at the right-hand side of the array. These solution vectors can be generated at the same rate as the input data vectors are fed in, and both processes can run simultaneously without interference.

4. Including re-orthogonalizations. In [12], it was shown how additional re-orthogonalizations stabilize the overall round-off error propagation in the updating scheme. In the algorithmic description of §2, $T_i^{(k)}$ is an approximate re-orthogonalization and normalization of rows p and q in the V-matrix. The row indices p and q are chosen as functions of k and i, in a cyclic manner. Furthermore, the re-orthogonalization scheme was shown to converge quadratically. In view of efficient parallel implementation, we first reorganize this re-orthogonalization scheme. The modified scheme is then easily mapped onto the systolic array. The computational overhead



Input Transformation Parameters

$$c^{\phi}, s^{\phi} \leftarrow c^{\phi}_{in}, s^{\phi}_{in}$$

Apply Transformation

$$\begin{bmatrix} a^{\star}_{q} & a^{\star}_{q+1} \\ v_{p,q} & v_{p,q+1} \\ v_{p+1,q} & v_{p+1,q+1} \end{bmatrix} \leftarrow \begin{bmatrix} a^{\star}_{q} & a^{\star}_{q+1} \\ v_{p,q} & v_{p,q+1} \\ v_{p+1,q} & v_{p+1\ q+1} \end{bmatrix} \begin{bmatrix} s^{\phi} & c^{\phi} \\ c^{\phi} & -s^{\phi} \end{bmatrix}$$

Propagate $a^{\star}_{q}, a^{\star}_{q+1}$

$$\circ, \diamond \leftarrow a^{\star}_{q}, a^{\star}_{q+1}$$

Propagate Transformation Parameters

$$c^{\phi}_{out}, s^{\phi}_{out} \leftarrow c^{\phi}, s^{\phi}$$

if $q =$ **even**

Input a_p, a_{p+1} and Intermediate Results

$$a_p, a_{p+1}, x, y \leftarrow (a_p)_{in}, (a_{p+1})_{in}, x_{in}, y_{in}$$

Update Intermediate Result

$$\begin{aligned} x &\leftarrow x + v_{p,q} \cdot a_p + v_{p+1,q} \cdot a_{p+1} \\ y &\leftarrow y + v_{p,q+1} \cdot a_p + v_{p+1,q+1} \cdot a_{p+1} \end{aligned}$$

Propagate a_p, a_{p+1} and intermediate results

$$(a_p)_{out}, (a_{p+1})_{out}, x_{out}, y_{out} \leftarrow a_p, a_{p+1}, x, y$$

end

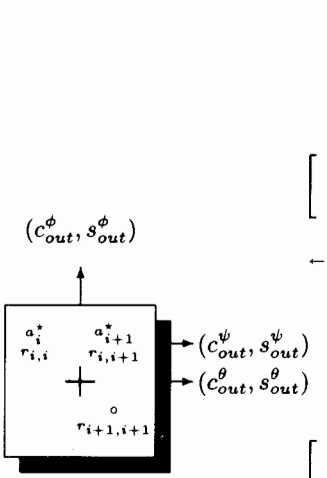
DISPLAY 1. *Internal cell V-matrix.*

turns out to be negligible, as the square part of the array (V-matrix) so far remained underloaded, compared to the triangular part (see below for figures).

First of all, as the re-orthogonalization scheme cyclicly adjusts the row vectors in the V-matrix, it is straightforward to introduce additional *row permutations* in the square part of the array. The 2×2 blocks in the square part then correspond to column transformations (SVD scheme) and row permutations (re-orthogonalization scheme). Orthogonal column transformations clearly do not affect the norms and inner products of the rows, except for local rounding errors assumed smaller than the accumulated errors. Hence, the column transformations are assumed not to interfere with the re-orthogonalization and thus need not be considered anymore. As for the row permutations, subsequent positions for the elements in the first column of V are indicated in Fig. 5, for a (fairly) arbitrary initial row numbering (as an example, the 2×2 block in the upper-left corner in Fig. 5(a) interchanges elements 4 and 5, etc.).

Let us now focus on *one* single row (row 1) and see how it can (approximately) be normalized and orthogonalized with respect to *all other* rows Later, we will use this in an overall procedure.

