Parallel VLSI Matrix Pencil Algorithm for High Resolution Direction Finding

Alle-Jan van der Veen, *Student Member, IEEE*, and Ed F. Deprettere, *Senior Member, IEEE*

***Abstract*—In this paper, we consider algorithms to find the directions of arrival (DOA's) of multiple signals from measurements on an array of antenna doublets (ESPRIT method), and their parallel implementation in VLSI. In particular, we look for algorithms that allow large scale pipelining and use only robust, unitary transformations. The main problem is to find the generalized Schur decomposition (GSD) of a matrix pencil. We tackle it using a modified Stewart Jacobi approach for which convergence is improved and parameter computations are simplified. The resulting architecture is a two-layer Jacobi array that can handle all the subproblems: two *QR* factorizations, two SVD's, and a single GSD. An ideal processing element is the pipelined CORDIC.**

I. Introduction

A FUNDAMENTAL problem in (real-time) signal processing is the recovery of deterministic signal parameters from noisy observations. One interesting case is the estimation of the directions of arrival (DOA's) of a number of waves impinging on an array of sensors or antennas. Equivalent problems are the harmonic and echo retrieval problems. Our objective here is to study how these problems can be translated into VLSI architectures that exhibit the special properties of "integral parallelism" or pipelining, and intrinsic numerical accuracy.

The keynote related to integral parallelism is that at no point in the algorithm a large accumulation of data is allowed to occur. An algorithm may typically be represented by a precedence or activity graph [2]. If this graph contains long critical cycles, then the processing of a new set of data has to wait for the conclusion of the current cycle. The data involved has to be stored temporally, and the algorithm essentially breaks down into two distinct stages. An example is the traditional solution of a system of linear equations $Ax = b$ using a QR factorization of A: $A = QR$. The backsubstituting stage $Rx = Q^{-1}b$ will use the data from the factorization in reverse order, requiring large amounts of internal storage: the precedence graph shows a large critical cycle. The difficulty here is solved by utilizing a different algorithm, called orthogonal Faddeev [3].

The algorithms for DOA may be studied for a similar purpose. We strive for what we call a "maximal pipelining" of data: a data flow that allows a regular timing and that is as much unidirectional from input to output as is possible. Another consideration is the uniformity of the various stages comprising the pipeline. If all stages share a common structure, the mapping of the algorithm onto a reduced size processor array will be feasible. For the same reason, we want to minimize the broadcast of data and control parameters among processors. Massive pipelining and optimal processor utilization becomes possible this way.

A requirement related to such integral parallelism is that all processing must be accurate, since iterative (convergence) loops are to be avoided. As a result, we shall only allow operations without error amplification, in practice orthogonal (unitary) transformations. Our architecture synthesis problem then consists of finding a pipelined architecture of processing elements executing unitary transformations.

In this paper, we focus on the DOA problem using the ESPRIT model [4]. This problem is solved by a matrix pencil approach in which the generalized eigenvalues are determined of a pair of data matrices. The resulting pencil algorithm can be subdivided into a sequence of a few stages, each of which can be implemented on a parallel machine. It turns out that the same processor array can be used for each stage. Moreover, as only unitary transformations are to be applied, the main processing element turns out to be a CORDIC, which is a device that can compute the angle of a two-dimensional vector or that is able to rotate a two-dimensional vector over a given angle.

The outline of the paper is as follows. In Section II, the ESPRIT technique for the DOA problem is quickly reviewed. In Section III, we present a parallel algorithm to solve the problem with a high degree of pipelining and robustness. The algorithm consists of two parallel SVD's, followed by a generalized Schur decomposition (GSD). Both decompositions are first phrased in terms of the Jacobi iteration algorithm (Section IV), and then more explicitly using CORDIC processors (Sections V–VII). In Section VII, we also discuss a modification to the Schur decomposition algorithm due to Stewart [5]. It is based on the use of "inner rotations" followed by permutations, rather than "outer rotations," and reveals improved convergence in a number of examples. Finally, in Section VIII, we consider the mapping of the subproblems on a single parallel array of CORDIC processors.

II. ESPRIT Direction-of-Arrival Estimation Model

The aim of the direction-of-arrival (DOA) estimation problem is the determination of the angles of arrival of a number of signals impinging on a sensor array. The ESPRIT model of Roy [4] and Paulraj *et al.* [6] for high-resolution DOA estimation is discussed below. The model can be extended to include wideband signals [7] and is also directly applicable to the spectral estimation problem [8]. Various other DOA models exist, see, e.g., [9] for an overview.

Consider a planar array composed of m pairs of pairwise identical sensors (doublets). The displacement Δ between the two sensors in each doublet is constant, but the sensor charac-

Manuscript received July 22, 1989; revised March 3, 1990. This work was supported by the Dutch National Applied Science Foundation under Contract STW DEL 47.0643. This work was presented at the SPIE Conference on Advanced Algorithms and Architectures III, 1988.

The authors are with the Department of Electrical Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands.

IEEE Log Number 9041155.

1053-587X/91/0200-0383$01.00 © 1991 IEEE

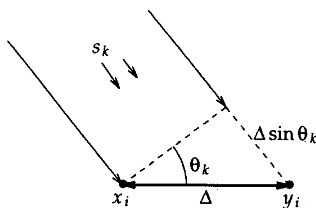

Fig. 1. Angle of arrival (one doublet).

teristics are unknown and the geometry of the doublets is arbitrary. We thus have two identical, although displaced, arrays. Incident on both arrays are d narrow-band noncoherent signals $s_k(t) = \hat{s}_k e^{j\omega_0 t}$, each having an unknown complex amplitude $\hat{s}_k$, and a known center frequency ω_0 which is the same for all signals. Noise is present at each sensor, and is assumed to be additive, stationary, and of zero mean. The signal received by sensor i in the first array at time t_j can then be modeled by

$$x_i(t_j) = \sum_{k=1}^{d} a_{ik} s_k(t_j) + n_{xi}(t_j) \tag{2.1}$$

where a_{ik} is the gain of sensor i in the direction of signal s_k, and n_{xi} is the noise present at sensor i. If the number of time samples is N, we can collect the N observations of the m sensors in a matrix X:

$$X = AS + N_x \tag{2.2}$$

where $X \triangleq [x_i(t_j)]$, $A \triangleq [a_{ik}]$, $S \triangleq [s_k(t_j)]$, and $N_x \triangleq [n_{xi}(t_j)]$. X is called the data matrix (dimension $m \times N$), A is the array gain matrix ($m \times d$), and S is the signal matrix ($d \times N$). Matrices A and S are not known.

We also measure the signals at the sensors of the displaced arrays. Due to the spatial distance between the two sensors in a doublet, each signal s_k experiences a phase shift (delay) ϕ_k, which is directly dependent on the incident angle θ_k of the signal s_k and is in complex notation given by $\phi_k = e^{-j\omega_0 \Delta \cdot \sin\theta_k / c}$ (see Fig. 1), where Δ is the constant displacement and c is the signal propagation velocity. Thus, the signal received by sensor i of the second array, at time t_j, can be modeled as

$$y_i(t_j) = \sum_{k=1}^{d} a_{ik} \phi_k s_k(t_j) + n_{yi}(t_j) \tag{2.3}$$

and the data matrix for this array, Y say, will now obey

$$Y = A\Phi S + N_y \tag{2.4}$$

where $\Phi = \text{diag}(\phi_1, \cdots, \phi_d)$. The direction finding problem thus reduces to estimating Φ, from which the θ_k can be computed directly.

III. A Pencil Solution to the DOA Problem

A. Problems

We are now able to state the DOA problem in the following concise way.

Given two data matrices X and Y (dimensions $m \times N$), such that

$$X = AS + N_x$$
$$Y = A\Phi S + N_y \tag{3.1}$$

where A is an unknown array gain matrix ($m \times d$), S is an unknown signal matrix ($d \times N$), $\Phi = \text{diag}(\phi_1, \cdots, \phi_d)$ is a diagonal matrix of phase shifts, and N_x and N_y are independent noise matrices ($m \times N$).

