Chapter 1
Numerical Algebra and Adaptive Array Processing

Computing Linear Algebra-based Algorithms in VLSI

RECENT developments in modern signal processing rely heavily on efficient manipulation of linear algebra-based algorithms. Not only are the complexity and efficiency of the algorithms of great concern, but the numerical stability and parallel computation are also key factors. The use of VLSI provides the design and realization of application-specific structures and architectures for computationally-intensive algorithms such as those used in linear algebra-based signal processing or in efficient solutions of sparse matrices. In fact, a growing number of important techniques are matrix-based and historically more closely related to linear algebra than to signal processing. Many of these techniques are becoming increasingly important in signal processing and need to be blended with traditional algorithms in a compatible and complementary way. A signal processing viewpoint brought to computational problems in linear algebra can potentially lead to new approaches to those problems.

The goal of this chapter is to present the impact of VLSI on the computation of numerical algebra algorithms as well as their application to modern signal processing, in particular in adaptive array processing where numerical algebra is a basic tool for design, analysis, and implementation.

Singular-value and eigenvalue problems have been the key computational kernel of many signal processing applications such as statistical spectrum estimation, direction finding and beamformation, noise cancellation, digital filter design, and image processing [3, 22, 27]. Although they are powerful tools for many applications, the major hurdle in solving these problems is the high computational complexity. Thus, parallel processing for these problems is essential. The paper in [9] is considered one of the earliest works on this subject. Since then, much interest has been generated for parallel computation of numerical algebra problems using VLSI processors. It turns out that most of the algebra-based operations can be efficiently implemented on systolic array when only local interconnections are allowed. In [9], parallel Jacobi-like algorithms are presented for computing a singular-value decomposition (SVD) of an $m \times n$ matrix and an eigendecomposition of an $n \times n$ symmetric matrix. A linear array is proposed for the SVD which can be computed in $O(mnS)$ time, where S is typically less than 10. A square array of $O(n^2)$ processors, which can compute the symmetric eigenvalue problem in $O(nS)$ time is also considered, where S is the number of sweeps. The reprinted paper by Hsiao and Delosme proposes a CORDIC-based parallel SVD using a square array of processors that can handle complex-valued data in various signal processing applications. Related material can be found in [39, 44–46].

The problem with these systolic arrays is that they can perform computation on matrices of only a given size. In many signal processing applications, the dimension of the data matrix can be very large. As a result, the design of an extremely large-size array processor becomes impractical and not cost-effective. One solution is to solve large-size problems on small-size array processors. The next reprinted paper by Schreiber presents two modified algorithms and a modified array which solves the size-matching problem.

The above array processors solve the SVD of a fixed data matrix. Very often, in adaptive signal processing in particular, the data matrix will grow in time as new rows of data are appended to it. In order to avoid the high complexity of recomputing the SVD at each time instant, it is more effective to update the SVD from previously available decomposition. The following reprinted paper by Moonen, Van Dooren, and Vandewalle shows that the mapping of an SVD updating algorithm can be mapped onto a systolic array with high throughput. Interested readers can find more discussion on related issued in [2, 41, 75].

Many signal processing data matrices are drawn from certain delay-and-shift operations so that special structural properties are displaced explicitly or implicitly. The Toeplitz structure is one of the most common. The reprinted paper by Kung and Hu proposes a pipelined lattice structure for linear system equations of Toeplitz form. A parallel algorithm of $O(n)$ computing time is proposed to perform on a linear array of $O(n)$ processors, where n is the dimension of the Toeplitz matrix. The Toeplitz structure is exploited to derive a fast computing algorithm so that such a complexity can be achieved. There are many works that take advantage of the structural properties to develop efficient algorithms and architectures. Interested readers can refer to examples [6, 8, 16, 32, 40, 64, 83] for more details.

The QR decomposition (QRD) is a powerful tool to solve the least-squares (LS) problems. We will examine the usefulness of the QRD in several adaptive signal processing reprinted papers

in this chapter. Indeed, the above considered SVD problem may require a preliminary QRD step. However, the existing QRD arrays in [4, 28, 42, 48, 68] are very different from the above-mentioned SVD structures. Therefore, interfacing becomes a serious problem. In Luk's reprinted paper, a QRD array processor is proposed so that both QRD and SVD can be computed on the same mesh-connected processors.

Indeed, there are many ways of computing QRD in array processors, and there is no best method. For example, in the above situation, a good QRD array must be compatible with the SVD array. However, this is not the case for the following adaptive beamforming applications. We will see that the requirements are different for the computational structures.

The objective of an adaptive beamformer is to select a set of weights to produce a desired beam pattern so that undesired noise and jamming signals can be nulled out without hindering the signals of interest. In most situations, the targets or environments keep changing so it is necessary to change the weights. A conventional technique that has been employed for three decades is the least-mean-squares (LMS) algorithm, which is a closed-loop gradient descent algorithm. The major drawback is its slow convergence rate and sensitivity to the conditioning of the data matrices. In many modern systems where rapid convergence and high cancellation performance are essential, open-loop techniques formulated as a LS minimization problem have been of great interest [48]. It then becomes a recursive LS (RLS) problem due to its adaptive nature. No surprise, QRD has been the most popular technique for its numerical stability and parallel nature. In the reprinted paper by Hargrave, Ward, and McWhirter, a QRD RLS algorithm implementing the Gentleman/Kung QRD systolic array [21] is presented. The paper shows that the RLS algorithm can be naturally implemented on the QRD systolic array using the Givens rotation. Therefore, the high throughput adaptive beamformer can now be realized on parallel processors to satisfy high performance requirements.

The kind of beamformer discussed above is called the sidelobe canceller. Another very popular class of beamformers, called minimum variance distortionless response (MVDR) beamforming, fixes the output gain toward some directions while minimizing the total output power. Obviously, the noise and jamming signals will be nullified in the minimization process. The MVDR problem can be formulated as a linear-constrained RLS problem, where the desired signal directions can be specified in the linear constraints so that the gain can be fixed. The solution looks very different from that of a sidelobe canceller. In the next reprinted paper by McWhirter and Shepherd, a clever way of implementating the MVDR on the QRD RLS systolic array is presented. The key is to avoid back-substitution so that the data flow is consistent enough to fully guarantee pipeline processing by matching the MVDR processing to the QRD RLS formulation. Some related issues including parallel weight extraction can be found in [63, 67, 73, 74, 77]. Application of Givens transformation based on CORDIC rotations implemented on a restructurable VLSI systolic array architecture for adaptive nulling of interfering signals in a multielement antenna phase-array radar capable of performing three billion operations are described in [7, 58]. In recent years, many adaptive and some nonadaptive beamforming array algorithms and architectures have been proposed for microwave radar and wireless basestation communication, as well as acoustic sonar, hearing aid, teleconference, and speech recognition interference rejection, direction-of-arrival (DOA), and SNR enhancement applications [1, 11–14, 17, 18, 24, 25, 33, 34, 43, 50, 54–57, 60–62, 65, 69, 71, 76, 78, 80–82, 84].

Householder transformation is well known for its effectiveness in performing many numerical algebra algorithms, including the QRD. It is superior to the Givens rotation both in terms of complexity and stability. There have been some attempts to map the Householder transformation onto array processors. However, due to its vectorized computational nature, the complexity of matrix-vector processing cells are not realizable. In the reprinted paper by Liu, Hsieh, and Yao, a modified algorithm is used to perform a two-level pipeline at both vector and word levels. It is shown that the Givens rotation-based array is a special case of the more general Householder array. The block processing property can be used to handle higher sampling rates at the cost of linear complexity increase. Readers can find more discussion on Householder transformation in [59, 72].

The Givens rotation is a versatile computational tool in many matrix computations. When it comes to VLSI implementation, the required square root and division arithemetics will occupy much silicon area and take many cycles to complete. In [31] all the existing square root-free algorithms have been unified into a single family [23, 26]. In the paper by Frantzeskakis and Liu, a family of square root and division-free algorithms is presented. Detailed implementations for RLS and constrained RLS are considered. It is interesting to see that square-root and division algorithms are not necessarily good for all cases. In some cases, a square root-free family can be better.

Given all the algorithms and architectures discussed so far, it is not difficult to realize that there are many common relations among them. The reprinted paper by McWhirter takes an overall view of this interesting observation and tries to unify these algorithms and architectures into the framework of what McWhirter calls "algorithmic engineering." Additional papers on numerical algebra and adaptive array processing are given in [5, 10, 15, 19, 20, 29, 30, 35–38, 47, 49, 51–53, 66, 70, 79, 81].

References

[1] S. P. Applebaum, "Adaptive arrays," Rept. SURC TR 66-001, Syracuse Univ. Research Center, 1996.

[2] A. H. Abdallah and Y. H. Hu, "Parallel VLSI computing array implementation for signal subspace updating algorithm," IEEE Trans. Signal Processing, Vol. 37, No. 5, pp. 742–748, May 1989.

[3] H. M. Ahmed, J.-M. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," IEEE Computer, Vol. 15, No. 1, pp. 65–82, Jan. 1982.

[4] J. L. Barlow and I. C. F. Ipsen, "Scaled Givens rotations for the solution of linear least-squares problems on systolic arrays," SIAM Jour. Sci. Statist. Computing, Vol. 8, No. 5, pp. 716–733, Sept. 1987.

[5] A. Bojanczyk, R. P. Brent, and H. T. Kung, "Numerically stable solution of dense systems of linear equations using mesh-connected processors," SIAM Jour. Sci. Statist. Computing, Vol. 5, No. 1, pp. 95–104, Mar. 1984.

[6] A. W. Bojanczyk, R. P. Brent, and F. R. De Hoog, "Linearly connected arrays for Toeplitz least-squares problems," Jour. of Parallel and Distributed Computing, Vol. 9, No. 3, pp. 261–270, July 1990.

[7] G. D. Bolstad and K. B. Neeld, "A CORDIC based DSP element for adaptive signal processing," *Proc. Digital Signal Processing Technology,* ed. P. Papamichalis and R. Kerwin, SPIE Press, pp. 291–313, 1995.

[8] R. P. Brent and F. T. Luk, "A systolic array for the linear time solution of Toeplitz systems of equations," Jour. VLSI Comput. Systems, Vol. 1, No. 1, pp. 1–23, 1983.

[9] R. P. Brent and F. T. Luk, "The solution of singular value and symmetric eigenvalue problems on multiprocessor array," SIAM Jour. Sci. Statist. Computing, Vol. 6, No. 1, pp. 69–84, Jan. 1985.

[10] P. Comon and Y. Robert, "A systolic array for computing BA^{-1}," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 35, No. 6, pp. 717–723, June 1987.

[11] I. Claesson, S. E. Nordholm, B. A. Bengtsson, and P. Erikson, "A multi-DSP implementation of a broadband adaptive beamformer for use in a hand-free mobile radio telephone," IEEE Trans. Vehicul. Tech., Vol. 40, No. 1, pp. 194–202, Feb. 1991.

[12] H. Cox, R. M. Zeskind, and M. M. Owen, "Robust adaptive beamforming," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 35, No. 10, pp. 1365–1376, Oct. 1987.

[13] M. H. Er and A. Cantoni, "Derivative constraints for broadband element space antenna processor," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 31, No. 6, pp. 1378–1393, Dec. 1983.

[14] O. L. Frost, "An algorithm for linearly constrained adaptive array processing," Proc. IEEE, Vol. 60, No. 4, pp. 926–935, Mar. 1972.

[15] J. M. Delosme, "Bit-level systolic algorithms for real-time symmetric and hermitian eigenvalue problems," Journal of VLSI Signal Processing, Vol. 4, No. 1, pp. 69–88, Feb. 1992.

[16] W. H. Fang and A. E. Yagle, "A systolic architecture for new split algorithms for arbitrary Toeplitz-plus-Hankel matrices," IEEE Trans. Signal Processing, Vol. 42, No. 2, pp. 485–489, Feb. 1994.

[17] J. L. Flanagan, J. D. Johnston, R. Zahn, and G. Elko, "Computer-steered microphone arrays for sound-transduction in large room," Jour. Acous. Soc. Amer., Vol. 78, No. 3, pp. 1508–1518, Nov. 1985.

[18] W. F. Gabriel, "Adaptive arrays—an introduction," Proc. IEEE, Vol. 64, No. 2, pp. 239–272, Feb. 1976.

[19] F. M. F. Gaston, G. W. Irwin, and J.G. McWhirter, "Systolic square root covariance Kalman filtering," Journal VLSI Signal Processing, Vol. 2, No. 1, pp. 37–49, 1990.

[20] W. M. Gentleman, "Least squares computations by Givens transformations without square roots," Jour. Inst. Math. Appls., Vol. 12, pp. 329–336, 1973.

[21] W. M. Gentleman and H. T. Kung, "Matrix triangularization by systolic arrays," Proc. SPIE, Vol. 298, pp. 19–26, 1981.

[22] G. H. Golub and C. F. Van Loan, *Matrix Computations,* 2nd ed., Johns Hopkins Univ. Press, 1989.

[23] J. Gotze and U. Schwiegelshohn, "A square root and division-free Givens rotation for solving least squares problems on systolic arrays," SIAM Jour. Sci. Statist. Computing, Vol. 12, No. 4, pp. 800–807, July 1991.

[24] J. E. Greenberg and P. M. Zurek, "Evaluation of an adaptive beamforming method for hearing aids," Jour. Acous. Soc. Amer., Vol. 91, No. 3, pp. 1662–1672, Mar. 1992.

[25] L. J. Griffiths and C. W. Jim, "An alternate approach to linearly constrained adaptive arrays," IEEE Trans. on Antenna Prop., Vol. 30, No. 1, pp. 27–34, Jan. 1982.

[26] S. Hammarling, "A note on modifications to the Givens plane rotation," Jour. Inst. Math. Applicat., Vol. 13, No. 2, pp. 215–218, Apr. 1974.

[27] S. Haykin, *Adaptive Filter Theory,* Prentice-Hall, 1986.

[28] D. E. Heller and I. C. F. Ipsen, "Systolic networks for orthogonal decompositions," SIAM Jour. Sci. Statist. Computing, Vol. 4, No. 2, pp. 261–269, June 1983.

[29] S. F. Hsieh, K. J. R. Liu, and K. Yao, "Systolic implementations of up/down-dating Cholesky factorization using vectorized Gram-Schmidt pseudo-orthogonalization," Journal of VLSI Signal Processing, Vol. 3, No. 3, pp. 151–161, 1991.

[30] S. F. Hsieh, K. J. R. Liu, and K. Yao, "Dual-state systolic architectures for up/downdating RLS adaptive filtering," IEEE Trans. Circuits and Syst. II, Vol. 39, No. 6, pp. 382–385, June 1992.

[31] S. F. Hsieh, K. J. R. Liu, and K. Yao, "A unified approach for QRD-based recursive least-squares estimation without square roots," IEEE Trans. Signal Processing, Vol. 41, No. 3, pp. 1405–1409, Mar. 1993.

[32] Y. H. Hu and S. Y. Kung, "Toeplitz eigensystem solver," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 33, No. 4, pp. 1264–1271, Oct. 1982.

[33] J. E. Hudson, *Adaptive Array Principle,* Peter Peregrinus, 1981.

[34] D. Korompis, K. Yao, R. E. Hudson, and F. Lorenzelli, "Microphone array processing for hearing aid applications," *VLSI Signal Processing VI,* ed. J. Rabaey, P. M. Chau, and J. Eldon, IEEE Press, pp. 13–23, 1994.

[35] P. F. C. Krekel and E. F. Deprettere, "A systolic algorithm and architecture for solving sets of linear equations with multiband coefficient matrix," Journal of VLSI Signal Processing, Vol. 1, No. 2, pp. 143–152, Oct. 1989.

[36] G. L. Lawson and R. J. Hanson, *Solving Least-Squares Problems,* Prentice-Hall, 1974.

[37] H. Leung and S. Haykin, "Stability of recursive QRD LS algorithms using finite-precision systolic array implementation," IEEE Trans. Acoust., Speech, and Signal Processing, Vol. 37, No. 5, pp. 760–763, May 1989.

[38] P. S. Lewis, "Systolic architectures for adaptive multichannel least squares lattice filters," Journal of VLSI Signal Processing, Vol. 2, No. 1, pp. 29–36, 1990.

[39] K. J. R. Liu and K. Yao, "Multiphase systolic algorithms for spectral decomposition," IEEE Trans. Signal Processing, Vol. 40, No. 1, pp. 190–201, Jan. 1992.

[40] K. J. R. Liu and S. F. Hsieh, "Fast orthogonalization algorithm and parallel architecture for AR spectral estimation based on forward-backward linear prediction," IEEE Trans. Signal Processing, Vol. 41, No. 3, pp. 1453–1458, Mar. 1993.

[41] F. Lorenzelli, P. C. Hansen, T. F. Chan, and K. Yao, "A systolic implementation of the Chan/Foster RRQR algorithm," IEEE Trans. Signal Processing, Vol. 42, No. 8, pp. 2205–2208, Aug. 1994.

[42] F. Lorenzelli and K. Yao, "A linear systolic array for recursive least squares," IEEE Trans. Signal Processing, Vol. 43, No. 12, pp. 3014–3021, Dec. 1995.

[43] F. Lorenzelli, A. Wang, D. Korompis, R. E. Hudson, and K. Yao, "Subband processing for broadband microphone arrays," Jour. of VLSI Signal Processing, Vol. 12, No. 3/4, pp. 221–234, Sept. 1996.

[44] F. T. Luk, "A parallel method for computing the generalized singular value decomposition," Jour. of Parallel and Distributed Computing, Vol. 2, pp. 250–260, 1985.

[45] F. T. Luk, "A triangular processor array for computing singular values," Linear Algebra Appl., Vol. 77, pp. 259–273, 1986.

[46] F. T. Luk and S. Qiao, "Computing the CS-decomposition on systolic arrays," SIAM Jour. Sci. Statist. Computing, Vol. 7, No. 4, pp. 1121–1125, Oct. 1986.

[47] F. T. Luk and S. Qiao, "Analysis of a recursive least squares signal-processing algorithm," SIAM Jour. Sci. Statis. Computing, Vol. 10, No. 3, pp. 407–418, May 1989.

[48] J. G. McWhirter, "Recursive least-squares minimization using a systolic array," Proc. SPIE, Vol. 431, pp. 415–431, 1983.

[49] G. M. Megson, "A Fast Faddeev Array," IEEE Trans. Computers, Vol. 41, No. 12, pp. 1594–1600, Dec. 1992.

[50] R. Monzingo and T. Miller, *Introduction to Adaptive Arrays,* Wiley, 1980.

[51] M. Moonen and J. Vandewalle, "Square root covariance algorithm for constrained recursive least squares estimation," Journal of VLSI Signal Processing, Vol. 3, No. 3, pp. 163–172, Sept. 1991.

[52] M. Moonen and J. Vandewalle, "A systolic array for recursive least squares computations," IEEE Trans. Signal Processing, Vol. 41, No. 2, pp. 906–912, Feb. 1993.

[53] M. Moonen, F. J. Vanpoucke, and E. F. Deprettere, "Parallel and adaptive high-resolution direction finding," IEEE Trans. Signal Processing, Vol. 42, No. 9, pp. 2439–2448, Sept. 1994.

[54] S. Oh and V. Wiswanathan, "Microphone array for hand-free voice communication in a car," *Modern Methods of Speech Processing,* ed. R. P. Ramachandran and R. Mammone, Kluwer, pp. 351–376, 1995.

[55] A. Paulraj and T. Kailath, "Eigenstructure methods for direction of arrival estimation in the presence of unknown noise fields," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 34, No. 1, pp. 13–20, Feb. 1986.

[56] P. M. Peterson, "Adaptive array processing for multiple microphone hearing aids," RLE Tech. Rept. No. 541, Mass. Inst. of Tech. 1989.

[57] S. U. K. Pillai, *Array Signal Processing,* Springer-Verlag, 1989.

[58] C. M. Rader, "VLSI systolic arrays for adaptive nulling," IEEE Signal Processing, Vol. 13, No. 4, pp. 29–49, July 1996.

[59] C. M. Rader and A. O. Steinhardt, "Hyperbolic Householder transformations," IEEE Trans. Acoust., Speech, and Signal Processing, Vol. 34, No. 6, pp. 1589–1602, Dec. 1986.

[60] I. S. Reed, J. D. Mallet, and L. E. Brennan, "Rapid convergence rate in adaptive arrays," IEEE Trans. Aerospace Elec. Sys., Vol. 10, No. 6, Nov. 1974.

[61] V. U. Reddy, A. Paulraj, and T. Kailath, "Performance analysis of the optimum beamformer in the presence of correlated sources and its behavior under spatial smoothing," IEEE Trans. Acous., Speech, Signal Processing, Vol. 35, No. 7, pp. 527–536, July 1987.

[62] R. Roy and T. Kailath, "ESPIRIT—estimation of signal parameters via rotational invariance techniques," IEEE Trans. Acous., Speech, Signal Processing, Vol. 37, No. 7, pp. 984–995, July 1989.

[63] H. Sakai, "Recursive least-squares algorithms of modified Gram-Schmidt type for parallel weight extraction," IEEE Trans. Signal Processing, Vol. 42, No. 2, pp. 429–434, Feb. 1994.

[64] A. H. Sayed, H. Lev-Ari, and T. Kailath, "Time-varying displacement structure and triangular arrays," IEEE Trans. on Signal Processing, Vol. 42, No. 5., pp. 1052–1062, May 1994.

[65] R. O. Schmidt, "Multiple emitter location and signal parameter estimation," IEEE Trans. Antenna Propag., Vol. 34, No. 3, pp. 276–280, Mar. 1986.

[66] R. Schreiber, "Bidiagonalization and symmetric tridiagonalization by systolic arrays," Journal of VLSI Signal Processing, Vol. 1, No. 4, pp. 279–285, Apr. 1990.

[67] R. Schreiber, "Implementation of adaptive array algorithms," IEEE Trans. Acoust., Speech, and Signal Processing, Vol. 34, No. 10, pp. 1038–1045, Oct. 1986.

[68] R. Schreiber and W. Tang, "On systolic arrays for updating the Cholesky factorization," BIT, Vol. 26, p. 451–466, 1986.

[69] T. J. Shan, M. Wax, and T. Kailath, "On spatial smoothing for direction-of-arrival estimation of coherent signals," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 33, No. 3, pp. 806–811, June 1985.

[70] N. R. Shanbhag and K. K. Parhi, "Relaxed look-ahead pipelined LMS adaptive filters and their application to ADPCM coder," IEEE Trans. on Circuits and Systems II, Vol. 40, No. 12, pp. 753–766, Dec. 1993.

[71] R. W. Stadler and W. M. Rabinowitz, "On the potential of fixed arrays for hearing aids," Jour. Acous. Soc. Amer., Vol. 94, No. 3, pp. 1332–1342, Sept. 1993.

[72] A. Steinhardt, "Householder transformation," IEEE ASSP Magazine, Vol. 15, No. 5, pp. 4–12, July 1988.

[73] P. Strobach, "A generalization of McWhirter's formulas," IEEE Trans. on Signal Processing, Vol. 41, No. 8, pp. 2607–2617, Aug. 1993.

[74] C. F. T. Tang, K. J. R. Liu, and S. Tretter, "Optimal weight extraction for adaptive beamforming using systolic arrays," IEEE Trans. Aerosp., Electron. Syst., Vol. 30, No. 2, pp. 367–385, Apr. 1994.

[75] A. J. van der Veen and E. F. Deprettere, "Parallel VLSI Matrix Pencil Algorithm for High Resolution Direction Finding," IEEE Trans. on Signal Processing, Vol. 39, No. 2, pp. 383–394, Feb. 1991.

[76] B. Van Veen and K. M. Buckley, "Beamforming: a versatile approach to spatial filtering," IEEE Signal Processing, Vol. 5, No. 2, pp. 4–24, Apr. 1988.

[77] B. Van Veen, "Systolic preprocessors for linearly constrained beamforming," IEEE Trans. on Signal Processing, Vol. 37, No. 4, pp. 600–604, Apr. 1989.

[78] B. Van Veen, "Minimum variance beamforming," *Adaptive Radar Detection and Estimation,* ed. S. Haykin and A. Steinhardt, Wiley, pp. 161–236, 1992.

[79] A. P. Varvitsiotis and S. Theodoridis, "A pipelined structure for QR adaptive LS system identification," IEEE Trans. Signal Processing, Vol. 39, No. 8, pp. 1920–1923, Aug. 1991.

[80] M. Viberg and B. Ottersten, "Sensor array processing based on subspace fitting," IEEE Trans. Signal Processing, Vol. 39, No. 5, pp. 1110–1121, May 1992.

[81] H. Wang and M. Kaveh, "Coherent single-subspace averaging for the detection and estimation of angles of arriving multiple wide-band sources," IEEE Trans. Acous., Speech, and Signal Processing, Vol. 33, No. 4, pp. 823–831, Aug. 1985.

[82] A. Wang, K. Yao, R. E. Hudson, D. Korompis, F. Lorenzelli, S. D. Soli, and S. Gao, "A high performance microphone array system for hearing aid applications," Proc. ICASSP, pp. 3197–3220, May 1996.

[83] C. J. Zarowski, "A Schur algorithm and linearly connected processor array for Toeplitz-plus-Hankel matrices," IEEE Trans. Signal Processing, Vol. 40, No. 8, pp. 2065–2078, Aug. 1992.

[84] M. D. Zoltowski, "Beamspace ML bearing estimation for adaptive phased array radar," in *Adaptive Radar Detection and Estimation,* ed. S. Haykin and A. Steinhardt, Wiley, pp. 237–332, 1992.

Parallel Singular Value Decomposition of Complex Matrices Using Multidimensional CORDIC Algorithms

Shen-Fu Hsiao, *Member, IEEE*, and Jean-Marc Delosme

***Abstract*— The singular value decomposition (SVD) of complex matrices is computed in a highly parallel fashion on a square array of processors using Kogbetliantz's analog of Jacobi's eigenvalue decomposition method. To gain further speed, new algorithms for the basic SVD operations are proposed and their implementation as specialized processors is presented. The algorithms are 3-D and 4-D extensions of the CORDIC algorithm for plane rotations. When these extensions are used in concert with an additive decomposition of 2×2 complex matrices, which enhances parallelism, and with low resolution rotations early on in the SVD process, which reduce operation count, a fivefold speedup can be achieved over the fastest alternative approach.**

I. Introduction

CONCURRENTLY with the rapid increase in computing power over the last two decades, signal processing algorithms based on the computation-intensive singular value decomposition (SVD) have become increasingly popular. An international workshop on *SVD and Signal Processing* has convened regularly since 1987. Since SVD algorithms consist essentially of the repeated computation of orthogonal transformations, when very high throughputs are required the design of the arithmetic units to be used as building blocks for the parallel computation of the orthogonal transformations becomes critical as it may provide an order of magnitude speed-up.

The coordinate rotation digital computer (CORDIC) [34], [35] provides a good model for such arithmetic units, as it enables the efficient implementation of plane rotations using simple hardware components, mainly adders and shifters. Moreover, if needed, its computations may be accelerated to some extent with redundant arithmetic techniques [15], [29], [32]. Thus, much research has been directed toward integrating the CORDIC arithmetic algorithms and parallel matrix algorithms to obtain specialized parallel architectures for basic problems such as QR decomposition [2], [22], [23], eigenvalue and singular value decomposition [4], [5], [6], [8], [20], [37], or for applications such as filtering [14], [24], and array signal processing [33].

However, when dealing with complex data, the maximal throughputs of parallel arrays built out of CORDIC units which process two real numbers at a time are well below the throughputs achievable for real data sets of equal size. In order to come close to the real data throughputs, further parallelism must be revealed. This may be accomplished by using rotation algorithms extending the CORDIC concept to vectors in spaces with more than two dimensions. In this paper, such multidimensional CORDIC algorithms are developed and employed to speed up the computation of the SVD of complex matrices.

The paper is organized as follows. In Section II, the original two-dimensional (2-D) CORDIC algorithm is generalized to multidimensional spaces. In Section III, the three- (3-D) and four-dimensional (4-D) CORDIC algorithms are used for the parallel computation of the SVD of complex matrices using the Jacobi–Kogbetliantz method [17]. An additive decomposition of complex 2×2 matrices is employed to parallelize the two-sided rotations which make up the core of the method. Speed and area performances are compared with an architecture based on 2-D CORDIC's. In Section IV, we show how to apply partial resolution CORDIC algorithms to further improve the computation speed of the parallel SVD. Results of experiments on various classes of matrices are included.

II. CORDIC Algorithms for Euclidean Rotations

A. Basic Theory

A CORDIC algorithm implements an n-dimensional (n-D) rotation as a product of elementary rotations selected such that a vector is rotated using a small number of shifts and additions. Our starting point is a formula due to Cayley [16] allowing any n-D rotation R with no eigenvalue equal to -1 to be represented as

$$R = (I - T)(I + T)^{-1} \tag{1}$$

where I is the $n \times n$ identity matrix and T is $n \times n$ skew-symmetric (i.e., $T^T = -T$). Since

$$R = \frac{1}{|I + T|}(I - T)Adj(I + T) \tag{2}$$

Manuscript received December 28, 1994; revised September 15, 1995. This work was supported in part by the National Science Council of Taiwan under Contract NSC 84-2215-E-110-010 and by the Defense Advanced Research Project Agency (DARPA) under Contract N00014-88-K-0573. The associate editor coordinating the review of this paper and approving it for publication was Dr. K. J. Ray Liu.

S.-F. Hsiao is with the Institute of Computer Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C.

J.-M. Delosme is with the Department of Electrical Engineering, Yale University, New Haven, CT 06520 USA.

Publisher Item Identifier S 1053-587X(96)02401-2.

Reprinted from *IEEE Transactions on Signal Processing*, pp. 685-697, March 1996.

where $|I+T|$ and $Adj(I+T)$ denote the determinant and the adjoint of $I+T$, the entries of R are represented as rational functions of the entries of T.

A *rational* CORDIC algorithm is defined by a family of elementary rotations of the general form

$$R(S_k, t_k) = (I - S_k t_k)(I + S_k t_k)^{-1} \tag{3}$$

where S_k is a skew-symmetric matrix of signs, whose entries are either ± 1 or 0, and the *rotation parameter* t_k is a negative integral power of two. Thus the multiplication of a vector by an *unscaled rotation*

$$(I - S_k t_k) Adj(I + S_k t_k)$$

calls for a few shifts and additions which can be implemented combinationally, in parallel, as one CORDIC step or *rotation iteration* [9].

What about the divisions by $|I + S_k t_k|$? Assume that it is possible for $|I + S_k t_k|$ to be independent of S_k. Then, for any given set of parameters t_k, the product of the scaling factors

$$\prod_k \frac{1}{|I + S_k t_k|}$$

is a constant. By decomposing this constant normalization factor into the product of several factors such that the multiplication of an n-D vector by each of these factors is implemented on the same hardware as an unscaled rotation in a single, *scaling iteration*, the whole set of divisions is replaced by a few scaling iterations.

Upon expanding the determinant $|I + S_k t_k|$ we see that there are in general many ways to make it independent of S_k. For instance, if $n = 3$, $|I + S_k t_k| = 1 + n_k t_k^2$, where n_k is the number of nonzero entries above the diagonal of S_k, hence we only need to fix n_k for each k. The families of elementary rotations considered in this paper are such that all matrices S_k within a family have a given pattern of zero entries, independent of k. Outside of this zero pattern, the entries can take both sign values, $+1$ and -1. Consider now the case $n = 4$ and assume that the entries of S_k outside the diagonal are all nonzero. The determinant $|I + S_k t_k|$ is not independent of the sign values but, according to its expansion, a sufficient condition for independence is that the products of entries 12 and 34, 13 and 24, and 14 and 23 be constant. The matrices S_k have different sign structures according to the value, $+1$ or -1, selected for each one of these products. Here again, for every family of elementary rotations considered in this paper, once the sign structure of one matrix S_k is selected that ensures the independence of $|I + S_k t_k|$ on S_k, the same structure is imposed on all the other matrices S_k. A set of independent entries of S_k is summed up in a *sign vector* s_k.

For $n = 2$, $t_k = 2^{-k}$ and

$$S_k = \begin{pmatrix} 0 & -s_k \\ s_k & 0 \end{pmatrix}$$

we obtain the *rational CORDIC* decomposition of a plane rotation

$$R_2 = \prod_{k=0}^{b} R_2(s_k) \tag{4}$$

where $R_2(s_k)$ is an abbreviation for the elementary rotation $R_2(S_k, t_k)$, exhibiting only the free parameter $s_k = \pm 1$, and is equal to

$$\begin{aligned} &\frac{1}{|I + S_k t_k|}(I - S_k t_k) Adj(I + S_k t_k) \\ &\quad = \frac{1}{1 + t_k^2}(I - S_k t_k)^2 \\ &\quad = \frac{1}{1 + t_k^2}\begin{pmatrix} 1 - t_k^2 & 2 s_k t_k \\ -2 s_k t_k & 1 - t_k^2 \end{pmatrix}. \end{aligned} \tag{5}$$

This decomposition has been used to speed up the Jacobi algorithm for the SVD of real matrices [8], [12], to obtain a redundant arithmetic rotation algorithm with constant scaling [32], and to calculate the trigonometric functions $\sin^{-1}, \cos^{-1}$ [28].

Since $R_2(s_k)$ is the square of

$$R_2^{1/2}(s_k) = \frac{1}{\sqrt{1 + t_k^2}}(I - S_k t_k) \tag{6}$$

where the numerator entries are simple polynomials in t_k, we also have the *square-root CORDIC* decomposition of a plane rotation

$$\prod_{k=0}^{b} R_2^{1/2}(s_k) = \prod_{k=0}^{b} \frac{1}{\sqrt{1 + t_k^2}}\begin{pmatrix} 1 & s_k t_k \\ -s_k t_k & 1 \end{pmatrix} \tag{7}$$

where the signs s_k are properly selected and the rotation parameter t_k is equal to the tangent of the kth elementary rotation angle θ_k, i.e., $t_k = \tan \theta_k$. This decomposition is the basis of the original 2-D CORDIC algorithm [34], [35]. Its angular resolution is $\theta_b \simeq 2^{-b}$.