1. In a first step, the norm (squared) and inner products are computed as a matrix-vector product



Compute Rotation Angles ϕ, θ

$$c^{\phi}, s^{\phi}, c^{\theta}, s^{\theta} \leftarrow \cos\phi, \sin\phi, \cos\theta, \sin\theta$$

Apply Transformation

$$\begin{bmatrix} a^{\star}_i & a^{\star}_{i+1} \\ r_{i,i} & 0 \\ & r_{i+1,i+1} \end{bmatrix} \leftarrow \begin{bmatrix} 1 & & \\ & s^{\theta} & c^{\theta} \\ & c^{\theta} & -s^{\theta} \end{bmatrix} \begin{bmatrix} a^{\star}_i & a^{\star}_{i+1} \\ r_{i,i} & r_{i,i+1} \\ & r_{i+1,i+1} \end{bmatrix} \begin{bmatrix} s^{\phi} & c^{\phi} \\ c^{\phi} & -s^{\phi} \end{bmatrix}$$

Compute Rotation Angle ψ

$$c^{\psi}, s^{\psi} \leftarrow \cos\psi, \sin\psi$$

Apply Transformation

$$\begin{bmatrix} r_{i,i} & r_{i,i+1} \\ 0 & a^{\star}_{i+1} \end{bmatrix} \leftarrow \begin{bmatrix} c^{\psi} & s^{\psi} \\ -s^{\psi} & c^{\psi} \end{bmatrix} \begin{bmatrix} r_{i,i} & 0 \\ a^{\star}_i & a^{\star}_{i+1} \end{bmatrix}$$

Propagate $a^{\star}_{i+1}$

$$\circ \leftarrow a^{\star}_{i+1}$$

Propagate Transformation Parameters

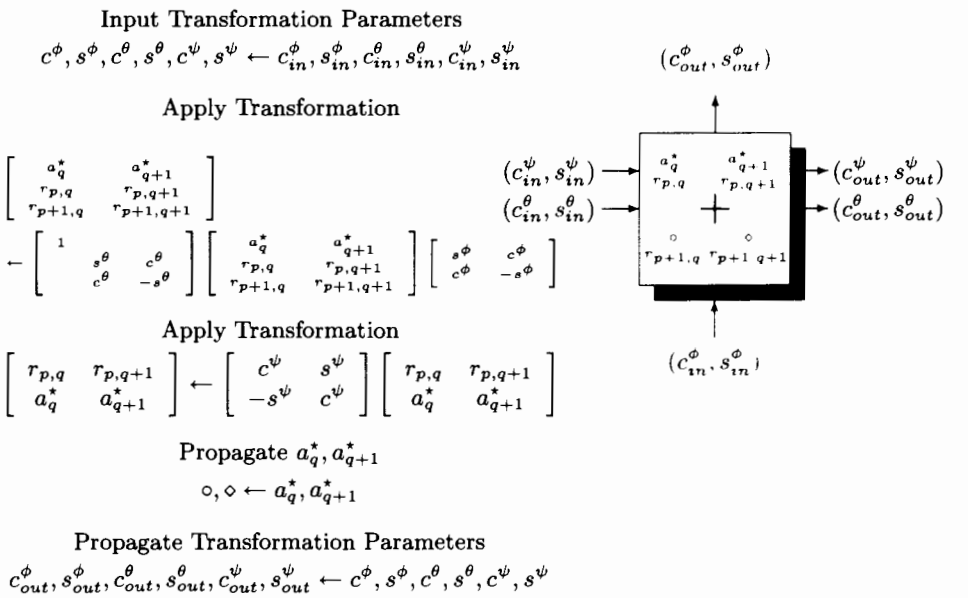
$$c^{\phi}_{out}, s^{\phi}_{out}, c^{\theta}_{out}, s^{\theta}_{out}, c^{\psi}_{out}, s^{\psi}_{out} \leftarrow c^{\phi}, s^{\phi}, c^{\theta}, s^{\theta}, c^{\psi}, s^{\psi}$$

DISPLAY 2. *Boundary cell R-factor.*

$$x_1 \stackrel{\text{def}}{=} V \cdot v_1 = \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} \cdot \begin{bmatrix} v_1 \end{bmatrix} \stackrel{\text{def}}{=} \begin{bmatrix} \xi_{11} \\ \xi_{12} \\ \vdots \\ \xi_{1n} \end{bmatrix},$$

where v_i^T is the ith row in V. On the systolic array, where v_1 initially resides in the bottom row, it suffices to propagate the v_1-components upwards, and accumulate the inner products from the left to the right (Figs. 5(a)–(h), where pq is shorthand for ξ_{pq}). The resulting x_1-vector components run out at the right-hand side.