Then find d (the number of signals impinging on the array) and Φ, assuming that the matrices A and S are of full rank d, and $N \geq m \geq d$. Note that all matrices are complex. ■

Assume for the moment that there is no noise present. To solve the problem in this case, it is sufficient to form the matrix pencil $X - \lambda Y$ [10], and to find for $i = 1, \cdots, d$ those values λ_i for which the rank of $X - \lambda_i Y$ is one less than the rank of X and Y. Indeed, we now have $X - \lambda Y = A(I - \lambda\Phi)S$, so when λ equals one of the ϕ_k^{-1}, the rank of the pencil is reduced by one. Hence, from these rank reducing numbers we can determine the signal parameters. These numbers are equal to the generalized eigenvalues of X and Y in case these matrices are square, as generalized eigenvalues λ are defined to be the nontrivial solutions to $Xx = \lambda Yx$ [10].

In practice, however, the data is corrupted by noise. Noise introduces new rank reducing numbers that do not correspond to incident signals, and also reduces the accuracy of the other rank reducing numbers. To improve accuracy, a large quantity of data samples are taken (N large), resulting in nonsquare data matrices X and Y. However, generalized eigenvalues are not defined for nonsquare matrices, and a different method is needed to compute the rank reducing numbers. Some solution methods are discussed by Golub and van Loan [10], Roy [4], Speiser [11], van Loan [12], and Ouibrahim [13]. Most methods reduce the problem to the generalized eigenvalue problem by computing data covariance matrices. In general, the implicit squaring of data involved here tends to make matrix condition numbers worse, thus deteriorating numerical stability and accuracy. One method that operates directly on the observed data is the total least squares approach discussed by Roy [4]. Details about TLS techniques can be found in [10], [14], [15].

TLS, like ordinary least squares, is a technique for solving an overdetermined set of linear equations $Fx = b$. An exact solution x exists only if b is in the column space of F. If b has been disturbed by noise, an LS approximation is obtained by projecting b onto this column space. TLS recognizes the fact that F may also contain errors, and projects F and b onto a "common" subspace, having minimal distance to both b and the columns of F. This technique is also valid if x and b are extended to matrices. From the observation that, without noise in (3.1), X and Y share a common column space $span(A)$ and a common row space $span(S^H)$, it is concluded that the TLS technique can be used to estimate these spaces if there is noise, in which case the data matrix X corresponds to F, and the data matrix Y to b. By exploiting these subspace properties, the following pencil algorithm can be derived, as was done by Roy [4].

B. Original TLS Pencil Method

The original TLS pencil method [4] is as follows.

1) Use the TLS technique to determine the common row space of X and Y. In the noise free case this space is equal to the row space of S and is of dimension d. X and Y are projected onto this space, which reduces their dimension to $m \times d$.

2) In the same way, determine the common column space of X and Y, thus estimating the column space of A. This estimation is used to further reduce the dimensions of X and Y to $d \times d$.

3) Next, proceed as in the noise free case. Compute the generalized eigenvalues of the resulting pair of $d \times d$ data matrices,

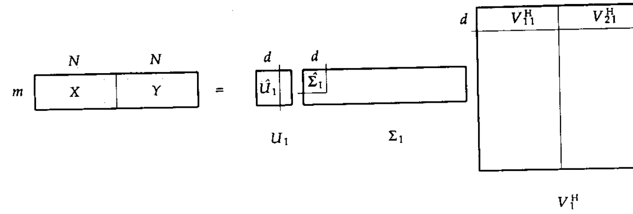

Fig. 2. TLS estimation of *span*(A).

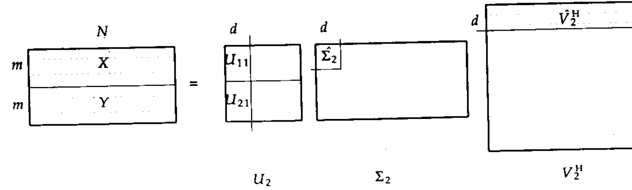

Fig. 3. TLS estimation of *span*(S^H).

which make up the entries of Φ. From these, compute the angles of arrival.

C. Parallel TLS Pencil Method

In the above method, first the common row space is computed, and the matrices are projected onto this space; then, from the resulting matrices the common column space is computed. We will now describe a method to compute both subspaces in parallel, and reduce the dimensions of X and Y to $d \times d$ at the same time. Simulations show that both methods yield essentially the same numerical results.

The TLS approximation to project X and Y onto their d-dimensional common column space can be computed directly from the singular value decomposition of $[X \quad Y]$:

$$[X \quad Y] = U_1 \Sigma_1 V_1^H \tag{3.2}$$

which is visualized in Fig. 2. In this decomposition, U_1 and V_1 are unitary matrices characterizing the column space and row space of $[X \quad Y]$, respectively. The positive diagonal matrix Σ_1 collects the singular values σ_i, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_m$, that are weighting the importance of vectors in U_1 and V_1. The $(m - d)$ smallest σ's as well as the corresponding columns in U_1 and V_1 are removed by TLS so as to bring the system back to a noise-free-like case in which only d nonzero singular values are present. Thus, from Fig. 2 the data matrices X and Y are approximated by

$$\begin{aligned} X &\approx \hat{U}_1 \hat{\Sigma}_1 V_{11}^H \\ Y &\approx \hat{U}_1 \hat{\Sigma}_1 V_{21}^H \end{aligned} \tag{3.3}$$

where $\hat{\Sigma}_1$ is the $d \times d$ diagonal matrix containing the d largest singular values of $[X \quad Y]$. The matrix $\hat{U}_1$ is the TLS approximation of the column space of A.

In the same way, and concurrently, we can estimate the common row space of X and Y by computing the SVD

$$\begin{bmatrix} X \\ Y \end{bmatrix} = U_2 \Sigma_2 V_2^H \tag{3.4}$$

which is visualized in Fig. 3. Proceeding in exactly the same way as above we construct noise-free-like versions of the data matrices X and Y satisfying

$$\begin{aligned} X &\approx U_{11} \hat{\Sigma}_2 \hat{V}_2^H \\ Y &\approx U_{21} \hat{\Sigma}_2 \hat{V}_2^H. \end{aligned} \tag{3.5}$$

The matrix $\hat{V}_2$ characterizes the estimated common row space of X and Y, hence the row space of S. Again, $\hat{\Sigma}_2$ is the $d \times d$ diagonal matrix of the d largest singular values of $\left[\begin{smallmatrix} X \\ Y \end{smallmatrix}\right]$.

We now have two TLS approximations (3.3) and (3.5) of X and Y: one in terms of a common column space $\hat{U}_1$, the other in terms of a common row space $\hat{V}_2$. The idea, now, is that they can be combined into one approximation by stating

$$\begin{aligned} X &\approx \hat{U}_1 E_x \hat{V}_2^H \\ Y &\approx \hat{U}_1 E_y \hat{V}_2^H \end{aligned} \tag{3.6}$$

thereby introducing square matrices E_x and E_y (dimensions $d \times d$), which are such that the generalized eigenvalues of the matrix pair (E_x, E_y) are equal to the rank reducing numbers of the pencil of these TLS approximations of X and Y.

The matrices E_x and E_y can be computed conveniently in the following way. Using (3.6) and the $d \times d$ identities

$$\begin{aligned} \hat{U}_1^H \hat{U}_1 &= I_{d \times d} \\ \hat{V}_2^H \hat{V}_2 &= I_{d \times d} \end{aligned} \tag{3.7}$$

we derive

$$\begin{aligned} E_x &= \hat{U}_1^H (\hat{U}_1 E_x \hat{V}_2^H) \hat{V}_2 \approx \hat{U}_1^H X \hat{V}_2 \\ E_y &= \hat{U}_1^H (\hat{U}_1 E_y \hat{V}_2^H) \hat{V}_2 \approx \hat{U}_1^H Y \hat{V}_2. \end{aligned} \tag{3.8}$$

We can also derive from (3.4) and $V_2^H \hat{V}_2 = \left[\begin{smallmatrix} I_{d \times d} \\ 0 \end{smallmatrix}\right]$ the equalities

$$\begin{aligned} X \hat{V}_2 &= U_{11} \hat{\Sigma}_2 \\ Y \hat{V}_2 &= U_{21} \hat{\Sigma}_2. \end{aligned} \tag{3.9}$$

Combining (3.8) and (3.9) we arrive at the following elegant decomposition of E_x and E_y:

$$\begin{aligned} E_x &\approx \hat{U}_1^H U_{11} \hat{\Sigma}_2 \\ E_y &\approx \hat{U}_1^H U_{21} \hat{\Sigma}_2. \end{aligned} \tag{3.10}$$

As we are only interested in the generalized eigenvalues of the pair (E_x, E_y) we can even omit the nonsingular matrix multiplier $\hat{\Sigma}_2$.