Let us use the 2-D square-root CORDIC algorithm to illustrate some basic notions. A rotation iteration is implemented by two parallel shift-and-add operations. For a given value b, the normalization factor

$$\prod_{k=0}^{b} (1 + t_k^2)^{-1/2}$$

is approximated off-line by the product of terms of the form 1 ± 2^{-l}. As a result, each scaling iteration also amounts to two parallel shift-and-add operations [1], [7].

The CORDIC algorithms have two modes, *evaluation* and *application*. In the evaluation mode, a sequence of $b + 1$ rotations is applied to a vector $[x_1\ x_2]^T$ to bring it along the first canonical axis $[\pm 1\ 0]^T$. The control sign s_k depends on the signs of the components at the previous iteration according to $s_k = sign(x_{1,k-1} \cdot x_{2,k-1})$ so that the accumulated angle

$$\sum_{k=0}^{b} s_k \theta_k$$

approximates the desired rotation angle $\theta = \tan^{-1}(x_2/x_1)$. There are two different types of CORDIC algorithms depending on whether the rotation angle θ is explicitly calculated or not. *Explicit* CORDIC evaluation calculates the angle

$$\theta = \sum s_k \theta_k$$

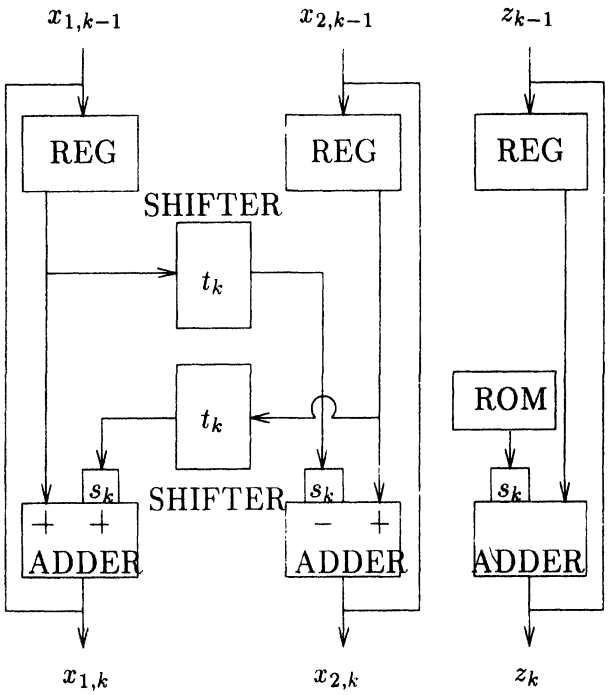


Fig. 1. Architecture of a 2-D CORDIC processor.

by accumulating all elementary angles $\theta_k = \tan^{-1}(t_k)$ to obtain the overall rotation angle θ. In contrast, *implicit* CORDIC evaluation generates only the control signs s_k since they make up a convenient encoding of the rotation angle θ. For many computational problems, including SVD computations, the explicit evaluation of the rotation angles can be bypassed [8], [12].

In the application mode, the explicit CORDIC algorithm decomposes the given angle θ into $\sum_k s_k\theta_k$ and the signs $s_k = \pm 1$ are used in the CORDIC rotation iterations. In the implicit CORDIC algorithm the control signs s_k themselves are given and thus can be used directly to rotate vectors. These signs are usually available as the result of a preliminary evaluation. Implicit CORDIC algorithms are advantageous for multiprocessor array implementations as they enable the evaluation and application processes to be overlapped [8], [12]. Furthermore, the implicit form avoids the sequentiality inherent in the definition of angles in higher dimensions.

Fig. 1 illustrates the architecture of a 2-D CORDIC processor. The Z accumulator part (including a register, a ROM storing the elementary angles θ_k, and an adder), computes or encodes the rotation angle. It is used only in the explicit CORDIC algorithm.

B. Householder CORDIC Algorithms

In high-dimensional CORDIC algorithms, the evaluation mode amounts to bringing an n-D vector $[x_1 \cdots x_n]^T$ along the first canonical axis $[\pm 1\ 0 \cdots\ 0]^T$ via a sequence of elementary n-D rotations and the application mode consists in applying the same sequence of elementary rotations to other vectors. The selection rule for the control signs in the evaluation mode should be simple. Observing that, to first order in t_k

$$R_n(S_k, t_k) \equiv (I - S_k t_k)(I + S_k t_k)^{-1} \simeq I - 2S_k t_k$$

and that we want to zero out the last $n-1$ components of the vector, we choose S_k of the form

$$\begin{pmatrix} 0 & -s_{1,k} & \cdots & -s_{n-1,k} \\ s_{1,k} & & & \\ \vdots & & \overline{S_k} & \\ s_{n-1,k} & & & \end{pmatrix} \tag{8}$$

and select the control signs $s_{i,k}$ according to $s_{i,k} = sign(x_{1,k-1} \cdot x_{i+1,k-1})$, $1 \le i \le n-1$, where $x_{i,k-1}$ denotes the ith component of the vector at the beginning of the kth iteration. According to the first-order approximation of $R_n(S_k, t_k)$, the magnitude of the first component always increases and, if the other components are sufficiently small with respect to the first component, their increments have signs opposite to their own. If $\overline{S_k} = 0$, this last observation holds without the restriction on the magnitude of the components and convergence is then easier to achieve.

With the choice $\overline{S_k} = 0$, $|I + S_k t_k| = 1 + (n-1)t_k^2$ is independent of s_k, as desired. The *Householder CORDIC* algorithm [22] results, in which the elementary rotation matrices $R_n(s_k) \equiv R_n(S_k, t_k)$ may be expressed as the product of two *Householder reflections*

$$R_n(s_k) = \left(I - 2\frac{e_1 e_1^T}{e_1^T e_1}\right)\left(I - 2\frac{u_k u_k^T}{u_k^T u_k}\right) \tag{9}$$

with

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{and} \quad u_k = \begin{bmatrix} 1 \\ s_k t_k \end{bmatrix} = \begin{bmatrix} 1 \\ s_{1,k} t_k \\ \vdots \\ s_{n-1,k} t_k \end{bmatrix}.$$

For $n = 4$, the elementary rotation matrices are in (10), which appears at the bottom of this page.

The Householder CORDIC algorithm consists of a sequence of elementary Householder transformations. While the use of sequences of Householder transformations in the context of VLSI signal processing is not new, see, e.g., [26], the Householder CORDIC transformations have the distinction of being "elementary" and hence easily implemented with shifts and adds.

$$R_4(s_k) = R_4\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{bmatrix}\right) = \frac{1}{1+3t_k^2}\begin{pmatrix} 1-3t_k^2 & 2s_{1,k}t_k & 2s_{2,k}t_k & 2s_{3,k}t_k \\ -2s_{1,k}t_k & 1+t_k^2 & -2s_{1,k}s_{2,k}t_k^2 & -2s_{1,k}s_{3,k}t_k^2 \\ -2s_{2,k}t_k & -2s_{2,k}s_{1,k}t_k^2 & 1+t_k^2 & -2s_{2,k}s_{3,k}t_k^2 \\ -2s_{3,k}t_k & -2s_{3,k}s_{1,k}t_k^2 & -2s_{3,k}s_{2,k}t_k^2 & 1+t_k^2 \end{pmatrix} \tag{10}$$

C. 3-D Rational CORDIC Algorithm

Other choices for $\overline{S_k}$ are possible that also lead to $|I+S_kt_k|$ independent of s_k. For instance, for $n=3$, the choice

$$\overline{S_k} = \begin{pmatrix} 0 & -s_{1,k} \\ s_{1,k} & 0 \end{pmatrix}$$

gives the elementary rotation matrices, which are shown in (11) at the bottom of this page. Factoring out the scaling term $1/(1+3t_k^2)$ yields an unscaled elementary rotation that can be implemented by simple arithmetic blocks: shifters, carry-save-adders (CSA) and adders. For lack of a better name (for the elementary rotations in the 3-D Householder CORDIC algorithm are also rational in t_k) we call the associated CORDIC algorithm the *3-D rational* CORDIC algorithm [9], [13].

D. Quaternion CORDIC Algorithms

For $n=4$ one may impose that all the matrices $\overline{S_k}$ be equal to either

$$\begin{pmatrix} 0 & s_{3,k} & -s_{2,k} \\ -s_{3,k} & 0 & s_{1,k} \\ s_{2,k} & -s_{1,k} & 0 \end{pmatrix} \text{ or } \begin{pmatrix} 0 & -s_{3,k} & s_{2,k} \\ s_{3,k} & 0 & -s_{1,k} \\ -s_{2,k} & s_{1,k} & 0 \end{pmatrix}.$$

In both cases

$$|I + S_kt_k| = (1+3t_k^2)^2$$

and

$$Adj(I + S_kt_k) = (1+3t_k^2)(I - S_kt_k)$$

so that, in a way similar to the 2-D case, the elementary rotations are the square of

$$\frac{1}{\sqrt{1+3t_k^2}}(I - S_kt_k).$$

These "square-root" matrices are unit quaternions, and thus, we obtain two *quaternion*, or *4-D square-root, CORDIC* algorithms [9], [13] with the two families of elementary rotation matrices, which are shown in (12) and (13) at the bottom of this page. The superscripts in $R_4^{1/2}(s_k)$ and $R_4^{-1/2}(s_k)$ merely indicate that the matrices are square roots of rational matrices. ($R_4^{1/2}(s_k)$ is not a square root of $R_4(s_k)$ and $R_4^{-1/2}(s_k)$ is not the inverse of $R_4^{1/2}(s_k)$.)

The selection of the rotation parameter sequence to ensure convergence of the 4-D Householder CORDIC algorithm is $t_k = 2^{-k}$, as in the 2-D case [22]. For the 3-D rational and 4-D square-root CORDIC algorithms, convergence is more difficult because $\overline{S_k} \neq 0$ in (8) while the selection rule for the control signs in the evaluation mode remains

$$s_{i,k} = \text{sign}\,(x_{1,k-1} \cdot x_{i+1,k-1}), \quad 1 \leq i \leq n-1.$$

Consequently, a few of the values 2^{-k} must be repeated, especially in the early iterations [9], [13].

E. Architecture of Processing Units

A possible architecture for a 4-D Householder CORDIC processing unit is shown in Fig. 2(a). It comprises four blocks, each implementing the multiplication of a 4-D vector by one of the rows of the matrix in (10). For instance, the block at the top left computes the product by the first row, $[1 - 2t_k^2 - t_k^2 \quad \pm 2t_k \quad \pm 2t_k \quad \pm 2t_k]$, by means of two shifters and a 6-to-2 carry-save-adder array followed by a fast two-input adder. Both 6-to-2 or 5-to-2 CSA arrays are built out of 3-to-2 CSA cells. The 3-D rational CORDIC algorithm described in Section II-C may be implemented using three blocks identical to the top left block in Fig. 2(a), but with different shifted data components as inputs. Thus if the 5-to-2 CSA arrays in Fig. 2(a) are replaced by 6-to-2 CSA arrays, and further control is added to the routing channels and multiplexers, a "unified" 4-D CORDIC unit illustrated in Fig. 2(b) is obtained that can implement not only 4-D Householder CORDIC operations but also the 3-D and two 2-D rational CORDIC operations. The area of the unified 4-D CORDIC unit, $\mathcal{A}_{4D,\,\text{unified}}$, is close to $\mathcal{A}_{4D}$, the area of the 4-D Householder CORDIC unit since both units require as many shifters and adders.

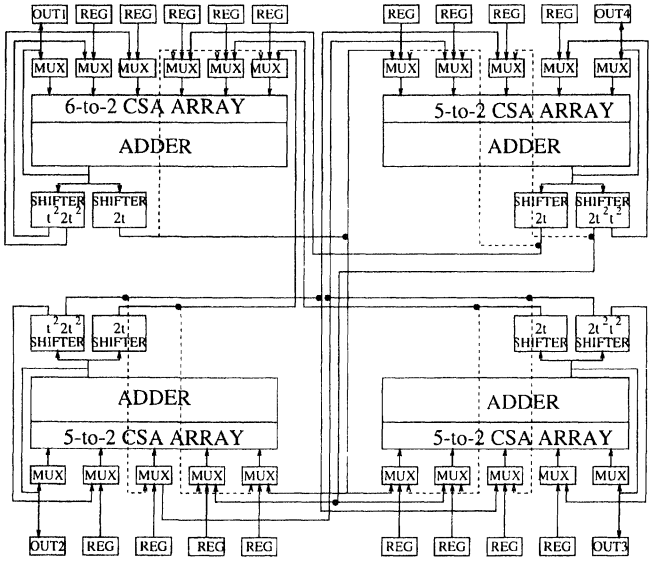
The architecture for a quaternion unit is simpler, with four shifters implementing multiplication by t_k, four CSA arrays with 4-to-2 CSA cells, and four adders [12].

In the next section, we will use the multidimensional CORDIC algorithms and the processing units just described for the parallel computation of the SVD of complex matrices. In the process we will see some surprising relations among the above algorithms.

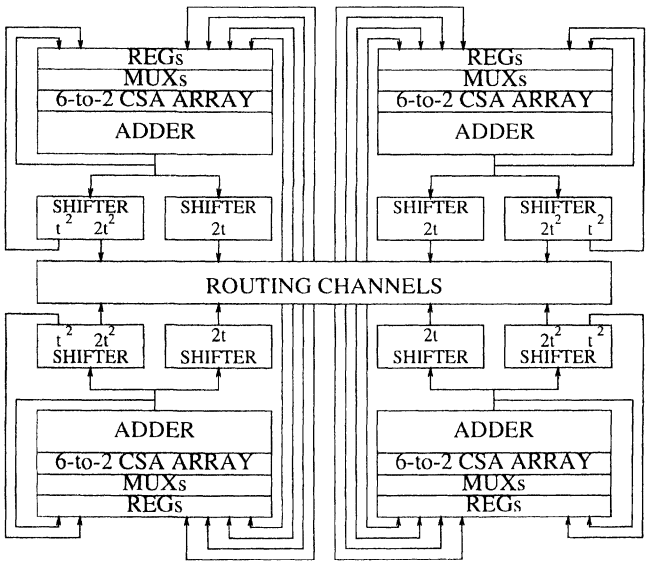
$$R_3'\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \end{bmatrix}\right) = \frac{1}{1+3t_k^2}\begin{pmatrix} 1-t_k^2 & 2s_{1,k}t_k - 2s_{1,k}s_{2,k}t_k^2 & 2s_{2,k}t_k + 2t_k^2 \\ -2s_{1,k}t_k - 2s_{1,k}s_{2,k}t_k^2 & 1-t_k^2 & 2s_{1,k}t_k - 2s_{1,k}s_{2,k}t_k^2 \\ -2s_{2,k}t_k + 2t_k^2 & -2s_{1,k}t_k - 2s_{1,k}s_{2,k}t_k^2 & 1-t_k^2 \end{pmatrix} \tag{11}$$

$$R_4^{1/2}(s_k) = R_4^{1/2}\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{bmatrix}\right) = \frac{1}{\sqrt{1+3t_k^2}}\begin{pmatrix} 1 & s_{1,k}t_k & s_{2,k}t_k & s_{3,k}t_k \\ -s_{1,k}t_k & 1 & -s_{3,k}t_k & s_{2,k}t_k \\ -s_{2,k}t_k & s_{3,k}t_k & 1 & -s_{1,k}t_k \\ -s_{3,k}t_k & -s_{2,k}t_k & s_{1,k}t_k & 1 \end{pmatrix} \tag{12}$$

$$R_4^{-1/2}(s_k) = R_4^{-1/2}\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{bmatrix}\right) = \frac{1}{\sqrt{1+3t_k^2}}\begin{pmatrix} 1 & s_{1,k}t_k & s_{2,k}t_k & s_{3,k}t_k \\ -s_{1,k}t_k & 1 & s_{3,k}t_k & -s_{2,k}t_k \\ -s_{2,k}t_k & -s_{3,k}t_k & 1 & s_{1,k}t_k \\ -s_{3,k}t_k & s_{2,k}t_k & -s_{1,k}t_k & 1 \end{pmatrix} \tag{13}$$



(a)



(b)

Fig. 2. (a) Four-dimensional Householder CORDIC processor architecture; (b) 4-D unified CORDIC processor architecture.

III. Parallel Singular Value Decomposition

The singular value decomposition of an $l \times m$ $(l \geq m)$ complex matrix A of rank $r > 0$ can be expressed as

$$A = USV^H$$

where the matrices U $(l \times l)$ and V $(m \times m)$ are unitary, V^H is the conjugate transpose of V, and the real matrix S $(l \times m)$ is diagonal with r positive singular values. If $l \neq m$, the SVD is computed in two phases: the QR decomposition of A is performed, i.e.

$$A = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$$

where Q is an $l \times l$ unitary matrix and R is an $m \times m$ upper triangular matrix, and is followed by the SVD of the square matrix R. Readers can refer to [6], [11], [22], [23] for the implementation of the first phase using CORDIC algorithms. From now on, we consider the SVD of an $m \times m$ square matrix A. We shall present parallel implementations of the complex SVD using a variant of the Jacobi–Kogbetliantz method in which the basic rotations are realized with 3-D and 4-D CORDIC algorithms.

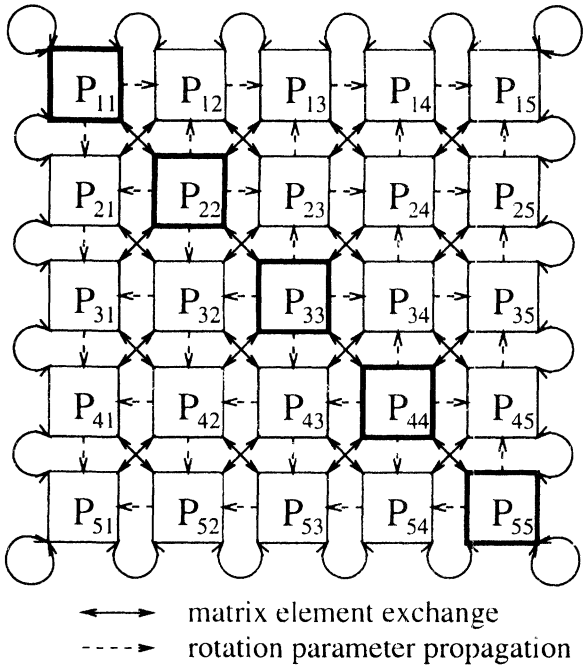


Fig. 3. Systolic array architecture for the Jacobi SVD algorithm.

A. VLSI Array Architecture for the SVD

The Jacobi method reduces to zero the off-diagonal entries of an $m \times m$ complex matrix A by applying complex plane rotations both from its left and from its right. The matrix A is eventually transformed into S, the diagonal matrix of singular values, while U and V may be obtained as aggregated products of the applied rotations [17]. The main operation in the method is the evaluation of the SVD of a 2×2 matrix, which can be expressed as a two-sided rotation

$$R^l \begin{pmatrix} a & b \\ c & d \end{pmatrix} R^r = \begin{pmatrix} a' & 0 \\ 0 & d' \end{pmatrix}$$

where R^l and R^r are the left- and right-side rotations selected to diagonalize the matrix.

A VLSI array architecture for the parallel SVD of A is shown in Fig. 3 (for $m = 10$) where each processor performs two-sided rotations on 2×2 submatrices of A [3]. When m is large, the above architecture, which is matched to the problem size, is viewed as *virtual* and is mapped onto a smaller physical array, see e.g. [11]. Assume m is even. Initially, processor P_{IJ} contains the 2×2 submatrix

$$A_{IJ} = \begin{pmatrix} a_{2I-1,2J-1} & a_{2I-1,2J} \\ a_{2I,2J-1} & a_{2I,2J} \end{pmatrix}.$$

At each Jacobi step, diagonal processor P_{II} performs on the 2×2 submatrix A_{II} a two-sided rotation $R_I^l A_{II} R_I^r$,

where R_I^l and R_I^r are selected to zero out the off-diagonal elements of A_{II}. Off-diagonal processor P_{IJ} applies the corresponding two-sided rotation $R_I^l A_{IJ} R_J^r$ with R_I^l from P_{II} and R_J^r from P_{JJ}. Then each processor exchanges matrix elements with neighbors according to the parallel ordering in [3]. In particular, the annihilated off-diagonal elements in the diagonal processors are swapped with off-diagonal elements from the neighboring off-diagonal processors.

Communications between processors are of two types. One is the exchange along diagonal links of the matrix elements at the end of each step. In $(m-1)$ steps, every off-diagonal element is brought once into a diagonal processor for evaluation. The other is the propagation along rows and columns of the parameters representing the rotations evaluated in the diagonal processors during each Jacobi step. (With implicit CORDIC algorithms only bits are sent at each step, using a limited broadcast scheme [8], [11], [12]. This enables the overlap of the evaluations in the diagonal processors and of the applications in the off-diagonal processors.)

B. Even-plus-Odd Decomposition of 2×2 *Complex Matrices*

Yang and Böhme [37] have used an additive decomposition of 2×2 real matrices in order to parallelize the evaluation *and application* of real two-sided rotations using explicit CORDIC. The approach has been extended to the real case with implicit CORDIC in [10] and to the complex case in [12]. Indeed, a 2×2 complex matrix M can be decomposed as

$$\begin{aligned} M &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ &= \frac{1}{2}\begin{pmatrix} a+\bar{d} & b-\bar{c} \\ c-\bar{b} & \bar{a}+d \end{pmatrix} + \frac{1}{2}\begin{pmatrix} a-\bar{d} & b+\bar{c} \\ \bar{b}+c & d-\bar{a} \end{pmatrix} \\ &= \begin{pmatrix} e_1 & e_2 \\ -\bar{e}_2 & \bar{e}_1 \end{pmatrix} + \begin{pmatrix} o_1 & o_2 \\ \bar{o}_2 & -\bar{o}_1 \end{pmatrix} = E + O \end{aligned} \tag{14}$$

where the overbar denotes the complex conjugate, and E and O are called complex *even* and *odd* matrices. The product of two even matrices or of two odd matrices is an even matrix while the product of an even matrix and an odd matrix is an odd matrix, thus justifying the names. A complex plane rotation can be represented by a complex even matrix

$$\begin{pmatrix} c_1 & s_1 \\ -\bar{s}_1 & \bar{c}_1 \end{pmatrix}$$

with the constraint $c_1\bar{c}_1 + s_1\bar{s}_1 = 1$.

C. Isomorphisms Between 2×2 *Complex Matrices and* 4×4 *Real Matrices*

To every 2×2 complex even matrix

$$\begin{aligned} E &= \begin{pmatrix} e_1 & e_2 \\ -\bar{e}_2 & \bar{e}_1 \end{pmatrix} \\ &= \begin{pmatrix} e_1^r + \jmath e_1^j & e_2^r + \jmath e_2^j \\ -e_2^r + \jmath e_2^j & e_1^r - \jmath e_1^j \end{pmatrix} \\ &= \begin{pmatrix} e_1^r & e_2^r \\ -e_2^r & e_1^r \end{pmatrix} + \jmath \begin{pmatrix} e_1^j & e_2^j \\ e_2^j & -e_1^j \end{pmatrix} \end{aligned}$$

correspond two distinct 4×4 real matrix representations

$$\begin{aligned} \boldsymbol{E}_1 &= \begin{pmatrix} e_1^r & e_1^j & e_2^r & e_2^j \\ -e_1^j & e_1^r & -e_2^j & e_2^r \\ -e_2^r & e_2^j & e_1^r & -e_1^j \\ -e_2^j & -e_2^r & e_1^j & e_1^r \end{pmatrix}, \\ \boldsymbol{E}_2 &= \begin{pmatrix} e_1^r & e_2^r & e_1^j & e_2^j \\ -e_2^r & e_1^r & e_2^j & -e_1^j \\ -e_1^j & -e_2^j & e_1^r & e_2^r \\ -e_2^j & e_1^j & -e_2^r & e_1^r \end{pmatrix}. \end{aligned} \tag{15}$$

The matrix E can be reconstructed from only the first column of $\boldsymbol{E}_1$ or $\boldsymbol{E}_2$. Similarly, to every 2×2 complex odd matrix O correspond two distinct 4×4 real matrix representations $\boldsymbol{O}_1$ and $\boldsymbol{O}_2$. The two 4×4 matrices in (15) lead to the two families of 4-D quaternion CORDIC elementary rotations in (12) and (13). $R_4^{1/2}(s_k)$, which corresponds to $\boldsymbol{E}_1$, is named the "standard" representation in [12]. Since a complex plane rotation can be represented by a complex even matrix with unit norm, it can be performed by a 4-D quaternion CORDIC algorithm.

Letting boldface $\boldsymbol{E}$ and $\boldsymbol{O}$ denote the standard 4×4 real matrix representations of the complex even matrix E and odd matrix O, we observe that

$$\begin{aligned} &\text{first column of } \boldsymbol{E} R_4^{1/2}\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{bmatrix}\right) \\ &\quad = \text{first column of } R_4^{-1/2}\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{bmatrix}\right)\boldsymbol{E} \\ &\text{first column of } \boldsymbol{O} R_4^{1/2}\left(\begin{bmatrix} s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{bmatrix}\right) \\ &\quad = \text{first column of } R_4^{-1/2}\left(\begin{bmatrix} s_{1,k} \\ -s_{2,k} \\ -s_{3,k} \end{bmatrix}\right)\boldsymbol{O}. \end{aligned} \tag{16}$$

The above *pseudo-commutativity* and *pseudo-anticommutativity* properties permit to move one of the plane rotations in a complex two-sided rotation from one side to the other, thus allowing the combination of the two plane rotations and hence their concurrent execution.

D. CORDIC Evaluations in the Diagonal Processors

Returning to the array architecture shown in Fig. 3 and considering the 2×2 submatrix in diagonal processor P_{II}, we adopt a two-phase approach to evaluate the left and right rotations R_I^l and R_I^r that make $R_I^l A_{II} R_I^r$ diagonal. The even-plus-odd decomposition of A_{II} is

$$\begin{aligned} A_{II} &= E_{II} + O_{II} \\ &= \begin{pmatrix} e_{1,II} & e_{2,II} \\ -\bar{e}_{2,II} & \bar{e}_{1,II} \end{pmatrix} + \begin{pmatrix} o_{1,II} & o_{2,II} \\ \bar{o}_{2,II} & -\bar{o}_{1,II} \end{pmatrix} \\ &= \begin{pmatrix} e_{1,II}^r + \jmath e_{1,II}^j & e_{2,II}^r + \jmath e_{2,II}^j \\ -e_{2,II}^r + \jmath e_{2,II}^j & e_{1,II}^r - \jmath e_{1,II}^j \end{pmatrix} \\ &\quad + \begin{pmatrix} o_{1,II}^r + \jmath o_{1,II}^j & o_{2,II}^r + \jmath o_{2,II}^j \\ o_{2,II}^r - \jmath o_{2,II}^j & -o_{1,II}^r + \jmath o_{1,II}^j \end{pmatrix}. \end{aligned}$$

From now on, we consider only the operations on the real 4×4 standard matrix representations of the involved 2×2 complex matrices, with the understanding that the final results can be readily transformed back into the complex matrix space.

First Evaluation Phase: Two sequences of 4-D quaternion CORDIC elementary rotation matrices with the same control signs are applied to the left and right sides of $\boldsymbol{A}_{II}$. The computation involved in the kth CORDIC elementary rotation is expressed as

$$\begin{aligned}\boldsymbol{A}_{II,k+1} &= \boldsymbol{E}_{II,k+1} + \boldsymbol{O}_{II,k+1} \\ &= R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right)\left(\boldsymbol{E}_{II,k} + \boldsymbol{O}_{II,k}\right) R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right)\end{aligned} \tag{17}$$

where the dependence of the control signs $\sigma_{1,k}, \sigma_{2,k}, \sigma_{3,k}$ on the processor index I is dropped to make the reading easier. Let $\mathbf{e}_{II,k}$ denote the first column of $\mathbf{E}_{II,k}$, and $\mathbf{o}_{II,k}$ denote the first column of $\mathbf{O}_{II,k}$. Using the pseudo-commutativity and pseudo-anticommutativity properties exhibited in (16), we get

$$\begin{aligned}\boldsymbol{e}_{II,k+1} &= R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) R_4^{-1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right)\ \boldsymbol{e}_{II,k} \\ \boldsymbol{o}_{II,k+1} &= R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) R_4^{-1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ -\sigma_{2,k}\\ -\sigma_{3,k}\end{bmatrix}\right)\ \boldsymbol{o}_{II,k}.\end{aligned}$$

Therefore, $\boldsymbol{e}_{II,k}$ is subjected to the kth CORDIC elementary rotation, which is shown in the first expression at the bottom of this page, and is the *4-D Householder CORDIC* elementary rotation matrix in (10). Similarly, $\boldsymbol{o}_{II,k}$ is subjected to the kth CORDIC elementary rotation, which is shown in the second expression at the bottom of this page, and is a symmetric permutation of an elementary rotation matrix in the 4-D Householder CORDIC algorithm.

The control signs $\sigma_{1,k}, \sigma_{2,k}, \sigma_{3,k}$ are selected to force to zero the last three components of $\boldsymbol{e}_{II}$. At the end of the first evaluation phase, we have $\boldsymbol{A}'_{II} = \boldsymbol{E}'_{II} + \boldsymbol{O}'_{II}$ where the first columns of $\boldsymbol{E}'_{II}$ and $\boldsymbol{O}'_{II}$ are

$$\begin{pmatrix} e^{r'}_{1,II}\\ 0\\ 0\\ 0\end{pmatrix} = \prod_k R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) R_4^{-1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right)\begin{pmatrix} e^{r}_{1,II}\\ -e^{j}_{1,II}\\ -e^{r}_{2,II}\\ -e^{j}_{2,II}\end{pmatrix} \tag{18}$$

$$\begin{pmatrix} o^{r'}_{1,II}\\ -o^{j'}_{1,II}\\ o^{r'}_{2,II}\\ o^{j'}_{2,II}\end{pmatrix} = \prod_k R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) R_4^{-1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ -\sigma_{2,k}\\ -\sigma_{3,k}\end{bmatrix}\right)\begin{pmatrix} o^{r}_{1,II}\\ -o^{j}_{1,II}\\ o^{r}_{2,II}\\ o^{j}_{2,II}\end{pmatrix}. \tag{19}$$

Reconstructing the 2×2 complex submatrix A'_{II} from $\boldsymbol{e}'_{II}$ and $\boldsymbol{o}'_{II}$, we obtain

$$\begin{aligned}A'_{II} &= E'_{II} + O'_{II} \\ &= \begin{pmatrix} e^{r'}_{1,II} & 0\\ 0 & e^{r'}_{1,II}\end{pmatrix} \\ &\quad + \begin{pmatrix} o^{r'}_{1,II} + \jmath o^{j'}_{1,II} & o^{r'}_{2,II} + \jmath o^{j'}_{2,II}\\ o^{r'}_{2,II} - \jmath o^{j}_{2,II} & -o^{r'}_{1,II} + \jmath o^{j'}_{1,II}\end{pmatrix}.\end{aligned}$$

The complex even matrix E'_{II} is real and diagonal at the end of that phase.

Second Evaluation Phase: In the second phase of the evaluation process, two sequences of 4-D quaternion elementary rotations with *opposite* control signs are applied to the left and right sides of $\boldsymbol{A}'_{II}$. Consider the kth CORDIC elementary rotation

$$\begin{aligned}\boldsymbol{A}'_{II,k+1} &= \boldsymbol{E}'_{II,k+1} + \boldsymbol{O}'_{II,k+1} \\ &= R_4^{1/2}\left(\begin{bmatrix}\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)\left(\boldsymbol{E}'_{II,k} + \boldsymbol{O}'_{II,k}\right) R_4^{1/2}\left(\begin{bmatrix}-\delta_{1,k}\\ -\delta_{2,k}\\ -\delta_{3,k}\end{bmatrix}\right).\end{aligned} \tag{20}$$

$$R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) R_4^{-1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) = \frac{1}{1+3t_k^2}\begin{pmatrix} 1-3t_k^2 & 2\sigma_{1,k}t_k & 2\sigma_{2,k}t_k & 2\sigma_{3,k}t_k\\ -2\sigma_{1,k}t_k & 1+t_k^2 & -2\sigma_{1,k}\sigma_{2,k}t_k^2 & -2\sigma_{1,k}\sigma_{3,k}t_k^2\\ -2\sigma_{2,k}t_k & -2\sigma_{2,k}\sigma_{1,k}t_k^2 & 1+t_k^2 & -2\sigma_{2,k}\sigma_{3,k}t_k^2\\ -2\sigma_{3,k}t_k & -2\sigma_{3,k}\sigma_{1,k}t_k^2 & -2\sigma_{3,k}\sigma_{2,k}t_k^2 & 1+t_k^2\end{pmatrix}$$

$$R_4^{1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ \sigma_{2,k}\\ \sigma_{3,k}\end{bmatrix}\right) R_4^{-1/2}\left(\begin{bmatrix}\sigma_{1,k}\\ -\sigma_{2,k}\\ -\sigma_{3,k}\end{bmatrix}\right) = \frac{1}{1+3t_k^2}\begin{pmatrix} 1+t_k^2 & 2\sigma_{1,k}t_k & -2\sigma_{1,k}\sigma_{3,k}t_k^2 & 2\sigma_{1,k}\sigma_{2,k}t_k^2\\ -2\sigma_{1,k}t_k & 1-3t_k^2 & -2\sigma_{3,k}t_k & 2\sigma_{2,k}t_k\\ -2\sigma_{1,k}\sigma_{3,k}t_k^2 & 2\sigma_{3,k}t_k & 1+t_k^2 & 2\sigma_{2,k}\sigma_{3,k}t_k^2\\ 2\sigma_{1,k}\sigma_{2,k}t_k^2 & -2\sigma_{2,k}t_k & 2\sigma_{2,k}\sigma_{3,k}t_k^2 & 1+t_k^2\end{pmatrix}$$

In terms of the first columns of $\boldsymbol{E}_{II,k}$, $\boldsymbol{E}_{II,k+1}$ and $\boldsymbol{O}_{II,k}$, $\boldsymbol{O}_{II,k+1}$, we have

$$\boldsymbol{e}'_{II,k+1} = R_4^{1/2}\left(\begin{bmatrix}\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}-\delta_{1,k}\\ -\delta_{2,k}\\ -\delta_{3,k}\end{bmatrix}\right)\ \boldsymbol{e}'_{II,k} \quad (21)$$

$$\boldsymbol{o}'_{II,k+1} = R_4^{1/2}\left(\begin{bmatrix}\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}-\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)\ \boldsymbol{o}'_{II,k}. \quad (22)$$

The operations in the above two equations are essentially the *3-D rational CORDIC elementary rotations*. Indeed, $\boldsymbol{e}'_{II,k}$ is rotated by the kth CORDIC elementary rotation, which is shown in the first expression at the bottom of the next page, where the 3×3 submatrix in the lower right corner is a 3-D rational CORDIC elementary rotation matrix similar to (11) if $\delta_{3,k} = \delta_{1,k}$ is imposed. Similarly, $\boldsymbol{o}'_{II,k}$ is subjected to the kth CORDIC elementary rotation (see the second expression at the bottom of this page). Crossing out the second row and second column, the remaining 3×3 submatrix is another 3-D rational CORDIC elementary rotation matrix if $\delta_{1,k} = \delta_{3,k}$. The control signs $\delta_{1,k} = \delta_{3,k}$ and $\delta_{2,k}$ are selected as indicated in Section II-D to force to zero the third and the fourth components of $\boldsymbol{o}'_{II}$. The zeros of $\boldsymbol{e}'_{II}$ created in the first evaluation phase are left unchanged.