2. In a second step, this x_1-vector is back-propagated to the left (Figs 5(e)–(t)). Due to the permutations along the way, the x_1-components reach the left-hand side of the array at the right time and in the right place, such that



DISPLAY 3. *Internal cell R-factor.*

3. in a third step, a *correction vector* can be computed as a matrix-vector product

$$y_1 \stackrel{\text{def}}{=} \left[\begin{array}{cccc} \frac{1}{2}(\xi_{11}-1) & \xi_{12} & \cdots & \xi_{1n} \end{array} \right] \cdot \left[\begin{array}{c} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{array} \right],$$

where the term $\frac{1}{2}(\xi_{11}-1)$ corresponds to the first-order term in the Taylor series expansion for the normalization of v_1. The y_1-vector components are accumulated from the bottom to the top, while x_1 is again being propagated to the right (Fig. 5(q)–(w)). Finally,

4. in a fourth step, v_1, which meanwhile moved on to the top row of the array, is adjusted with y_1:

$$v_1^\star \leftarrow v_1 - y_1.$$

These operations are performed in the top row of the array (Figs. 5(t)–(w)). One can check that if

$$\begin{aligned} v_1 \cdot v_1 &= 1 + O(\epsilon), \\ v_1 \cdot v_p &= O(\epsilon), \qquad p = 2, \ldots, n \end{aligned}$$

for some small $\epsilon \ll 1$, then

$$v_1^\star \cdot v_1^\star = 1 + O(\epsilon^2),$$

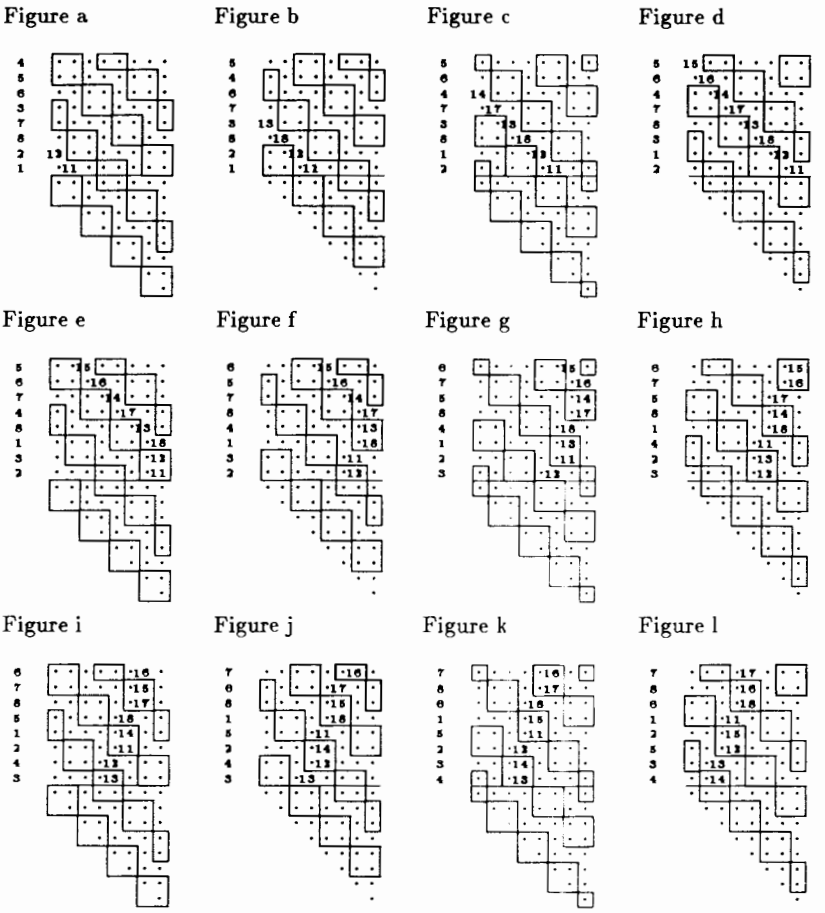


FIG. 5. *Re-orthogonalizations.*

$$v_1^* \cdot v_p = O(\epsilon^2), \qquad p = 2, \ldots, n.$$

The above procedure for v_1 should now be repeated for rows 2, 3, etc., and furthermore, everything should be pipelined. Obviously, one could start a similar procedure for v_2 in Fig. 5(e), for v_3 in Fig. 5(i), etc. The pipelining of such a scheme would be remarkably simple, but unfortunately there is something wrong with it. A slight modification is needed to make things work properly.