D. TLS Pencil Algorithm

We are now ready to state the essential steps of the TLS pencil algorithm:

1) Stack the measurements X and Y to form $[X \quad Y]$ and $\begin{bmatrix} X \\ Y \end{bmatrix}$. X and Y have dimension $m \times N$. Since N will eventually be large, these matrices are wide and relatively flat. Before proceeding with the TLS stage, we can reduce these matrices to square, upper triangular matrices R_1 and R_2 of dimension m and $2m$, by performing a preprocessing QR factorization

$$[X \quad Y] = [R_1 \quad 0]Q_1$$

$$\begin{bmatrix} X \\ Y \end{bmatrix} = [R_2 \quad 0]Q_2. \tag{3.11}$$

As the unitary matrices Q_1 and Q_2 are part of the unitary matrices V_1^H and V_2^H in the SVD's in (3.2) and (3.4), respectively, and as we will not use these matrices, we can throw them away and subsequently use the new, upper triangular matrices R_1 and R_2 instead of X and Y. This substantially reduces the amount of computations.

2) Compute (in parallel) the SVD's of the triangular matrices R_1 and R_2

$$U_1^H R_1 V_1 = \Sigma_1$$

$$U_2^H R_2 V_2 = \Sigma_2. \tag{3.12}$$

At the same time, construct a matrix E $(2m \times 2m)$ defined by

$$\begin{bmatrix} U_1^H & 0 \\ 0 & U_1^H \end{bmatrix} \cdot U_2 = E. \tag{3.13}$$

3) Estimate, if necessary, the number of signals d using Σ_1 and Σ_2. Then, from E, select the $d \times d$ submatrices $E(1:d, 1:d)$ and $E(m+1:m+d, 1:d)$ to obtain $\hat{U}_1^H U_{11}$ and $\hat{U}_1^H U_{21}$, respectively, as defined in the previous subsection.

4) Compute the generalized eigenvalues of the matrix pair $(\hat{U}_1^H U_{11}, \hat{U}_1^H U_{21})$. These are the TLS pencil solutions to the original matrix pencil $X - \lambda Y$. The generalized eigenvalues can be computed using a generalized Schur decomposition (GSD).

The three major steps in this algorithm are the QR factorizations, the double SVD, and the GSD. In the following sections, we shall derive a VLSI parallel array on which both the double SVD, as needed here, and the GSD can be computed. The QR factorization can also be performed on the same array, as in Luk [16], but this will not be discussed in this paper.

E. Parallel Array Definition

We close this section with a definition of the actual computations we want the processing array to perform. To begin with, in an actual parallel array implementation of the double SVD (3.12) and (3.13), it is convenient to avoid the duplication of U_1 rotations, as implied by (3.13), and to operate on matrices of equal dimension. At the expense of repeating computations, this can be done by replacing the SVD of the matrix R_1 by an SVD of an augmented matrix $\underline{R}_1$, defined by

$$\underline{R}_1 \triangleq \begin{bmatrix} R_1 & 0 \\ 0 & R_1 \end{bmatrix}. \tag{3.14}$$

This leads to the following parallel computation task:

$$\underline{U}_1^H \underline{R}_1 \underline{V}_1 = \underline{\Sigma}_1$$

$$U_2^H R_2 V_2 = \Sigma_2$$

$$\underline{U}_1^H I U_2 = E. \tag{3.15}$$

E is the desired matrix, and both $\underline{R}_1$ and R_2 are upper triangular. In the next sections we will show how $\underline{\Sigma}_1$, Σ_2 can be computed by iteratively operating upon $\underline{R}_1$, R_2; at the same time, the U-operations can be applied to the identity matrix I, resulting in the parallel construction of E. All matrices are now of equal dimension $2m \times 2m$. This is the first decomposition the array has to support.

Then, to compute the generalized eigenvalues, we use a generalized Schur decomposition algorithm. The GSD of a pair of matrices (A, B) is defined as

$$Q^H AZ = S \qquad Q, Z \text{ unitary}$$

$$Q^H BZ = T \qquad S, T \text{ upper triangular.} \tag{3.16}$$

The generalized eigenvalues of the pair (A, B) are equal to those of the pair (S, T), and can simply be extracted from their diagonals. The computation of the pair (S, T) is the second decomposition the array has to support.

In the first part of this paper we have seen that the DOA problem can be modeled by a nonsquare matrix pencil of two data matrices. A two stage unitary method to solve this pencil consists of i) a TLS approximation that separates the signal space from the noise space, which is done by computing two SVD's in parallel, and ii) a GSD that solves the problem in the noise free case. The basic operations in both decompositions are two-sided unitary transformations (U, V, or Q, Z), operating on two data matrices. So, both decompositions are of the same type, and in fact both can be computed using generalizations of the Jacobi method. This is the topic of the second part of the paper. Since the Jacobi method can often easily be extended from one matrix (single SVD/Schur decomposition) to two matrices (double SVD/GSD), we will focus on the single matrix case as much as possible.

IV. The Jacobi Iteration Method

Jacobi iteration methods are algorithms to compute the following general type of decomposition of a matrix A:

$$Q^H AZ = R \qquad Q, Z \text{ unitary.} \tag{4.1}$$

The result matrix R typically contains a lot of zero entries. It can be diagonal, in which case the SVD of A is obtained, or upper triangular for a Schur decomposition. In the latter case, $Q = Z$. Some recent papers dealing with SVD using Jacobi methods are due to Brent *et al.* [17], and Brent and Luk, [18], see also [19]. In contrast to the Jacobi-SVD, the Jacobi approach to the computation of the Schur decomposition of non-Hermitian matrices is not so well understood. Heuristic algorithms have been derived by Stewart [5] and Eberlein [20]. We restate the most important issues here.

The basic operation in the Jacobi method is a plane rotation $Q_{ij}(\theta)$ in the (i, j) plane, which is defined as a matrix which

is an identity matrix, except for the entries (i, i), (i, j), (j, i) and (j, j) which together form the 2×2 rotation matrix

$$\begin{bmatrix} q_{ii} & q_{ij} \\ q_{ji} & q_{jj} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}. \tag{4.2}$$

(In the next section, we extend this to the complex case.) To compute the decomposition (4.1) iteratively, Q and Z are composed of a sequence of plane rotations Q_{ij} and Z_{ij}, operating on the pair of rows (i, j) and the pair of columns (i, j) of A, respectively. All Jacobi methods are based on the following three notions:

1) The basic operation is the computation of the desired decomposition for the 2×2 submatrix $\left[\begin{smallmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{smallmatrix}\right]$, $i < j$. The two resulting plane rotations $Q_{ij}(\theta_1)$ and $Z_{ij}(\theta_2)$ are then applied to the matrix A via $A := Q_{ij}^H A Z_{ij}$. This results in the annihilation of the entries (a_{ij}, a_{ji}) in case of SVD, or the entry a_{ji} in case of the Schur decomposition. The new matrix A differs from the old A only in rows and columns i and j. GSD is a relatively simple extension of the ordinary Schur decomposition to a pair of matrices (A, B): rotation angles are computed from corresponding 2×2 submatrices of A and B, and the resulting plane rotations are applied to both A and B in order to annihilate the entries a_{ji} and b_{ji} simultaneously. The basic 2×2 problems (the calculation of θ_1 and θ_2) for SVD and GSD are deferred to Sections VI and VII.

2) Step 1 must be done systematically (according to some feasible ordering) for all pairs (i, j), $i < j$, to zero each pair of entries (a_{ij}, a_{ji}), respectively the entry a_{ji}, at least once. This set of operations is called a sweep. In practice, a limited number of sweeps (say 10) is needed for global convergence in case of SVD, and convergence is proven to be ultimately quadratic [21]. For GSD, the number of sweeps depends on the normality of the matrix. For strongly nonnormal matrices, convergence may be slow or even absent, and no convergence proof has been given so far.

3) Independent pairs of rows and columns can be operated upon in parallel. This is what makes the Jacobi methods highly suitable for parallel implementations.

In most multiprocessor implementations of algorithms, only nearest neighbor communication is allowed, implying here that only adjacent rows and columns are paired. The simultaneous processing of as many independent pairs $(i, i + 1)$ as possible results in a partitioning of the matrix into 2×2 submatrices (Fig. 4(a)). This subdivision can be done in two ways: the alternative grid is also shown (Fig. 4(b)). Each 2×2 block corresponds to a processor in which the four entries reside. To annihilate an entry $a_{i+1,i}$, the diagonal processor in which this entry resides computes rotations $Q_{i,i+1}(\theta_1)$ and $Z_{i,i+1}(\theta_2)$, which subsequently are applied to the corresponding rows and columns.