Thus, at the end of the second evaluation phase, we have $\boldsymbol{A}''_{II} = \boldsymbol{E}''_{II} + \boldsymbol{O}''_{II}$ with the first columns of $\boldsymbol{E}''_{II}$ and $\boldsymbol{O}''_{II}$ equal to

$$\begin{pmatrix} e^{r''}_{1,II}\\ 0\\ 0\\ 0\end{pmatrix} = \prod_k R_4^{1/2}\left(\begin{bmatrix}\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}-\delta_{1,k}\\ -\delta_{2,k}\\ -\delta_{3,k}\end{bmatrix}\right)\begin{pmatrix} e^{r'}_{1,II}\\ 0\\ 0\\ 0\end{pmatrix} \quad (23)$$

$$\begin{pmatrix} o^{r''}_{1,II}\\ -o^{j''}_{1,II}\\ 0\\ 0\end{pmatrix} = \prod_k R_4^{1/2}\left(\begin{bmatrix}\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}-\delta_{1,k}\\ \delta_{2,k}\\ \delta_{3,k}\end{bmatrix}\right)\begin{pmatrix} o^{r'}_{1,II}\\ -o^{j'}_{1,II}\\ o^{r'}_{2,II}\\ o^{j'}_{2,II}\end{pmatrix}. \quad (24)$$

The computations in (23) may seem superfluous. However they must be implemented when the resolution of the CORDIC algorithms is limited—as in the faster version presented in Section IV—and hence the "zero" entries are not exactly zero. The reconstructed 2×2 complex matrix A''_{II} in the diagonal processor P_{II} becomes

$$A''_{II} = E''_{II} + O''_{II} = \begin{pmatrix} e^{r''}_{1,II} & 0\\ 0 & e^{r''}_{1,II}\end{pmatrix} + \begin{pmatrix} o^{r''}_{1,II} + jo^{j''}_{1,II} & 0\\ 0 & -o^{r''}_{1,II} + jo^{j''}_{1,II}\end{pmatrix}.$$

Thus at the end of the two-phase evaluation, the 2×2 complex submatrix A''_{II} in processor P_{II} is diagonal with *complex* (not real) diagonal elements.

E. CORDIC Applications in the Off-Diagonal Processors

The off-diagonal processor P_{IJ} performs on the left and right sides of $\boldsymbol{A}_{IJ}$ the corresponding CORDIC applications with the control signs from processors P_{II} and P_{JJ}, respectively. After the application process, $\boldsymbol{A}_{IJ}$ is transformed into

$$\boldsymbol{A}''_{IJ} = \boldsymbol{E}''_{IJ} + \boldsymbol{O}''_{IJ}$$

where $\boldsymbol{E}''_{IJ}$ and $\boldsymbol{O}''_{IJ}$ are determined from their first columns $\boldsymbol{e}''_{IJ}$ and $\boldsymbol{o}''_{IJ}$ with

$$\boldsymbol{e}''_{IJ} = \prod_k R_4^{1/2}\left(\begin{bmatrix}\delta^I_{1,k}\\ \delta^I_{2,k}\\ \delta^I_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}-\delta^J_{1,k}\\ -\delta^J_{2,k}\\ -\delta^J_{3,k}\end{bmatrix}\right) \times \prod_k R_4^{1/2}\left(\begin{bmatrix}\sigma^I_{1,k}\\ \sigma^I_{2,k}\\ \sigma^I_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}\sigma^J_{1,k}\\ \sigma^J_{2,k}\\ \sigma^J_{3,k}\end{bmatrix}\right)\ \boldsymbol{e}_{IJ} \quad (25)$$

$$\boldsymbol{o}''_{IJ} = \prod_k R_4^{1/2}\left(\begin{bmatrix}\delta^I_{1,k}\\ \delta^I_{2,k}\\ \delta^I_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}-\delta^J_{1,k}\\ \delta^J_{2,k}\\ \delta^J_{3,k}\end{bmatrix}\right) \times \prod_k R_4^{1/2}\left(\begin{bmatrix}\sigma^I_{1,k}\\ \sigma^I_{2,k}\\ \sigma^I_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}\sigma^J_{1,k}\\ -\sigma^J_{2,k}\\ -\sigma^J_{3,k}\end{bmatrix}\right)\ \boldsymbol{o}_{IJ} \quad (26)$$

where the superscripts I and J on the control signs represent the indices of the processors from which the control signs are generated. Depending on the ± 1 values of the parameters, the

$$\frac{1}{1+3t_k^2} \times \begin{pmatrix} 1+3t_k^2 & 0 & 0 & 0\\ 0 & 1-t_k^2 & -2\delta_{3,k}t_k + 2\delta_{1,k}\delta_{2,k}t_k^2 & 2\delta_{2,k}t + 2\delta_{1,k}\delta_{2,k}t_k^2\\ 0 & 2\delta_{3,k}t_k + 2\delta_{1,k}\delta_{2,k}t_k^2 & 1-t_k^2 & -2\delta_{1,k}t_k + \delta_{2,k}\delta_{3,k}t_k^2\\ 0 & -2\delta_{2,k}t_k + 2\delta_{1,k}\delta_{3,k}t_k^2 & 2\delta_{1,k}t_k + 2\delta_{2,k}\delta_{3,k}t_k^2 & 1-t_k^2\end{pmatrix}$$

$$\frac{1}{1+3t_k^2} \times \begin{pmatrix} 1-t_k^2 & 0 & 2\delta_{2,k}t_k + 2\delta_{1,k}\delta_{3,k}t_k^2 & 2\delta_{3,k}t_k - 2\delta_{1,k}\delta_{2,k}t_k^2\\ 0 & 1+3t_k^2 & 0 & 0\\ -2\delta_{2,k}t_k + 2\delta_{1,k}\delta_{3,k}t_k^2 & 0 & 1-t_k^2 & -2\delta_{1,k}t_k - 2\delta_{2,k}\delta_{3,k}t_k^2\\ -2\delta_{3,k}t_k - 2\delta_{1,k}\delta_{2,k}t_k^2 & 0 & 2\delta_{1,k}t_k - 2\delta_{2,k}\delta_{3,k}t_k^2 & 1-t_k^2\end{pmatrix}.$$

product

$$R_4^{1/2}\left(\begin{bmatrix}\alpha_{1,k}\\ \alpha_{2,k}\\ \alpha_{3,k}\end{bmatrix}\right)R_4^{-1/2}\left(\begin{bmatrix}\beta_{1,k}\\ \beta_{2,k}\\ \beta_{3,k}\end{bmatrix}\right)$$

is equal, up to symmetric permutations of rows and columns, to an elementary rotation matrix of either the 4-D Householder CORDIC algorithm or a slight extension of the 3-D rational CORDIC algorithm, where the sign in $\overline{S_k}$ in Section II-C is independent of the other two signs. Therefore, the CORDIC applications in the off-diagonal processor P_{IJ} consist of two consecutive sequences of 3-D rational CORDIC or 4-D Householder CORDIC elementary rotations applied in parallel to the first columns of the even and odd matrices $\boldsymbol{E}_{IJ}$ and $\boldsymbol{O}_{IJ}$. Each processor in Fig. 3 consists of two 4-D unified CORDIC processors, which can perform 4-D Householder CORDIC operations as well as 3-D rational CORDIC operations. Thus the area of the diagonal or off-diagonal processor is

$$\mathcal{A}_{II} = \mathcal{A}_{IJ} = 2\mathcal{A}_{4D,\text{unified}}$$

where $\mathcal{A}_{4D,\text{unified}}$ denotes the area of a unified 4-D CORDIC processor in Fig. 2(b). The delay of each Jacobi SVD step is $\mathcal{T}_{\text{SVD}} = 2\mathcal{T}_{3D}^{'}$ where $\mathcal{T}_{3D}^{'}$ denotes the computation time of one 3-D rational CORDIC operation. The additional hardware requirements for the concurrent computation of the unitary matrices U and V are discussed in Section III-H.

F. SVD of Hermitian Matrices

If the $m \times m$ matrix A is Hermitian matrix, so is A_{II}, the 2×2 submatrix in the diagonal processor P_{II}. In this special case, the even matrix E_{II} is diagonal, and both E_{II} and O_{II} have real diagonal elements. Hence, the first evaluation phase—described by (18) and (19)—is redundant and can be skipped. Therefore, the evaluation and application operations can be accomplished within the computation time of one 3-D rational CORDIC operation instead of the time of two such operations [12]. Furthermore, after the evaluation, the diagonalized matrix $A_{II}^{''}$ is real.

G. Real Diagonalization in the Diagonal Processors

Since in general the Jacobi SVD algorithm converges to a diagonal matrix with *complex* diagonal elements, additional operations on the complex diagonal matrix are required in order to force the diagonal elements real and positive. Considering the 2×2 complex diagonal submatrix in the diagonal processor P_{II} after the convergence of the Jacobi algorithm, two 2-D CORDIC operations are required to annihilate the imaginary parts of the two complex diagonal elements. Afterward, sign inversion may be needed to make the diagonal elements positive. The above computation occurs in the diagonal processors only after the Jacobi algorithm converges, hence the magnitudes of the off-diagonal elements are negligible. The computation time of this last phase is very small compared to the total computation time of the Jacobi algorithm. This phase is not needed in the Hermitian case.

H. Construction of the Unitary Matrices U and V

The unitary matrices U and V of the SVD of an $m \times m$ matrix $A = USV^H$ can be calculated concurrently with the computation of S if each processing element in the array of Fig. 3 also contains two pairs of quaternion CORDIC processors, each pair in charge of accumulating the product of the rotations applied from the left or from the right of A. More specifically, assume that initially a processing element P_{IJ} holds, in addition to A_{IJ}, two 2×2 submatrices U_{IJ}^H and V_{IJ} where U_{IJ}^H and V_{IJ} are null matrices if $I \neq J$ and are identity matrices if $I = J$. Considering the update for U_{IJ}^H, concurrently with the CORDIC evaluations or applications discussed in Sections III-D and E, two identical quaternion CORDIC operations, representing the left rotation, are performed separately on the first columns of the two 4×4 real matrices that represent the even and the odd parts of the current iterate of U_{IJ}^H. The two resultant 4-D column vectors are then used to reconstruct the new U_{IJ}^H. The above operations are performed in a pair of quaternion CORDIC processors inside processing element P_{IJ}. After each Jacobi step, the entries of the iterate of U_{IJ}^H are exchanged among its neighbors. Similar operations for the update of V_{IJ} are performed in another pair of quaternion CORDIC processors. The two unitary matrices U^H and V are thus generated when the Jacobi SVD converges. (When A is not square, USV^H is the SVD of the matrix R in the decomposition $A = QR$ and the product of the first m columns of Q by U must be computed on the CORDIC array. A procedure detailed in [11] solves that problem.)

I. Comparison of Two CORDIC-Based Jacobi SVD Methods

In [6], Cavallaro and Elster proposed the Jacobi SVD of complex matrices using the 2-D explicit CORDIC algorithm. The total computation time for one Jacobi step is $\mathcal{T}_{\text{SVD}} = 9.75\mathcal{T}_{2D}^{1/2}$ where $\mathcal{T}_{2D}^{1/2}$ denotes the computation time of one 2-D CORDIC operation with the elementary rotations defined in (7). The 0.75 factor accounts for the CORDIC scaling iterations [5]. The areas of the diagonal and off-diagonal processors (not including computation of U and V) are $\mathcal{A}_{II} = \mathcal{A}_{IJ} = 4\mathcal{A}_{2D,\text{exp}}^{1/2}$ where $A_{2D,\text{exp}}^{1/2}$ is the area of a 2-D explicit CORDIC processor shown in Fig. 1. For our Jacobi SVD method using higher dimensional CORDIC algorithms, the computation time of one Jacobi step is reduced to $\mathcal{T}_{\text{SVD}} = 2.15\mathcal{T}_{3D}^{'}$ where $\mathcal{T}_{3D}^{'}$ is the computation time of one 3-D rational CORDIC operation. The factor 0.15 accounts for the computation time of the scaling iterations on a 4-D unified CORDIC processor. The processor area (not including the computation of U and V) is $\mathcal{A}_{II} = \mathcal{A}_{IJ} = 2\mathcal{A}_{4D,\text{unified}}$. Fig. 4 details the computation time of one Jacobi step in these two methods.

Assuming 32-bit accuracy and double-metal scalable CMOS 1-μm technology

$$A_{2D,\text{exp}}^{1/2} \simeq 2100 \times 2000\ \mu\text{m}^2\ , \qquad T_{2D}^{1/2} \simeq 0.63\ \mu\text{s}$$
$$A_{4D,\text{unified}} \simeq 3600 \times 4500\ \mu\text{m}^2\ , \qquad T_{3D}^{'} \simeq 0.80\ \mu\text{s}.$$

The above area and time estimates take into consideration the contribution of the interconnection wires, drivers, registers,

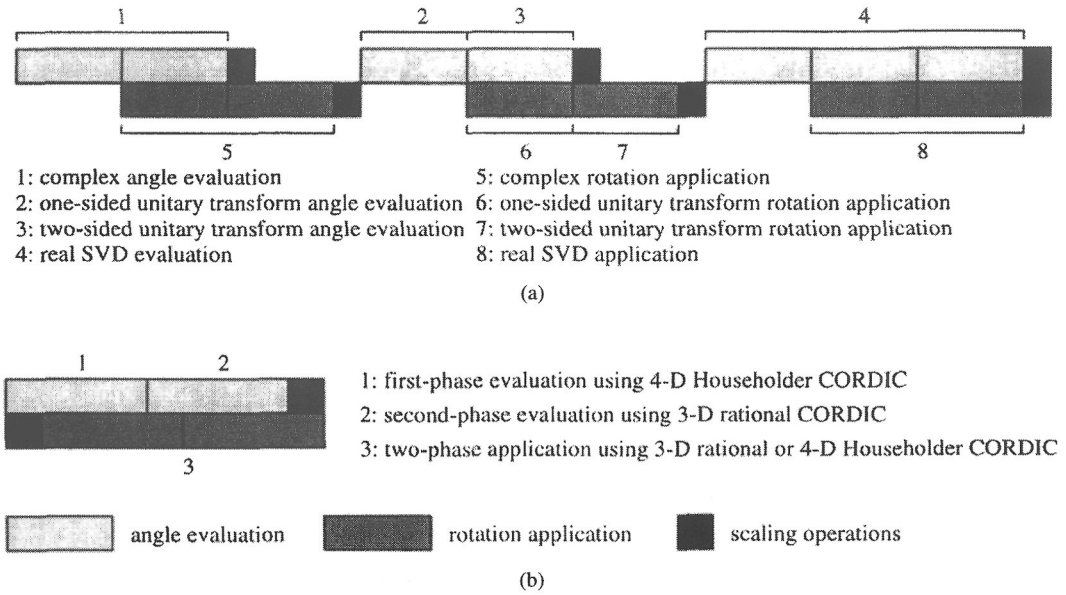


Fig. 4. Comparison of two CORDIC-based Jacobi methods for complex matrices: (a) Cavallaro and Elster's method; (b) proposed method. For each method the top (respectively, bottom) trace refers to the diagonal (respectively, off-diagonal) processors.

multiplexers, and other data steering units. The delay estimates are obtained using SPICE. The adders are designed combining multiple-output domino logic with carry-look-ahead [25]. The shifters are barrel shifters with switches implemented by gate logic to minimize delay. Using the above time estimates, each Jacobi step takes about 6.1 μs in Cavallaro and Elster's method and only 1.7 μs in our method. Thus a speed-up factor of about 3.5 is achieved while the hardware complexity in each processor is barely doubled.

For comparison, on a single-CPU SPARC-20 workstation and using a variant optimized for the SPARC CPU of the procedure in [6], the diagonalization of a 2×2 complex matrix takes around 37 μs while a corresponding two-sided complex rotation application takes around 13 μs. Thus the computation time of a Jacobi step for a 50×50 complex matrix is approximately 8.7 ms on a SPARC-20. The computation time for the SVD of a 50×50 complex matrix on a single-CPU SPARC-20 is about 1.6 s if the triangular structure of the matrix A is preserved by the Jacobi steps [27]. Sequential software packages typically use the more efficient Golub–Kahan algorithm (QR algorithm) to compute the SVD [17]. For example, MATLAB computes the SVD of the same 50×50 complex matrix within 0.4 s on a SPARC-20 workstation. Unfortunately, the most efficient sequential SVD algorithms are much less suited to processor array implementations than the Jacobi algorithm. As will be seen in the next section, the SVD of the 50×50 complex matrix on a 25×25 processor array implementing our method takes 0.13 to 0.25 ms.

Recently, Hemkumar and Cavallaro [19], [20] have proposed an approach which improves upon the original Cavallaro and Elster's method by using two pairs of "inner" and "outer" transformations and by adopting Yang and Böhme's method [37] for the last outer transformation. With this approach, the computation time of one Jacobi step is reduced to $\mathcal{T}_{\text{SVD}} = 7\mathcal{T}_{2D}^{1/2}$ while the area complexity is still $\mathcal{A}_{II} = \mathcal{A}_{IJ} = 4\mathcal{A}_{2D,\text{exp}}^{1/2}$. Our method is still about 2.6 times faster and, as will be seen in the next section, because it uses implicit CORDIC algorithms for all the transformations, its speed can be further doubled at almost no extra hardware cost.

IV. Jacobi SVD with Partial CORDIC

In the Jacobi SVD algorithm, the zeros created at each step are smeared by subsequent rotations. Thus the exact annihilation of matrix entries is not necessary during the early steps of the algorithm, a property exploited in the "threshold Jacobi method" [36]. With an *implicit* CORDIC algorithm one may apply in the early steps of the Jacobi algorithm "partial-resolution" CORDIC operations to bring the off-diagonal elements just within some sufficiently small magnitude, thus reducing the total number of CORDIC iterations [9], [12].

In a partial CORDIC algorithm only a contiguous subsequence of the sequence of rotation iterations in the original, "full," CORDIC algorithm is applied. The first and last rotation parameters, 2^{-p} and 2^{-q}, with p and q the first and last shift, may change from one operation to another. The *window size* $w = q - p + 1$ is the total number of rotation iterations in one CORDIC operation. In the standard, full-window, CORDIC algorithm, the window size is fixed to cover the desired range and achieve maximal resolution. In a unit-window partial CORDIC algorithm, $p = q$ and each CORDIC operation consists in a single rotation iteration [18], [21], [31].

In [18], Götze *et al.* proposed a Jacobi-like algorithm for the parallel eigenvalue computations of real symmetric matrices in which a unit-window partial CORDIC operation is performed on every 2×2 submatrix on the diagonal. While the approach has been recently extended to Hermitian matrices [21], its generalization to the SVD of general complex matrices seems quite difficult. The process of determining $p(= q)$ for each diagonal submatrix involves several comparisons, and the associated scaling operations—potentially different from processor to processor—create extra computation overhead.

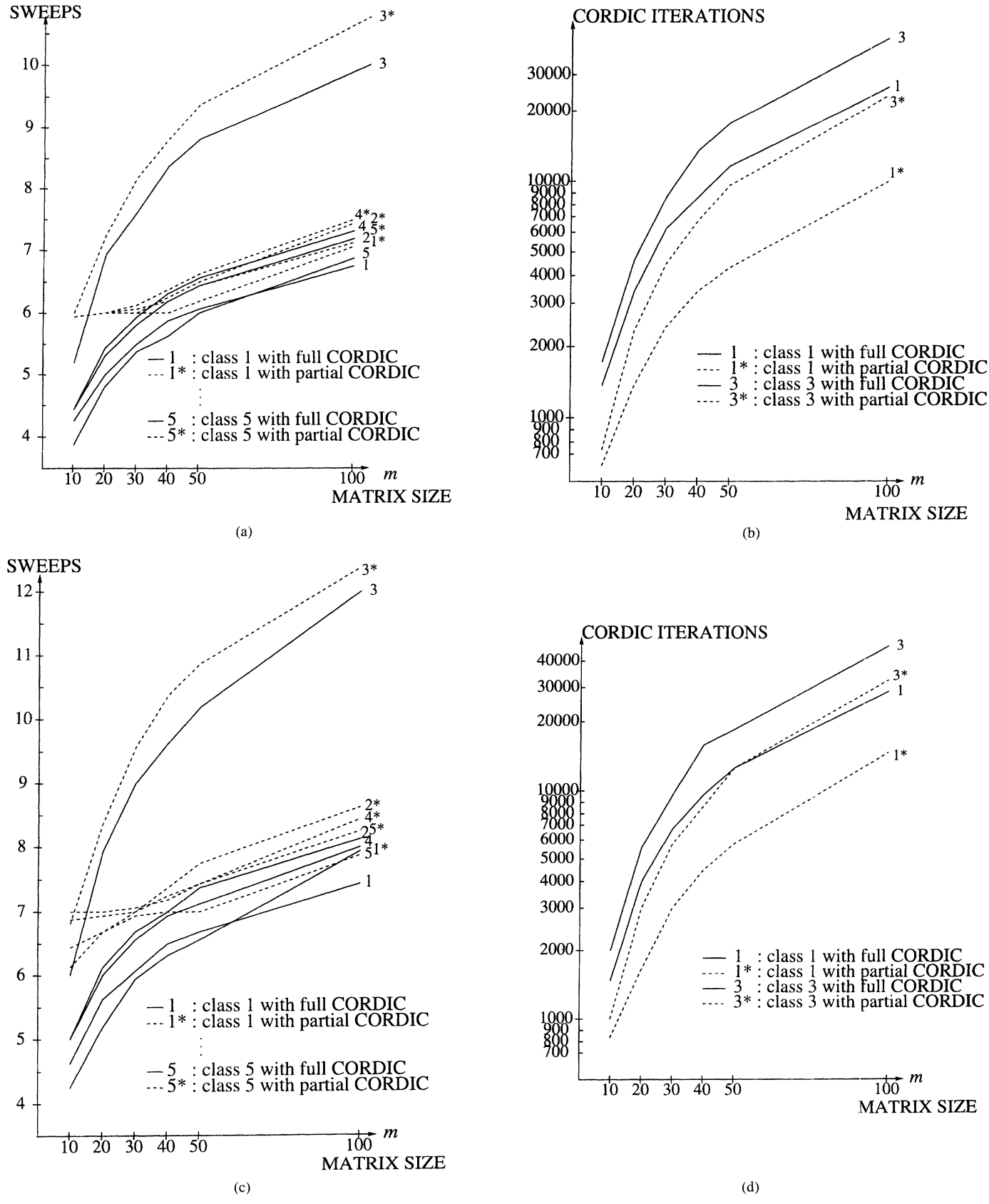


Fig. 5. (a) Number of sweeps versus matrix size m for Hermitian matrices; (b) number of CORDIC rotation iterations versus m for Hermitian matrices; (c) number of sweeps versus m for general complex matrices; (d) number of CORDIC rotation iterations versus m for general complex matrices.

We propose that p and q be identical for all 2×2 submatrices and remain unchanged throughout each sweep in order to simplify the scaling. Thus if no scaling CORDIC iterations were applied, the final matrices obtained after convergence would just be scaled versions of the true results (assuming no arithmetic overflow occurred). For this discussion we shall

make the pessimistic assumption that at each step scaling is performed in the same way as in Fig. 4(b) and with the same number of iterations as for the full CORDIC algorithm. We present here only the best of several strategies for updating p and q, as came out from our experiments.

1) Start with $p^{(1)} = 1$ and $q^{(1)} = q_0$. The initial value q_0 is selected equal to 4.
2) At the beginning of each sweep, increase the resolution—until reaching full resolution (shift b)—according to $p^{(i+1)} = p^{(i)}, q^{(i+1)} = \min(q^{(i)} + \triangle q, b)$, where the increment $\triangle q$ is selected equal to 4.

We shall see that this scheme provides a speed-up factor of about 2 on top of the speed-up obtained by using the even-plus-odd decomposition discussed in Section III.

Since the convergence rate of the Jacobi SVD algorithm depends on the distribution of the singular values of the matrices, we generated for our experiments random matrices belonging to several classes with different singular value distributions, and applied to these matrices the Jacobi method with full CORDIC and with partial CORDIC algorithms. The matrices considered belong to five classes, listed below. For all these classes, the random variables are independent, the $m \times m$ matrices D_k are diagonal and real, and the matrices U and V are random unitary matrices with columns uniformly distributed on the m-dimensional unit sphere (subject to the orthogonality constraints).

- Class 1: the real and imaginary parts of all the entries in the matrix are uniformly distributed between -1 and 1.
- Class 2: $A = UD_2V^H$ where D_2 contains singular values uniformly distributed in an interval.
- Class 3: $A = UD_3V^H$, where the diagonal entries of D_3 are lumped into two groups, each with very small standard deviation compared to the difference of the means of the two groups. Thus matrix A tends to have multiple singular values.
- Class 4: $A = UD_4V^H$, where D_4 has two diagonal entries very close to each other while the others are uniformly distributed in an interval.
- Class 5: $A = UD_5V^H$, where the two extreme diagonal entries in D_5 have a large ratio (i.e., A is ill-conditioned) while the others are uniformly distributed in an interval.

The stop criterion for the Jacobi algorithm in our experiments was that the sum of the squares of the off-diagonal elements be less than 10^{-12} times the sum of the squares of all the elements. Fig. 5 shows, for selected matrix sizes and 100 independent trials per data point, the maximum numbers of sweeps and of CORDIC rotation iterations for the SVD of Hermitian matrices and of general complex matrices using full and partial CORDIC algorithms. For instance, for 100×100 general complex matrices, the Jacobi method using full CORDIC requires about 0.61 ms for matrices in class 1 and 0.97 ms for matrices in class 3. With partial CORDIC, the total computation times drop, respectively, to 0.34 and 0.71 ms. For 50×50 matrices in class 1, the required computation times are 0.25 ms with full CORDIC and 0.13 ms with partial CORDIC. (The array data input/output is at most a few percent of the computation time. Furthermore, when computing the SVD of a batch of matrices, it may be overlapped with the computations.) From the experiment, we make the following observations:

1) Convergence for matrices in class 3 requires more sweeps, and thus more CORDIC iterations, than for matrices in other classes.
2) The test matrices in classes other than 3 have similar convergence rates.
3) The speed-up of the Jacobi SVD using partial CORDIC with respect to using full CORDIC is close to 2 except for the matrices in class 3, for which the speed-up seldom exceeds 1.5.
4) The average number of sweeps for a matrix of given class and dimension is at most 1.5 less than the maximum shown in Fig. 5.

In a systolic implementation such as the one shown in Fig. 3, classical matrix-dependent termination criteria for the Jacobi algorithm are very expensive. Thus a good strategy is to stop after a predetermined number of sweeps, possibly dependent on the dimension of the matrix. According to Fig. 5, a very simple strategy is to stop after 13 sweeps for every matrix of size up to 100. Larger matrices require larger numbers of sweeps. In [3] where the SVD of real matrices is considered, Brent *et al.* suggested to select 10 as the number of sweeps for matrices of practical size (say, $m < 1000$) because they considered only random matrices belonging to class 1. But as we have seen, matrices in class 3 require more sweeps for convergence. We also tested other matrices, proposed in [30], but the convergence rates were *not* worse than for the matrices in class 3.

V. Conclusion

The core operation in a highly parallel version of the Jacobi method for the singular-value decomposition of complex matrices is the complex two-sided rotation of a 2×2 matrix. Because any 2×2 complex matrix can be decomposed into the sum of two special complex matrices—a scaled rotation and a scaled reflection—the operation is equivalent to a left-sided complex rotation on each matrix in the decomposition, and thus amounts to two concurrent complex rotations on the first column of each matrix.

With the original CORDIC algorithm, the real rotation of a two-dimensional real vector may be performed on both components in parallel. Multidimensional CORDIC algorithms enable the complex rotation of a 2-D complex vector to be performed almost as quickly as a real 2-D rotation by computing all components simultaneously. With these new CORDIC algorithms the two one-sided complex rotations are performed on all components in parallel. Computation time is further reduced by using rotations whose resolution increases from coarse to fine as the Jacobi method proceeds. A speed-up of 5 or more is achieved with respect to parallel implementations based on the original CORDIC algorithm.

We have described a parallel implementation of the Jacobi algorithm on a square array of 4-D CORDIC processors. Since 4-D CORDIC processors can with little extra control implement two 2-D CORDIC operations in parallel, we are

advocating the use of an array of 4-D CORDIC processors to implement the SVD of both real and complex matrices as well as eigendecompositions, QR decompositions, and other matrix operations common in signal processing. If higher speeds are desired, a promising route is to generalize to multidimensional CORDIC algorithms the redundant arithmetic methods presented in [29] and use redundant arithmetic in conjunction with the methods presented in this paper.

REFERENCES

[1] H. M. Ahmed, "Signal processing algorithms and architectures," Ph.D. dissertation, Dept. Elec. Eng., Stanford Univ., Stanford, CA, June 1982.

[2] H. M. Ahmed, J.-M. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *IEEE Comput.*, pp. 65–82, Jan. 1982.

[3] 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, no. 1, pp. 69–84, Jan. 1985.

[4] 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. 1, no. 3, pp. 242–270, 1985.

[5] J. R. Cavallaro and F. T. Luk, "CORDIC arithmetic for an SVD processor," *J. Parallel Distrib. Comput.*, no. 5, pp. 271–290, June 1988.

[6] J. R. Cavallaro and A. C. Elster, "A CORDIC processor array for the SVD of a complex matrix," in R. Vaccaro, Ed., *SVD and Signal Processing II.* Amsterdam, The Netherlands: Elsevier, 1991, pp. 227–239.

[7] J.-M. Delosme, "VLSI implementation of rotations in pseudo-Euclidean spaces," in *Proc. IEEE ICASSP*, Apr. 1983, pp. 927–930.

[8] ——, "A processor for two-dimensional symmetric eigenvalue and singular value arrays," in *Proc. 21st Asilomar Conf. on Circuits, Systems, and Computers*, Nov. 1987, pp. 217–221.

[9] ——, "CORDIC algorithms: Theory and extensions," in *Advanced Algorithms and Architectures for Signal Processing IV, Proc. SPIE*, vol. 1152, Aug. 1989, pp. 131–145.

[10] ——, "Bit-level systolic algorithm for the symmetric eigenvalue problem," in *Proc. Int. Conf. on Application Specific Array Processors* , Princeton, NJ, Sept. 1990, pp. 770–781.

[11] ——, "Parallel implementations of the SVD using implicit CORDIC arithmetic," in *SVD and Signal Processing II: Algorithms, Analysis and Applications,* R. Vaccaro, Ed. Amsterdam, The Netherlands: Elsevier, 1991, pp. 33–56.

[12] ——, "Bit-level systolic algorithms for real symmetric and Hermitian eigenvalue problems," *J. VLSI Signal Processing*, vol. 4, pp. 69–88, 1992.

[13] J.-M. Delosme and S.-F. Hsiao, "CORDIC algorithms in four dimensions," in *Advanced Algorithms and Architectures for Signal Processing IV, Proc. SPIE* vol. 1348, pp. 349–360, July 1990.

[14] E. F. Deprettere, D. Dewilde, and R. Udo, "Pipelined CORDIC architectures for fast VLSI filtering and array processing," in *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, Mar. 1984, pp. 3:41A.6.1–41A.6.4.

[15] M. D. Ercegovac and T. Lang, "Redundant and on-line CORDIC: Application to matrix triangularization and SVD," *IEEE Trans. Comput.*, vol. 39, no. 6, pp. 725–740, June 1990.

[16] F. R. Gantmacher, *The Theory of Matrices.* New York: Chelsea, 1959.

[17] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed. Baltimore, MD: Johns Hopkins Univ. Press, 1989, pp. 444–459.

[18] J. Götze, S. Paul, and M. Sauer, "An efficient Jacobi-like algorithm for parallel eigenvalue computation," *IEEE Trans. Comput.*, vol. 42, pp. 1058–1065, Sept. 1993.

[19] N. D. Hemkumar and J. R. Cavallaro, "Efficient complex matrix transformations with CORDIC," in *Proc. IEEE 11th Symp. on Computer Arithmetic* (Windsor, Ont., Canada, June 1993) pp. 122–129.