As for the processing of v_2, one easily checks that the computed inner product ξ_{21} equals $v_2 \cdot v_1 = v_1 \cdot v_2 = \xi_{12}$, while it *should* equal $v_2 \cdot v_1^*$. A similar problem occurs with ξ_{31} and ξ_{32}, etc. In general, problems occur when computing inner products with *ascending rows*, which still have to be adjusted in the top row of the array before the relevant inner product can be computed. This problem is readily solved as follows. Instead of computing inner products with *all* other rows, we only take *descending rows* into account. This is easily done by assigning *tags* to the rows, where, e.g., a 0-tag indicates an ascending row, and a 1-tag indicates a descending row. Tags are

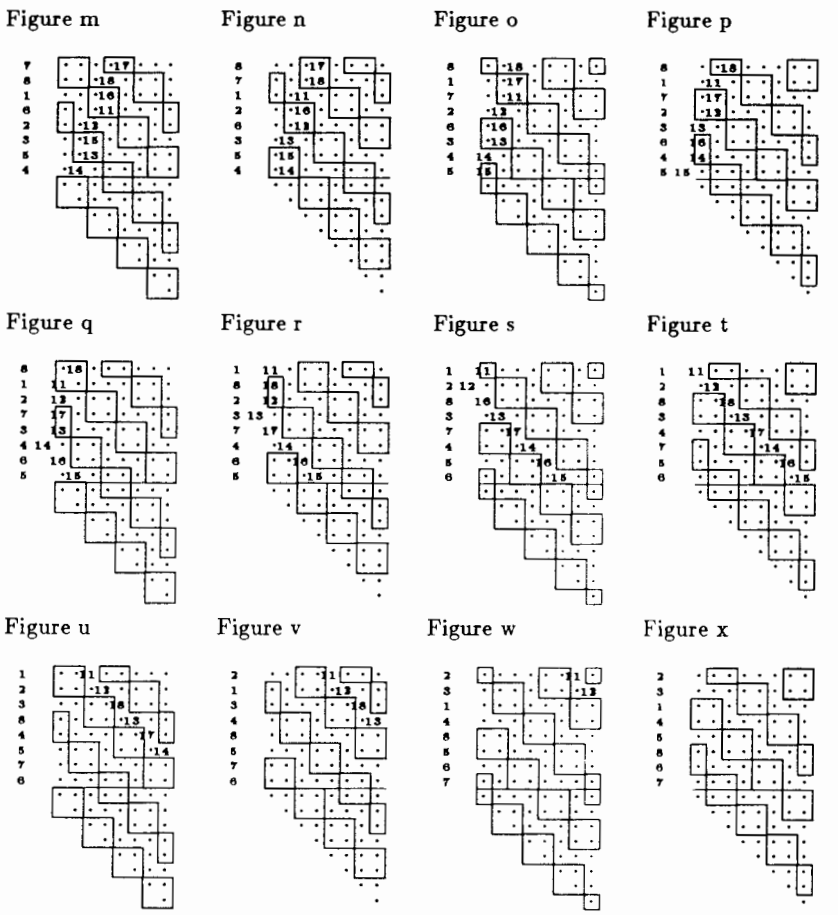


FIG. 5 (*continued*).

reset at the top and the bottom of the array. Computing an x_i vector is then done as follows:

$$x_i \stackrel{\text{def}}{=} V \cdot v_i = \begin{bmatrix} \boxed{\text{TAG}_1 v_1^T} \\ \boxed{\text{TAG}_2 v_2^T} \\ \vdots \\ \boxed{\text{TAG}_n v_n^T} \end{bmatrix} \cdot \begin{bmatrix} \boxed{v_i} \end{bmatrix} \stackrel{\text{def}}{=} \begin{bmatrix} \xi_{i1} \\ \xi_{i2} \\ \vdots \\ \xi_{in} \end{bmatrix}.$$

For the 8×8 example of Fig. 5, one can check that the result of this is that v_1 is orthogonalized onto v_2, v_3, v_4, and v_5 ($\xi_{16} = \xi_{17} = \xi_{18} = 0$, because rows 6, 7, and 8 are ascending rows at that time, i.e., $\text{TAG}_6 = \text{TAG}_7 = \text{TAG}_8 = 0$). Similarly, v_2 is orthogonalized onto v_3, v_4, v_5, and v_6 ($\xi_{27} = \xi_{28} = \xi_{21} = 0$), etc. The resulting ordering is recast as follows (*pp* refers to the normalization of row p, whereas *pq* for

TABLE 1

	SVD updating	Re-orthog.	Total
Internal cell	16×	8×	24×
V-matrix	10+	8+	18+
Internal cell	28×		28×
R-matrix	14+		14+
Diagonal cell	25×		25×
R-matrix	13+		13+
	9÷		9÷
	4√·		4√·

$p \neq q$ refers to the orthogonalization of row p onto row q)

11	12	13	14	15							
	22	23	24	25	26						
		33	34	35	36	37					
			44	45	46	47	48				
				55	56	57	58	51			
					66	67	68	61	62		
						77	78	71	72	73	
							88	81	82	83	84

In such a *sweep*, each combination appears at least once (wherewith $4 = \frac{n}{2}$ combinations appear twice, e.g., 15 (51), etc.). In the general case, the rows are cyclicly normalized and orthogonalized onto the $\frac{n}{2}$ succeeding rows. One can easily prove that if $\|VV^T - I\|_F = O(\epsilon)$ before a particular sweep, then $\|VV^T - I\|_F = O(\epsilon^2)$ after the sweep. In other words, the quadratic convergence rate is maintained. On the other hand, it is seen that one single sweep takes twice the computation time for a sweep in a "normal" cyclic by rows or odd–even ordering. This is hardly an objection, as the re-orthogonalization is only meant to keep V *reasonably close* to orthogonal. The error analysis in [12] thus still applies (with slightly adjusted constant coefficients).