We now turn to the question of what ordering scheme is to be used to zero the off-diagonal entries. The purpose is to annihilate in turn each entry a_{ji} (for SVD also a_{ij}) for all pairs (i, j), $i < j$ in a sweep. As in Fig. 4, only entries $a_{i+1,i}$ and $a_{i,i+1}$ along the first subdiagonals can be annihilated, we must move the other off-diagonal entries towards the diagonal. This movement is obtained by interchanging, for each 2×2 block, the entries residing in this block, which takes place after the plane rotation has been performed on these entries. A convenient scheme to do this is the "odd-even" ordering proposed by Stewart [5]. By using permutations to interchange the two adjacent rows $(i, i + 1)$ and also the two corresponding columns,

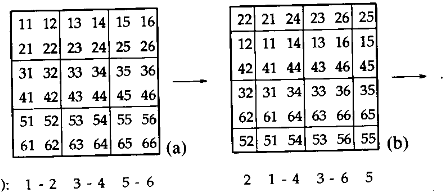

Fig. 4. (a) Partitioning of the matrix, and (b) result after one step of the odd-even ordering.

	time	
1st sweep:	1.	1–2 3–4 5–6
	2.	2 1–4 3–6 5
	3.	2–4 1–6 3–5
	4.	4 2–6 1–5 3
	5.	4–6 2–5 1–3
	6.	6 4–5 2–3 1
2nd sweep:	7.	6–5 4–3 2–1
	..	....

Fig. 5. Odd-even iteration scheme.

and by alternating the two possible processor grids (termed "raster shifting") on odd and even time steps, all pairs (i, j) are obtained. This is demonstrated in Figs. 4(b) and 5 for $m = 6$, where dashes indicate an active (i, j) pair. Instead of the above iteration scheme, other "equivalent" orderings can be used (see [19], [22] for an overview), but we prefer the odd-even ordering, as it preserves the initial upper triangular form of the matrices $\underline{R}_1$ and R_2 (3.15) during SVD computations.

Instead of an ordinary permutation, a rotation over $\pi/2$ is often used because it has essentially the same interchanging properties, but can be combined nicely with the $Q(\theta_1)$ and $Z(\theta_2)$ rotations. In general, more than one pair of rotation angles (θ_1, θ_2) can be found that solves the 2×2 decomposition. To ensure an effective movement of off-diagonal entries, the combined effect of the rotation followed by the permutation should result in "almost" a permutation. Because a large rotation tends to undo the interchanging effect of the permutation, the smallest rotation angle θ_S ("inner rotation") is chosen, and is often combined with the permutation into one rotation over an angle of $\theta_S + \pi/2$. Alternatively (and equivalently in the case of SVD), if the largest rotation angle θ_L is close enough to $\pi/2$, this rotation can be used instead of the combination. This rotation is then called an "outer rotation," and will be discussed in more detail in Sections VI and VII.

V. Basic CORDIC Arithmetic

As may be clear by now, the basic operation in the Jacobi iteration consists of a plane rotation. The corresponding tool to use is a CORDIC processor [23], which is a device that rotates a real 2-D input vector over a given angle, or computes angles in a vectorization mode by rotating an input vector to the positive X axis. The latter operation also yields the norm of the input vector. Special purpose SVD CORDIC's do exist and have recently been designed in [24], [25]. However, as we also use CORDIC's for computing the Schur decomposition, we will use a general purpose VLSI CORDIC. Our CORDIC [26] is bit-level pipelined, resulting in a high throughput rate for vector operations. The net effect of pipelining is that angles are computed in a fraction of the time needed for computing the result

of a rotation. This is emphasized in the coming figures by drawing the angle flow vertically (indicating parallelism), as opposed to the data, which flows from left to right. One side effect of the way our bit-level pipelining is implemented is that the angle bits are a nonbinary coding of the rotation angle. While this results in a higher throughput rate, it also means that the sum of two rotation angles and half the angle (as needed in the next section) cannot be obtained simply from an addition or bit shift of the corresponding angle code vectors.

Since we operate in the complex field, it is useful to construct a complex CORDIC. This CORDIC will, of course, be composed of real CORDIC's. First, we define a complex rotation as follows:

$$R(\theta, \phi_1, \phi_2) \triangleq \begin{bmatrix} e^{j\phi_1} & \\ & e^{j\phi_2} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{5.1}$$

where $c = \cos\theta$; $s = \sin\theta$.

Complex and real CORDIC representations of this rotation are shown in Fig. 6, wherein $[x \;\; y] = [x_r + jx_i \;\; y_r + jy_i]$ and $[x' \;\; y'] = [x \;\; y]R(\theta, \phi_1, \phi_2)$ are complex 2-D row vectors. Note that the first matrix in (5.1) is strictly speaking not a rotation, unless $\phi_1 = -\phi_2$. The definition presented here is chosen to simplify the computation of a real θ from complex vectors. Indeed, in vectorization mode we want to compute (θ, ϕ_1, ϕ_2) so that $[x \;\; y]R(\theta, \phi_1, \phi_2) = [x' \;\; 0]$, from which

$$\tan\theta = \frac{s}{c} = -\frac{e^{j\theta_2}}{e^{j\phi_1}} \cdot \frac{y}{x}. \tag{5.2}$$

To ensure that $\tan\theta$ is real, we choose ϕ_2 and ϕ_1 so as to make the numerator, respectively, the denominator in (5.2) real. Using a complex CORDIC, this is done by vectorizing the vector $[x \;\; y]$, switching each CORDIC in Fig. 6 to vectorization mode. Of course, ϕ_1 and ϕ_2 could be combined into one ϕ, but the computation of this ϕ would involve more operations than is the case here. Moreover, we can now use the same structure of Fig. 6 for both vectorization and rotation mode. As an aside, note that in (5.2) negative angles are computed. This is because CORDIC performs a vectorization by rotating its input vector counterclockwise towards the positive X axis. Usually, the angles obtained in this way are directly applicable to successive vector rotations. In the rare cases where positive angles are needed we will denote this by drawing a small "+" sign at the output of the CORDIC.

Finally, a rotation operating on two m-dimensional column vectors $[x \;\; y]$ to obtain $[x \;\; y]R(\theta, \phi_1, \phi_2)$ can be implemented by vertically stacking m complex rotors, acting in parallel (Fig. 7). This operation is used in the Jacobi iteration array to apply plain rotations Q_{ij} and Z_{ij} to a pair (i, j) of rows or columns, respectively. For notational convenience, the symbol $R(\theta)$ will be used for both real and complex rotations.

VI. Singular Value Decomposition

In this section, we will focus on the SVD of a 2×2 complex matrix A,

$$Q^H(\theta_1)AZ(\theta_2) = \Sigma \tag{6.1}$$

in which we assume (according to (3.15)) that A is upper triangular with real main diagonal entries:

$$A = \begin{bmatrix} a & ce^{j\alpha} \\ 0 & b \end{bmatrix}. \tag{6.2}$$

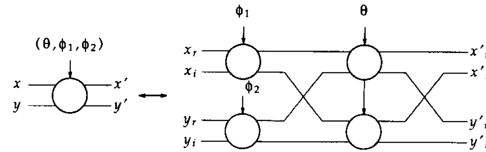

Fig. 6. Basic complex rotation.

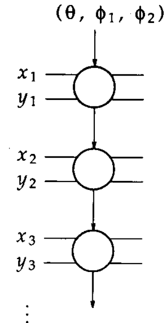

Fig. 7. Plane rotation of two column vectors.

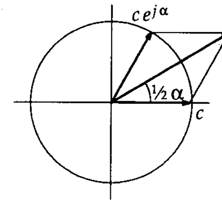

Fig. 8. Computing $\frac{1}{2}\alpha$ from $ce^{j\alpha}$.