[20] ——, "Redundant and on-line CORDIC for unitary transformations," *IEEE Trans. Comput.*, vol. 43, pp. 941–954, Aug. 1994.

[21] ——, "Jacobi-like matrix factorizations with CORDIC-based Inexact diagonalizations," in *Proc. 5th SIAM Conf. on Applied Linear Algebra* (Snowbird, UT, June 1994), pp. 295–299.

[22] S.-F. Hsiao and J.-M. Delosme, "Householder CORDIC algorithms," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 990–1001, Aug. 1995.

[23] Y. H. Hu, "CORDIC-based VLSI architecture for digital signal processing," *IEEE Signal Processing Mag.*, vol. 9, no. 3, pp. 17–35, July 1992.

[24] Y. H. Hu and H. E. Liao, "CALF: A CORDIC adaptive lattice filter," *IEEE Trans. Signal Processing*, vol. 40, no. 4, pp. 990–993, Apr. 1992.

[25] I. S. Hwang and A. L. Fisher, "Ultrafast compact 32-bit CMOS adders in multiple-output domino logic," *IEEE J. Solid State. Circuits*, vol. 24, no. 2, pp. 358–369, Apr. 1989.

[26] K. J. R. Liu, S. F. Hsieh, and K. Yao, "Systolic block Householder transformation for RLS algorithm with two-level pipelined implementation," *IEEE Trans. Signal Processing*, vol. 40, no. 4, pp. 946–958, Apr. 1992.

[27] F. T. Luk, "A triangular processor array for computing singular values," *Linear Alg. Appl.*, vol. 77, pp. 259–273, 1986.

[28] C. Mazenc, X. Merrheim, and J.-M. Muller, "Computing functions $\cos^{-1}$ and $\sin^{-1}$ using Cordic," *IEEE Trans. Comput.*, vol. 42, no. 1, pp. 118–122, Jan. 1993.

[29] X. Merrheim, "Bases Discrètes et Calcul des Fonctions Elémentaires par Matériel," Ph.D. dissertation, Ecole Normale Supérieure de Lyon, Lyon, France, Feb. 1994.

[30] P. P. M. De Rijk, "A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 2, pp. 359–371, Mar. 1989.

[31] D. E. Schimmel, "Bit-level Jacobi-like algorithms for eigenvalue and singular value decompositions," Ph.D dissertation. Dept. Elec. Eng., Cornell Univ., Ithaca, NY, Jan. 1991.

[32] N. Takagi, T. Asada, and S. Yajima, "Redundant CORDIC methods with a constant scale factor for sine and cosine computation," *IEEE Trans. Comput.*, vol. 40, pp. 989–995, 1991.

[33] A.-J. van der Veen and E. F. Deprettere, "Parallel VLSI matrix pencil algorithm for high resolution direction finding," *IEEE Trans. Signal Processing,* vol. 39, pp. 383–394, Feb. 1991.

[34] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electron. Comput.*, vol. EC-8, no. 3, pp. 330–334, Sept. 1959.

[35] J. S. Walther, "A unified algorithm for elementary functions," in *Proc. AFIPS Conf.*, vol. 38, 1971, pp. 379–385.

[36] J. H. Wilkinson, *The Algebraic Eigenvalue Problem.* Oxford, UK: Clarendon, 1988.

[37] B. Yang and J. F. Böhme, "Reducing the computations of the singular value decomposition array given by Brenk and Luk," *SIAM J. Matrix Anal. Appl.*, vol. 12, no. 4, pp. 713–725, Oct. 1991.

SOLVING EIGENVALUE AND SINGULAR VALUE PROBLEMS ON AN UNDERSIZED SYSTOLIC ARRAY*

ROBERT SCHREIBER†

Abstract. Systolic architectures due to Brent, Luk and Van Loan are today the most promising method for computing the symmetric eigenvalue and singular value decompositions in real time. These systolic arrays, however, are only able to decompose matrices of a given fixed size. Here we present two modified algorithms and a modified array that do not have this disadvantage. The results of a numerical experiment show that a combination of one of our new algorithms and the modified array can decompose matrices of arbitrary size with little or no loss of efficiency.

Key words. eigenvalue problems, singular value decomposition, systolic computation, parallel computation

1. Introduction. Systolic arrays are of significant and growing importance in numerical computing [12], especially in matrix computation and its applications in digital signal processing [13]. There is now considerable interest in systolic computation of the singular value decomposition [2], [4], [6], [10] and the symmetric eigenvalue problem [1], [8].

To date, the most powerful systolic array for the eigenvalues of a symmetric $n \times n$ matrix is a square $n/2 \times n/2$ array due to Brent and Luk. This array implements a certain cyclic Jacobi method. It takes $O(n)$ time to perform a sweep of the method, and $O(\log n)$ sweeps for the method to converge [1].

Brent and Luk have also invented a closely related $(n/2)$-processor linear array for computing the singular value decomposition (SVD) of an $m \times n$ matrix A. An SVD of A is a factorization $A = U\Sigma V^T$, where V is orthogonal, Σ is nonnegative and diagonal, and U is $m \times n$ with orthonormal columns. This array implements a cyclic Hestenes algorithm that, in real arithmetic, is an exact analogue of their Jacobi method applied to the eigenproblem for A^TA. The array requires $O(mn)$ time for a sweep, and $O(\log n)$ sweeps for convergence [2].

A new array, much like the eigenvalue array, is reported by Brent, Luk and Van Loan to be capable of finding the SVD in time $O(m + n \log n)$ [3].

The purpose of this paper is to consider an important, indeed an essential problem concerning the practical use of these arrays. How, with an array of a given fixed size, can we decompose matrices of arbitrarily large size?

2. Systolic arrays for the Jacobi and Hestenes methods. We shall concentrate on Hestenes' method for the SVD. Starting with the given matrix A, we build an orthogonal matrix V such that AV has orthogonal columns. Thus

$$AV = U\Sigma,$$

where U has orthonormal columns and Σ is nonnegative and diagonal. An SVD is given by $A = U\Sigma V^T$.

To construct V, we take $A^{(0)} = A$, and iterate

$$A^{(i+1)} = A^{(i)}Q^{(i)}, \qquad i = 0, 1, \cdots,$$

* Received by the editors June 14, 1983, and in final revised form November 12, 1984. A preliminary version of this paper appeared in *Real-Time Signal Processing* VI, SPIE vol. 431 (1983), pp. 72-77. This research was partially supported by the U.S. Office of Naval Research under contract N00014-82-K-0703.

† Guiltech Research Company, 255 San Geronimo Way, Sunnyvale, California 94086.

with $Q^{(i)}$ orthogonal, until some matrix $A^{(i)}$ has orthogonal columns. $Q^{(i)}$ is chosen to be a product of $n(n-1)/2$ plane rotations

$$Q^{(i)} = \prod_{j=1}^{n(n-1)/2} Q_j^{(i)}.$$

Every possible pair (r, s), $1 \leqq r < s \leqq n$, is associated with one of the rotations $Q_j^{(i)}$ (the association is independent of i) in this way: the rotation $Q_j^{(i)}$ is chosen to make columns r and s of

$$A^{(i)}\left(\prod_{k=1}^{j} Q_k^{(i)}\right)$$

orthogonal. The process of going from $A^{(i)}$ to $A^{(i+1)}$ is called a "sweep." Every permutation of the set of pairs corresponds to a different cyclic Hestenes method.

The correspondence with the Jacobi method is this. The sequence $A^{(i)T}A^{(i)}$ converges to the diagonal matrix Σ^2 of eigenvalues of A^TA. Moreover,

$$A^{(i+1)T}A^{(i+1)} = Q^{(i)T}(A^{(i)T}A^{(i)})Q^{(i)},$$

where $Q^{(i)}$ is the product of $n(n-1)/2$ of Jacobi rotations that zero, in some cyclic order, the off-diagonal elements of $A^{(i)T}A^{(i)}$.

The permutation chosen by Brent and Luk allows the rotations to be applied in parallel in groups of $n/2$. Their permutation consists of $n-1$ groups of $n/2$ pairs such that, in each group, every column occurs once. Thus, the $n/2$ rotations corresponding to a pair-group commute. They can be applied in any order or, in fact, in parallel.

The SVD array is shown in Fig. 1. There are $n/2$ processors. Each processor holds two matrix columns. Initially processor i holds column $2i-1$ in its "left memory" and column $2i$ in its "right memory."

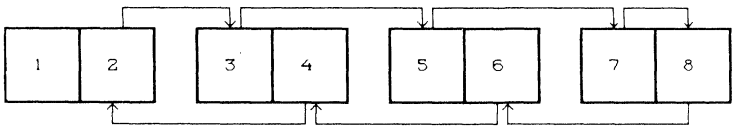


FIG. 1. *The* SVD *array*; $n = 8$.

In each cycle, each processor computes and applies to its two columns a plane rotation that makes them orthogonal. Next, using the connections shown in Fig. 1, columns move to neighboring processors. This produces a new set of $n/2$ column-pairs.

After $n-1$ cycles, $n(n-1)/2$ pairs of columns have been orthogonalized. It can be shown (by a parity argument) that no pair occurs twice during this time. Thus, every pair is orthogonalized exactly once. We call the process of orthogonalizing all pairs, in this parallel order, an A-sweep.

A diagram (given in [2] originally) showing the movement of columns through the array, very important in the considerations to follow, is given in Fig. 2.

3. Solving larger problems. We now consider the problem of finding an SVD when A has n columns, the array has p processors, and $n > 2p$.

The usual approach to this problem is to imagine that a "virtual" array, large enough to solve the problem (having $\lceil n/2 \rceil$ or more processors), is to be simulated by the given small physical array. Moreover, the simulation must be efficient. The array should not spend a large amount of time loading and unloading data.

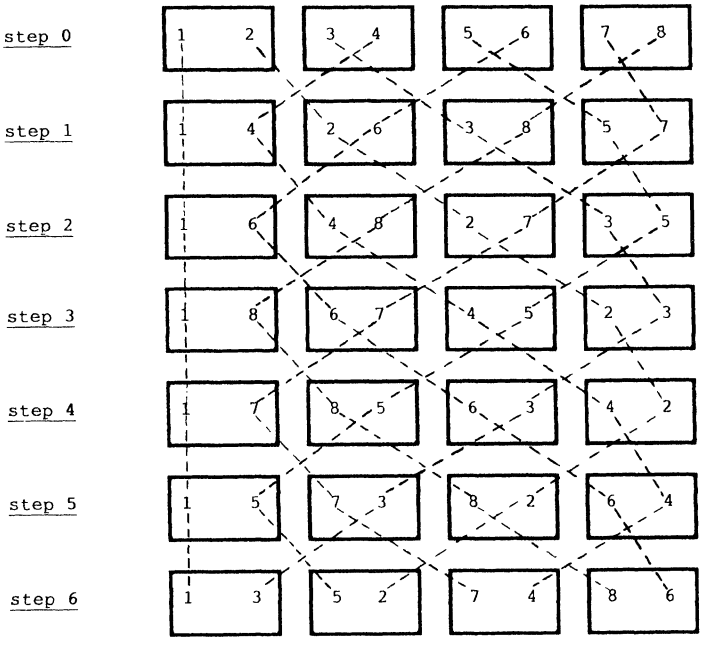


FIG. 2. *Flow of data in the* SVD *array*; $n = 8$.

For some arrays, this simulation is trivial. One finds a subarray of the virtual array, of the same size as the physical array, for which all the input streams are known. Clearly the action of such a subarray can be carried out and its outputs stored. These outputs then become the inputs to other subarrays. This process continues until, subarray by subarray, the computation of the entire virtual array has been performed. If this technique is possible, we say that the array is "decomposable." The various matrix multiply arrays [7], the array of Gentleman and Kung for QR factorization [5] and the array of Schreiber and Kuekes for solving triangular systems [9] are good examples of decomposable arrays.

Some arrays are indecomposable: the Kung-Leiserson band-matrix LU factorization array, for example [7].

Consider the 4×4 matrix multiplication array shown in Fig. 3. It computes $C + AB$ where C and B are $4 \times m$ and A is 4×4. Suppose we have a 2×2 array of the same type. With it, $C + AB$ can be computed using a block algorithm. Partition A, B and C so that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \qquad C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix},$$

where the blocks are 2×2 or $2 \times m$. We use the array to carry out these operations:

1. $C_1 := A_{11}B_1$;
2. $C_2 := A_{21}B_1$;
3. $C_1 := C_1 + A_{12}B_2$;
4. $C_2 := C_2 + A_{22}B_2$.

An equivalent viewpoint is that we use the array to emulate (that is, perform the work of) four 2×2 sections of the 4×4 array; the sections are shown in dotted outline in Fig. 3. Note that the input data for each section is known at the time the section is emulated by the 2×2 array, either because that data is input data or because it is

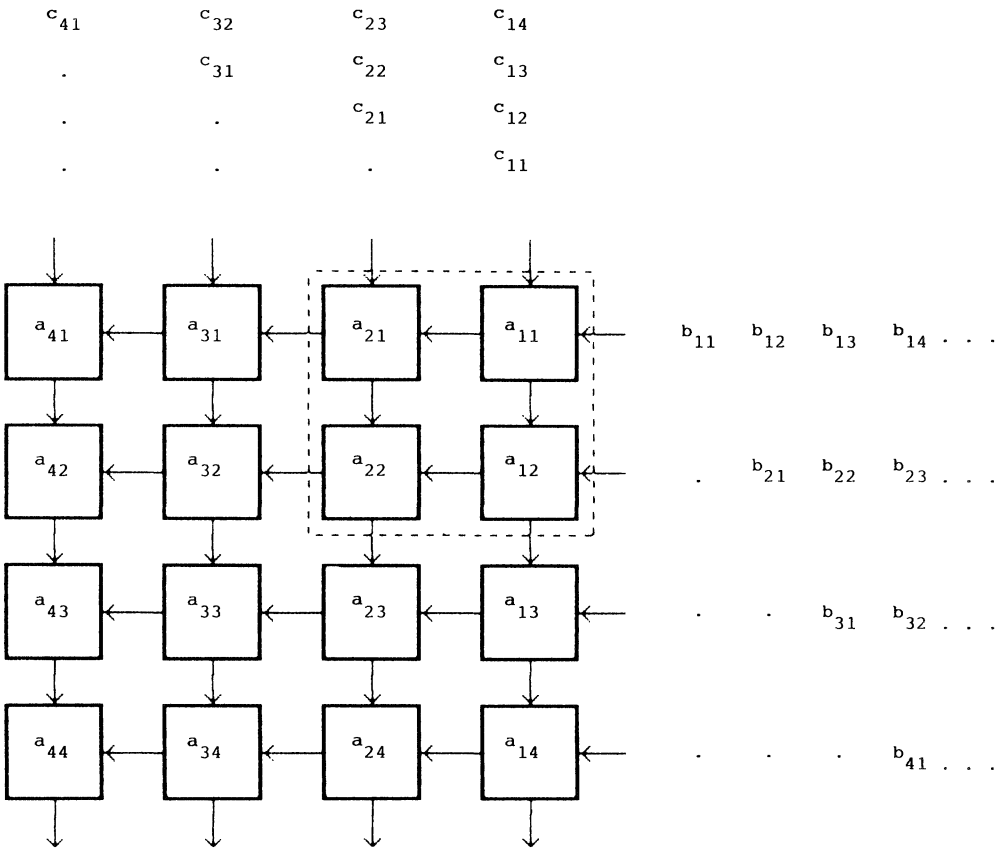


FIG. 3. *A matrix multiplication array*: $C \leftarrow C + AB$.

output from another part of the array whose action has already been emulated. This sort of decomposition of an array can work only when there are no cycles in the flow of data in the array. This is true of the decomposable arrays mentioned above.

The SVD array of § 2 is indecomposable. Consider Fig. 2. Suppose a two-processor array is available. It cannot efficiently simulate the four-processor array because there does not exist a two-processor segment of Fig. 2 for which only known data enters. If this diagram is cut by a vertical line, data flows across the line in both directions, every cycle. The data cannot be known if only the computations on one side of the line have been performed.

Here we shall present a solution to this problem. The idea is to have a given p-processor array simulate a pq-processor "superarray" which is not of the Brent–Luk type. Moreover, the superarray is decomposable. In its space-time dataflow graph, the processors occur in groups of p. For long periods of either p or $2p-1$ cycles, no data flows between groups. Thus, the physical array can efficiently carry out the computation of the superarray, group-by-group.

We give two such superarrays. The first implements a Hestenes method in which a "sweep" corresponds to a permutation of a multiset of off-diagonal pairs. There is some redundancy: some pairs are generated and orthogonalized several times. The second implements a cyclic Hestenes method with a permutation different from the one used in an A-sweep. For this method, a minor change must be made to the array.

We have compared these new sweeps to the A-sweep. These experiments indicate that the first superarray is about 20–60% less efficient than the Brent–Luk array, while the second superarray is virtually equal to the Brent–Luk array in efficiency.

3.1. Method A. This method is easiest to explain in terms of an example. Suppose we have a 4 processor array. Suppose there are 16 columns in A. We proceed as follows:

1. Load columns 1–8 and perform an A-sweep;
2. Load columns 9–16 and perform an A-sweep;
3. Load columns 1–4, 13–16; perform an A-sweep;
4. Load columns 5–8, 9–12; perform an A-sweep;
5. Load columns 1–4, 9–12; perform an A-sweep;
6. Load columns 13–16, 5–8; perform an A-sweep.

Steps 1–6 together constitute an A-supersweep. During an A-supersweep, every column pair is orthogonalized. Some are orthogonalized more than once.

To describe the general case, suppose there is a p-processor array, and $n = 2pq$ (pad A with zero columns, if necessary, so that $2p$ divides n). Imagine that the matrix A consists of $2q$ supercolumns: supercolumn A_i consists of columns

$$p(i-1)+1, \cdots, pi.$$

Now consider a q-superprocessor virtual superarray. Each superprocessor holds two supercolumns (one in each of its left and right memories). In one supercycle the superprocessors each perform a single A-sweep over the $2p$ columns in their memory.

(Obviously we can simulate a supercycle of a superprocessor using one p-processor Brent–Luk array and $2p-1$ cycles of time. Moreover, we can be loading the data for the next supercycle and unloading the data from the preceding supercycle at the same time as we process the data for the current supercycle.)

Initially, supercolumns A_1 and A_2 are in superprocessor 1, A_3 and A_4 in superprocessor 2, etc.

Between supercycles, the supercolumns move to neighboring superprocessors. The scheme for moving supercolumns is precisely the same as the scheme for moving ordinary columns in a q-processor Brent–Luk array.

After $2q-1$ supercycles, we have performed an A-sweep on every pair of supercolumns exactly once. Together these $2q-1$ supercycles constitute an A-supersweep. During an A-supersweep, every pair of columns of A is orthogonalized. If two columns are in different supercolumns, then they are orthogonalized once, during the supercycle in which their containing supercolumns occupy the same superprocessor. If they are in the same supercolumn, then they are orthogonalized $2q-1$ times.

In units of cycles, the time for an A-supersweep, T_{AS}, is

$$\begin{aligned} T_{AS} &= (2q-1) \text{ supercycles} * (2p-1) \text{ cycles/supercycle} \\ &= (2q-1)(2p-1). \end{aligned}$$

(Of course, the simulation by a p-processor array takes q times this time.) The time for an A-sweep over n columns, T_A, is

$$T_A = n-1 = 2pq-1.$$

Thus, the A-supersweep takes longer; the ratio of times satisfies

$$\frac{9}{7} \leqq \frac{T_{AS}}{T_A} < 2.$$

(The lower bound arises in the simplest nontrivial case $p = q = 2$.)

There is little theoretical basis for comparing the effectiveness of A-supersweeps and A-sweeps in reducing the nonorthogonality of the columns of A. We have therefore performed an experiment. A set of square matrices A whose elements were random and uniformly distributed in $[-1, 1]$ was generated. Both A-supersweeps and A-sweeps were used until the sum-of-squares of the off-diagonal elements of A^TA was reduced to 10^{-12} times its initial value. We show the results in Table 1. The number of test matrices, the average number of sweeps, the largest number for any test matrix, and the relative time

$$\rho \equiv T_{AS} * \text{average-sweeps } (AS) / T_A * \text{average-sweeps } (A)$$

are shown.

Evidently one A-supersweep is more effective in reducing nonorthogonality than one A-sweep. This is not surprising, since more orthogonalizations are performed. Their cost-effectiveness, however, is roughly 20–60% less.

TABLE 1
Comparison of A-sweeps and A-supersweeps.

				Average		Maximum		
p	q	n	Trials	A super	A	A super	A	ρ
2	2	8	320	3.98	4.33	5	5	1.18
2	4	16	160	5.10	5.38	6	7	1.33
2	8	32	80	6.18	6.29	7	7	1.43
4	2	16	160	4.80	5.40	5	6	1.24
4	4	32	80	5.99	6.31	7	7	1.50
4	8	64	20	7.05	7.55	8	8	1.57
8	2	32	80	5.25	6.28	6	7	1.21
8	4	64	10	6.60	7.60	7	8	1.45
16	2	64	20	6.00	7.30	6	8	1.21

In order to gauge the reliability of the statistics generated by this experiment, we also measured the standard deviations of the sampled data. In all cases, the standard deviations were less than 0.5. For the samples of size 80 or more, the standard errors of the means are no more than 0.06, so these statistics are quite reliable. For the samples of sizes 20 and 10, these data may be in error by as much as 10%.

3.2. Method B. Method A suffers some loss of speed, because in an A-supersweep some column-pairs are orthogonalized many times. By making a small modification to the Brent–Luk array and using the new array as our basic tool, we can simulate a new supersweep, called an AB-supersweep, during which every column-pair is orthogonalized exactly once.

Figure 3 shows the modified array. The connection from processor 1 to processor p is new. Note that a ring connected set of processors can easily simulate this structure. This array is still able to perform A-sweeps over sets of $2p$ columns. But it can also perform a second type of sweep, which we call an "AB-sweep," and which we now describe.

In an AB-sweep, a pair (A, B) of supercolumns, each consisting of p columns, is loaded into the array. During the sweep, all pairs (a, b) $a \in A$, $b \in B$ are orthogonalized exactly once. But no pairs from $A \times A$ or $B \times B$ are orthogonalized.

To implement an AB-sweep, place the columns of A in the p left memories and the columns of B in the p right memories of the processors. (The set of left (respectively

right) processor memories is the superprocessor's left (respectively right) memory, rather than the memories of the leftmost (respectively rightmost) $p/2$ processors.) Processors do precisely what they did before: orthogonalize their two columns. Between cycles, A remains stationary while B rotates one position, using the connections shown as solid lines in Fig. 4.

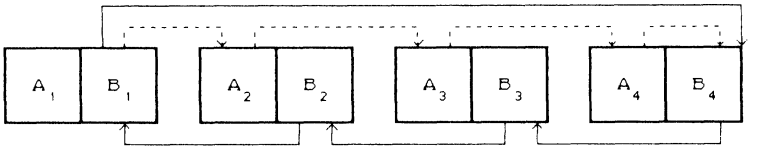


FIG. 4. *The modified* SVD *array*; $n = 8$.

An AB-supersweep is as follows. Again we work with $2q$ supercolumns of p columns each. The initial configuration is as for an A-supersweep. During the first supercycle, which takes $2p-1$ cycles, every superprocessor performs an A-sweep on the $2p$ columns in its memory. On subsequent supercycles, all superprocessors perform AB-sweeps, where the sets A and B are the two supercolumns in its memory. Between supercycles, supercolumns move as before.

It is easy to see that in an AB-supersweep, every column pair is orthogonalized once. Thus this scheme implements a true cyclic Hestenes method. The permutation differs, nevertheless, from the permutation used in an A-sweep.

Again, we have compared the new scheme to the A-sweep by an experiment. The experiment setup was precisely the same as for the previous experiment. The results are shown in Table 2.

TABLE 2
Comparison of A-sweeps and AB-supersweeps.

				Averages		Maxima	
p	q	n	Trials	AB super	A	AB super	A
2	2	8	320	4.32	4.33	5	5
2	4	16	160	5.35	5.38	6	7
2	8	32	80	6.36	6.29	7	7
4	2	16	160	5.36	5.40	6	6
4	4	32	80	6.18	6.31	7	7
4	8	64	20	7.50	7.55	8	8
8	2	32	80	6.13	6.28	7	7
8	4	64	10	7.10	7.60	8	8
16	2	64	20	7.00	7.30	7	8

Evidently, AB-supersweeps are as effective as A-sweeps. The standard deviations of the number of AB-supersweeps needed were also all less than 0.5.

3.3. Earlier work. Another scheme for solving problems with an undersized array was proposed in [3], a paper that deals with a square SVD array. The proposal is to use a block method, in which the SVDs of diagonal blocks are computed in the given array.

Applied to the linear SVD array, this idea is much like our A-supersweep scheme, except that a superprocessor iterates to convergence instead of performing only one A-sweep.

TABLE 3
Comparison of block-Jacobi A-sweeps and A-supersweeps.

p	q	n	Trials	Ratio
2	2	8	40	2.4
2	4	16	40	2.7
2	8	32	10	3.1
4	2	16	40	2.9
4	4	32	10	4.2
4	8	64	5	4.3
8	2	32	10	3.7
8	4	64	5	5.2
16	2	64	5	3.7

The extra time needed for this convergence leads to inefficiency in this scheme. The data in Table 3 show this. These give the result of a numerical experiment; the setup was the same as for previous experiments. For several values of p and q we show the ratio of the number of operations used by the block method to the number used by the AB-supersweep method discussed earlier. For this experiment, we stopped a superprocessor, which was working on the $n \times 2p$ matrix B, from further iteration when the sum of the squares of the off-diagonal elements of B^TB was less than 10^{-12} times the sum of the squares of the diagonal elements of B^TB.

Thus, the present schemes require fewer computations than the block method of [3]. On the other hand, they may require more input/output from the array. The choice between them will turn on factors such as the relative speed of computation and input/output that depend on how the array is implemented.

Clearly there is a family of methods of this type parameterized by the number of sweeps over each block. Our experiments illuminate two extreme cases.

4. The eigenvalue array. In this section we show how the two ideas for problem decomposition in § 3 can be used to solve large eigenvalue problems on the square array of [1]. This array solves the symmetric eigenvalue problem in time $O(n \log n)$ and is the fastest array known for that problem. The same ideas also apply, in exactly the same way, to the SVD array of [3].

The eigenvalue array is a $p \times p$ array that holds a $2p \times 2p$ symmetric matrix. Each processor p_{ij} holds a 2×2 submatrix b_{ij}: initially,

$$b_{ij} = \begin{bmatrix} a_{2i-1,2j-1} & a_{2i-1,2j} \\ a_{2i,2j-1} & a_{2i,2} \end{bmatrix}.$$

At each cycle, each diagonal processor p_{ii} generates a plane rotation r_i such that $r_i b_{ii} r_i^T$ is diagonal. The rotations are then sent from the diagonal processor to all processors in the same row and the same column. The off-diagonal processor p_{ij} (if $i<j$) on receiving rotations r_i and r_j, computes a new block $b_{ij} = r_i b_{ij} r_i^T$. After the rotations have been applied by the off-diagonal processors, columns and rows are interchanged. Adjacent processors exchange data with their neighbors to the right and left to permute the matrix columns as in the SVD array (Fig. 1). Then processors exchange data with the processors above and below to permute the matrix rows in the same way. After $2p-1$ cycles, all off-diagonal elements have been annihilated once: this is one sweep.

It is not necessary to broadcast rotations to an entire row or column of the array. Instead, rotations move through one processor per cycle. Thus, if rotations are generated at time $t=0$, they are applied at time t by the processors $p_{i,i\pm t}$ in diagonal t. Likewise,

the process of exchanging data occurs in a wave. It begins with exchanges between the diagonal processors and those in diagonals ± 1 at time 2. At time $t+1$, $0 < t < p-1$, processors of diagonals $\pm t$ and $\pm(t+1)$ exchange data. The second set of rotations is generated at the diagonal at time 3, the next at time 6, etc. Thus, the last rotations are generated at time $6p-6$, and the whole sweep is finished at time $7p-7$.

The decomposition of this array works as follows. Partition the given $n \times n$ matrix A as

$$\begin{matrix} A_{11} & A_{12} & \cdots & A_{1,2q} \\ A_{21} & A_{22} & \cdots & A_{2,2q} \\ \vdots & & & \vdots \\ A_{2q,1} & A_{2q,2} & \cdots & A_{2q,2q} \end{matrix}.$$

We imagine a virtual $q \times q$ superarray with superprocessors P_{ij} that hold a 2×2 block matrix B_{ij} having $p \times p$ blocks: initially,

$$B_{ij} = \begin{bmatrix} A_{2i-1,2j-1} & A_{2i-1,2j} \\ A_{2i,2j-1} & A_{2i,2j} \end{bmatrix}.$$

At each supercycle, each diagonal superprocessor P_{ii} generates an orthogonal product of plane rotations R_i by performing one sweep of the Jacobi method with the parallel order as described in the last paragraph. The rotations are then sent from the diagonal

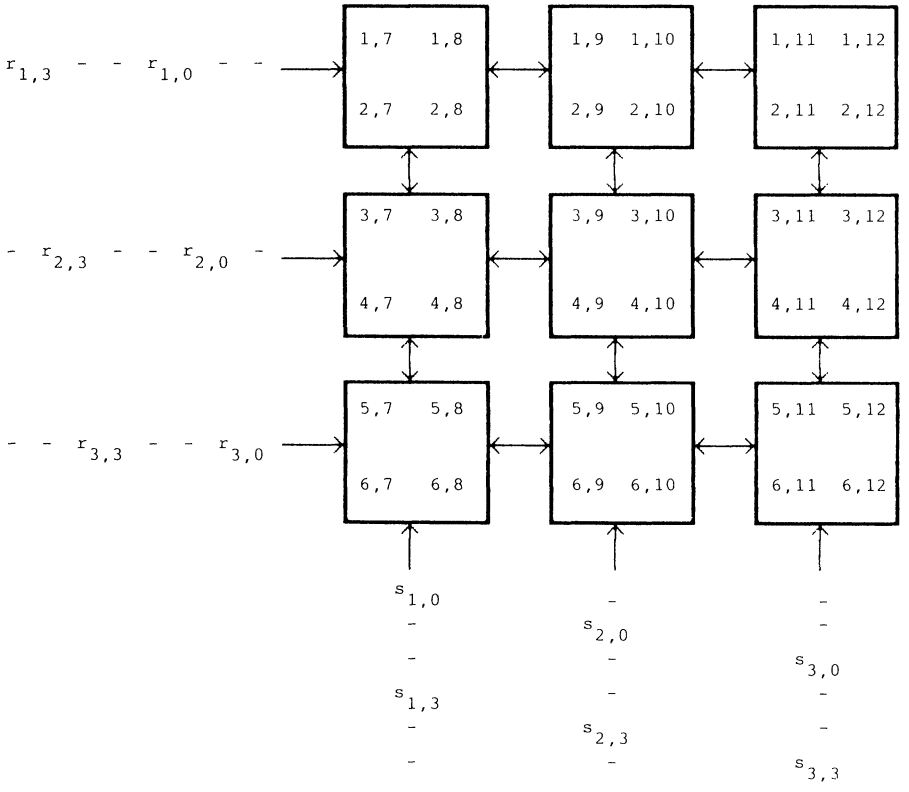


FIG. 5. *Operation of an off-diagonal superprocessor.*

superprocessor to all superprocessors in the same row and the same column. The off-diagonal superprocessor P_{ij} (if $i<j$), on receiving rotation-products R_i and R_j, computes a new block $B_{ij} = R_i B_{ij} R_j^T$. After the rotations have been applied by the off-diagonal processors, block columns and rows are interchanged. Adjacent superprocessors exchange data with their neighbors to the right and left to permute the block columns as in the SVD (Fig. 1). Then superprocessors exchange data with the superprocessors above and below to permute the block rows in the same way.

A single square $p \times p$ eigenvalue array can emulate this superarray efficiently. Indeed, the diagonal superprocessor is a $p \times p$ eigenvalue array. We assume that the rotations generated by this array flow out at the edges and can be stored for later use. We need only show that the off-diagonal superprocessors can be emulated. For this to work, the p diagonal processors must change their roles, becoming ordinary off-diagonal processors. We assume an off-diagonal block B_{ij} is loaded into the array. The rotations that make up R_i and R_j are sent into the array at its left and bottom edges. The individual plane rotations are sent into the array at the same relative places and times as when they left the array after being generated. They flow through the array and are applied to the matrix elements as in the eigenvalue array. Interchange of rows and columns begins at the lower left corner of the array and moves in a wave toward the upper right corner.

Figure 5 illustrates this. The orthogonal matrix R_i is a product of plane rotations that we denote by $r_{k,t}$, where $r_{k,t}$ is the rotation generated by processor p_{kk} at time $3t$ while the array was emulating the diagonal superprocessor P_{ii}. We denote by $s_{k,t}$ the constituent rotation of R_j that was generated by p_{kk} at time $3t$ while the array was emulating the diagonal superprocessor P_{jj}. Thus, $1 \leqq k \leqq p$ and $0 \leqq t \leqq 2p-2$.

Acknowledgment. This work was done in Stockholm while I was a guest of the Royal Institute of Technology, Department of Numerical Analysis and Computer Science, and of Uppsala University, Department of Computer Science. I thank Charles Van Loan, Erik Tidén and Björn Lisper for their comments.

REFERENCES

[1] R. P. BRENT AND F. T. LUK, *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays*, this Journal, 6 (1985), pp. 69-84.

[2] ———, *A systolic architecture for the singular value decomposition*, Technical Report TR-CS-82-09, Dept. Computer Science, The Australian National University, Canberra, 1982.

[3] R. P. BRENT, F. T. LUK AND C. VAN LOAN, *Computation of the singular value decomposition using mesh-connected processors*, Technical Report TR 82-528, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1983; J. VLSI and Computer Systems, to appear.