Complete processor descriptions are left out for the sake of brevity. Let it suffice to give an operation count for different kinds of processors. See Table 1, from which it follows that the re-orthogonalizations do not increase the load of the critical cells.

Finally, as the rows of V continuously interchange, each input vector a in Fig. 1 should be permuted accordingly, before multiplication (see §2, $a^t \cdot V = (a^t \cdot P^t) \cdot (P \cdot V)$, where P is a permutation matrix). One can straightforwardly design a kind of "preprocessor" for a, which outputs the right components of a at the right time. For the sake of brevity, we will not go into details here.

5. Square root-free algorithms. The *throughput* in the parallel SVD updating array is essentially determined by the computation times for the processors on the main diagonal to calculate the rotation angles both for the QR updatings and the SVD steps. In general, these computations require, respectively, one and three square roots, which appears to be the main computational bottleneck. Gentleman developed a square root-free procedure for QR updating [1], [4], [7] where use is made of a (one-sided) factorization of the R-matrix. The SVD schemes as such, however, do not lend themselves to square root-free implementation. Still, in [3] a few alternative SVD schemes have been investigated based on *approximate* formulas for the computation of either $\tan\theta$ or $\tan\phi$. When combined with a (generalized) Gentleman procedure with a two-sided factorization of the R-factor, these schemes eventually yield square

root-free SVD updating algorithms. Implementation on a systolic array hardly imposes any changes when compared to the conventional algorithm. Furthermore, as the approximate formulas for the rotation angles are in fact (at least) first-order approximations, the (first-order) performance analysis in [12] still applies. In other words, the same upper bounds for the tracking error are valid, even when approximate formulas are used.

5.1. Square root-free SVD computations. The SVD procedure is seen to reduce to solving elementary 2×2 SVDs on the main diagonal (§2). For an *approximate* SVD computation, the relevant transformation formula becomes

$$\begin{bmatrix} r^*_{i,i} & r^*_{i,i+1} \\ 0 & r^*_{i+1,i+1} \end{bmatrix} = \begin{bmatrix} -\sin\theta & \cos\theta \\ \cos\theta & \sin\theta \end{bmatrix} \begin{bmatrix} r_{i,i} & r_{i,i+1} \\ 0 & r_{i+1,i+1} \end{bmatrix} \begin{bmatrix} -\sin\phi & \cos\phi \\ \cos\phi & \sin\phi \end{bmatrix},$$

where $r_{i,i+1}$ is only being approximately annihilated ($|r^*_{i,i+1}| \leq |r_{i,i+1}|$). In particular, the following approximate schemes from [3] turn out to be very useful for our purpose. (For details, refer to [3].)

if $|r_{i,i}| \geq |r_{i+1,i+1}|$

$$\sigma = \frac{r_{i+1,i+1} r_{i,i+1}}{r^2_{i,i} - r^2_{i+1,i+1} + r^2_{i,i+1}}$$

approximation 1: $\tan\theta = \sigma$
approximation 2: $\tan\theta = \frac{\sigma}{1+\sigma^2}$

$$\tan\phi = \frac{r_{i+1,i+1}\tan\theta + r_{i,i+1}}{r_{i,i}}$$

if $|r_{i,i}| \leq |r_{i+1,i+1}|$

$$\sigma = \frac{r_{i,i} r_{i,i+1}}{r^2_{i+1,i+1} - r^2_{i,i} + r^2_{i,i+1}}$$

approximation 1: $\tan\phi = \sigma$
approximation 2: $\tan\phi = \frac{\sigma}{1+\sigma^2}$

$$\tan\theta = \frac{r_{i,i}\tan\phi - r_{i,i+1}}{r_{i+1,i+1}}$$

In the sequel, we consider only $|r_{i,i}| \geq |r_{i+1,i+1}|$, as the derived formulas can straightforwardly be adapted for the other case. These approximate schemes still require two square roots for the computation of $\cos\phi$ and $\cos\theta$.