Using complex rotations as defined in (5.1), a simple transition to the real case can be made by first rotating this matrix in the complex plane,

$$\begin{bmatrix} e^{-j\phi} & \\ & e^{j\phi} \end{bmatrix} \begin{bmatrix} a & ce^{j\alpha} \\ 0 & b \end{bmatrix} \begin{bmatrix} e^{j\phi} & \\ & e^{-j\phi} \end{bmatrix} = \begin{bmatrix} a & c \\ 0 & b \end{bmatrix} \tag{6.3}$$

from which $\phi = \frac{1}{2}\alpha$. We can compute ϕ from $ce^{j\alpha}$ as suggested by the geometry depicted in Fig. 8, leading to the CORDIC implementation shown in Fig. 9(a). This also yields c. Next, the real part of the complex rotations are used to solve the real SVD:

$$\begin{bmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{bmatrix} \begin{bmatrix} a & c \\ 0 & b \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \tag{6.4}$$

from which the two angles θ_1 and θ_2 can be computed as [17]

$$\tan(\theta_1 + \theta_2) = \frac{c}{b - a}$$

$$\tan(\theta_1 - \theta_2) = \frac{c}{b + a}. \tag{6.5}$$

A (real) CORDIC implementation to compute θ_1 is shown in Fig. 9(b). In this figure, $[x \;\; y] = [(b + a) \;\; c]R(\theta_1 + \theta_2)$, and $x' = \sqrt{(b + a)^2 + c^2}$ is the norm of $[x \;\; y]$. Since the angle between this vector $[x \;\; y]$ and the positive X axis is $2\theta_1$, the angle between the vector $[x + x' \;\; y]$ equals θ_1. After com-

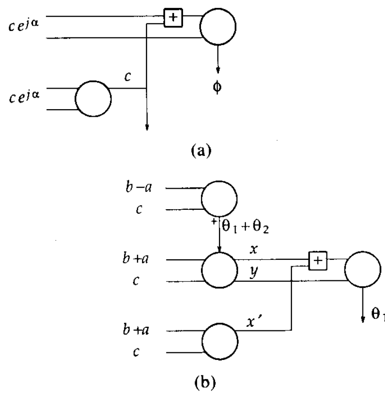

Fig. 9. SVD (using real CORDIC's). (a) Computation of $\phi = \frac{1}{2}\alpha$. (b) Computation of θ_1.

puting θ_1 from this vector, an easy way to obtain θ_2 is by first applying $Q(\theta_1)$ to A to get $Q^H A$, as in Fig. 7, and then using $Z(\theta_2)$ to make this matrix upper triangular (in fact: diagonal). θ_2 thus follows from one vectorization.

VII. Generalized Schur Decomposition

The last step in the direction finding algorithm is the computation of the generalized Schur decomposition for two square non-Hermitian matrices, A and B,

$$\begin{aligned} Q^H AZ &= S \qquad Q, Z \text{ unitary} \\ Q^H BZ &= T \qquad S, T \text{ upper triangular.} \end{aligned} \tag{7.1}$$

We will use the "Jacobi-like" Schur algorithm due to Stewart [5], extended in a straightforward way to yield the generalized decomposition. The same kind of extension has been used by Luk to compute a generalized SVD [27]. The basic operation in the extended algorithm is the computation of rotation angles for Q and Z that make 2×2 submatrices of both A and B upper triangular. Stewart's algorithm uses the odd-even ordering, and "outer rotations" that combine rotations with permutations. As Eberlein has pointed out [20], this algorithm does not always converge. She uses a different ordering scheme, both "inner" and "outer" rotations, and some heuristics to improve convergence. In this section, we derive an alternative modification to the Stewart algorithm, that uses the original ordering scheme. We give reasons why the modifications will fundamentally improve the convergence behavior of the algorithm. Simulations affirm that this is indeed the case, and that the modified algorithm has convergence properties similar to a "clean" version of Eberlein's algorithm. But first, we present a CORDIC implementation for solving the 2×2 GSD problem.

A. Solution of the 2×2 GSD Problem

The closed form solution of the 2×2 GSD problem gives rise to two coupled quadratic equations, which, unlike the SVD problem, are not easily implemented using CORDIC hardware. To tackle this, we do not compute the exact decomposition, but instead derive an accurate estimation using standard QZ iteration, which is quite fast for 2×2 matrices. The QZ method implicitly performs a QR iteration on AB^{-1}, and was derived by Moler and Stewart [28], see also [10]. The idea is that, since the Jacobi iteration algorithm is itself a convergence process, it

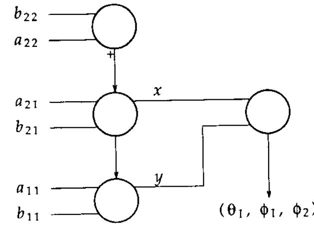

Fig. 10. GSD-computation of $Q(\theta_1)$.

is possible to use eigenvalue estimations within this process without affecting global convergence properties too much. We have observed that a fairly limited number of QZ iterations (two) provide estimations that are accurate enough for use in the Jacobi algorithm. This will be shown in Tables I–VI.

One iteration of the QZ algorithm can be summarized as follows.

1) Determine Q so that $Q^H(A - \sigma B)$ is upper triangular. Here, σ is a suitable shift (discussed below). Then apply Q to A and B, yielding $Q^H A$ and $Q^H B$.

2) Determine Z so that $Q^H BZ$ is upper triangular. Apply Z to A and B, yielding $Q^H AZ$ and $Q^H BZ$.

As in the QR iteration, a shift σ is needed to ensure convergence when both eigenvalues have almost equal absolute values [10], which frequently is the case in the DOA problem. The shift should be close to an eigenvalue of AB^{-1}, hence a suitable and simple choice for σ is a_{22}/b_{22}. It is known from QR iteration theory that a_{22}/b_{22} will converge to the eigenvalue that is closest to the σ used in the iteration step, resulting in a Q rotation which is closest to identity (again denoted as inner rotation). We will take advantage of this knowledge in the next subsection.

We now describe a CORDIC implementation for executing this 2×2 QZ algorithm. From step 1, $Q(\theta_1)$ can be computed as

$$\frac{e^{j\phi_1}}{e^{j\phi_2}} \cdot \tan\theta_1 = \frac{a_{21} - \sigma b_{21}}{a_{11} - \sigma b_{11}}$$

$$= \frac{a_{21} - \dfrac{a_{22}}{b_{22}} \cdot b_{21}}{a_{11} - \dfrac{a_{22}}{b_{22}} \cdot b_{11}} = \frac{a_{21}c - b_{21}s}{a_{11}c - b_{11}s} \triangleq \frac{y}{x} \tag{7.2}$$

in which the angle defined by

$$\frac{s}{c} \triangleq \frac{a_{22}}{b_{22}} \tag{7.3}$$

is computed using a CORDIC vectorization. After this, x and y are found using CORDIC rotations over this angle, and $(\theta_1, \phi_1, \phi_2)$ can be computed in a second stage (Fig. 10). In general, all matrix entries are complex, so complex CORDIC's as in Fig. 6 are used here. After applying Q to A and B, the proper $Z(\theta_2)$ rotation (step 2) is found by making the new B upper triangular. Again, this is done using one vectorization. Note the close resemblance of Fig. 10 and the SVD computations in Fig. 9.

The QZ iteration algorithm consists of a repeated execution of the above process on the same 2×2 block (we use two iterations). This loop is a major difference with SVD, where no

TABLE I
GSD ESPRIT Convergence Test. Test Matrices Used in Table II

$$A = \begin{bmatrix} .7018 - .0039i & -.1682 - .0087i & -.0013 - .0237i & .0184 - .0122i \\ .0220 - .0018i & .6928 - .0069i & -.3152 - .0335i & -.1024 - .0188i \\ .0000 + .0004i & .0031 - .0016i & .6088 + .0064i & .0309 - .1119i \\ .0003 - .0008i & .0004 + .0005i & .0005 - .0935i & .4256 + .3374i \end{bmatrix}$$

$$B = \begin{bmatrix} .4557 - .5467i & .1124 - .1202i & .0329 + .0024i & .0030 + .0165i \\ -.0129 + .0165i & .4547 - .5076i & .2284 - .2181i & .0846 - .0708i \\ -.0003 - .0004i & -.0013 + .0031i & .4657 - .4164i & -.0408 - .0073i \\ .0004 + .0007i & .0000 - .0001i & .0216 + .0676i & -.2284 - .5650i \end{bmatrix}$$

TABLE II
GSD ESPRIT Convergence Test

Sweep	Stewart Exact	"Eberlein" Exact	Modified Exact	Modified Approx (2)
0	1.3e − 1	1.3e − 1	1.3e − 1	1.3e − 1
1	5.5e − 3	8.9e − 3	5.5e − 3	5.4e − 1
2	3.7e − 3	2.7e − 4	6.8e − 3	3.3e − 2
3	2.4e − 3	2.6e − 7	1.6e − 3	1.4e − 2
4	1.8e − 3	2.4e − 13	5.9e − 5	3.0e − 3
5	1.2e − 3	2.1e − 25	8.1e − 8	1.7e − 4
6	9.1e − 4		1.5e − 13	7.0e − 7
7	5.8e − 4		5.6e − 25	1.2e − 11
8	4.5e − 4			3.1e − 21
9	2.9e − 4			
10	2.2e − 4			

Convergence is measured via the Frobenius norm of the strictly lower (or upper) triangle of AB^{-1}. Both exact 2 × 2 GSD's and approximations using 2 QZ iterations have been used. "Eberlein" is actually implemented as parallel ordering, and use of θ_S and permutations only.