[4] ALAN M. FINN, FRANKLIN T. LUK AND CHRISTOPHER POTTLE, *Systolic array computation of the singular value decomposition*, in Real-Time Signal Processing V, Joel Trimble, ed., SPIE, 341 (1982), pp. 35-43.

[5] W. M. GENTLEMAN AND H. T. KUNG, *Matrix triangularization by systolic array*, in Real-Time Signal Processing IV, Tian F. Tao, ed., SPIE, 298 (1981), pp. 19-26.

[6] DON E. HELLER AND ILSE C. F. IPSEN, *Systolic networks for orthogonal decompositions*, this Journal, 4 (1983), pp. 261-269.

[7] H. T. KUNG AND C. E. LEISERSON, *Systolic arrays (for VLSI)*, in Sparse Matrix Proceedings, I. S. Duff and G. W. Stewart, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1978, pp. 256-282. A slightly different version appears in § 8.3 of Introduction to VLSI Systems by C. A. Mead and L. A. Conway, Addison-Wesley, Reading, MA, 1980.

[8] ROBERT SCHREIBER, *Systolic arrays for eigenvalue computation*, in Real-Time Signal Processing V, Joel Trimble, ed., SPIE, 341 (1982), pp. 27-34.

[9] ROBERT SCHREIBER AND PHILIP J. KUEKES, *Systolic linear algebra machines in digital signal processing*, in VLSI and Modern Signal Processing, S. Y. Kung, H. Whitehouse and T. Kailath, eds., Prentice-Hall, Englewood Cliffs, NJ, 1984, pp. 389–405.

[10] R. SCHREIBER, *A systolic architecture for singular value decomposition*, in Proc. 1er Colloque International sur les Méthodes Vectorielles et Paralèles en Calcul Scientifique, Electricité de France Bulletin de la Direction des Etudes at Recherches, Serie C, No. 1, 1983, pp. 143–148.

[11] ———, *On the systolic array of Brent, Luk and Van Loan*, in Real-Time Signal Processing VI, Keith Bromley, ed., SPIE, 431 (1983), pp. 72–77.

[12] ———, *Systolic arrays: high performance parallel machines for matrix computation*, in Elliptic Problem Solvers II, Garrett Birkhoff and Arthur Schoenstadt, eds., Academic Press, New York, 1984, pp. 187–194.

[13] JEFFREY M. SPEISER AND HARPER J. WHITEHOUSE, *Parallel processing algorithms and architectures for real-time signal processing*, in Real-Time Signal Processing IV, Tian F. Tao, ed., SPIE, 298 (1981), pp. 2–9.

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.

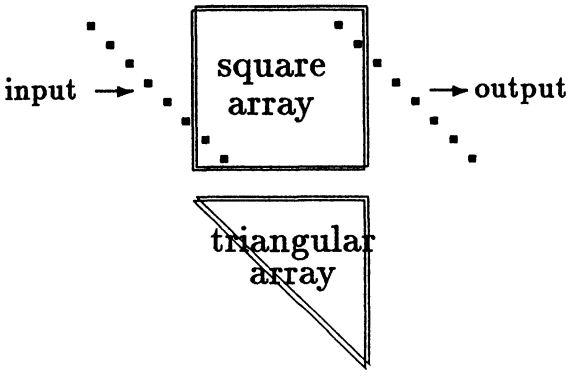


FIG. 1. *Overview* SVD *updating array.*

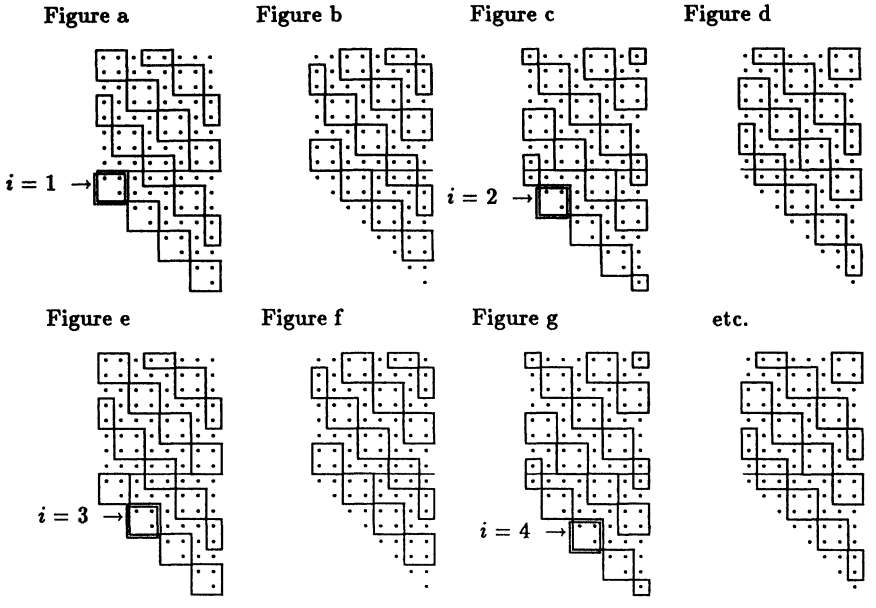


FIG. 2. SVD *array.*

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

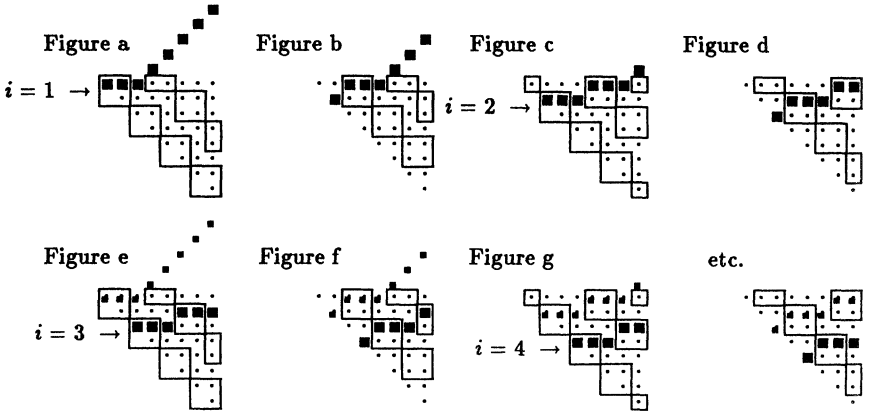


FIG. 3. *Modified Gentleman–Kung array.*

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

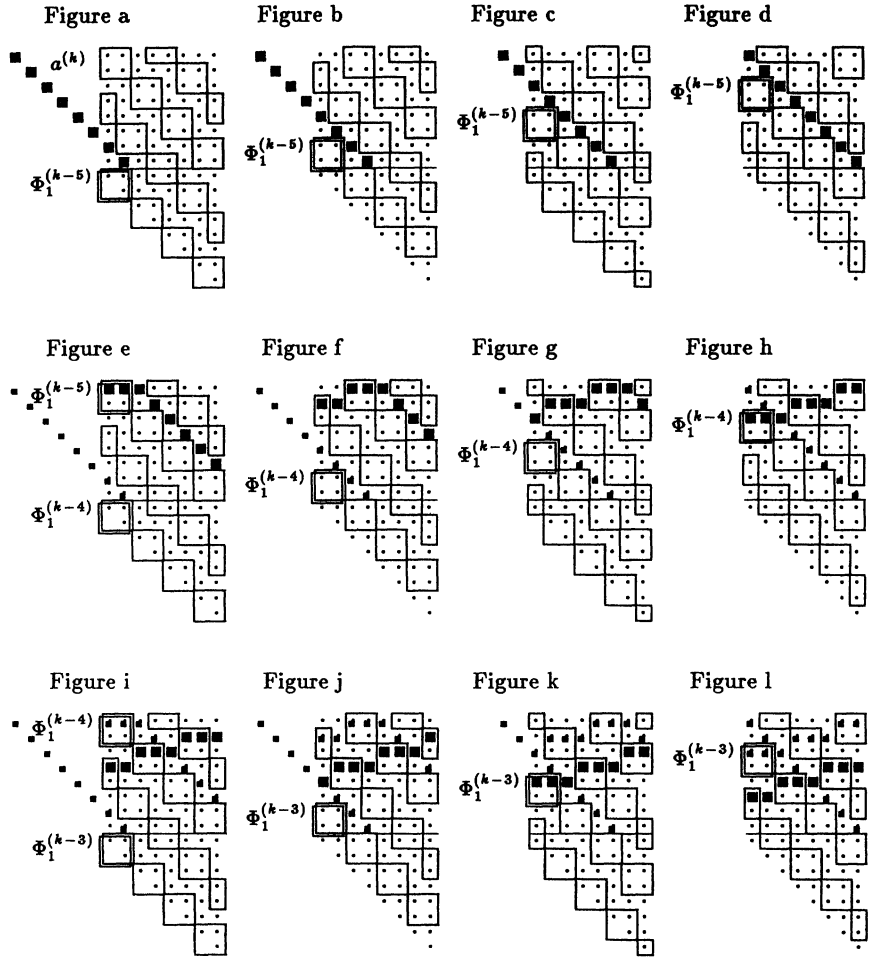


FIG. 4. SVD *updating array.*

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

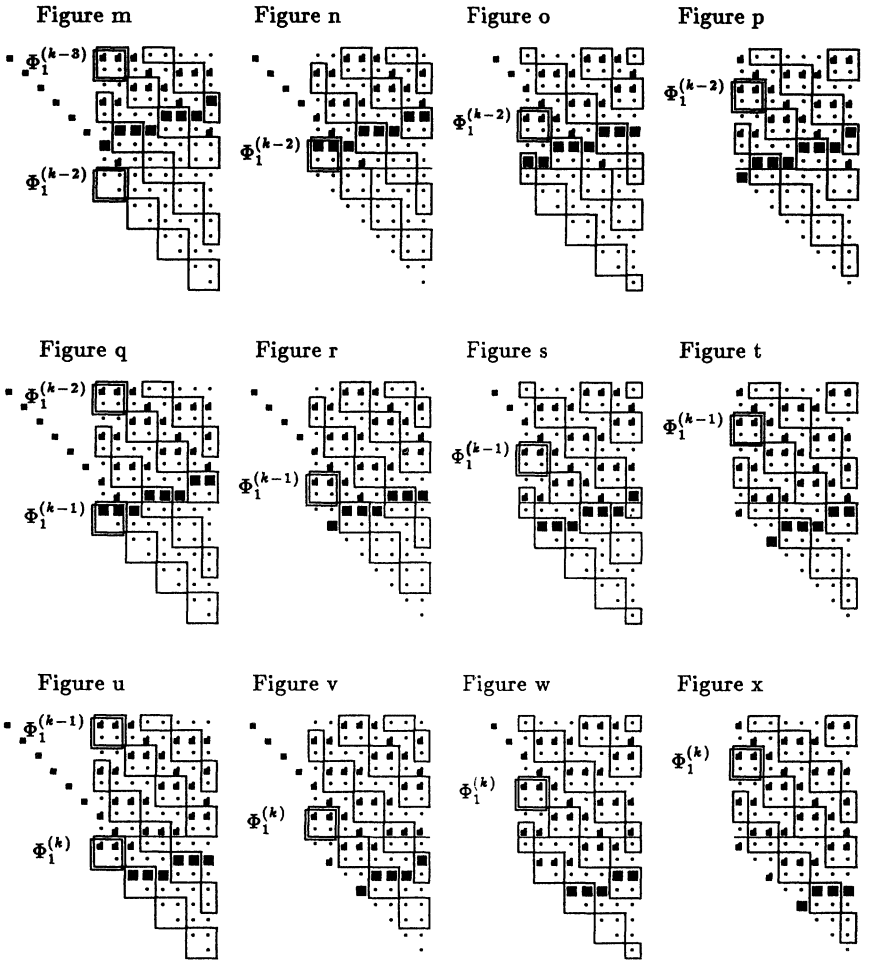


FIG. 4 (*continued*).

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 $O(n^0)$.

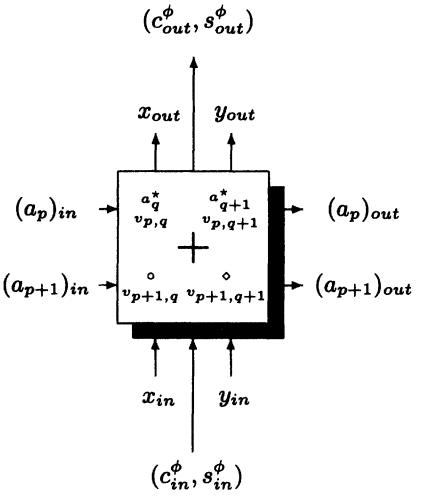
Let us now focus on the transformations in $\Phi^{(\natural)}$ and the way these can be processed. One of these transformations is, e.g., $\Phi_1^{(k-5)}$ (see §2 for notation), which is computed on the main diagonal in Fig. 4(a) (double frame). While propagating downwards, the $a_\star^{(\natural)^T}$-vector crosses the upgoing rotation $\Phi_1^{(k-5)}$ in Fig. 4(e). At this point, this transformation can straightforwardly be applied to the available $a_\star^{(\natural)^T}$-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 $a_\star^{(\natural)}$-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 $a_\star^{(\natural)}$-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

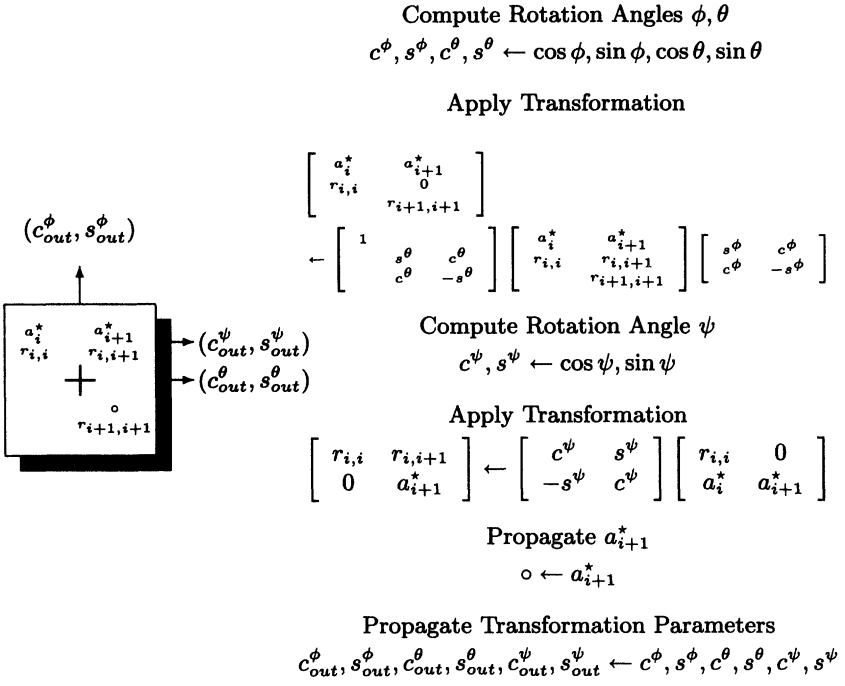
DISPLAY 1. *Internal cell V-matrix.*

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

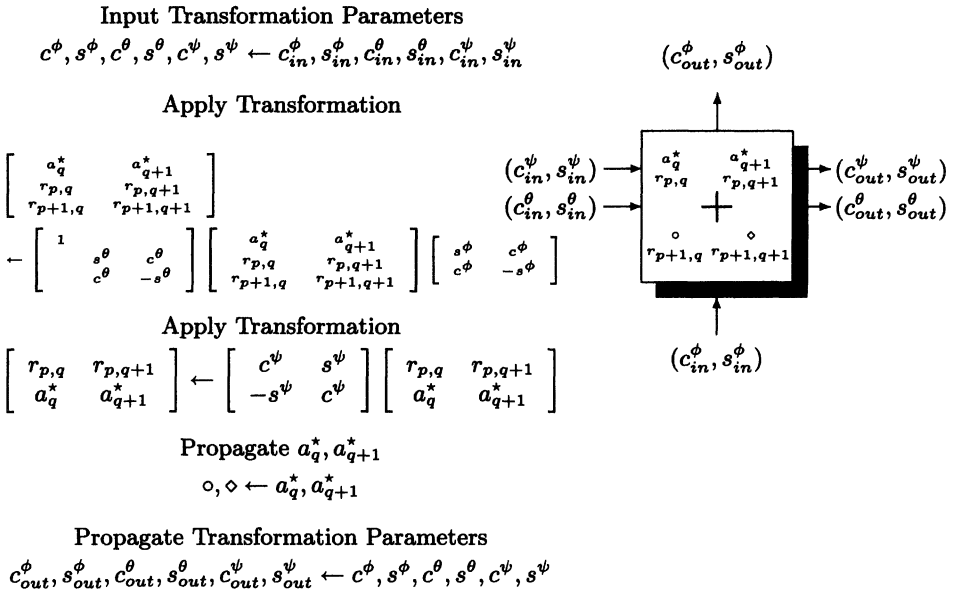


DISPLAY 2. *Boundary cell R-factor.*

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

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

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

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

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

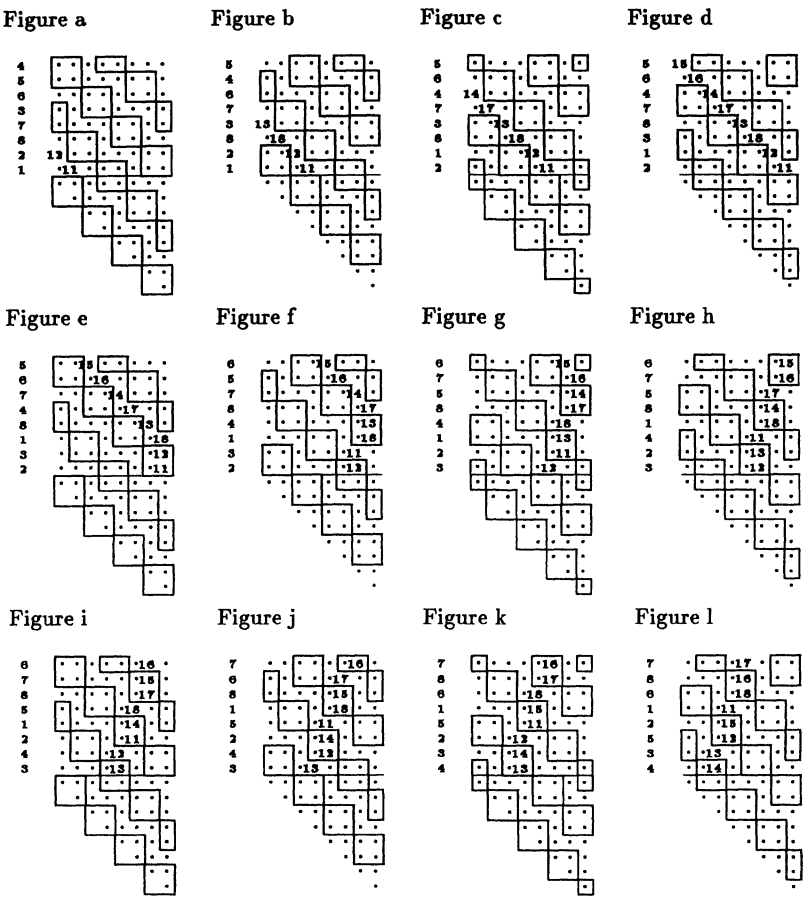


FIG. 5. *Re-orthogonalizations.*

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

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 $v_2 \cdot v_1^\star$. 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

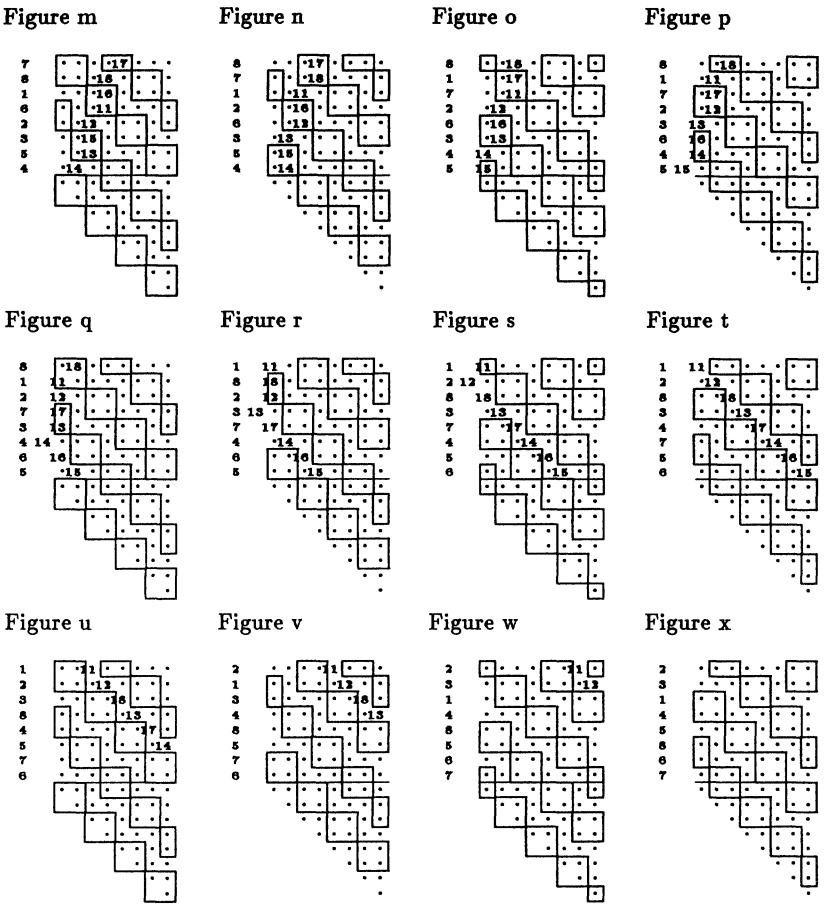


FIG. 5 (*continued*).

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$\sqrt{\cdot}$		4$\sqrt{\cdot}$

$p \neq q$ 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

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

with scale factor updating (notice the implicit row permutation)

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

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

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:

$$\begin{aligned}
&\begin{bmatrix} \sqrt{d^{row^*}_i} & 0 \\ 0 & \sqrt{d^{row^*}_{i+1}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}^{\circ}_{i,i} & \bar{r}^{\circ}_{i,i+1} \\ \bar{r}^{\circ}_{i+1,i} & \bar{r}^{\circ}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d^{col}_i} & 0 \\ 0 & \sqrt{d^{col}_{i+1}} \end{bmatrix} \cdot \begin{bmatrix} -\sin\phi & \cos\phi \\ \cos\phi & \sin\phi \end{bmatrix} \\
&= \begin{bmatrix} \sqrt{d^{row^*}_i} & 0 \\ 0 & \sqrt{d^{row^*}_{i+1}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}^{\circ}_{i,i} & \bar{r}^{\circ}_{i,i+1} \\ \bar{r}^{\circ}_{i+1,i} & \bar{r}^{\circ}_{i+1,i+1} \end{bmatrix} \\
&\qquad \cdot \begin{bmatrix} -\tan\phi\sqrt{\frac{d^{col}_i}{d^{col}_{i+1}}} & 1 \\ 1 & \tan\phi\sqrt{\frac{d^{col}_{i+1}}{d^{col}_i}} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d^{col}_{i+1}}\cos\phi & 0 \\ 0 & \sqrt{d^{col}_i}\cos\phi \end{bmatrix} \\
&= \begin{bmatrix} \sqrt{d^{row^*}_i} & 0 \\ 0 & \sqrt{d^{row^*}_{i+1}} \end{bmatrix} \cdot \begin{bmatrix} \bar{r}^{*}_{i,i} & \bar{r}^{*}_{i,i+1} \\ 0 & \bar{r}^{*}_{i+1,i+1} \end{bmatrix} \cdot \begin{bmatrix} \sqrt{d^{col^*}_i} & 0 \\ 0 & \sqrt{d^{col^*}_{i+1}} \end{bmatrix},
\end{aligned}$$

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

$$\begin{aligned}
d^{col^*}_i &= d^{col}_{i+1}\cos^2\phi, \\
d^{col^*}_{i+1} &= d^{col}_{i}\cos^2\phi,
\end{aligned}$$

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

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

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

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

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

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.

A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems

SUN-YUAN KUNG, MEMBER, IEEE, AND YU HEN HU, STUDENT MEMBER, IEEE

Abstract—**The design of VLSI parallel processors requires a fundamental understanding of the parallel computing algorithm and an appreciation of the implementational constraint on communications. Based on such consideration, this paper develops a highly concurrent Toeplitz system solver, featuring maximum parallelism and localized communication. More precisely, a highly parallel algorithm is proposed which achieves $O(N)$ computing time with a linear array of $O(N)$ processors. This compares very favorably to the $O(N \log_2 N)$ computing time attainable with the traditional Levinson algorithm implemented in parallel. Furthermore, to comply with the communication constraint, a pipelined processor architecture is proposed which uses only localized interconnections and yet retains the maximum parallelism attainable.**

I. Introduction

With rapidly growing microelectronics technology leading the way, modern signal processor architectures are undergoing a major revolution. The availability of low cost, fast VLSI (very large scale integration) devices promises the practice of cost-effective, high-speed parallel processing of large volumes of data. This will make possible an ultra-high throughput rate and, therefore, indicates a major technological breakthrough for real-time signal processing applications. On the other hand, it has become more critical than ever to have a fundamental understanding of the algorithm structure, architecture, and implementation constraints in order to realize the full potential of VLSI computing power [1]. In this paper, the two most critical issues—the parallel computing algorithm and the VLSI architectural constraint—will be considered.

Traditionally, computational complexity is measured in terms of number of the arithmetic operations required in an algorithm. With the emergence of inexpensive (VLSI) parallel computing capability, this criterion will soon become obsolete since the most efficient algorithms for sequential machines are not necessarily the most efficient for parallel machines. Taking into account the parallelism, an up to date and more practical criterion appears to be the processing throughput rate attainable in parallel computation [2], [3]. Therefore, in formulating a parallel algorithm, the first important question should be: *How can we structure the algorithm to achieve the maximum parallelism and, therefore, the maximum throughput rate?*

Communication constraint represents another fundamental impact of VLSI device technology on the architecture design and implementation consideration. In VLSI systems, communication tends to be very restrictive as it accounts for most time, area, and energy consumption. Therefore, architectures which require only localized communication, such as a *systolic array* [4] and a *wavefront array processor* [5]-[7], become very attractive for VLSI implementation. Thus, the second important question should be: *How can we cope with the communication constraint so as to compromise least in processing throughput rate?*

Obviously, the above two questions are mutually dependent, and their answers often affect each other iteratively throughout the design process. Therefore, they deserve an integrated solution which should provide general guidelines for designing VLSI parallel processing architectures. The motivation of this paper, therefore, is to demonstrate a design methodology following these general guidelines. Nevertheless, we shall limit ourselves to the specific task of solving Toeplitz systems [8] which, we consider, is one of the most important signal processing problems.

A Toeplitz system is a set of linear system equations

$$\mathbf{Tx} = \mathbf{y} \tag{1.1}$$

with T being a Toeplitz matrix, i.e., the (i, j)th element $t_{ij} = t_{i-j} = t_k$, $-N \leqslant k \leqslant N$. (Throughout this paper, we assume that T is an $(N+1) \times (N+1)$ real matrix.) This system arises in numerous widespread applications ranging from speech, image, and neurophysics to radar, sonar, geophysics, and astronomical signal processing [9]-[11]. The contribution of this paper lies in the development of a highly concurrent algorithm and pipelined architecture which is able to solve a Toeplitz system in $O(N)$ processing time in an array processor as opposed to $O(N^3)$ for the general (sequential) Gauss elimination procedure or $O(N^2)$ for the (sequential) Levinson algorithm (cf. Section II). In addition, the design methodology demonstrated in this paper should also help answer some fundamental problems faced in designing VLSI parallel processor architectures.

The organization of this paper is as follows. In Section II, a conventional (Levinson) algorithm for solving a Toeplitz system and its inherent limitation for parallel processing are examined. In Section III, a highly concurrent algorithm is developed. In Section IV, to naturally map this algorithm onto a parallel computing system, a lattice-connected processors array is proposed. The complete Toeplitz system solver is discussed in Section V, and lastly, some system applications, implementations, and multichannel extensions are included in Section VI.

II. Preliminary Review

It is known that the conventional approach (e.g., Gauss elimination procedure) for solving a linear system takes $O(N^3)$ arithmetic operations with each operation containing one mul-

Manuscript received March 5, 1982; revised August 3, 1982. This work was supported in part by the Office of Naval Research under Contract N00014-80-C-0457, N00014-81-K-0191 and by the National Science Foundation under Grant ECS-80-16581.

The authors are with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089.

Reprinted from *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 66-76, February 1983.

tiplication and one addition. By making use of the Toeplitz structure, several fast algorithms are now available to solve a Toeplitz system in $O(N^2)$ operations or even less [12]-[17]. Among them, the most popular is the Levinson algorithm [12].

The Levinson algorithm was originally proposed by Norman Levinson in 1947, and since then, a number of variants have been proposed [18]-[23]. Basically, the Levinson algorithm recursively solves for the (kth-order) solution $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ in the equation depicted below:

$$T_k \begin{bmatrix} 1 & b_{kk} \\ a_{1k} & \vdots \\ \vdots & b_{1k} \\ a_{kk} & 1 \end{bmatrix} = \begin{bmatrix} E_k & 0 \\ 0 & \vdots \\ \vdots & 0 \\ 0 & E_k \end{bmatrix} = \begin{bmatrix} E_k & \mathbf{0}_k \\ \mathbf{0}_k & E_k \end{bmatrix} \qquad (2.1)$$

$$\boldsymbol{a}_k \quad \boldsymbol{b}_k$$

where T_k denotes the $(k+1) \times (k+1)$ leading principal minor[1] of T and E_k is some nonzero number. The recursive procedure efficiently utilizes $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ to derive the solution for the $(k+1)$th-order equation. Then, by induction, the Nth-order equation can be solved. This recursive procedure can be explained in three steps.

Step 1: Let

$$T_{k+1} \begin{bmatrix} \boldsymbol{a}_k & 0 \\ 0 & \boldsymbol{b}_k \end{bmatrix} = \begin{bmatrix} E_k & S_k \\ \mathbf{0}_k & \mathbf{0}_k \\ Q_k & E_k \end{bmatrix} \qquad (2.2)$$

where Q_k and S_k are obtained via *inner product operation:*

$$Q_k = [t_{k+1}, \cdots, t_1]\, \boldsymbol{a}_k \qquad (2.3a)$$

$$S_k = [t_{-k-1}, \cdots, t_{-1}]\, \boldsymbol{b}_k. \qquad (2.3b)$$

Step 2: $\{\boldsymbol{a}_{k+1}, \boldsymbol{b}_{k+1}\}$ are derived as

$$[\boldsymbol{a}_{k+1} \quad \boldsymbol{b}_{k+1}] = \begin{bmatrix} \boldsymbol{a}_k & 0 \\ 0 & \boldsymbol{b}_k \end{bmatrix} \begin{bmatrix} 1 & Kr^{(k+1)} \\ Ke^{(k+1)} & 1 \end{bmatrix} \qquad (2.4)$$

where[2]

$$Kr^{(k+1)} = -S_k/E_k \quad \text{and} \quad Ke^{(k+1)} = -Q_k/E_k. \qquad (2.5)$$

Step 3: Consequently, we have

$$T_{k+1}[\ \mathrm{a}_{k+1} \quad \mathrm{b}_{k+1}\ ] = \begin{bmatrix} E_{k+1} & 0_k \\ 0_k & E_{k+1} \end{bmatrix} \qquad (2.6)$$

where

$$E_{k+1} = E_k + Ke^{(k+1)} S_k = E_k + Kr^{(k+1)} Q_k$$
$$= E_k(1 - Ke^{(k+1)} Kr^{(k+1)}) \qquad (2.7)$$

and we are ready for the next recursion.

The computation should stop at the end of the Nth recursion. A simple calculation shows that the total number of operations required is approximately $2N^2$.

[1] Throughout this paper, we shall assume that each leading principle minor of the matrix T is nonsingular.

[2] In the literature, $\{Ke, Kr\}$ are called "reflection coefficients"; see, e.g., [13].

In the case of a symmetric Toeplitz system, by symmetry, we have $Ke^{(i)} = Kr^{(i)}$ and that $a_{ik} = b_{ik}$ [cf. (2.1)], and therefore the number of operations can be reduced to one half, that is, N^2.

To explicitly solve the Toeplitz system, we note that the $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ vectors produced in the Levinson algorithm constitute a *UDL* decomposition of the T^{-1} matrix, namely,

$$T^{-1} = \begin{bmatrix} 1 & b_{11} & \cdots & b_{NN} \\ 0 & 1 & & \\ & & \ddots & \\ & & & 1 \quad b_{1N} \\ 0 & & 0 & 1 \end{bmatrix} \begin{bmatrix} E_0 & & & 0 \\ & E_1 & & \\ & & \ddots & \\ 0 & & & E_N \end{bmatrix}^{-1}$$
$$\cdot \begin{bmatrix} 1 & 0 & & 0 \\ a_{11} & 1 & & \\ & & \ddots & \\ a_{NN} & \cdots & a_{1N} & 1 \end{bmatrix} \qquad (2.8)$$
$$\triangleq U_B D_E L_A.$$

Therefore, the solution of the Toeplitz system can be computed as $\boldsymbol{x} = U_B D_E L_A \boldsymbol{y}$.