The above approximate formulas can, however, be combined with a (generalized) Gentleman procedure, where use is made of a two-sided factorization of the R-matrix

$$R = D^{\frac{1}{2}}_{row} \cdot \bar{R} \cdot D^{\frac{1}{2}}_{col}$$

and where only $\bar{R}$, D_{row}, and D_{col} are stored (in the sequel, an overbar always refers to a factorization).

Let us first rewrite the first approximate formula for $\tan\theta$:

$$\tan\theta = \underbrace{\left(\frac{r_{i+1,i+1}^2}{r_{i,i}^2 - r_{i+1,i+1}^2 + r_{i,i+1}^2}\right)}_{\rho} \cdot \frac{r_{i,i+1}}{r_{i+1,i+1}}.$$

As ρ contains only squared values, it can be computed from the factorization of R as well:

$$\rho = \frac{d_{i+1}^{row} d_{i+1}^{col} \bar{r}_{i+1,i+1}^2}{d_i^{row} d_i^{col} \bar{r}_{i,i}^2 - d_{i+1}^{row} d_{i+1}^{col} \bar{r}_{i+1,i+1}^2 + d_i^{row} d_{i+1}^{col} \bar{r}_{i,i+1}^2}.$$

Obviously, a similar formula for ρ can be derived from the second approximate formula for $\tan\theta$ (which has better convergence properties; see [3]). Applying Gentleman's procedure to the *row transformation* then gives

$$\begin{bmatrix} -\sin\theta & \cos\theta \\ \cos\theta & \sin\theta \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_i^{row}} & 0 \\ 0 & \sqrt{d_{i+1}^{row}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}_{i,i} & \bar{r}_{i,i+1} \\ 0 & \bar{r}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_i^{col}} & 0 \\ 0 & \sqrt{d_{i+1}^{col}} \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{d_{i+1}^{row}}\cos\theta & 0 \\ 0 & \sqrt{d_i^{row}}\cos\theta \end{bmatrix} \cdot \begin{bmatrix} -\tan\theta\sqrt{\frac{d_i^{row}}{d_{i+1}^{row}}} & 1 \\ 1 & \tan\theta\sqrt{\frac{d_{i+1}^{row}}{d_i^{row}}} \end{bmatrix}$$

$$\cdot \begin{bmatrix} \bar{r}_{i,i} & \bar{r}_{i,i+1} \\ 0 & \bar{r}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_i^{col}} & 0 \\ 0 & \sqrt{d_{i+1}^{col}} \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{d_i^{row^*}} & 0 \\ 0 & \sqrt{d_{i+1}^{row^*}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}_{i,i}^{\circ} & \bar{r}_{i,i+1}^{\circ} \\ \bar{r}_{i+1,i}^{\circ} & \bar{r}_{i+1,i+1}^{\circ} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_i^{col}} & 0 \\ 0 & \sqrt{d_{i+1}^{col}} \end{bmatrix}.$$

With

$$\tan\theta = \rho \cdot \frac{\bar{r}_{i,i+1}\sqrt{d_i^{row}}}{\bar{r}_{i+1,i+1}\sqrt{d_{i+1}^{row}}},$$

this leads to row transformation formulas

$$\begin{bmatrix} \bar{r}_{i,i}^{\circ} & \bar{r}_{i,i+1}^{\circ} \\ \bar{r}_{i+1,i}^{\circ} & \bar{r}_{i+1,i+1}^{\circ} \end{bmatrix} = \begin{bmatrix} -\rho \cdot \frac{\bar{r}_{i,i+1} d_i^{row}}{\bar{r}_{i+1,i+1} d_{i+1}^{row}} & 1 \\ 1 & \rho \cdot \frac{r_{i,i+1}}{\bar{r}_{i+1,i+1}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}_{i,i} & \bar{r}_{i,i+1} \\ 0 & \bar{r}_{i+1,i+1} \end{bmatrix}$$

with scale factor updating (notice the implicit row permutation)

$$d_i^{row^*} = d_{i+1}^{row}\cos^2\theta,$$
$$d_{i+1}^{row^*} = d_i^{row}\cos^2\theta,$$

$$\cos^2\theta = \frac{1}{1 + \rho^2 \cdot \frac{\bar{r}_{i,i+1}^2 d_i^{row}}{\bar{r}_{i+1,i+1}^2 d_{i+1}^{row}}}.$$

Note that due to the 1's in the first transformation formula, a 50 percent saving in the number of multiplications is obtained in the off-diagonal processors.