TABLE III
GSD ESPRIT Convergence Test

Scenario	Stewart Exact	"Eberlein" Exact	Modified Exact	Modified Approx (2)
1	no conv.	13–15	13–16	12–21
2	no conv.	11–14	12–17	12–17
3	no conv.	9–13	7–9	9–11

Test matrices in an 8 × 8 example are from ESPRIT Scenarios (1)–(3). (1) Eight signals impinging on the array at angles increasing by 10°; (2) same case but two of the signals are now spaced by 5° from other signals; (3) 6 signals spaced by 10°, but evaluated for $d = 8$ (as due to overestimation). For each test run, the sweep is marked for which the norm of the error first drops below 10^{-14}; a dozen tests are run for each scenario; the table lists the range of the set of marked sweeps.

iteration is needed and new 2 × 2 problems result after each application of Q and Z due to the raster shifting. In an actual implementation, we want the structure of the GSD and SVD computations to be similar. A simple way to achieve this is obtained by "unfolding" the loop and making the raster shift conditional: to iterate, skip the raster shift, and the processors will operate on the same 2 × 2 matrices in the next time step. To end the loop, perform a raster shift.

B. Sweeps and Convergence

We have experimented with a generalized Schur decomposition based on Stewart's Schur algorithm [5]. This GSD turned out to have bad convergence properties in a number of simulations (see Tables I–III for an ESPRIT simulation example). Because the same problem also arises in the single Schur decomposition ($Q^H AQ = S$), we will focus on this decomposition for a while. Of particular interest is the near-convergence behavior of algorithms, for an algorithm should sustain this convergence rather than diverge.

In Stewart's Schur algorithm, the odd-even ordering is used, and 2 × 2 submatrices are made upper triangular using the largest of the two possible angles (called θ_S and θ_L). It is assumed that this θ_L is close enough to $\frac{1}{2}\pi$ to effect the required circulation of off-diagonal entries. However, as Eberlein [20] has pointed out, it is not guaranteed that θ_L is even larger than $\pi/4$. In particular, if the current 2 × 2 matrix is almost upper triangular (which is the case when the full matrix is near convergence):

$$\begin{bmatrix} c_{11} & c_{12} \\ \epsilon & c_{22} \end{bmatrix} \xrightarrow{\theta} \begin{bmatrix} c'_{11} & c'_{12} \\ 0 & c'_{22} \end{bmatrix} \tag{7.4}$$

then, for ϵ small in comparison with $c_{11} - c_{22}$, the tangents of θ_S and θ_L can be approximated by

$$t_S \approx \frac{\epsilon}{c_{11} - c_{22}}$$
$$t_L \approx \frac{c_{11} - c_{22}}{c_{12}}. \tag{7.5}$$

If the matrix is nonnormal, we could have $c_{12} > c_{11} - c_{22}$, hence $\theta_L < \pi/4$, and there is no rotation that is large enough to satisfy the assumptions stated above, i.e., that effectively permutes the matrix and keeps it upper triangular at the same time.[1] The choice to keep the matrix upper triangular, then, obstructs the necessary permutation. As this can occur even near convergence, the unconditional use of θ_L is not the right thing to do. This is confirmed by the simulation results.

We can overcome this problem by making the following two observations:

[1]Note that in SVD, a $|\theta_L| > \pi/4$ always exists because a permuted diagonal matrix is again diagonal, so $\theta_L = \theta_S \pm \pi/2$.

TABLE IV
GSD CONVERGENCE FOR STEWART MATRICES, $\alpha = 0.1$

Sweep	Stewart Exact	"Eberlein" Exact	Modified Exact	Modified Approx (2)
0	1.7e + 0	1.7e + 0	1.7e + 0	1.7e + 0
1	4.1e − 1	2.2e − 1	4.1e − 1	4.1e − 1
2	1.1e − 2	1.3e − 2	2.7e − 2	2.6e − 2
3	1.1e − 4	2.0e − 5	6.1e − 4	6.1e − 4
4	3.2e − 7	8.4e − 10	5.0e − 8	6.8e − 7
5	1.6e − 9	3.0e − 15	2.7e − 11	9.0e − 12
6	5.9e − 12	8.7e − 23	7.6e − 20	2.5e − 19
7	3.1e − 14			
8	1.6e − 16			

TABLE V
GSD CONVERGENCE FOR STEWART MATRICES, $\alpha = 1$

Sweep	Stewart Exact	"Eberlein" Exact	Modified Exact	Modified Approx (2)
0	2.1e + 0	2.1e + 0	2.1e + 0	2.1e + 0
1	1.2e + 0	9.1e − 1	1.2e + 0	9.3e − 1
2	4.2e − 1	2.5e − 1	6.7e − 1	1.9e − 1
3	1.5e − 1	2.4e − 2	3.8e − 1	3.5e − 2
4	4.0e − 2	1.2e − 3	8.0e − 2	1.7e − 3
5	1.1e − 2	3.1e − 6	3.8e − 2	1.1e − 5
6	3.8e − 3	3.3e − 10	3.9e − 4	7.1e − 9
7	1.0e − 3	4.9e − 14	5.2e − 7	9.8e − 15
8	3.9e − 4	1.7e − 20	3.5e − 11	
9	1.1e − 4		5.3e − 18	
10	4.1e − 5			
11	1.1e − 5			
12	4.3e − 6			
13	1.2e − 6			
14	4.6e − 7			

TABLE VI
GSD CONVERGENCE FOR STEWART MATRICES, $\alpha = 10$

Sweep	Stewart Exact	"Eberlein" Exact	Modified Exact	Modified Approx (3)
0	1.5e + 1	1.5e + 1	1.5e + 1	1.5e + 1
1	3.4e + 0	3.5e + 0	3.4e + 0	4.4e + 0
2	1.4e + 0	3.1e + 0	1.6e + 0	6.8e + 0
...				
13	3.2e − 1	1.2e − 1	4.5e − 3	4.5e − 2
14	3.0e − 1	7.5e − 2	3.3e − 4	9.6e − 2
15	2.8e − 1	1.3e − 1	1.9e − 6	4.4e − 2
16	2.6e − 1	9.8e − 2	1.3e − 9	5.9e − 3
17	2.5e − 1	4.8e − 2	2.0e − 14	1.2e − 4
18	2.4e − 1	1.4e − 2		1.3e − 7
19	2.3e − 1	2.7e − 3		1.2e − 11
20	2.1e − 1	6.9e − 5		2.1e − 18
21	2.0e − 1	4.0e − 7		
22	2.0e − 1	4.8e − 10		
23	1.9e − 1	5.7e − 14		

1) Near convergence, ϵ is very small, hence (from 7.5) θ_S is very small.