In order to meet the extremely high throughput rate requirement in many real-time signal processing applications, the potential of parallel computing has to be utilized. However, for parallel execution of the Levinson algorithm on a linear processor array with N processing elements, the parallelism will be severely hampered by the presence of the inner product operations (2.3). More precisely, in each recursion step, the inner product operation will be the bottleneck of the computation, since it requires a minimum execution time of $\log_2 k$ units [3]. Consequently, to compute all the N recursions, the total parallel computing time amounts to $O(N \log_2 N)$ on a linear processor array. This not only unnecessarily slows down the parallel processing speed, but also accounts for considerable waste of processors. In the next section, we shall develop an algorithm which avoids the inner product operations, and therefore achieves much higher parallelism. More precisely, this algorithm will attain a processing time of $O(N)$ time units [as opposed to $O(N \log_2 N)$] on a vector array of $O(N)$ processing elements.

III. A Highly Concurrent Algorithm

In this section, we shall present a highly concurrent algorithm. Mathematically, this algorithm can find its roots back to the now classical Schur's algorithm [24] as first pointed out by Dewilde *et al.* [25]. The matrix formulation used in the algorithm bears a strong similarity with an earlier work by Bareiss [26] and later, in a different fashion, by Rissanen [27] and Morf [16]. However, our derivation is independent of the previous works. Moreover, in our formulation, 1) the parallelism and the (localized) date dependency inherent in the algorithm are explicitly exposed, and 2) there surfaces a natural topological mapping from the mathematical algorithm to the computing structure to be discussed in the next section. Therefore, our formulation is included below for the purpose of a clearer and easier presentation.

A. Main Algorithm

The function of the proposed algorithm is to perform a triangular decomposition on matrix T, that is,[3]

$$T = \bar{L}D\bar{U} = \bar{L}U(U = D\bar{U}) \tag{3.1}$$

where D is a diagonal matrix. Then the solution $\boldsymbol{x}$ of (1.1) can be solved explicitly with back substitution:

$$\boldsymbol{x} = T^{-1}\boldsymbol{y} = U^{-1}\bar{L}^{-1}\boldsymbol{y}. \tag{3.2}$$

To set up for a new recursive procedure, we consider an *augmented* Toeplitz matrix of T, say $\tilde{T}$, which is a natural extension of T as illustrated by the (4 × 4) example below:

$$\tilde{T} = \left[\begin{array}{cccc:cccc} t_3 & t_2 & t_1 & t_0 & t_{-1} & t_{-2} & t_{-3} \\ 0 & t_3 & t_2 & t_1 & t_0 & t_{-1} & t_{-2} \\ 0 & 0 & t_3 & t_2 & t_1 & t_0 & t_{-1} \\ 0 & 0 & 0 & \underbrace{t_3 \quad t_2 \quad t_1 \quad t_0}_{T} & & & \end{array}\right]. \tag{3.3}$$

Substituting T by $\tilde{T}$ in (3.3), we have

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}}_{\tilde{L}} \tilde{T} = \underbrace{\left[\begin{array}{ccc:c} X & X & X & u_{11} \quad u_{12} \quad u_{13} \quad u_{14} \\ X & X & X & 0 \quad u_{22} \quad u_{23} \quad u_{24} \\ X & X & X & 0 \quad 0 \quad u_{33} \quad u_{34} \\ X & X & X & \underbrace{0 \quad 0 \quad 0 \quad u_{44}}_{U} \end{array}\right]}_{} \tag{3.4}$$

where $\tilde{L} \triangleq \bar{L}^{-1}$ and the X's denote DON'T CARE entries. Our strategy is then to construct the matrices $\tilde{L}$ and U by creating all the "zeros" below the diagonal in the U matrix. Let us consider the following equation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tilde{T} = \left[\begin{array}{cccc:ccc} t_3 & t_2 & t_1 & t_0 & t_{-1} & t_{-2} & t_{-3} \\ 0 & t_3 & t_2 & t_1 & t_0 & t_{-1} & t_{-2} \end{array}\right]. \tag{3.5}$$

Now perform row operations on both sides of (3.5) such that

$$\begin{bmatrix} 1 & Kr^{(2)} \\ Ke^{(2)} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tilde{T} = \begin{bmatrix} 1 & Kr^{(2)} & 0 & 0 \\ Ke^{(2)} & 1 & 0 & 0 \end{bmatrix} \tilde{T}$$

$$= \begin{bmatrix} v^{(2)} \\ u^{(2)} \end{bmatrix} = \left[\begin{array}{ccc:ccc} v_3^{(2)} & v_2^{(2)} & v_1^{(2)} & v_0^{(2)} & 0 & v_{-2}^{(2)} & v_{-3}^{(2)} \\ u_3^{(2)} & u_2^{(2)} & u_1^{(2)} & 0 & u_{-1}^{(2)} & u_{-2}^{(2)} & u_{-3}^{(2)} \end{array}\right] \tag{3.6}$$

where[4]

$$Ke^{(2)} = -t_1/t_0; \qquad Kr^{(2)} = -t_{-1}/t_0 \tag{3.7}$$

Compare the second row of the RHS (right-hand side) of (3.4) to that of (3.6), i.e., $\boldsymbol{u}^{(2)}$; it is clear that a zero is created by the row operation and the desired second rows of $\tilde{L}$ and U are obtained:

$$\tilde{L}_2 = [1_{21} \quad 1 \quad 0 \quad 0] = [Ke^{(2)} \quad 1 \quad 0 \quad 0]$$

$$U_2 = [0 \quad u_{22} \quad u_{23} \quad u_{24}] = [0 \quad u_{-1}^{(2)} \quad u_{-2}^{(2)} \quad u_{-3}^{(2)}].$$

To compute the third rows of the $\tilde{L}$ and U matrices, the same strategy can be used. For this purpose, we first *right-shift* $[Ke^{(2)} \; 1 \; 0 \; 0]$ such that $[Ke^{(2)} \; 1 \; 0 \; 0] \rightarrow [0 \; Ke^{(2)} \; 1 \; 0]$. By the Toeplitz structure of the $\tilde{T}$ matrix, $\boldsymbol{u}^{(2)}$ in (3.6) will also be right-shifted accordingly. Thus, we have the following equation:

$$\begin{bmatrix} 1 & Kr^{(2)} & 0 & 0 \\ 0 & Ke^{(2)} & 1 & 0 \end{bmatrix} \tilde{T}$$

$$= \left[\begin{array}{ccc:ccc} v_3^{(2)} & v_2^{(2)} & v_1^{(2)} & v_0^{(2)} & 0 & v_{-2}^{(2)} & v_{-3}^{(2)} \\ 0 & u_3^{(2)} & u_2^{(2)} & u_1^{(2)} & 0 & u_{-1}^{(2)} & u_{-2}^{(2)} \end{array}\right]. \tag{3.8}$$

Note that through this shift operation, the two 0's created in the previous recursion on the RHS are realigned into the same column so as to remain uneffected by the row operations in the next recursion. With this precaution, a similar procedure as in the previous recursion can be repeated:

$$\begin{bmatrix} 1 & Kr^{(3)} \\ Ke^{(3)} & 1 \end{bmatrix} \begin{bmatrix} 1 & Kr^{(2)} & 0 & 0 \\ 0 & Ke^{(2)} & 1 & 0 \end{bmatrix} \tilde{T}$$

$$= \begin{bmatrix} 1 & X & Kr^{(3)} & 0 \\ Ke^{(3)} & X & 1 & 0 \end{bmatrix} \tilde{T} \tag{3.9a}$$

$$= \begin{bmatrix} \boldsymbol{v}^{(3)} \\ \boldsymbol{u}^{(3)} \end{bmatrix} = \left[\begin{array}{ccc:ccc} v_3^{(3)} & v_2^{(3)} & v_1^{(3)} & v_0^{(3)} & 0 & 0 & v_{-3}^{(3)} \\ u_3^{(3)} & u_2^{(3)} & u_1^{(3)} & 0 & 0 & u_{-2}^{(3)} & u_{-3}^{(3)} \end{array}\right] \tag{3.9b}$$

where

$$Ke^{(3)} = -u_1^{(2)}/v_0^{(2)} \quad \text{and} \quad Kr^{(3)} = -v_{-2}^{(2)}/u_{-1}^{(2)}. \tag{3.10}$$

By comparing $\boldsymbol{u}^{(3)}$ to the third row on the RHS of (3.4), clearly, the third rows of the $\tilde{L}$ and U matrices are obtained:

$$\tilde{L}_3 = [l_{31} \quad l_{32} \quad 1 \quad 0] = [Ke^{(3)} \quad (Ke^{(3)}Kr^{(2)} + Ke^{(2)}) \quad 1 \quad 0]$$

$$U_3 = [0 \quad 0 \quad u_{33} \quad u_{34}] = [0 \quad 0 \quad u_{-2}^{(3)} \quad u_{-3}^{(3)}].$$

This completes the second recursion. By induction, future recursions can be carried out in the same manner until all the rows of the $\tilde{L}$ and U matrices are computed. Summarizing the above procedure, several observations can be made.

1) *The shift operation in each recursion is natural due to the Toeplitz structure of the $\tilde{T}$ matrix since it retains the zeros produced in the previous recursion.* The purpose of the shift is to realign these zeros with those of the auxiliary vector $\boldsymbol{v}$ [cf. (3.8) such that these zeros will remain unaffected by the upcoming row operations. This also explains the purpose and the necessity of computing for the auxiliary vectors $\boldsymbol{v}$ in each recursion.

2) Note that $\tilde{L}$ is nothing but the L_A matrix in (2.8) since from (2.8) we have

[3]The overbar "$\bar{\ }$" of the triangular matrix L (or U) indicates that L (or U) has 1's along its diagonal.

[4]These $\{Ke, Kr\}$ play the same role as those in the Levinson algorithm, and therefore will also be called *reflection coefficients* in the following discussion.

$$T = L_A^{-1} D_E^{-1} U_B^{-1}.$$

Comparing the above equation to (3.1), and noting that the LDU factorization of a given matrix is unique,[5] we have $\bar{L} = L_A^{-1}$, $\bar{U} = U_B^{-1}$, and $D = D_E^{-1} = \text{diag}\,[E_0, \cdots, E_N]$. In this sense, the algorithm proposed can be regarded as a generalization of the conventional Levinson algorithm. More precisely, the $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ vectors are actually embedded in the $2 \times N$ matrix on the LHS in the kth recursion. For example, in the first recursion [cf. (3.6)], we have $a_{11} = Ke^{(2)}$, $b_{11} = Kr^{(2)}$. In the second recursion [cf. (3.9a)], we have

$$\begin{bmatrix} 1 & Kr^{(3)} \\ Ke^{(3)} & 1 \end{bmatrix} \begin{bmatrix} 1 & a_{11} & 0 & 0 \\ 0 & b_{11} & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & a_{12} & a_{22} & 0 \\ b_{22} & b_{12} & 1 & 0 \end{bmatrix}. \tag{3.11}$$

3) This new formula *completely avoids the need of inner product operations;* therefore, the bottleneck incurred in parallel execution of the Levinson algorithm no longer exists. Furthermore, the reflection coefficient computation and row operations in the formulation are very suitable for parallel execution; hence, this new algorithm is inherently highly concurrent. As a consequence, it has the following parallel formulation.

Main Algorithm

INITIAL CONDITIONS:

$$v_k^{(1)} = u_k^{(1)} = t_k \qquad (-N \leqslant k \leqslant N) \tag{3.12a}$$

FOR $i = 1$ UNTIL N DO BEGIN

IN PARALLEL DO BEGIN[6]

$$Ke^{(i+1)} = -u_1^{(i)} [v_0^{(i)}]^{-1} \tag{3.12b}$$

$$Kr^{(i+1)} = -v_{-i}^{(i)} [u_{-i+1}^{(i)}]^{-1} \tag{3.12c}$$

END IN PARALLEL DO;

IN PARALLEL FOR $-N \leq k \leq N$ DO BEGIN

$$v_k^{(i+1)} = v_k^{(i)} + Kr^{(i+1)} u_{k+1}^{(i)} \tag{3.12d}$$

$$u_k^{(i+1)} = u_{k+1}^{(i)} + Ke^{(i+1)} v_k^{(i)} \tag{3.12e}$$

END IN PARALLEL DO;

OUTPUT;

END FOR LOOP;

4) Based on the above formulation, it is clear that *with $O(N)$ processing elements connected in a linear array, the parallel computing time for each recursion can take as little as two time units* (one for reflection coefficient computation, the other for row operation). *For N recursions, this amounts to a total parallel processing time of $O(N)$ time units as opposed to $O(N \log_2 N)$ in the Levinson algorithm.*

B. Duality of the Main Algorithm

1) *Duality:* In order to execute the back substitution step (3.2) to solve a Toeplitz system, both the $\tilde{L}$ and U matrices have to be computed. However, the equation for computing the $\tilde{L}$ $(= \bar{L}^{-1})$ matrix is purposely left out in the above main algorithm. *This is because each column of the L $(= \bar{L}D)$ matrix is also provided by the vector v while each row of the U matrix is provided by the vector u.* More precisely, it is proved in Appendix A that

$$T = \begin{bmatrix} v_0^{(1)} & 0 & \cdots & 0 \\ v_1^{(1)} & v_0^{(2)} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \\ v_N^{(1)} & v_{N-1}^{(2)} & \cdots & v_0^{(N+1)} \end{bmatrix} \begin{bmatrix} E_0 & & & 0 \\ & E_1 & & \\ & & \ddots & \\ 0 & & & E_N \end{bmatrix}^{-1} \cdot \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & \cdots & u_{-N}^{(1)} \\ 0 & u_{-1}^{(2)} & \cdots & u_{-N}^{(2)} \\ & \ddots & & \vdots \\ 0 & & 0 & u_{-N}^{(N+1)} \end{bmatrix} \tag{3.13a}$$

$$= LD^{-1}U. \tag{3.13b}$$

Consequently, (3.12) alone is sufficient for $\bar{L}D\bar{U}$ factorization of T matrix and the computation of the $\tilde{L}$ matrix can be saved (see also [28]).

2) Symmetric Toeplitz Systems: Additional computational savings are possible in solving a *symmetric* Toeplitz system. In this case, $L = U$, and hence [from (3.12)] $Ke^{(i)} = Kr^{(i)}$ for each i. Consequently, the computations of $v_k^{(i)}$ and $u_k^{(i)}$ for $k > 0$ become redundant and may be omitted. *This leads to a savings of one half of the computations as compared to the nonsymmetrical case.*

In practice, symmetric Toeplitz systems arise much more often than nonsymmetric ones, and are therefore of much greater importance from an applicational point of view. In the next section, a pipelined computing structure for concurrent processing of symmetric Toeplitz systems solutions will be discussed.

IV. Pipelined Lattice Processor

A. Parallel Lattice Computing Structure

In this section, we consider the implementation of the parallel algorithm on a VLSI chip. The major computation in the algorithm lies in the linear combinations of two vectors in each recursion. Therefore, a parallel computing structure with a linear processor array as depicted in Fig. 1 is proposed.

The configuration consists of a series of modular processing cells, termed *lattice cells*, to perform row operations. Each cell is composed of an upper and a lower processing element (PE) with lattice connections. The only exception is the upper PE in the ⟨0⟩ cell which is a divider cell for the computation of reflection coefficients.

During each recursion, the reflection coefficient is first computed in the divider cell and then broadcast to all the lattice cells through the global (horizontal) interconnections. Then the row operations are performed simultaneously in all the lattice cells. Upon completion, the result in each upper PE is left-shifted to its immediate left neighbor, preparing for the next recursion.[7] Meanwhile, the contents of the lower PE's,

[5] This is true provided that all the leading principal minors of the given matrix are nonsingular.

[6] IN PARALLEL DO indicates that the operations within this block can be executed in parallel.

[7] This corresponds to the shift operation discussed in Section III. Note that since only the relative position between the data in the upper and lower PE's is of importance, the left-shift for the upper PE's ($\boldsymbol{v}$ vector) is equivalent to the right-shift for the lower PE's ($\boldsymbol{u}$ vector), as described earlier.

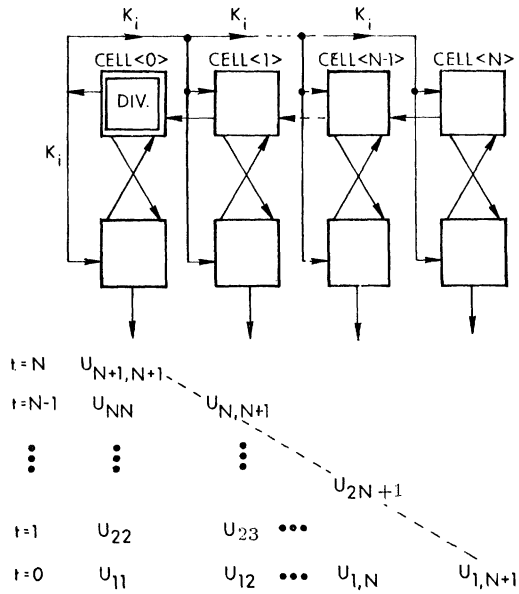


Fig. 1. Parallel lattice computing structure.

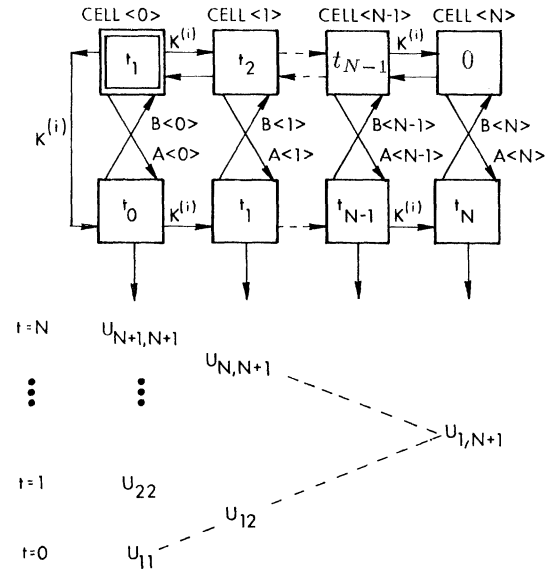


Fig. 2. Pipelined lattice computing structure.

which correspond to a row of the U matrix, can be output, and the recursion is thus completed.

The operation is completed after N such recursions. Let τ_1 denote the time interval needed for division, and let τ_2 denote the time interval for each lattice operation (multiplication and addition); then the total computing time will be $N(\tau_1 + \tau_2)$.

B. Pipelined Lattice Computing Structure

To accomplish maximal parallelism, the parallel lattice computing structure just proposed relies heavily on global communication. As mentioned earlier, this may cause certain difficulties on, for example, synchronization, longer delay, larger power, and chip area consumption in a VLSI system. Therefore, in the following, we shall propose a modified version of the lattice computing structure which eliminates the need for global communication without compromising the parallelism.

The general configuration (Fig. 2) of the modified lattice computing structure remains largely resembling the one in Fig. 1, except that the global communication links are replaced by nearest neighbor interconnections. To achieve maximal parallelism in this locally connected computing network, we must resort to a pipelined operation which renders efficient and smooth data flow. This leads to a useful notion of computational wavefront [5], [6], [29]-[31].

C. Computational Wavefront

Roughly speaking, a computational wavefront in a computing structure corresponds to the computational activity incurred in one recursion step in a recursive (parallel) algorithm. As an example, the computational wavefront of the first recursion is examined below.

Suppose that the data are initially placed in the registers of the PE's such that $B_{\langle 0\rangle} = t_0$ and $A_{\langle m-1\rangle} = B_{\langle m\rangle} = t_m$ for $m = 1, 2, \cdots, N$ where the $A_{\langle m\rangle}$ register is in the mth upper PE and $B_{\langle m\rangle}$ in the mth lower PE (cf. Fig. 2). The process starts with the $\langle 0\rangle$ cell (divider cell) where the reflection coefficient is computed and stored in register $C_{\langle 0\rangle}$:

$$C_{\langle 0\rangle} \Leftarrow A_{\langle 0\rangle}/B_{\langle 0\rangle}. \tag{4.1}$$

The computation activity then propagates to the $\langle 1\rangle$ cell (after propagating $C_{\langle 0\rangle}$ to $C_{\langle 1\rangle}$) where the following computations are executed simultaneously in upper and lower PE's:[8]

$$A_{\langle 1\rangle} \Leftarrow A_{\langle 1\rangle} - C_{\langle 1\rangle} \times B_{\langle 1\rangle}$$

(in the upper PE of the $\langle 1\rangle$ cell) (4.2a)

$$B_{\langle 1\rangle} \Leftarrow B_{\langle 1\rangle} - C_{\langle 1\rangle} \times A_{\langle 1\rangle}$$

(in the lower PE of the $\langle 1\rangle$ cell). (4.2b)

Upon completion of the execution, the $\langle 1\rangle$ cell propagates its new content in the upper PE, $A_{\langle 1\rangle}$, to its left neighbor, $A_{\langle 0\rangle}$, to prepare for the next recursion. Meanwhile, it also sends the content of $C_{\langle 1\rangle}$ to $C_{\langle 2\rangle}$ so that the computation activity continues propagating to the $\langle 2\rangle$ cell. The next front of activity will be at the $\langle 3\rangle$ cell, then the $\langle 4\rangle$ cell, and so on. As a consequence, a computational wavefront is created traveling across the linear processor array. (It may be noted that the wave propagation implies localized data flow.) Once the wavefront sweeps through all the cells, the first recursion is completed. The content in $B_{\langle k\rangle}$ ($k = 0, 1, \cdots, N-1$) is output downward as soon as it is available.

As the first computational wavefront propagates, the second recursion can be executed concurrently *by pipelining a second wavefront as soon as the content in $A_{\langle 1\rangle}$ is sent to the $A_{\langle 0\rangle}$ register in the $\langle 0\rangle$ cell.* Therefore, the time interval between the first and second wavefronts is estimated to be $\tau_1 + \tau_2$. (Note that it takes a $\tau_1 + \tau_2$ time interval before $A_{\langle 1\rangle}$ is made available if data transfer time is considered negligible.) The second wavefront strongly resembles that of the first one. For example, the $\langle 0\rangle$ cell first computes the second reflection coefficient according to (4.1), then forward $C_{\langle 0\rangle}$ to the $\langle 1\rangle$ cell where (4.2) will be performed. After this, the second wavefront will propagate to cells $\langle 2\rangle$, $\langle 3\rangle$, and so on. Once the wavefront arrives and executes at the $(N-1)$th cell, the second recursion is completed. Again, the content in $B_{\langle k\rangle}$, which is nothing but $u^{(3)}_{-k-2}$ ($k = 0, 1, \cdots, N-2$), is output downward as soon as it is available.

[8]The lower PE of the $\langle 0\rangle$ cell should also perform the operation $B_{\langle 0\rangle} \Leftarrow B_{\langle 0\rangle} - C_{\langle 0\rangle} X A_{\langle 0\rangle}$ at this moment. This accounts for the computation of the D matrix.

The same pipelining scheme can be repeated for the third, and eventually all the recursions. When all the N wavefronts are generated and operations are executed, the parallel algorithm is completed. Since the wavefronts are generated consecutively at a rate of one wavefront per $\tau_1 + \tau_2$ time interval, the total computing time will be $N(\tau_1 + \tau_2)$.[9] Therefore, the pipelined operation takes the processing time $N(\tau_1 + \tau_2)$, which is the same as that of the parallel operation with global communication. From now on, the proposed pipelined computing structure will be called the pipelined lattice processor (PLP). We note that the PLP has accomplished the design goal of using only local interconnections and yet not sacrificing any degree of parallelism attainable.

D. Programming Aspects

In the above, we have successfully exploited the notion of the computational wavefront to portray the pipelined operations on the lattice array. However, in general, it is even more preferable to have an appropriate language to program pipelined algorithms.

In a recent paper [5], [6], it is noted that the wavefront notion has indeed much more widespread applications. Namely, it is applicable to a large class of parallel matrix algorithms, especially to those with the so-called *locality* and *recursivity* nature. Therefore, this same notion has been further extended to a wavefront-oriented language suitable for programming pipelined algorithms on programmable (pipelined) array processors [5], [6], [29]-[31]. Although further elaboration on the structure of the language is beyond the scope of this paper, for illustration purpose, we have included in Appendix B one such programming example for the above pipelined algorithm.

The wavefront language does not only facilitate programming most pipelined algorithms, but also offers a convenient tool for the subsequent simulation and verification tasks. The simulation results of the PLP yield a series of snapshots of the computation wavefronts in the PLP which match with the result theoretically predicted [31].

V. Complete Toeplitz System Solvers

So far, we have used the PLP to decompose a Toeplitz system into lower and upper triangular matrices, i.e., $T = U^t D^{-1} U$ in $O(N)$ time units (assume that T is symmetric). To completely solve the Toeplitz system, however, an explicit solution $\boldsymbol{x}$ has to be derived. Therefore, subsequent operations are needed and should also be performed in $O(N)$ time units. Based on different inversion formulas for the Toeplitz matrix T, we shall present three different organizations—corresponding to three different methods for the subsequent computations—of a complete Toeplitz system solver. All of these utilize linear processor array and achieve $O(N)$ units of processing time.

A. Back-Substitution Method

This method involves computing $\boldsymbol{x}$ via (3.2) (for the symmetric case):

$$\boldsymbol{x} = T^{-1}\boldsymbol{y} = U^{-1}D(U^t)^{-1}\boldsymbol{y} \tag{5.1}$$

which can be separated in two back-substitution steps:

$$\boldsymbol{g} = D(U^t)^{-1}\boldsymbol{y} \tag{5.2a}$$

and

$$\boldsymbol{x} = U^{-1}\boldsymbol{g}. \tag{5.2b}$$

Back substitution is a standard matrix operation for solving linear systems, which also enjoys a pipelined operation. Therefore, it can be implemented with a locally connected linear processor array. As it is a rather well-known procedure [4], [7], the detail is omitted here.

A complete Toeplitz system solver depicted in Fig. 3 is constituted by a PLP and a (pipelined) back-substitution processor. The processor in the upper part is the PLP where the outputs are fed into the lower part—the back-substitution processor. Upon receiving the data, the back-substitution processor (initially stores the $\boldsymbol{y}$ vector) will perform the first back substitution (i.e., $\boldsymbol{g} = D(U^t)^{-1}\boldsymbol{y}$). The elements of the U^t matrix (output from PLP) are stored in the LIFO (last-in-first-out) memory stack which serves as a matrix transposer. The scaling operator $D = \text{diag}[u_{00}, \cdots, u_{NN}]$ will operate on $(U^t)^{-1}\boldsymbol{y}$ to obtain the $\boldsymbol{g}$ vector which then is stored in the G-LIFO stack. After the first back substitution, the second step (5.2b) starts immediately with the U matrix and $\boldsymbol{g}$ vector fetched from the memory stacks. Finally, the output $\boldsymbol{x}$ is obtained from the left end of the back-substitution processor. Note that the first back substitution can be executed concurrently with the LU decomposition in order to save processing time.

B. A Method Based on LU Decomposition of T^{-1}

Recall that in Section II, the $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ vectors computed from the Levinson algorithm constitute a *UDL* decomposition of the T^{-1} matrix, and thus facilitate an explicit solution for the Toeplitz system (2.8). Now, this formula can be utilized to provide a different organization of the Toeplitz system solver. In doing so, we have to compute the $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ vectors. However, in Section III-A, we note that the Levinson procedure is inherently imbedded in the new algorithm, and that the computation of $\{\boldsymbol{a}_k, \boldsymbol{b}_k\}$ requires virtually the same kind of row operations [see, e.g., (3.11)]. Therefore, the PLP processor can also be utilized to produce these vectors. For this, a duplicate PLP (without a divider cell) can be attached to the original one. The new PLP will have different initial conditions: all of its PE's are stored with "0," except the ⟨1⟩ cell where "1" is stored. During execution, the new PLP is triggered by the reflection coefficients sending from the original PLP for performing lattice operations. Then the results, $\boldsymbol{a}_k$ vectors [cf. (3.11)] in the lower PE's, will be output to a linear processor array for matrix-vector multiplication: $\boldsymbol{x} = L_A^t D L_A \boldsymbol{y}$. This is obtained from (2.8) with $U_B = L_A^t$ (i.e., the symmetric case). Details of this multiplication operation are omitted here since they are again accessible in the literature [1], [4], [7].

Cybenko [32] has shown that the Levinson algorithm (in the sequential computation scheme) is numerically comparable to the Cholesky factorization method which is known to be numerically stable. Since (2.8) is a formula based on the Levinson algorithm, the numerical stability of the above method in the pipelined computation scheme is expected.

[9] Note that the Nth recursion will be initiated at the time $(N-1)(\tau_1 + \tau_2)$, and by that time there will be only one computation [i.e., (4.1)] to be performed in the last recursion (cf. Section III).

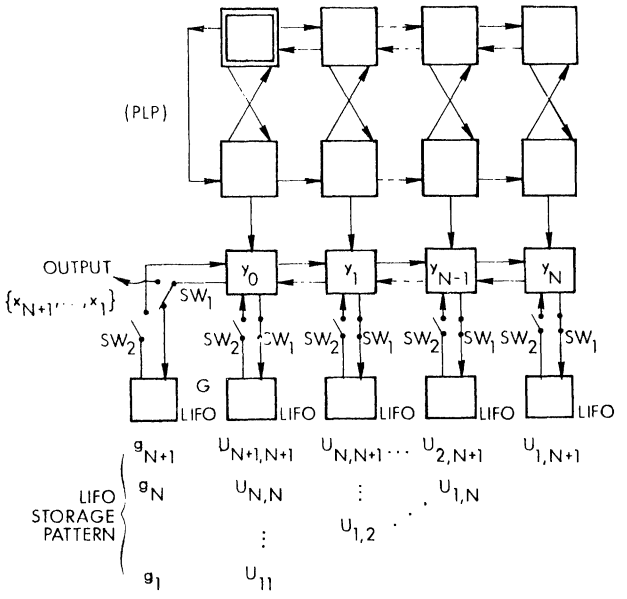


Fig. 3. Complete Toeplitz system solver.

C. A Different Organization Based on Gohberg's Formula

In addition to the *UDL* decomposition, another Toeplitz inversion formula proposed by Gohberg [33] can also be applied for an alternative organization. This formula is based only on the knowledge of $\{\boldsymbol{a}_N, \boldsymbol{b}_N\}$ and E_N:

$$T^{-1} = \frac{1}{E_N}\begin{bmatrix} 1 & & \\ a_{1N} & 1 & 0 \\ \vdots & \ddots & \ddots \\ a_{NN} & a_{1N} & 1 \end{bmatrix}\begin{bmatrix} 1 & b_{1N} & \cdots & b_{NN} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & b_{1N} \\ 0 & \cdots & & 1 \end{bmatrix}$$

$$- \frac{1}{E_N}\begin{bmatrix} 0 & & \\ a_{NN} & 0 & 0 \\ \vdots & \ddots & \ddots \\ a_{1N} & a_{NN} & 0 \end{bmatrix}\begin{bmatrix} 0 & b_{NN} & \cdots & b_{1N} \\ \vdots & 0 & \ddots & \vdots \\ \vdots & & \ddots & b_{NN} \\ 0 & \cdots & & 0 \end{bmatrix} \tag{5.4a}$$

$$= L_a U_a + L_b U_b. \tag{5.4b}$$

Hence, $\boldsymbol{x}$ can be solved explicitly from

$$\boldsymbol{x} = T^{-1}\boldsymbol{y} = L_a U_a \boldsymbol{y} + L_b U_b \boldsymbol{y} = L_a \boldsymbol{w} + L_b \boldsymbol{z} \tag{5.5}$$

where $\boldsymbol{w} = U_a \boldsymbol{y}$, $\boldsymbol{z} = U_b \boldsymbol{y}$. Note that due to the particular triangular Toeplitz structure, (5.5) can be essentially regarded as linear convolution operations.

The appended PLP in Section V-B is again applicable to produce the needed $\{\boldsymbol{a}_N, \boldsymbol{b}_N\}$. On completion of the lattice operation, these vectors will be ready in the upper and lower PE's for the convolution operation according to Gohberg's formula. For this convolution operation, two linear array processors can be utilized or, when the length of the vector is large, than the FFT (fast Fourier transform) processor may be used for fast processing.

As a brief comparison of the three different organizations proposed, note that the first organization utilizes only one set of PLP; hence, it needs fewer PE's. On the other hand, in the third organization, no memory modules (LIFO) are needed. As for the processing speed, it depends very much on the size of the processor array and the applications. Generally speaking, the first method is perhaps the fastest for moderate matrix size. When N is large and the FFT is worth utilizing, the third method will become more favorable.

VI. Applications, Implementations, and Extensions

A. Applications to Array Signal Processing

From a practical point of view, the proposed Toeplitz system solver has many immediate signal processing applications. For example, it can be applied to maximum likelihood (ML) and maximum entropy (ME) adaptive array signal processing[10] problems [9]. In the ML method, the wave-vector spectrum is computed as

$$\sigma_G^2 = [E^t(f, \boldsymbol{v}) S_x^{-1}(f) E(f, \boldsymbol{v})]^{-1}$$

where $S_x(f)$ is an $N \times N$ cross-covariance matrix and $E(f, \boldsymbol{v})$ is an $N \times 1$ vector denoting the array phasing vector corresponding to the wave direction [9]. Suppose that the input sensor array is uniformly spaced, and that the statistical environment is stationary, then the $S_x(f)$ matrix to be inverted in the above equation will be a Toeplitz matrix. Then the PLP can be applied to efficiently compute the spectrum σ_G^2.

For a better illustration, we use the following sample system specifications. Suppose that the number of the sensors $N = 32$ and that the temporal frequency bandwidth is $F = 50$ Hz and the frequency resolution desired is $\delta F = 0.1$ Hz. Then for one set of input data, the total computation time required is

$$(F/\delta F) \times 2Nt_0 = 32\,000\, t_0.$$

Now, we must have $32\,000\, t_0 \leqslant T$ where $T = 1/\delta F = 10$ s is the processing time allowed for one set of data, provided a real-time processing rate is required. Hence, each arithmetic operation must take no more than 10 s/32 000 = 312.5 μs (that is, $t_0 \leqslant 312.5$ μs). This time frame can be easily complied by a commercially available ALU, and therefore, a real-time signal processing rate is trivially attainable.