The *column transformation* should then annihilate $r^{\circ}_{i+1,i}$ in order to preserve the triangular structure. With

$$\tan\phi = \frac{\bar{r}^{\circ}_{i+1,i+1}\sqrt{d^{col}_{i+1}}}{\bar{r}^{\circ}_{i+1,i}\sqrt{d^{col}_{i}}}$$

we can again apply Gentleman's procedure as follows:

$$\begin{bmatrix} \sqrt{d_i^{row^*}} & 0 \\ 0 & \sqrt{d_{i+1}^{row^*}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}^{\circ}_{i,i} & \bar{r}^{\circ}_{i,i+1} \\ \bar{r}^{\circ}_{i+1,i} & \bar{r}^{\circ}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_i^{col}} & 0 \\ 0 & \sqrt{d_{i+1}^{col}} \end{bmatrix} \cdot \begin{bmatrix} -\sin\phi & \cos\phi \\ \cos\phi & \sin\phi \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{d_i^{row^*}} & 0 \\ 0 & \sqrt{d_{i+1}^{row^*}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}^{\circ}_{i,i} & \bar{r}^{\circ}_{i,i+1} \\ \bar{r}^{\circ}_{i+1,i} & \bar{r}^{\circ}_{i+1,i+1} \end{bmatrix}$$

$$\cdot \begin{bmatrix} -\tan\phi\sqrt{\frac{d^{col}_i}{d^{col}_{i+1}}} & 1 \\ 1 & \tan\phi\sqrt{\frac{d^{col}_{i+1}}{d^{col}_i}} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_{i+1}^{col}}\cos\phi & 0 \\ 0 & \sqrt{d_i^{col}}\cos\phi \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{d_i^{row^*}} & 0 \\ 0 & \sqrt{d_{i+1}^{row^*}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}^{*}_{i,i} & \bar{r}^{*}_{i,i+1} \\ 0 & \bar{r}^{*}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d_i^{col^*}} & 0 \\ 0 & \sqrt{d_{i+1}^{col^*}} \end{bmatrix},$$

which then leads to column transformation formulas

$$\begin{bmatrix} \bar{r}^{*}_{i,i} & \bar{r}^{*}_{i,i+1} \\ 0 & \bar{r}^{*}_{i+1,i+1} \end{bmatrix} = \begin{bmatrix} \bar{r}^{\circ}_{i,i} & \bar{r}^{\circ}_{i,i+1} \\ \bar{r}^{\circ}_{i+1,i} & \bar{r}^{\circ}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} -\frac{\bar{r}^{\circ}_{i+1,i+1}}{\bar{r}^{\circ}_{i+1,i}} & 1 \\ 1 & \frac{\bar{r}^{\circ}_{i+1,i+1}d^{col}_{i+1}}{\bar{r}^{\circ}_{i+1,i}d^{col}_{i}} \end{bmatrix}$$

with scale factor updating

$$d_i^{col^*} = d_{i+1}^{col}\cos^2\phi,$$
$$d_{i+1}^{col^*} = d_i^{col}\cos^2\phi,$$

$$\cos^2\phi = \frac{1}{1 + \frac{\bar{r}^{\circ 2}_{i+1,i+1}d^{col}_{i+1}}{\bar{r}^{\circ 2}_{i+1,i}d^{col}_{i}}}.$$

5.2. Square root-free SVD updating. The above approximate SVD schemes straightforwardly combine with the square root-free QR updating procedure into a square root-free SVD updating procedure. At a certain time step, the data matrix is reduced to R, which is stored in factorized form

$$R = D_{row}^{\frac{1}{2}} \cdot \bar{R} \cdot D_{col}^{\frac{1}{2}}.$$

Furthermore, the same column scaling is applied to the V-matrix

$$V = \bar{V} \cdot D_{col}^{\frac{1}{2}},$$

where $\bar{V}$ is stored instead of V. The reason for this is twofold. First, the column rotations to be applied to the V-matrix are computed as modified Givens rotations. Explicitly applying these transformations to an unfactorized V would then necessarily require square roots. Second, a new row vector a^T to be updated immediately gets

TABLE 2

	SVD updating	Re-orthog.	Total
Internal cell	10×	12×	22×
V-matrix	10+	8+	18+
Internal cell	14×		14×
R-matrix	14+		14+
Diagonal cell	35×		35×
R-matrix	17+		17+
	9÷		9÷

the correct column scaling from the matrix-vector product $a^T \cdot \bar{V}$, so that the QR updating can then be carried out as if there were no column scaling at all. The updating can indeed be described as follows:

$$\begin{aligned}\begin{bmatrix} A \\ a^T \end{bmatrix} &= \begin{bmatrix} U & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} D_{row}^{\frac{1}{2}} \cdot \bar{R} \cdot D_{col}^{\frac{1}{2}} \\ a^T \cdot (\bar{V} \cdot D_{col}^{\frac{1}{2}}) \end{bmatrix} \cdot V^T \\ &= \begin{bmatrix} U & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} D_{row}^{\frac{1}{2}} & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \bar{R} \\ a^T \cdot (\bar{V}) \end{bmatrix} \cdot D_{col}^{\frac{1}{2}} \cdot V^T \\ &= \text{etc.}\end{aligned}$$

The factor in the midst of this expression can then be reduced to a triangular factor by QR updating, making use of modified Givens rotations. The further reduction of the resulting triangular factor can be carried out next, as detailed in the previous section.