2) In the Jacobi algorithm, effective permutations are essential, hence the basic step should be the use of a rotation angle as small as possible (which is θ_S) followed by a permutation. The point is that the resulting net angle $\theta_S + \frac{1}{2}\pi$ is not equivalent to θ_L, as we now end up with a lower triangular matrix

$$\begin{bmatrix} c_{11} & c_{12} \\ \epsilon & c_{22} \end{bmatrix} \xrightarrow{\theta_S + (1/2)\pi} \begin{bmatrix} c'_{22} & 0 \\ c'_{12} & c'_{11} \end{bmatrix}. \tag{7.6}$$

During a sweep, using (7.6) as a basic step, a matrix that is initially almost upper triangular is turned into an almost lower triangular matrix:

$$\begin{bmatrix} \times & \times & \times & \times \\ \epsilon & \times & \times & \times \\ \epsilon & \epsilon & \times & \times \\ \epsilon & \epsilon & \epsilon & \times \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} \times & 0 & \times & \times \\ \times & \times & \times & \times \\ \epsilon & \epsilon & \times & 0 \\ \epsilon & \epsilon & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \epsilon' & \times \\ \epsilon & \times & 0 & \epsilon' \\ \times & \times & \times & \times \\ \epsilon & \times & \epsilon & \times \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} \times & 0 & \epsilon' & \epsilon' \\ \times & \times & \times & \epsilon' \\ \times & \epsilon & \times & 0 \\ \times & \times & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \epsilon' & \epsilon' & \epsilon' \\ \times & \times & 0 & \epsilon' \\ \times & \times & \times & \epsilon' \\ \times & \times & \times & \times \end{bmatrix} \rightarrow \cdots$$

where $\times$ is generic for an arbitrary matrix element. The small entries ϵ are turned into 0 using (7.6), but fill-ins ϵ' result from subsequent rotations. After one sweep, the matrix has turned to (almost) lower triangular form due to the permutations. In the next (even) sweep, the matrix is made upper triangular again, now using

$$\begin{bmatrix} c_{11} & \epsilon \\ c_{21} & c_{22} \end{bmatrix} \xrightarrow{(1/2)\pi + \theta_S} \begin{bmatrix} c'_{22} & c'_{21} \\ 0 & c'_{11} \end{bmatrix} \tag{7.7}$$

as a basic step. In an implementation, we can use the QZ iteration algorithm of the previous subsection to rotate over θ_S. During odd sweeps, we subsequently apply the permutation, yielding (7.6). During even sweeps, we first apply the permutation, and then perform the QZ iteration, which yields (7.7).

We have run several ESPRIT test scenarios to demonstrate the convergence properties of the GSD algorithms. ESPRIT data typically has eigenvalues spaced closely together on the unit circle in the complex plane (corresponding to the directions of arrival), but we have also run tests in which some of the eigenvalues have random positions, as caused by an overestimation of the number of signals. Comparing Stewart's algorithm with the modified algorithm described above (Tables I–III), we observe that Stewart's method hardly converges, whereas the modified algorithm ultimately shows a quadratic convergence behavior. This increase of speed is not because the use of θ_S produces smaller fill-ins ϵ': it can be shown using (7.5) that, in principle, the magnitude of a fill-in can be the same as in the θ_L case. Convergence is faster now due solely to the more effective

permutation habit of this algorithm. We have run the same tests on algorithms in which the 2 × 2 GSD is approximated using a few QZ iterations. The results show that when two iterations are used, on the average only a few extra sweeps are needed.

The algorithm described by Eberlein [20] uses a different ordering (the Brent–Luk "parallel" ordering [17]), and a rather obscure combination of θ_S, θ_L and permutations, governed by a number of (heuristic) decision rules. We have checked the performance of a "clean" version of this algorithm, using θ_S only, and permutations according to the parallel ordering scheme. By doing so, the actual difference between this algorithm and our modified Stewart algorithm narrows down to the use of a different ordering scheme. The tests show that the resulting convergence speeds are similar. This suggest that both orderings are equivalent for Jacobi–Schur, in the same way as has already been derived by Luk for Jacobi-SVD [22].

All above results are confirmed by a test case that appeared both in the Stewart and in the Eberlein papers (Tables IV–VI). The convergence speed is measured for matrices of the form $A = U(D + \alpha F)U^H$, where U is unitary, D is diagonal, F is strictly upper triangular, and the parameter α controls the departure of A from normality. Only when $\alpha = 10$, two QZ iterations are not sufficient to ensure convergence, and three iterations are needed. It is remarked here that, for all Schur algorithms until now, the speed of convergence still strongly depends on the normality of the matrix. Although the use of θ_S in combination with permutations empirically results in faster convergence, no rigid proof has been presented stating why this is so. In literature until now, only for normal matrices a quantitative analysis of convergence is given [29].

VIII. SVD/GSD Parallel Array Architecture

The purpose of this section is to describe in general terms a parallel array of processors on which Jacobi iteration algorithms can be performed. According to (3.15), we need a three-layer array for the SVD computations in the ESPRIT algorithm: two layers for operating with plane rotations upon $\underline{R}_1$ and R_2 to make these matrices diagonal, and one layer to apply the corresponding rotations to I to construct the result matrix E. (It will eventually be possible to combine $\underline{R}_1$ and R_2 into one layer, since both matrices remain upper triangular.) A two-layer array is needed for the GSD computations in (3.17). Since the multilayer array is a rather straightforward extension from the one-layer case, we will focus in this section only on the latter case. Systolic arrays for the Jacobi algorithm have already been discussed by many authors (see, e.g., [17]–[19]). The major new consideration here is that we require the array to make efficient use of a number of pipelined CORDIC processors, which leads towards a different architecture.

A. Stage Definition

The various operations of the Jacobi iteration algorithm (computation of rotation parameters and application of plane rotations) can be partitioned into a sequence of stages, defining a stage as a set of independent parallel tasks. First we observe that we can operate on all independent pairs $(i, i + 1)$ of rows or columns in parallel, but not on rows and columns at the same time, because these have matrix entries in common. Hence Q and Z rotations go into two different stages. Also, from Figs. 9 and 10, the computation of θ_1 for each 2 × 2 main diagonal submatrix $(i, i + 1)$ is partitioned into two stages: a preprocessing stage to compute $[x \;\; y]$ in Fig. 10 for GSD, or $[x + x' \;\; y]$ in Fig. 9 for SVD, followed by the actual computation of $(\theta_1, \phi_1, \phi_2)$ in the next stage. The basic time steps, then, are as follows.

1) Compute parameters $[x \;\; y]$ or $[x + x' \;\; y]$ for all 2 × 2 main diagonal submatrices $(i, i + 1)$ in parallel.

2) Compute plane rotations $Q_{i,i+1}(\theta_1)$ by vectorizing these parameters. Immediately apply these rotations to the corresponding rows, using hardware as in Fig. 7.

3) Compute plane rotations $Z_{i,i+1}(\theta_2)$. In general, these follow from simple 2 × 2 QR factorizations (one vectorization). Immediately apply these rotations to the corresponding columns.

4) If necessary, apply a raster shift to switch between operating on pairs $(i, i + 1)$ for odd i and even i. SVD always needs a raster shift. GSD, however, skips the first of every two shifts to implement an unfolded QZ iteration loop of two cycles.

B. Pipelined Processor Array

At this point, we have divided the algorithm into a sequence of stages. We can now map each stage onto a number of parallel operating CORDIC's, so that each processor in a stage operates on pipelined data. We want the order in which the matrix entries are processed and are shifted out of a stage to be suitable for the next stage, so that this stage can commence processing as soon as the first entries leave the current stage. This will create a FIFO pipeline of stages, in which the number of stages actually implemented may be chosen independently of the dimension of the matrix. A simple, one-directional data movement thus results.

The main problem here is to find a processing ordering of data that enables a smooth transition from row to column processing and also needs only local data communication. A suitable ordering is obtained by subdividing the matrix into diagonals which are processed one after another. The main diagonal is to be processed first, because new rotations are computed from these entries, but the other diagonals can be ordered in a number of ways, one of which is shown in Fig. 11(a). The numbers indicate the ordering of diagonals, hence correspond to the time step at which a diagonal is processed in a stage. Entries having equal number labels are processed in parallel. Other subdivision schemes are possible (see, e.g., [17]), but the scheme in Fig. 11 ensures that all diagonal vectors are of equal length; consequently, the number of processors used in a stage is constant at each time step. The resulting global timing regime is both regular and simple. The FIFO's in which the rows of the matrix are stored send one entry into the array at every time step. All processors in each stage consist of pipelined CORDIC's, and a number of stages may be active (in pipeline) at the same time, each one processing diagonal vectors.

As may be seen from the scheme and the matrix grid in Fig. 11(a), a row or column processor always operates on a pair of entries that are separated at most two time steps. Thus, attached to each processor is a buffer representing a delay of two time steps, in which the processor collects the data it needs to operate upon. Because all diagonals experience the same delay, the global timing is kept regular.

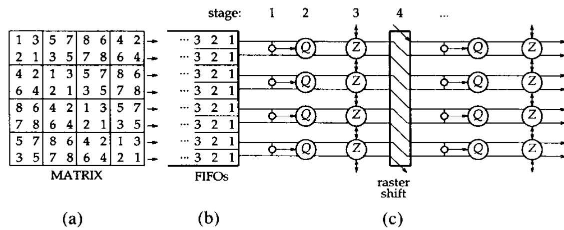

Fig. 11. Pipelined stages of parallel CORDIC processors. (a) Matrix. (b) Resulting ordering. (c) Parallel array.