Obviously, the specifications quoted above are oversimplified. In practical situations, one will probably face a larger sensor array with higher bandwidth; hence, many more operations will probably be needed. On the other hand, in terms of VLSI technology, the size of the PLP can be made much larger, and the multiplication speed can be 10^3-10^4 times faster than that given above. Therefore, for most cases, the real-time processing rate can still be achievable, and with VLSI implementation of PLP, it is also rather affordable.

B. Implementation

In a joint effort between the Hughes Research Laboratory (at Malibu, CA) and the University of Southern California, Los Angeles, a VLSI Toeplitz system solver chip is implemented [34], [35].

This design contains 16 stages of processors with 28 bit fixed-point arithmetic units working on 8 bit (input dynamic range) input data. Suppose that the system clock is operating on a 4 MHz rate; it is estimated that a 16 X 16 Toeplitz system equation can be solved in about 0.6 ms (minisecond). The design is based on NMOS depletion load technology. In the

[10]The term "array processing" in signal processing commonly refers to the processing of signals from a multiple sensor array.

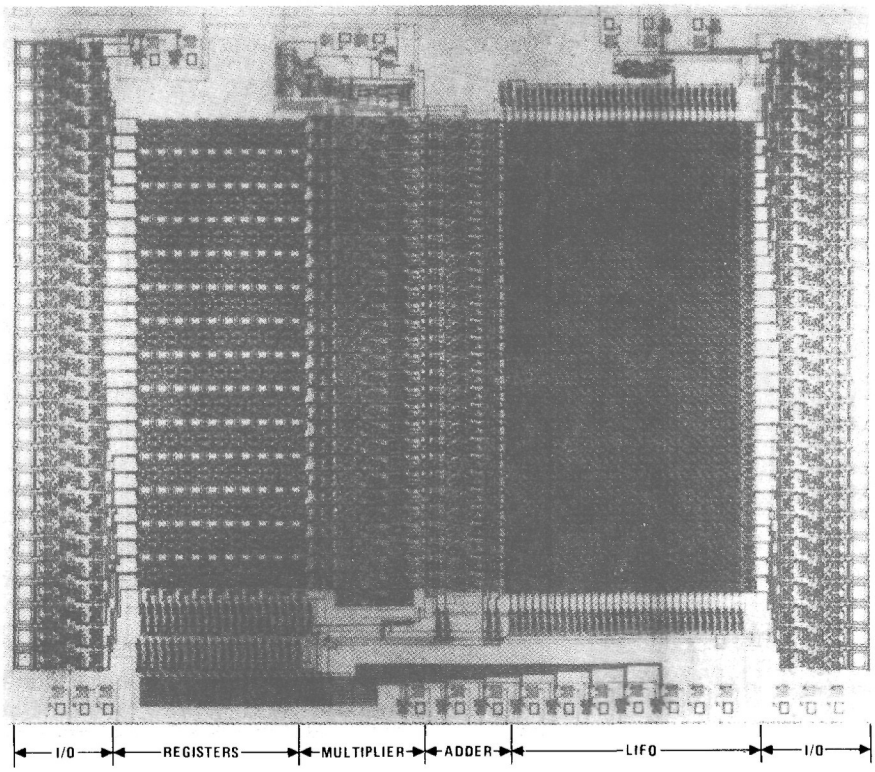


Fig. 4. Chip layout of Toeplitz systems solver. (Courtesy Hughes Research Laboratory, Malibu, CA.)

chip design, 6 μm feature size ($\lambda = 6$ μm) is used, and a single stage of processor will occupy a whole chip (230×300 mil^2). The chip layout is shown in Fig. 4. It is estimated that with the feature size reduced to 1 μm, 32 stages can be implemented on a single chip in a later implementation phase. A full report on the final implementation will appear in a future publication.

C. Extension to Multichannel Signal Processing

Quite often, one has to deal with multichannel signal processing which involves a number of input signals simultaneously. In this situation and under the stationary statistic assumption, the main computation involved is usually to solve a *block* Toeplitz system:

$$\boldsymbol{B} = \begin{bmatrix} B_0 & B_{-1} & \cdots & B_{-N} \\ B_1 & B_0 & & \\ \vdots & & \ddots & \\ B_N & & & B_0 \end{bmatrix}$$

where each block element B_k is by itself a $q \times q$ matrix. *The extension of the scalar case (nonsymmetric) main algorithm to the matrix case can be carried out easily. However, the dual relations established in the scalar case fail to extend to the matrix case* due to the noncommutability nature of matrix multiplication. Consequently, we need a total of $2N$ block-matrix processors (as opposed to N) to implement the corresponding lattice computing structure for a symmetrical block Toeplitz system. To strive towards a saving of half the required processors, we propose the following further modification.

D. Normalized Levinson Algorithm

Recently, a normalized version of the Levinson algorithm was proposed by Vieara *et al.* [36]. This algorithm employs a matrix square root procedure to accomplish normalization of the reflection coefficient.[11] It turns out that the very same version may be applied to the parallel algorithm to retain the duality relations and whereby save half of the hardware. Roughly speaking, during the execution of each recursion, each entry of the $\boldsymbol{u}$ and $\boldsymbol{v}$ vector is modified by[12]

$$\overset{*}{V}{}_k^{(i)} = [V_0^{(i)}]^{-1/2} V_k^{(i)}$$

$$\overset{*}{U}{}_k^{(i)} = [U_{-i+1}^{(i)}]^{-1/2} U_k^{(i)}.$$

By substituting the above relation into the new algorithm (in multichannel formulation), a similar analysis as in Section III-A can be carried out and a *normalized* version of the new algorithm can be derived. For completeness, this normalized (high parallelism) algorithm is present in Appendix C.

VII. Conclusion

Two fundamental issues should be kept in mind in designing modern VLSI signal processors. The first is to formulate the signal processing algorithm to allow the maximal extent of parallel processing. The second is to make sure that the parallel architecture meets the (localized) communication constraint imposed by the device technology.

In this paper, we have demonstrated an integrated approach to the above issues by tackling a specific (but practically

[11] That is, for each reflection coefficient matrix $K^{(i)}$, its L_2 norm is less than or equal to unity provided that the B matrix is nonnegative definite.

[12] For a symmetric nonnegative definite matrix M, $M^{1/2}$ is defined as $M = M^{1/2}[M^{1/2}]^t$.

important) problem–solving Toeplitz systems. We have proposed a highly concurrent algorithm which enjoys $O(N)$ computing time by a vector processor array with $O(N)$ processors. This is to be compared to the $O(N^2)$ processing time of the Levinson algorithms when implemented sequentially or $O(N \log_2 N)$ when implemented in parallel. Furthermore, we have developed a pipelined lattice processor (PLP) to implement the new algorithm. The PLP architecture employs a localized communication scheme without sacrificing the overall parallel processing speed.

Appendix A
Proof of (3.13)

The duality in (3.13) originates from the symmetric row operation in the algorithm (3.12). Suppose that a matrix $\hat{T} \triangleq T^t$ is introduced. Due to Toeplitz structure, we know that $\hat{T}$ is also a Toeplitz matrix with $\hat{t}_k = t_{-k}$ for $-N \leqslant k \leqslant N$. Obviously, the algorithm (3.12) can be applied on $\hat{T}$ as well. Thus,[13]

$$\hat{T} = \hat{L}\hat{D}^{-1}\hat{U}. \tag{A.1}$$

Taking the transpose on both sides of (A.1) and using the uniqueness property of triangular factorization, it is easy to show that

$$\hat{L} = U^t, \quad \hat{D} = D, \quad \text{and} \quad \hat{U} = L^t. \tag{A.2}$$

On the other hand, by carrying out the algorithm (3.12), we find that

$$1)\ \hat{Ke}^{(i)} = \hat{Kr}^{(i)}, \qquad \hat{Kr}^{(i)} = \hat{Ke}^{(i)} \tag{A.3}$$

$$2)\ \hat{v}_k^{(i)} = u_{-k-i+1}^{(i)}, \qquad \hat{u}_k^{(i)} = v_{-k-i+1}^{(i)}. \tag{A.4}$$

Equations (A.3) and (A.4) are proved by induction as follows.

For $i = 1$, (A.4) is true from (3.12a). Then, from (3.12b) and (3.12c),

$$\hat{Ke}^{(2)} = -u_1^{(1)}/v_0^{(1)} = -t_{-1}/t_0 = Kr^{(2)} \tag{A.5a}$$

$$\hat{Kr}^{(2)} = -u_{-1}^{(1)}/v_0^{(1)} = -t_1/t_0 = Ke^{(2)}. \tag{A.5b}$$

Suppose that for $i = I$, (A.3) and (A.4) are valid; let $i = I + 1$:

$$\hat{Ke}^{(I+1)} = -\hat{u}_1^{(I)}/\hat{v}_0^{(I)} = -v_{-I}^{(I)}/u_{-I+1}^{(I)} = Kr^{(I+1)} \tag{A.6a}$$

$$\hat{Kr}^{(I+1)} = -\hat{v}_{-I}^{(I)}/\hat{u}_{-I+1}^{(I)} = -u_1^{(I)}/v_0^{(I)} = Ke^{(I+1)} \tag{A.6b}$$

Moreover,

$$\begin{aligned}\hat{v}_k^{(I+1)} &= \hat{v}_k^{(I)} + \hat{Kr}^{(I+1)}\hat{u}_{k+1}^{(I)} \\ &= u_{-k-I+1}^{(I)} + Ke^{(I+1)}v_{-k-I}^{(I)} = u_{-k-(I+1)+1}^{(I+1)}\end{aligned} \tag{A.7a}$$

$$\begin{aligned}u_k^{(I+1)} &= u_{k+1}^{(I)} + Ke^{(I+1)}v_k^{(I)} \\ &= v_{-k-I}^{(I)} + Kr^{(I+1)}u_{-k-I+1}^{(I)} = v_{-k-(I+1)+1}^{(I+1)}.\end{aligned} \tag{A.7b}$$

By mathematical induction, (A.3) and (A.4) are proved.

From (A.1) and (A.4), it is clear that

$$\hat{U}_i = [0, \cdots, 0, v_0^{(i)}, \cdots, v_{N-i}^{(i)}].] \tag{A.8}$$

Using (A.8) and the relation $\hat{U} = L^t$ in (A.2), (3.13) follows.

[13] In Appendix A, all the symbols with "^" will be regarded as those associated with the $\hat{T}$ matrix.

Appendix B
Program for Pipelined Lattice Algorithm

(In the program presented below, the program constructs are close to those of Pascal language, while the arithmetic operations are similar to those of Assembly language. Note that the statement after the "!" sign is considered as a comment.)

Array Size: $2 \times N$ processing elements.

Computation: Pipelined lattice algorithm.

Initial: First row of the Toeplitz matrix T in the A register of the first row PE's, and the B register of the second row PE's.

Output: Rows of the U matrix ($T = U^t D^{-1} U$) are output from the second row PE's.

```
BEGIN
  SET COUNT 1;
  REPEAT
    WHILE WAVEFRONT IN ARRAY DO
      BEGIN
        CASE KIND =
          (1,*) : FLOW A, LEFT;
          (2,1),(2,*) : FLOW B, UP;
        ENDCASE;
      END;
    DECREMENT COUNT;
  UNTIL TERMINATED;
  SET COUNT N;
  REPEAT
    WHILE WAVEFRONT IN ARRAY DO
      BEGIN
        CASE KIND =
          (1,1),(1,*) : BEGIN
                          FETCH A, RIGHT;
                          FLOW A, DOWN;
                          FETCH B, DOWN;
                        END;
          (1,1) : BEGIN
                    DIV A,B,C;  ! C=A/B;
                    FLOW C, DOWN;
                  END;
          (1,*) : BEGIN
                    FETCH C, LEFT;
                    ! A <= A - B X C;
                    MULT B,C,R;
                    SUB A,R,A;
                    FLOW A, LEFT;
                  END;
          (2,1) : FETCH C, UP;
          (2,*) : FETCH C, LEFT;
          (2,1),(2,*) : BEGIN
                          FETCH A, UP;
                          ! B <= B - A X C;
                          MULT A,C,R;
                          SUB B,R,B;
                          FLOW B, UP;
                          FLOW B, DOWN;  ! OUTPUT;
                        END;
        ENDCASE;
        FLOW C, RIGHT;
```

END;
DECREMENT COUNT;
UNTIL TERMINATED;
ENDPROGRAM.

APPENDIX C
NORMALIZED ALGORITHM

'NITIAL CONDITIONS:

$$\overset{*}{V}_k^{(1)} = \overset{*}{U}_k^{(1)} = B_0^{-1/2} B_k \qquad (-N \leqslant k \leqslant 0)$$

FOR $i = 1$ UNTIL N DO
BEGIN

$$K^{(i+1)} = -\overset{*}{U}_1^{(i)} [\overset{*}{V}_0^{(i)}]^{-1}$$

IN PARALLEL DO
BEGIN

$$P_{i+1} = I - K^{(i+1)} [K^{(i+1)}]^t$$

$$Q_{i+1} = I - [K^{(i+1)}]^t K^{(i+1)}$$

END IN PARALLEL DO;

IN PARALLEL FOR $-N \leqslant K \leqslant -1$ DO BEGIN

$$\overset{*}{V}_k^{(i+1)} = P_{i+1}^{-1/2} [\overset{*}{V}_k^{(i)} + K^{(i+1)} \overset{*}{U}_{k+1}^{(i)}]$$

$$\overset{*}{V}_k^{(i+1)} = P_{i+1}^{-1/2} [\overset{*}{V}_k^{(i)} + K^{(i+1)} \overset{*}{U}_{k+1}^{(i)}]$$

END IN PARALLEL DO;
OUTPUT;
END FOR LOOP;
END NORMALIZED PARALLEL LEVINSON ALGORITHM.

ACKNOWLEDGMENT

The authors wish to thank Prof. T. Kailath at Stanford University, Stanford, CA, for many very valuable discussions, and the reviewers for bringing [26] and [28] to our attention and for many helpful comments.

REFERENCES

[1] Sun-Yuan Kung and Yu Hen Hu, A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz System, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 31, no. 1, Feburary 1983, pp. 66-76.

[2] L. S. Haynes, R. L. Lau, D. P. Siewiorek, and D. W. Mizell, "A survey of highly parallel computing," *IEEE Computer*, vol. 15, pp. 9-26, Jan. 1982.

[3] D. Heller, "A survey of parallel algorithms in numerical linear algebra," *SIAM Rev.*, vol. 20, pp. 740-777, Oct. 1978.

[4] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. Conf. VLSI*, California Inst. Technol., Pasadena, Jan. 1979, pp. 65-90.

[5] S. Y. Kung, R. J. Gal-Ezer, and K. S. Arun, "Wavefront array processor: Architecture, language and applications," in *Proc. Conf. Adv. Res. in VLSI*, Massachusetts Inst. Technol., Cambridge, Jan. 1982.

[6] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront array processor: Language, architecture, and applications," *IEEE Trans. Comput.*, Nov. 1982.

[7] S. Y. Kung, "VLSI array processor for signal processing," presented at the M.I.T. Conf. Adv. Res. on IC, Cambridge, MA, 1980.

[8] O. Grenander and G. Szego, *Toeplitz Forms and Their Applications.* Berkeley, CA: Univ. California Press, 1958.

[9] A. V. Oppenheim, Ed., *Applications of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[10] S. S. Haykin, Ed., *Nonlinear Methods of Spectral Analysis.* New York: Springer-Verlag, 1979.

[11] S. M. Kay and S. L. Marple, Jr., "Spectrum analysis–A modern perspective," *Proc. IEEE*, vol. 69, pp. 1380-1498, Nov. 1981.

[12] N. Levinson, "The Wiener RMS (root-mean-square) error criterion in filter design and prediction," *J. Math. Phys.*, vol. 25, pp 261-278, Jan. 1947.

[13] T. Kailath, "A view of three decades of linear filtering theory," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 145-181, Mar. 1974.

[14] A. K. Jain, "Fast inversion of banded Toeplitz matrices by circular decomposition," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-26, pp. 121-126, Apr. 1978.

[15] M. Morf, "Doubling algorithm for Toeplitz and related equation," in *Proc. ICCASP 80*, Denver, CO, Apr. 1980, pp. 954-959.

[16] —, "Fast algorithms for multivariable systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1974.

[17] S. Y. Kung and Y. H. Hu, "Fast and parallel algorithms for solving Toeplitz systems," presented at the Int. Symp. Mini and Microcomputers in Control and Measurement, San Francisco, CA, May 1981.

[18] J. Durbin, "The filtering of time series models," *Rev. Int. Statist. Inst.*, vol. 28, pp. 233-244, 1960.

[19] W. F. Trench, "An algorithm for inversion of finite Toeplitz matrices," *J. SIAM*, vol. 12, pp. 515-522, 1964.

[20] S. Zohar, "The algorithm of W. F. Trench," *J. Ass. Comput. Mach.*, vol. 16, no. 4, pp. 592-601, 1969.

[21] P. Whittle, "The analysis of multiple stationary time series," *J. Royal Statist. Soc.*, ser. b, vol. 15, pp. 125-139, 1953.

[22] R. Wiggins and E. A. Robinson, "Recursive solution to the multichannel filtering problems," *J. Geophys. Rev.*, vol. 70, no. 8, pp. 1885-1891, 1965.

[23] H. Akaike, "Block Toeplitz inversion," *SIAM J. Appl. Math.*, vol. 24, pp. 234-241, Mar. 1975.

[24] I. Schur, "Uber Potenzreihen die in Innern des Einheitskreises Beschrankt Sind," *J. Reine Angewandte Mathematik*, vol. 147, 1917, pp. 205-232.

[25] P. Dewilde, A. Vieira, and T. Kailath, "On a generalized Szego-Levinson realization algorithm for optimal linear predictors based on a network synthesis approach," *IEEE Trans. Circuits Syst.*, vol. CAS-25, 1978, pp. 663-675.

[26] E. H. Bareiss, "Numerical solution of linear equations with Toeplitz and vector Toeplitz matrices," *Numer. Math.*, vol. 13, pp. 404-424, 1969.

[27] J. Rissanen, "Algorithms for triangular decomposition of block Hankel and Toeplitz matrices with applications to factoring positive matrix polynomials," *Math. Comp.*, vol. 27, pp. 147-159, Jan. 1973.

[28] R. K. Brouwer, "Particular methods for solving particular systems of linear equations," M. Phil. thesis, Dep. Comput. Sci., Univ. Waterloo, Waterloo, Ont., Canada, July 1971.

[29] S. Y. Kung, "Matrix data flow in language for matrix operation dedicated array processors," in *Proc. ECCTD 1981*, The Hague, The Netherlands, Aug. 1981, pp. 393-398.

[30] S. Y. Kung, K. S. Arun, D. V. Bhaskar Rao, and Y. H. Hu, "A matrix data flow languate/architecture for parallel matrix operations based on computational wavefront concept," in *Proc. Carnegie-Mellon Univ. Conf. VLSI Syst. Computations*, Oct. 1981, pp. 226-234 (published by Comput. Sci. Press).

[31] S. Y. Kung *et al.*, "Highly parallel modern signal processing," Univ. Southern California, Los Angeles, USC-SRO Rep. 1, Oct. 1981.

[32] G. Cybenko, "The numerical stability of the Levinson-Durbin algorithm for Toeplitz systems of equations," *SIAM J. Sci. Statist. Comput.*, vol. 1, Sept. 1980.

[33] I. C. Gohberg and I. A. Fel'dman, *Convolution Equations and Projection Methods for Their Solutions* (transl. of Math. Monograph, vol. 41, Amer. Math. Soc., Providence, RI, 1974).

[34] J. G. Nash, S. Hansen, and G. R. Nudd, "VLSI processor array for matrix manipulation," in *VLSI System and Computations*, H. T. Kung. Ed. Baltimore, MD: Comput. Sci. Press, 1981, pp. 367-373.

[35] J. G. Nash and G. R. Nudd, "Concurrent VLSI architectures for two dimensional signal processing systems," presented at the USC workshop on VLSI and Modern Signal Processing, Los Angeles, CA, Nov. 1-3, 1982. The full paper appears in *VLSI and Modern Signal Processing*, S. Y. Kung *et al.*, Eds. Englewood Cliffs, NJ: Prentice-Hall.

[36] A. C. G. Vieara, "Matrix orthogonal polynomials, with applications to autoregressive modeling and ladder forms," Ph.D. dissertation, Stanford Univ., Stanford, CA, Dec. 1977.

Sun-Yuan Kung is with Dept. Electrical Engineering, Princeton University, Princeton, NJ 08540, USA. Email: kung@princeton.edu .

Yu Hen Hu is with Dept. Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706, USA. Email: hu@engr.wisc.edu .

A ROTATION METHOD FOR COMPUTING THE QR-DECOMPOSITION*

FRANKLIN T. LUK†

Abstract. A parallel method for computing the QR-decomposition of an $n \times n$ matrix is proposed. It requires $O(n^2)$ processors and $O(n)$ units of time. The method can be extended to handle an $m \times n$ matrix ($m \geqq n$). The requirements then become $O(n^2)$ processors and $O(m)$ time.

Key words. QR-decomposition, plane rotations, systolic arrays, real-time computation, VLSI

AMS(MOS) subject classifications. 65F30-68A10

1. Introduction.

Let $A \in R^{n \times n}$ and

$$A = QR \tag{1.1}$$

(Q orthogonal, R upper triangular) be its QR-decomposition (QRD). The sequential computation of this decomposition requires time $O(n^3)$. Often, the QRD of $A \in R^{m \times n}$ ($m \geqq n$) is desired; the required time becomes $O(mn^2)$. For real-time signal processing (cf. Bromley and Speiser [6]) fast parallel algorithms are needed and various methods [1], [2], [8], [9], [10], [12] (most of which are applicable only to square matrices) have been proposed in the literature. Ahmed, Delosme and Morf [1], Bojanczyk, Brent and Kung [2] and Gentleman and Kung [8] all use Givens rotations and a triangular array of $O(n^2)$ processors. Both [1] and [2] store Q (in product form) in the array and propagate R, whereas [8] stores R and propagates Q. The decomposition is computable in time $O(n)$. The technique in [8] can be applied to an $m \times n$ matrix, and it will use time $O(m)$. Heller-Ipsen [9] and Johnsson [10] both consider a banded matrix A, say with bandwidth w. Based on the Givens rotations, the technique of Heller and Ipsen requires time $O(n)$ and a rectangular array of wq processors, where q equals the number of subdiagonals of A. Johnsson discusses a parallel implementation of the Householder transformations. He uses w processors and $O(nw)$ units of time. Sameh [12] considers an $m \times n$ matrix and a ring of p processors. He describes procedures based on the Givens and the Householder transformations; his algorithms require time $O(mn^2/p)$.

The parallel algorithms of Brent, Luk and Van Loan [4], [5] for computing the ordinary and the generalized singular value decompositions may require a preliminary QRD step. However, the mesh-connected multiprocessor arrays in [4], [5] are very different from the QR-arrays in [1], [2], [8], [9], [10], [12] and the interfacing of different arrays can be a serious problem. In this paper we present a rotation method that computes the QRD using a mesh-connected processor array. Our idea is to determine the QRD of an $n \times n$ matrix by computing in parallel $\lfloor n/2 \rfloor$ two-by-two QRDs. This strategy of decomposing an n-by-n problem into $\lfloor n/2 \rfloor$ two-by-two subproblems has been used successfully by Brent and Luk [3] for the symmetric eigenvalue decomposition, by Brent, Luk and Van Loan [4] for the singular value decomposition, and by Stewart [13] for the Schur decomposition. The strategy in [3], [4] is to divide an $n \times n$ matrix into blocks of 2×2 submatrices and to assign one

* Received by the editors June 11, 1984, and in final form March 8, 1985. This work was supported in part by the National Science Foundation under grant MCS-8213718 and by the Office of Naval Research under contract N00014-85-K-0074.

† School of Electrical Engineering, Cornell University, Ithaca, New York 14853.

processor to each block, resulting in a mesh-connected grid of $(\lceil n/2 \rceil)^2$ processors. The Schur decomposition array in [13] consists of two computational networks, each one quite similar to the multiprocessor array in [3], [4]. A total of approximately $n^2/2$ processors are needed (see § 2 for a precise count). However, if we assign two nodes (one from each network) to a processor, we can simulate this complex array using a mesh-connected grid of processors (cf. O'Leary-Stewart [11]). Our QRD algorithm requires the Schur decomposition array, hence $O(n^2)$ processors.

In § 2 we present our new algorithm and prove that it always converges after $2n$ time steps. The algorithm is extended to handle an $m \times n$ $(m \geqq n)$ matrix in § 3. The requirements become $O(n^2)$ processors and $O(m)$ time.

2. The algorithm. We parallelize the computations by simultaneously triangularizing $\lfloor n/2 \rfloor$ two-by-two submatrices of $A \in R^{n \times n}$. Consider the basic transformation: a QRD with column pivoting is computed of the 2×2 matrix

$$B = \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix}.$$

We get

$$\hat{B} \equiv JB\Pi = \begin{pmatrix} \hat{a}_{ii} & \hat{a}_{ij} \\ 0 & \hat{a}_{jj} \end{pmatrix},$$

where

$$J = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \quad \text{and} \quad \Pi = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The rotation parameters are calculated using the formulae:

$$h = \sqrt{a_{ij}^2 + a_{jj}^2}, \quad c = a_{ij}/h, \quad s = a_{jj}/h.$$

The full transformation on A is defined by

$$T_{ij} : A \to \hat{A} \equiv J_{ij} A \Pi_{ij}, \tag{2.1}$$

where J_{ij} denotes a plane rotation and Π_{ij} a permutation, both in the (i, j)-plane. The transformation T_{ij} will annihilate the (j, i)-element.

Our new algorithm uses an "odd-even" ordering of Stewart [13]. His ordering is amply illustrated by the $n = 8$ case:

$$(i, j) = (1, 2), (3, 4), (5, 6), (7, 8), (2, 3), (4, 5), (6, 7).$$

Many results in this section come from [13] and an interested reader should consult that fine paper. Besides a parallel implementation, the "odd-even" ordering preserves the triangular structure of a given matrix (see Lemma 5). A bonus (unimportant here) of the ordering is that the sum of squares of the strictly lower triangular elements will decrease. More precisely, define

$$\sigma(A) \equiv \sum_{p>q} |a_{pq}|^2.$$

The transformation $T_{i,i+1}$ will produce a matrix $\hat{A}$ satisfying

$$\sigma(\hat{A}) = \sigma(A) - |a_{i+1,i}|^2.$$

Our procedure thus shares with other Jacobi methods (cf. [3], [4], [13]) the property that it drives the matrix to the desired form. An important difference is that our

algorithm is finite: it converges after $2n$ time steps (see Theorem 2), where one time step is defined as the time required to do a transformation T_{ij}. The orthogonal matrix Q is readily computable through accumulating the plane rotations. We present our new algorithm.

ALGORITHM QRD.
  $Q \leftarrow I$;
  for $t = 1, 2, \cdots, n$ do
    for $i = 1, 3, \cdots$ (i odd), $2, 4, \cdots$ (i even) do
      begin
      $A \leftarrow J_{i,i+1} A \Pi_{i,i+1}$;
      $Q \leftarrow Q J_{i,i+1}^T$
      end.

The column pivotings are essential. For example, Algorithm QRD without pivoting stagnates on a matrix with a zero subdiagonal:

$$A = \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ \times & 0 & \times & \times \\ \times & \times & 0 & \times \end{pmatrix}.$$

Interestingly, the pivotings nullify one another after $2n$ steps.

LEMMA 1. *All columns of A return to their original positions after $2n$ time steps.*

Proof. A column vector moves forwards (backwards) after each time step until it gets to the first (last) position, where it stays for one step. It then reverses direction and moves again. The required number of steps for all columns to return to their initial positions equals the sum of $2n-2$ (for moving) and 2 (for the rest periods at the two ends). □

Fig. 1 (cf. [13, Fig. 4.1]) exhibits these interchanges for the case $n = 6$. The numbers to the side are time steps, and the six numbers following them mark the positions of the original columns. A dash between two elements indicates an interchange that will take place at the next time step. We shall prove that the matrix is triangularized after at most $2n-2$ time steps. First, we introduce a notation and define a property indicating that a column vector is in "upper triangular" form.

Notation. Let $A^{(0)} = A$ and denote by $A^{(t)}$ the matrix A *after* time step t. Set also

$$A^{(t)} \equiv (a_1^{(t)}, \cdots, a_n^{(t)}) \equiv (a_{ij}^{(t)}).$$

0. 1-2 3-4 5-6
1. 2 1-4 3-6 5
2. 2-4 1-6 3-5
3. 4 2-6 1-5 3
4. 4-6 2-5 1-3
5. 6 4-5 2-3 1
6. 6-5 4-3 2-1
7. 5 6-3 4-1 2
8. 5-3 6-1 4-2
9. 3 5-1 6-2 4
10. 3-1 5-2 6-4
11. 1 3-2 5-4 6
12. 1-2 3-4 5-6

FIG. 1. *Positions of original columns after each time step.*

DEFINITION. We say $a_i^{(t)} \in U$ (the ith column is in "upper-triangular" form) if elements $a_{i+1,i}^{(t)}, \cdots, a_{n,i}^{(t)}$ all equal 0. □

Let us state and prove three lemmas.

LEMMA 2. *If $T_{n-1,n}$ is made at time step t, then $a_{n-1-i}^{(t+i)} \in U$, for $i = 0, 1, \cdots, n-2$.*

Proof. The transformation $T_{n-1,n}$ annihilates the $(n, n-1)$-element. So $a_{n-1}^{(t)} \in U$. Now use induction. At time step $t+j$ ($j \geqq 0$) we perform $T_{n-1-j,n-j}$ so that column $a_{n-1-j}^{(t+j)} \in U$. At the next time step, the rotation in $T_{n-2-j,n-1-j}$ will create a new zero in the $(n-1-j, n-2-j)$-position, and pivoting will bring zeros to the other subdiagonal positions of column $n-2-j$ from column $n-1-j$. Hence $a_{n-2-j}^{(t+j+1)} \in U$. □

LEMMA 3. *If $T_{n-1,n}$ is made at time step t, then $a_{2i-1}^{(t+n-2)} \in U$ for $i = 1, 2, \cdots, \lfloor n/2 \rfloor$.*

Proof. We perform the transformation $T_{n-1,n}$ at time steps $t, t+2, t+4, \cdots$. Hence $a_{n-1}^{(t+2j)} \in U$, for $j = 0, 1, \cdots$. Apply Lemma 2 to each of these vectors. □

LEMMA 4. *If $a_1^{(t)}, \cdots, a_i^{(t)}, a_{i+2}^{(t)}$ are all in U and $T_{i+1,i+2}$ is made at time step $t+1$, then $a_1^{(t+1)}, \cdots, a_{i+1}^{(t+1)}$ all belong to U.*

Proof. We can check that our basic transformation (2.1) applied to any two consecutive columns in $\{a_1^{(t)}, \cdots, a_i^{(t)}\}$ will not move the resulting pair out of U. The transformation $T_{i+1,i+2}$ will put column $a_{i+1}^{(t+1)}$ in U (see the proof of Lemma 2). □

In words, after transformation $T_{n-1,n}$ the $(n-1)$-st column satisfies the "upper-triangular" property. The column then moves left and picks up appropriate zero elements along the way (Lemma 2). The same event recurs every two time steps. Eventually all odd-numbered columns are in "upper-triangular" form (Lemma 3). After that, the first two columns get in U, then the first three columns, and so on (Lemma 4). We thus need only to determine when the transformation $T_{n-1,n}$ first occurs to compute the time at which the matrix becomes triangularized.

THEOREM 1. *The matrix $A^{(2n-3)}(A^{(2n-2)})$ is upper triangular for n even (n odd).*

Proof. For n even (n odd), we do the transformation $T_{n-1,n}$ at time step 1 (step 2). After $n-2$ additional time steps, all odd-numbered columns are in U (Lemma 3). At the next time step, T_{23} is made. We need $n-2$ time steps for columns $2, 3, \cdots, n-1$ to get in U (Lemma 4). □

It is clear now why the pivot block has been restricted to contiguous elements.

LEMMA 5. *Let $A^{(t)}$ be upper triangular. Then $A^{(t+1)}$ stays upper triangular.*

Proof. Apply $T_{i,i+1}$ ($1 \leqq i < n$) to $A^{(t)}$. Columns $a_i^{(t+1)}$, $a_{i+1}^{(t+1)}$ still belong to U. □

Since each column of A returns to its original position after $2n$ steps, we have proved our principal result.

THEOREM 2. *Algorithm* QRD *computes a* QR*-factorization of A after $2n$ steps.*

Figure 2 shows how a 6×6 matrix is triangularized after 9 steps. Steps 10 to 12 are necessary to return all columns to their original positions.

To implement the algorithm we associate a processor with each 2×2 block of four contiguous elements. The architecture is the same as the Schur decomposition array introduced in Stewart [13] and detailed in O'Leary-Stewart [11]. There are $(n^2+2n-6)/2$ processors for n even and $(n^2+2n-3)/2$ processors for n odd. Only nearest neighbor connections are required of the processors, since each needs only to receive rotations from some of its neighbors, apply them and pass them on to other neighbors. We do not assume broadcasting of the rotation parameters and so each cluster of rotations requires $(\lceil n/2 \rceil - 1)$ time steps to pass completely out of the matrix. Since the clusters follow each other at intervals of two time steps and the last ($2n$th) cluster begins at step $4n-1$ our algorithm requires a total of $(\lceil 9n/2 \rceil - 2)$ time steps. The rotations propagate through the matrix as shown in Fig. 3 [13].