From the above explanation, it follows that on a systolic array, a square root-free updating algorithm imposes hardly any changes. The diagonal matrices D_{row} and D_{col} are obviously stored in the processor elements on the main diagonal, and $\bar{R}$ and $\bar{V}$ are stored instead of R and V. The matrix-vector product $\bar{a}_\star^T = a^T \cdot \bar{V}$ is computed in the square part, and the $\bar{R}$-factor is updated with $\bar{a}_\star^T$ next, much like the R-factor was updated with $a_\star^T = a^T \cdot V$ in the original algorithm. All other operations are carried out much the same way, albeit that modified rotations are used throughout. When re-orthogonalizations are included, it is necessary to propagate the scale factors to the square array, along with the column transformation, such that the norms and inner products can be computed consistently. The rest is straightforward.

Finally, an operation count for a square root-free implementation is exhibited in Table 2. Note that the operation count for the diagonal processors depends heavily on the specific implementation. We refer to the literature for various (more efficient) implementations [1], [7]. The operation count for V-processors remains unchanged (as compared to Table 1), while the computational load for the diagonal processors is reduced to roughly the same level, apart from the divisions. The internal R-processors are seen to be underloaded this time, due to the reduction in the number of multiplications.

6. Conclusion. An approximate SVD updating procedure was mapped onto a systolic array with $O(n^2)$ parallelism for $O(n^2)$ complexity. By combining modified Givens rotations with approximate schemes for the computation of rotation angles in the SVD steps, all square roots can be avoided. In this way, a main computational bottleneck for the array implementation can be overcome.

REFERENCES

[1] J. L. BARLOW AND I. C. F. IPSEN, *Scaled Givens rotations for the solution of linear least squares problems on systolic arrays*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 716–733.

[2] R. P. BRENT AND F. T. LUK, *The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 69–84.

[3] J. P. CHARLIER, V. VANBEGIN, AND P. VAN DOOREN, *On efficient implementations of Kogbetliantz's algorithm for computing the singular value decomposition*, Numer. Math., 52 (1988), pp. 279–300.

[4] W. M. GENTLEMAN, *Least squares computations by Givens transformations without square roots*, J. Inst. Math. Appls., 12 (1973), pp. 329–336.

[5] W. M. GENTLEMAN AND H. T. KUNG, *Matrix triangularization by systolic arrays*, Real-Time Signal Processing IV, Proc. SPIE, Vol. 298, 1981, pp. 19–26.

[6] P. E. GILL, G. H. GOLUB, W. MURRAY, AND M. A. SAUNDERS, *Methods for modifying matrix factorizations*, Math. Comp., 28 (1974), pp. 505–535.

[7] S. HAMMARLING, *A note on modifications to the Givens plane rotations*, J. Inst. Math. Appls., 13 (1974), pp. 215–218.

[8] M. T. HEATH, A. J. LAUB, C. C. PAIGE, R. C. WARD, *Computing the singular value decomposition of a product of two matrices*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 1147–1159.

[9] E. KOGBETLIANTZ, *Solution of linear equations by diagonalization of coefficient matrices*, Quart. Appl. Math., 13 (1955), pp. 123–132.

[10] F. T. LUK, *A triangular processor array for computing singular values*, Linear Algebra Appl., 77 (1986), pp. 259–273.

[11] F. T. LUK AND H. PARK, *On the equivalence and convergence of parallel Jacobi* SVD *algorithms*, in Proc. SPIE, Vol. 826, Advanced Algorithms and Architectures for Signal Processing II, F. T. Luk, ed., 1987 pp. 152–159.

[12] M. MOONEN, P. VAN DOOREN, AND J. VANDEWALLE, *An* SVD *updating algorithm for subspace tracking*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 1015–1038.

[13] J. M. SPEISER, *Signal processing computational needs*, in Proc. SPIE, Vol. 696, Advanced Algorithms and Architectures for Signal Processing, J. M. Speiser, ed., 1986, pp. 2–6.

[14] G. W. STEWART, *A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 853–863.

[15] S. VAN HUFFEL AND J. VANDEWALLE, *Algebraic relations between classical regression and total least squares estimation*, Linear Algebra Appl., 93 (1987), pp. 149–162.