The processors in stage 1 catch the relevant entries of the 2 × 2 main diagonal submatrices as they come along, and compute the rotation parameters $[x \quad y]$ or $[x + x' \quad y]$. The parameters are sent ahead of the data stream to the processors in stage 2 (labeled Q). These processors first compute the corresponding complex rotation angle by vectorizing the rotation parameters, and immediately proceed to rotate the incoming pairs of rows over this compound angle. A processor in stage 3 (column operations) first computes the $Z(\theta_2)$ rotation by vectorizing a pair of two relevant diagonal entries. The data entering this processor is row oriented, hence the processing of columns is distributed over all processors in a stage. This implies that a processor can use an angle only twice, after which it must be passed to its neighboring processors, resulting in a propagation of angles at constant speed between adjacent processors. Angles also wrap around between the top and bottom processors in a stage, hence the processors are connected in a ring. The (conditional) raster shift is performed by a down-shift of rows, also with a wraparound to the top. The raster shift can be that simple because the rows are ordered with diagonal elements first, and this condition is preserved by the shift.

At the edges of the matrix some irregularities occur. After applying the raster shift, some edge entries only need row operations, others need only column operations, and no angle computations are needed for these entries (see Fig. 4). To prohibit the rotations that are not allowed, we have to tag the diagonal entries of the two rows at the boundaries of the matrix. This enables the processors computing angles to detect that no rotations are needed, thus to generate a "zero"-valued dummy rotation angle. The regularity and generality of the stages 2 and 3 are kept intact this way.

The number of stages actually implemented in hardware may be chosen independently of the dimension of the matrix. It is not too difficult to further reduce the number of processors in each stage to arrive at a fixed size array, but we omit the details here for brevity.

IX. Concluding Remarks

In this paper, we have described in three parts a parallel architecture for a matrix pencil application based on the ESPRIT narrow-band DOA model. Starting with the ESPRIT model, a matrix pencil algorithm has been derived that consists of two stages of related unitary matrix decompositions. The keynote of the second part is that both decompositions can be computed in the same way using two-sided rotations. It has been shown that the computation of rotation angles can be unified to a reasonable extent, which has led to an architecture on which both decompositions can be computed using only one type of processor. The purpose of this paper has been the presentation of a framework relating these three fields. Each of them deserves to be explored further. In particular, future investigations should show how the matrix pencil algorithm can be used for more general ESPRIT-based models incorporating, e.g., wide-band, multidimensional, correlated signals. Also, the GSD Jacobi method is in need of a more formal proof stating conditions for convergence of nonnormal matrices.

References

[1] A. J. van der Veen and E. F. Deprettere, "A parallel VLSI direction finding algorithm," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 975, pp. 289–299, 1988.
[2] G. Cohen, D. Dubois, J. P. Quadrat, and M. Vlot, "A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 3, pp. 210–220, 1985.
[3] K. Jainandunsing and E. F. Deprettere, "A new class of highly structured algorithms for solving systems of linear equations," *SIAM J. Sci. Stat. Comput.*, Sept. 1989.
[4] R. Roy, "ESPRIT," Ph.D. dissertation, Stanford University, Stanford, CA, 1987.
[5] G. W. Stewart, "A Jacobi-like algorithm for computing the Schur decomposition of a non-Hermitian matrix," *SIAM J. Sci. Stat. Comput.*, vol. 6, pp. 853–864, 1985.
[6] A. Paulraj, R. Roy, and T. Kailath, "A subspace rotation approach to signal parameter estimation," *Proc. IEEE*, vol. 74, no. 7, pp. 1044–1045, 1986.
[7] B. Ottersten and T. Kailath, "ESPRIT for wide-band signals," in *Proc. 21st Asilomar Conf. Signal Syst. Comput.*, 1988, pp. 98–102.
[8] S. Y. Kung, *VLSI Array Processors.* Englewood Cliffs, NJ: Prentice-Hall, 1987.
[9] V. U. Reddy, A. Paulray, and T. Kailath, "Lecture notes on array signal processing," Stanford University, Stanford, CA, 1987.
[10] G. H. Golub and C. F. van Loan, *Matrix Computations.* Baltimore, MD: John Hopkins University Press, 1984.
[11] J. M. Speiser, "Some observations concerning the ESPRIT direction finding method," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 826, pp. 178–185, 1987.
[12] C. F. van Loan, "A unitary method for the ESPRIT direction-of-arrival estimation algorithm," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 826, pp. 170–176, 1987.
[13] H. Ouibrahim, D. D. Weiner, and T. K. Sarkar, "A generalized approach to direction finding," presented at IEEE Mil. Comm. Conf., 1986.
[14] G. H. Golub and C. F. van Loan, "An analysis of the total least squares problem," *SIAM J. Numer. Anal.*, vol. 17, no. 6, pp. 883–892, 1980.
[15] S. van Huffel, "Analysis of the TLS problem and its use in parameter estimation: Computation, properties, and applications," Ph.D. dissertation, Kath. University Louvain, Louvain, Belgium, 1987.
[16] F. T. Luk, "Architectures for computing eigenvalues and SVD's," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 614, pp. 24–33, 1986.
[17] R. P. Brent, F. T. Luk, and C. F. van Loan, "Computation of the singular value decomposition using mesh-connected processors," *J. VLSI Comput. Syst.*, vol. 3, pp. 242–270, 1985.
[18] R. P. Brent and F. T. Luk, "The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM J. Sci. Stat. Comput.*, vol. 6, pp. 69–84, 1985.
[19] U. Schwiegelshohn and L. Thiele, "A systolic array for cyclic-by-rows Jacobi algorithms," *J. Parallel Distributed Comput.*, vol. 4, pp. 334–340, 1987.
[20] P. J. Eberlein, "On the Schur decomposition of a matrix for parallel computation," *IEEE Trans. Comput.*, vol. C-36, pp. 167–174, 1987.

[21] K. V. Fernando, "Global convergence of the cyclic Kogbetliantz method," Numerical Algorithms Group Ltd., Oxford, NAG Tech. Rep., 1986.

[22] F. T. Luk, "On the equivalence and convergence of parallel Jacobi SVD algorithms," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 826, pp. 152-159, 1987.

[23] J. E. Volder, "The CORDIC trigonometric computing technique," *IEEE Trans. Electron. Comput.*, vol. EC-9, pp. 227-231, 1960.

[24] J. R. Cavallaro and F. T. Luk, "CORDIC arithmetic for an SVD processor," in *IEEE Proc. 8th Symp. Comp. Arithmetic*, 1987, pp. 113-120.

[25] J. M. Delosme, "A processor for two-dimensional symmetric eigenvalue and singular value arrays," in *Proc. 21st Asilomar Conf. Signal Syst. Comput.*, 1988, pp. 217-221.

[26] A. A. J. de Lange, A. van der Hoeven, J. Bu, and E. F. Deprettere, "A floating-point pipelined CMOS CORDIC processor," in *Proc. ISCAS'88*, 1988.

[27] F. T. Luk, "A parallel method for computing the generalized SVD," *J. Parallel Distributed Comput.*, vol. 2, pp. 250-260, 1985.

[28] C. B. Moler and G. W. Stewart, "An algorithm for generalized matrix eigenvalue problems," *SIAM J. Numer. Anal.*, vol. 10, no. 2, pp. 241-256, 1973.

[29] P. Van Dooren and J. P. Charlier, "Jacobi-like algorithm for computing the generalized Schur form of a regular pencil," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 1152, Aug. 1989.

Alle-Jan van der Veen (S'87) was born in The Netherlands in 1966. He graduated from the Department of Electrical Engineering, Delft University of Technology, in 1988. He is currently with the Network Theory Section at the same university, where he is working towards the Ph.D. degree. His research interests include system identification, model reduction, time-varying network theory, and optimal control.

Ed F. Deprettere (M'83-SM'88) received the M.Sc. degree from the Ghent State University, Ghent, Belgium, in 1968, and the Ph.D. degree from the Delft University of Technology (DUT), Delft, The Netherlands, in 1981.

In 1970, he became a Research Assistant and Lecturer at the DUT, where he is now Associate Professor in the Department of Electrical Engineering, Network Theory Section. His current research interests are in VLSI and modern signal processing, in particular VLSI array processing and mapping of signal processing algorithms, network graphs, and matrix equations onto silicon.