As mentioned in § 1 we look for a new QRD algorithm to eliminate array interfacings in an SVD computation. All other quadratic QRD arrays [1], [2], [8] are

1.
× × × × × ×
0 × × × × ×
× × × × × ×
× × 0 × × ×
× × × × × ×
× × × × 0 ×

2.
× × × × × ×
× × × × × ×
× 0 × × × ×
× × × × × ×
× × × 0 × ×
× × × 0 × ×

3.
× × × × × ×
0 × × × × ×
× × × × × ×
× × 0 × × ×
× × 0 × × ×
× × 0 × 0 ×

4.
× × × × × ×
× × × × × ×
× 0 × × × ×
× 0 × × × ×
× 0 × 0 × ×
× 0 × 0 × ×

5.
× × × × × ×
0 × × × × ×
0 × × × × ×
0 × 0 × × ×
0 × 0 × × ×
0 × 0 × 0 ×

6.
× × × × × ×
0 × × × × ×
0̇ 0 × × × ×
0 0 × × × ×
0 0 × 0 × ×
0 0 × 0 × ×

7.
× × × × × ×
0 × × × × ×
0 0 × × × ×
0 0 0 × × ×
0 0 0 × × ×
0 0 0 × 0 ×

8.
× × × × × ×
0 × × × × ×
0 0 × × × ×
0 0 0 × × ×
0 0 0 0 × ×
0 0 0 0 × ×

9.
× × × × × ×
0 × × × × ×
0 0 × × × ×
0 0 0 × × ×
0 0 0 0 × ×
0 0 0 0 0 ×

10.
× × × × × ×
0 × × × × ×
0 0 × × × ×
0 0 0 × × ×
0 0 0 0 × ×
0 0 0 0 0 ×

11.
× × × × × ×
0 × × × × ×
0 0 × × × ×
0 0 0 × × ×
0 0 0 0 × ×
0 0 0 0 0 ×

12.
× × × × × ×
0 × × × × ×
0 0 × × × ×
0 0 0 × × ×
0 0 0 0 × ×
0 0 0 0 0 ×

FIG. 2. *The zero-nonzero structure after each time step.*

1.
1 1 × × × × × ×
1 1 × × × × × ×
× × 1 1 × × × ×
× × 1 1 × × × ×
× × × × 1 1 × ×
× × × × 1 1 × ×
× × × × × × 1 1
× × × × × × 1 1

2.
× × 1 1 × × × ×
× × 1 1 × × × ×
1 1 × × 1 1 × ×
1 1 × × 1 1 × ×
× × 1 1 × × 1 1
× × 1 1 × × 1 1
× × × × 1 1 × ×
× × × × 1 1 × ×

3.
× × × × 1 1 × ×
× 2 2 × 1 1 × ×
× 2 2 × × × 1 1
× × × 2 2 × 1 1
1 1 × 2 2 × × ×
1 1 × × × 2 2 ×
× × 1 1 × 2 2 ×
× × 1 1 × × × ×

4.
× 2 2 × × × 1 1
2 × × 2 2 × 1 1
2 × × 2 2 × × ×
× 2 2 × × 2 2 ×
× 2 2 × × 2 2 ×
× × × 2 2 × × 2
1 1 × 2 2 × × 2
1 1 × × × 2 2 ×

5.
3 3 × 2 2 × × ×
3 3 × × × 2 2 ×
× × 3 3 × 2 2 ×
2 × 3 3 × × × 2
2 × × × 3 3 × 2
× 2 2 × 3 3 × ×
× 2 2 × × × 3 3
× × × 2 2 × 3 3

6.
× × 3 3 × 2 2 ×
× × 3 3 × × × 2
3 3 × × 3 3 × 2
3 3 × × 3 3 × ×
× × 3 3 × × 3 3
2 × 3 3 × × 3 3
2 × × × 3 3 × ×
× 2 2 × 3 3 × ×

FIG. 3. *Propagations of the rotations.*

triangular structures composed of $n(n+1)/2$ processors (including the n delay registers in [1], [2]) and they require $3n-2$ time steps. Admittedly, Algorithm QRD needs a little more time and the array structure is slightly more complex. But the possibility of computing an SVD using just one programmable array of processors justifies the additional costs.

It is of independent interest to compare the methods in [1], [2], [8] with Algorithm QRD. The methods of Ahmed et al. [1] and of Gentleman-Kung [8] annihilate elements of A from top down by chess knight moves, while the method of Bojanczyk et al. [2] performs the Givens rotations from bottom up by "long" chess knight moves. These methods all require $3n-5$ stages, where one stage is defined to be a simultaneous application of disjoint plane rotations. Algorithm QRD requires $2n$ stages and creates zeros in a rather unusual manner. The precise way depends on whether n is even or odd and will be omitted. Figure 4 illustrates the three different orderings for $n=8$. Like other parallel Jacobi-type methods [3], [4], [13] Algorithm QRD cannot utilize any banded structure of A.

We complete our analysis by examining roundoff errors. The following lemma comes from Gentleman [7].

LEMMA 6. *If a sequence of plane rotations in a* QRD *scheme can be written as a sequence of s stages, then the final computed matrix obtained when this sequence of plane rotations is applied to a given matrix A will be the exact result of exact computations on a matrix whose difference from A is bounded in norm by* $\eta s(1+\eta)^{s-1}\|A\|_F$, *where* η *denotes a small multiple of the machine precision and* $\|\cdot\|_F$ *the Frobenius matrix norm.*

×	×	×	×	×	×	×	×
1	×	×	×	×	×	×	×
2	4	×	×	×	×	×	×
3	5	7	×	×	×	×	×
4	6	8	10	×	×	×	×
5	7	9	11	13	×	×	×
6	8	10	12	14	16	×	×
7	9	11	13	15	17	19	×

(a) Ahmed et al. [1] and Gentleman-Kung [8].

×	×	×	×	×	×	×	×
7	×	×	×	×	×	×	×
6	9	×	×	×	×	×	×
5	8	11	×	×	×	×	×
4	7	10	13	×	×	×	×
3	6	9	12	15	×	×	×
2	5	8	11	14	17	×	×
1	4	7	10	13	16	19	×

(b) Bojanczyk et al. [2].

×	×	×	×	×	×	×	×
15	×	×	×	×	×	×	×
14	16	×	×	×	×	×	×
13	15	11	×	×	×	×	×
12	14	10	16	×	×	×	×
11	13	9	15	7	×	×	×
10	12	8	14	6	16	×	×
9	11	7	13	5	15	3	×

(c) Algorithm QRD.

FIG. 4. *Three different orders of annihilations.*

We thus obtain a *better* error bound for Algorithm QRD ($s=2n$) than for the other three methods ($s=3n-5$). This result comes as a surprise since redundant zeros are created by our algorithm.

3. Rectangular matrices. In § 2 we consider only square matrices. If the matrix A has more rows than columns, i.e., $A \in R^{m\times n}$ with $m \geqq n$, then we may adopt one of two strategies. The first approach is to apply Algorithm QRD to the square matrix

$$\tilde{A} = (A|0) \in R^{m\times m}.$$

We get

$$\tilde{A} = Q\begin{pmatrix} R & 0 \\ 0 & 0 \end{pmatrix},$$

where $R \in R^{n\times n}$ and so

$$A = Q\begin{pmatrix} R \\ 0 \end{pmatrix}.$$

The procedure requires $O(m)$ time and $O(m^2)$ processors. Its advantage is that no additional hardware is needed, assuming that our processor array can handle $m \times m$ matrices. The problem is of course that this assumption may be wrong.

The second approach is appropriate for a very large m, particularly for $m \gg n$. For simplicity, assume that our array can handle $2n \times 2n$ matrices and that

$$k = m/n$$

is an integer. Partition the matrix in the form:

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{pmatrix},$$

where each block A_i is $n \times n$. We propose a procedure that eliminates the elements of A one block at a time.

ALGORITHM Block QRD.
Set $R_1 := A_k$;
For $i = 1, 2, \cdots, k-1$ do
Use Algorithm QRD to compute a $2n \times 2n$ QR-decomposition:

$$\begin{pmatrix} A_{k-i} & 0 \\ R_i & 0 \end{pmatrix} = Q_{i+1}\begin{pmatrix} R_{i+1} & 0 \\ 0 & 0 \end{pmatrix},$$

where $Q_{i+1} \in R^{2n\times 2n}$ and $R_{i+1} \in R^{n\times n}$.

We get the QR-decomposition of A from

$$Q = \begin{pmatrix} I_{k-2} & 0 \\ 0 & Q_2 \end{pmatrix}\begin{pmatrix} I_{k-3} & 0 & 0 \\ 0 & Q_3 & 0 \\ 0 & 0 & I_1 \end{pmatrix}\begin{pmatrix} I_{k-4} & 0 & 0 \\ 0 & Q_4 & 0 \\ 0 & 0 & I_2 \end{pmatrix}\cdots\begin{pmatrix} I_1 & 0 & 0 \\ 0 & Q_{k-1} & 0 \\ 0 & 0 & I_{k-3} \end{pmatrix}\begin{pmatrix} Q_k & 0 \\ 0 & I_{k-2} \end{pmatrix},$$

where I_j denotes the $jn \times jn$ identity matrix, and

$$R = \begin{pmatrix} R_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

The requirements are $O(n^2)$ processors and $O(m)$ time. Note that one may speed up the algorithm by using two or more processor arrays. With $\lfloor k/2 \rfloor$ arrays we need but $\lceil \log_2 k \rceil$ steps. Thus, excluding the costs of data input and output, the required time can be cut to $O(n \log k)$.

REFERENCES

[1] H. M. AHMED, J.-M. DELOSME AND M. MORF, *Highly concurrent computing structures for matrix arithmetic and signal processing*, Computer, 15 (Jan. 1982), pp. 65–82.

[2] A. BOJANCZYK, R. P. BRENT AND H. T. KUNG, *Numerically stable solution of dense systems of linear equations using mesh-connected processors*, this Journal, 5 (1984), pp. 95–104.

[3] R. P. BRENT AND F. T. LUK, *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays*, this Journal, 6 (1985), pp. 69–84.

[4] R. P. BRENT, F. T. LUK AND C. VAN LOAN, *Computation of the singular value decomposition using mesh-connected processors*, J. VLSI Computer Systems, 1 (1985), to appear.

[5] ———, *Computation of the generalized singular value decomposition using mesh-connected processors*, Proc. SPIE Vol. 431, Real Time Signal Processing, VI (1983), pp. 66–71.

[6] K. BROMLEY AND J. M. SPEISER, *Signal Processing Algorithms, Architectures, and Applications*, Tutorial 31, SPIE 27th Annual Internat. Tech. Symp., San Diego, Aug. 1983.

[7] W. M. GENTLEMAN, *Error analysis of QR decompositions by Givens transformations*, Lin. Alg. Applics., 10 (1975), pp. 189–197.

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

[9] D. E. HELLER AND I. C. F. IPSEN, *Systolic networks for orthogonal decompositions*, this Journal, 4 (1983), pp. 261–269.

[10] L. JOHNSSON, *A computational array for the* QR*-method*, Proc. Conf. Adv. Research in VLSI, P. Penfield, ed., Artech House, Inc., Dedham, MA, 1982, pp. 123–129.

[11] D. P. O'LEARY AND G. W. STEWART, *Data-flow algorithms for parallel matrix computations*, Tech. Report 1366, Computer Science Dept., Univ. Maryland, College Park, 1984.

[12] A. H. SAMEH, *Solving the linear least squares problem on a linear array of processors*, Proc. Purdue Workshop Algorithmically-Specialized Computer Organ., W. Lafayette, IN, Sept. 1982.

[13] G. W. STEWART, *A Jacobi-like algorithm for computing the Schur decomposition of a non-Hermitian matrix*, this Journal, 6 (1985), pp. 853–864.

A Novel Algorithm and Architecture for Adaptive Digital Beamforming

CHRISTOPHER R. WARD, PHILIP J. HARGRAVE, AND JOHN G. McWHIRTER

Abstract—A novel algorithm and architecture are described which have specific application to high performance, digital, adaptive beamforming. It is shown how a simple, linearly constrained adaptive combiner forms the basis for a wide range of adaptive antenna subsystems. The function of such an adaptive combiner is formulated as a recursive least squares minimization operation and the corresponding weight vector is obtained by means of the $Q-R$ decomposition algorithm using Givens rotations. An efficient pipelined architecture to implement this algorithm is also described. It takes the form of a triangular systolic/wavefront array and has many desirable features for very large scale integration (VLSI) system design.

I. Introduction

THE OBJECTIVE of an adaptive antenna is to select a set of amplitude and phase weights with which to combine the outputs from the elements in an array so as to produce a far-field pattern that, in some sense, optimizes the reception of a desired signal. The substantial improvements in system antijam performance offered by this form of array processing has meant that it is now becoming an essential requirement for many military radar, communications and navigation systems.

The key components of an adaptive antenna system are illustrated in Fig. 1(b). The amplitude and phase weights are selected by a beampattern controller that continuously updates them in response to the element outputs. In some systems the output from the beamformer is also monitored to provide a feedback control. In all cases the resulting array beampattern is continuously adjusted to ensure cancellation of interference and jamming sources.

The most commonly employed technique for deriving the adaptive weight vector uses a closed loop gradient descent algorithm where the weight updates are derived from estimates of the correlation between the signal in each channel and the summed output of the array. This process can be implemented in an analog fashion using correlation loops [1] or digitally in the form of the Widrow least mean square (LMS) algorithm [2]. The value of this approach should not be underestimated. Gradient descent algorithms are very cost-effective and extremely robust but unfortunately they are not suitable for all applications. The major problem with an adaptive beamformer based on a gradient descent process is one of poor convergence for a broad dynamic range signal environment. This constitutes a fundamental limitation for many modern systems where features such as improved antenna platform dynamics (in the tactical aircraft environment, for example), sophisticated jamming threats and agile waveform structures (as produced by frequency hopped, spread spectrum formats) produce a requirement for adaptive systems having rapid convergence and high cancellation performance.

In recent years, there has been considerable interest in the application of direct solution or "open loop" techniques to adaptive antenna processing in order to accommodate these increasing demands. In the context of adaptive antenna processing, these algorithms have the advantage of requiring only minimal input data to accurately describe the external environment and provide an antenna pattern capable of suppressing a wide dynamic range of jamming signals. Open loop algorithms may be explained most concisely by expressing the adaptive process as a least squares minimization problem. In fact, the least squares algorithm may be considered to define the optimal path of adaptation.

In this paper we describe a novel algorithm and architecture for high performance, digital, adaptive beamforming. The adaptive combiner function is formulated as a recursive least squares minimization process and the corresponding set of linear equations is solved using the $Q-R$ decomposition algorithm. It is further shown how the $Q-R$ algorithm can be implemented using an efficient pipelined architecture in the form of a triangular systolic array.

II. Basic Configurations

The form of adaptive combiner which we consider in this paper is illustrated in Fig. 1(b). The inputs to the combiner take the form of a primary signal $y(t)$ and set of $N-1$ (complex) auxiliary signals $\mathbf{x}(t)$. The weight vector $\mathbf{w}$ is adjusted to minimize the power of the combined output signal which is given by

$$e(t)=\mathbf{x}^T(t)\mathbf{w}+y(t). \tag{1}$$

This type of adaptive linear combiner may be used in a wide range of adaptive antenna applications.

It is well known, for example, how it may be applied to adaptive sidelobe cancellation. In this case the primary signal constitutes the output from a main (high gain) antenna while the auxiliary signals are obtained from an array of $N-1$ auxiliary antennas. The adaptive combiner serves to modify the beampattern of the overall antenna system by directing deep nulls toward jamming waveforms received via the sidelobes of the main antenna.

Manuscript received June 5, 1985; revised October 4, 1985. This work was supported by the Procurement Executive, U.K. Ministry of Defence.

C. R. Ward and P. J. Hargrave are with Standard Telecommunication Laboratories Ltd., London Road, Harlow, Essex, U.K. CM17 9NA.

J. G. McWhirter is with Royal Signals and Radar Establishment, St. Andrews Road, Great Malvern, Worcestershire, U.K., WR14 3PS.

IEEE Log Number 8407019.

Reprinted from *IEEE Transactions on Antennas and Propagation*, pp. 338-346, March 1986.

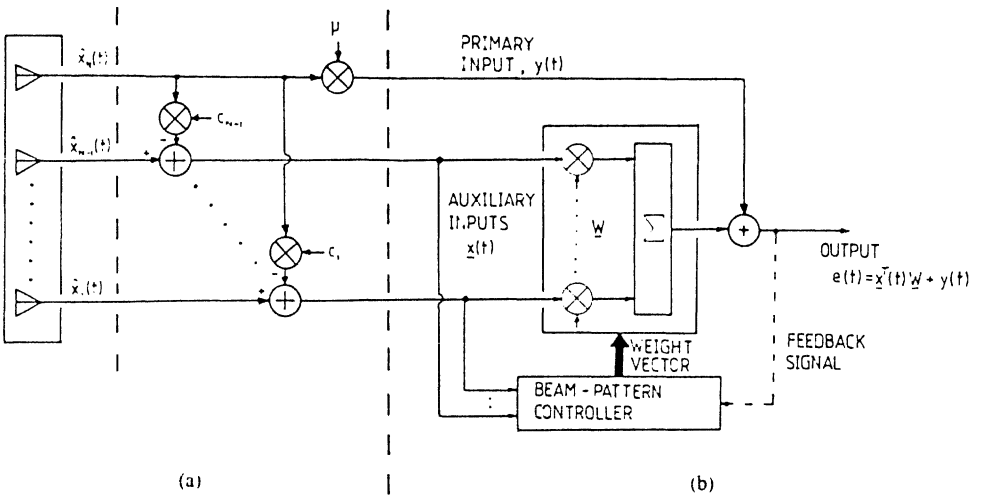


Fig. 1. Key components of an adaptive antenna processor. (a) Constraint preprocessor. (b) Adaptive combiner.

It is also well known how this form of adaptive combiner may be used in conjunction with a suitable reference signal to control a more general antenna array in which all of the elements are essentially equivalent. The reference signal, which is assumed to be correlated with the desired signal, provides the (negative) primary input to the combiner while the signals received by the antenna array provide the $N-1$ auxiliary inputs. In this case the weighted sum of the auxiliary inputs provides as close a match as possible to the reference signal and hence produces the desired output from the beamformer.

The basic combiner illustrated in Fig. 1(b) may also be used in the so-called "power inversion" mode which has particular application to communications. In this case the N antenna elements are assumed to be omnidirectional and of comparable gain. The received signals are fed into the combiner, one of them going to the primary channel and thus having its weight coefficient constrained to unity. The other $N-1$ signals enter the auxiliary channels with their adaptive weights initialized to zero and so, prior to adaptation, the overall beampattern is determined solely by the (omnidirectional) response of the "primary element." This "end-element clamped" configuration provides no inherent mechanism to inhibit the adaptive process from nulling the desired signal. However, the system is only allowed to adapt when the desired signal is known to be absent. When it is present, the weight vector is frozen thus allowing signal reception. This is referred to as the "power inversion" mode of operation because the differential interference powers received by the antenna elements are inverted by the combiner.

A particularly important application of adaptive antenna arrays requires the power of an N element combined signal

$$e(t)=\mathbf{x}^T(t)\mathbf{w} \tag{2}$$

to be minimized subject to a linear beam constraint of the form

$$\mathbf{c}^T\mathbf{w}=\mu. \tag{3}$$

This constraint ensures that the gain of the antenna array maintains a constant value μ in a given look direction specified by the vector $\mathbf{c}$. It is worth pointing out that the "end-element clamped" configuration described above constitutes a particularly simple form of linearly constrained process in which the constraint vector is given by

$$\mathbf{c}^T=(0,\ 0\text{------}0\ 1). \tag{4}$$

However, the incorporation of a general linear constraint is not so straightforward. A number of techniques have been proposed in the literature but in all cases the resulting implementation is extremely cumbersome. For example, Widrow [3] *et al.* suggested the injection of an artificial look direction signal into the antenna array receiver channels and introducing a corresponding reference signal into the adaptive process. This technique then requires an additional "slave processor" to apply the adapted weight vector. Frost [4] also showed how a general linear constraint could be incorporated into the adaptive process using projection operator techniques but the resulting algorithm is rather expensive in terms of computation.

We will now show how the general linear constraint in (3) may be incorporated in a much simpler way. It may be assumed without loss of generality that $c_N=1$ and so (3) may be expressed as

$$w_N=\mu-\mathbf{c}^T\mathbf{w} \tag{5}$$

where $\mathbf{c}$ and $\mathbf{w}$ denote the first $N-1$ elements of the vectors $\mathbf{c}$ and $\mathbf{w}$, respectively. Equations (2) and (3) can therefore be combined in the form

$$e(t)=(\hat{\mathbf{x}}^T(t)-x_N(t)\mathbf{c}^T)\mathbf{w}+\mu x_N(t) \tag{6}$$

where $\hat{\mathbf{x}}(t)$ denotes the vector of signals received by the first $N-1$ channels of the N element array. Since the constraint has been absorbed explicitly by eliminating the coefficient w_N and thereby removing the Nth degree of freedom, the power of the combined signal $e(t)$ may now be minimized with respect to the unconstrained $N-1$ element weight vector $\hat{\mathbf{w}}$. The form of (6) is therefore identical to that of (1) and so the output

power minimization may be carried out using the type of adaptive combiner illustrated in Fig. 1(a). The term $+\mu x_N(t)$ corresponds to the primary signal $y(t)$ while the transformed vector $\hat{\mathbf{x}}(t) - x_N(t)\mathbf{c}$ corresponds to the vector of auxiliary signals $\mathbf{x}(t)$. This input data transformation may be implemented using a simple linear preprocessor array of the type depicted in Fig. 1(a). In effect the Nth antenna signal is arbitrarily chosen as the primary combiner input. The corresponding antenna element is assumed to have omnidirectional coverage and the constraint preprocessor ensures that any signal which enters it from the required look direction is removed from the auxiliary channels before they enter the combiner. The adaptive nulling of this look direction signal is thus prevented.

From the discussion in this section it should be clear that the type of adaptive combiner illustrated in Fig. 1(b) has a wide range of applications in adaptive beamforming. In the remainder of this paper we concentrate on the development of a novel direct solution adaptive control technique which applies specifically to this basic configuration.

III. Least Squares Minimization

The function of the adaptive combiner in Fig. 1(b) will now be formulated in terms of least squares minimization. We denote the combined array output at time t_i by

$$e(t_i) = \mathbf{x}^T(t_i)\mathbf{w} + y(t_i). \tag{7}$$

where $\mathbf{x}(t_i)$ is the vector of (complex) auxiliary signals at time t_i and $y(t_i)$ is the corresponding sample of the (complex) primary signal. The residual signal power at time t_n is estimated by the quantity $E^2(n)$ where

$$E(n) = [\,|e(t_n)|^2 + \delta|e(t_{n-1})|^2 + \cdots + \delta^{n-1}|e(t_1)|^2\,]^{1/2}. \tag{8}$$

For the sake of generality this unnormalized estimator includes a simple "forget factor" δ which generates an exponential time window and localizes the averaging procedure.

Introducing a more compact matrix notation the estimator defined in (8) may be expressed in the form

$$E(n) = \|\mathbf{e}(n)\| \tag{9}$$

where

$$\mathbf{e}(n) = \mathbf{B}(n)\begin{pmatrix} e(t_1) \\ e(t_2) \\ \vdots \\ e(t_n) \end{pmatrix} \tag{10}$$

and

$$\mathbf{B}(n) = \text{diag}\,\{\beta^{n-1},\ \beta^{n-2},\ \cdots,\ 1\} \tag{11}$$

with $\beta^2 = \delta$.

Now from (7) it follows that the vector of residuals may be written in the form

$$\mathbf{e}(n) = \mathbf{X}(n)\mathbf{w} + \mathbf{y}(n) \tag{12}$$

where

$$\mathbf{X}(n) = \mathbf{B}(n)\begin{pmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_n) \end{pmatrix} \tag{13}$$

and

$$\mathbf{y}(n) = \mathbf{B}(n)\begin{pmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_n) \end{pmatrix}. \tag{14}$$

$\mathbf{X}(n)$ is simply the matrix of all data received by the weighted elements up to time t_n and $\mathbf{y}(n)$ is the corresponding vector of data in the primary or reference channel. The matrix $\mathbf{B}(n)$ takes account of the exponential time window and, for convenience, it has simply been absorbed into the definition of $\mathbf{e}(n)$, $\mathbf{y}(n)$ and $\mathbf{X}(n)$.

Determining the weight vector $\mathbf{w}(n)$ which minimizes $E^2(n)$ is referred to as least squares estimation [5]. The conventional approach to this problem is to derive an analytic expression for the complex gradient of the quantity $E^2(n)$ and determine the weight vector $\mathbf{w}(n)$ for which it vanishes. Now from (9) and (12) we have for the complex gradient

$$\nabla_w(E^2(n)) = 2\mathbf{X}^H(n)(\mathbf{X}(n)\mathbf{w} + \mathbf{y}(n)) \tag{15}$$

and setting the right side of this equation equal to zero leads to the well-known Wiener–Hopf equation:

$$\mathbf{M}(n)\mathbf{w}(n) + \rho(n) = 0 \tag{16}$$

where

$$\mathbf{M}(n) = \mathbf{X}^H(n)\mathbf{X}(n) \tag{17}$$

is the (estimated) covariance matrix and

$$\rho(n) = \mathbf{X}^H(n)\mathbf{y}(n) \tag{18}$$

is the estimated cross-correlation vector. The solution to (16) for nonsingular $\mathbf{M}(n)$ is clearly given by

$$\mathbf{w}(n) = -\mathbf{M}^{-1}(n)\rho(n) \tag{19}$$

and this provides an analytic expression for the optimum weight vector at time t_n.

In their classic paper, Reed, Mallet, and Brennan [6] suggested that the weight vector be obtained by solving (16) directly and showed that the problems of poor convergence associated with closed loop algorithms may be avoided in this way. This approach leads directly to the type of signal processing architecture which is illustrated schematically in Fig. 2. It comprises a number of distinct components—one to form and store the covariance matrix estimate, one to compute the solution of (16) and one to apply the resulting weight vector to the received signal data. These data must be stored in a suitable memory while the weight vector is being computed. The system also requires a number of high speed data communication buses and a sophisticated control unit to deliver the appropriate sequence of instructions to each

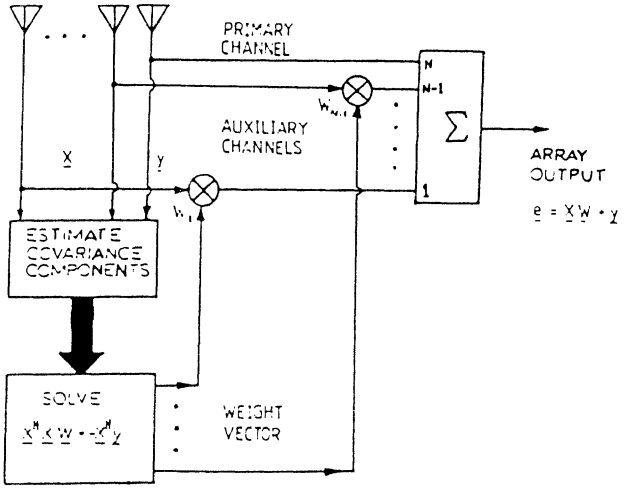


Fig. 2. Sample matrix inversion architecture.

component. This type of architecture is obviously complicated, extremely difficult to design and not very suitable for very large scale integration (VLSI).

Not only does the analytic solution given in (16) lead to a complicated circuit architecture, it is also very poor from the numerical point of view. The problem of solving a system of linear equations like those defined in (16) can be ill-conditioned and hence numerically unstable. Ill-conditioning occurs if the matrix has a very small determinant in which case the true solution can be subjected to large perturbations and still satisfy the equation quite accurately. The degree to which a system of linear equations is ill-conditioned is determined by the condition number of the coefficient matrix. The condition number of a matrix $\mathbf{A}$ is defined by

$$Cn(\mathbf{A}) = \lambda_1 / \lambda_p \tag{20}$$

where λ_1 and λ_p are the largest and smallest singular values, respectively, of the matrix $\mathbf{A}$. The larger $Cn(\mathbf{A})$, the more ill-conditioned is the system of equations. It follows from (17) that

$$Cn(\mathbf{M}(n)) = Cn(\mathbf{X}^H(n)\mathbf{X}(n)) = Cn^2(\mathbf{X}(n)) \tag{21}$$

and so the condition number of the estimated covariance matrix $\mathbf{M}(n)$ is much greater than that of the corresponding data matrix $\mathbf{X}(n)$. Any numerical algorithm which avoids forming the covariance matrix explicitly and operates directly on the data is likely to be much better conditioned.

IV. $Q-R$ Decomposition

An alternative approach to the least squares estimation problem which is particularly good in the numerical sense is that of orthogonal triangularization [7]. This is typified by the method known as $Q-R$ decomposition which we generalize here to the complex case. An $n \times n$ unitary[1] matrix $\mathbf{Q}(n)$ is generated such that

$$\mathbf{Q}(n)\mathbf{X}(n) = \begin{pmatrix} \mathbf{R}(n) \\ \hline \mathbf{0} \end{pmatrix} \tag{22}$$

[1] A matrix $\mathbf{A}$ is defined in this paper as being unitary if $\mathbf{A}^H\mathbf{A} = \mathbf{I}$. Matrix $\mathbf{A}$ is termed orthogonal if $\mathbf{A}^T\mathbf{A} = \mathbf{I}$.

where $\mathbf{R}(n)$ is an $(N-1)$ by $(N-1)$ upper triangular matrix. Then, since $\mathbf{Q}(n)$ is unitary we have

$$E(n) = \|\mathbf{e}(n)\| = \|\mathbf{Q}(n)\mathbf{e}(n)\| = \left\| \begin{pmatrix} \mathbf{R}(n) \\ \hline \mathbf{0} \end{pmatrix} \mathbf{w}(n) + \begin{pmatrix} \mathbf{u}(n) \\ \hline \mathbf{v}(n) \end{pmatrix} \right\| \tag{23}$$

where

$$\mathbf{u}(n) = \mathbf{P}(n)\mathbf{y}(n)$$

and

$$\mathbf{v}(n) = \mathbf{S}(n)\mathbf{y}(n). \tag{24}$$

$\mathbf{P}(n)$ and $\mathbf{S}(n)$ are simply the matrices of dimension $(N-1)$ by n and $(n-N+1)$ by n, respectively, which partition $\mathbf{Q}(n)$ in the form

$$\mathbf{Q}(n) = \begin{pmatrix} \mathbf{P}(n) \\ \hline \mathbf{S}(n) \end{pmatrix}. \tag{25}$$

It follows that the least squares weight vector $\mathbf{w}(n)$ must satisfy the equation

$$\mathbf{R}(n)\mathbf{w}(n) + \mathbf{u}(n) = \mathbf{0} \tag{26}$$

and hence

$$E(n) = \|\mathbf{v}(n)\|. \tag{27}$$

Since the matrix $\mathbf{R}(n)$ is upper triangular, (26) is much easier to solve than the Wiener–Hopf equation described earlier. The weight vector $\mathbf{w}(n)$ may be derived quite simply by a process of back-substitution. Equation (26) is also much better conditioned since the condition number of $\mathbf{R}(n)$ is given by

$$Cn(\mathbf{R}(n)) = Cn(\mathbf{Q}(n)\mathbf{X}(n)) = Cn(\mathbf{X}(n)). \tag{28}$$

This property follows directly from the fact that $\mathbf{Q}(n)$ is unitary.

Givens Rotations

The triangularization process may be carried out using either Householder transformations [7] or Givens rotations [8], [9], [10]. However the Givens rotation method is particularly suitable for the adaptive antenna application since it leads to a very efficient algorithm whereby the triangularization process is recursively updated as each new row of data enters the problem. A complex Givens rotation is an elementary transformation of the form

$$\begin{pmatrix} c & s^* \\ -s & c \end{pmatrix} \begin{pmatrix} 0 \cdots 0, & r_i \cdots r_k \cdots \\ 0 \cdots 0, & x_i \cdots x_k \cdots \end{pmatrix} = \begin{pmatrix} 0 \cdots 0, & r_i' \cdots r_k' \cdots \\ 0 \cdots 0, & 0 \cdots x_k' \cdots \end{pmatrix} \tag{29}$$

where the rotation coefficients, c and s, satisfy

$$-s \cdot r_i + c \cdot x_i = 0$$

$$s^*s + c^*c = 1$$

$$c^* = c \tag{30}$$

and are synonymous with the cosine and sine of an angular rotation in the multidimensional complex space. These relationships uniquely specify the rotation coefficients as

$$c = \frac{|r_i|}{\sqrt{x_i^* x_i + r_i^* r_i}} \tag{31}$$

and

$$s = \frac{x_i}{r_i} \cdot c. \tag{32}$$

A sequence of such elimination operations may be used to triangularize the matrix $\mathbf{X}(n)$ in the following recursive manner. Assume that the matrix $\mathbf{X}(n-1)$ has already been reduced to triangular form by the unitary transformation

$$\mathbf{Q}(n-1)\mathbf{X}(n-1) = \begin{pmatrix} \mathbf{R}(n-1) \\ \hline \mathbf{0} \end{pmatrix}. \tag{33}$$

Now define the unitary matrix

$$\bar{\mathbf{Q}}(n-1) = \left(\begin{array}{c|c} \mathbf{Q}(n-1) & \mathbf{0} \\ \hline \mathbf{0} & 1 \end{array}\right). \tag{34}$$

Clearly

$$\bar{\mathbf{Q}}(n-1)\mathbf{X}(n) = \bar{\mathbf{Q}}(n-1)\begin{pmatrix} \beta\mathbf{X}(n-1) \\ \hline \mathbf{x}^T(t_n) \end{pmatrix} = \begin{pmatrix} \beta\mathbf{R}(n-1) \\ \hline \mathbf{0} \\ \hline \mathbf{x}^T(t_n) \end{pmatrix} \tag{35}$$

and so the triangular process may be completed by the following sequence of operations. Rotate the $N-1$ element vector $\mathbf{x}^T(t_n)$ with the first row of $\beta\mathbf{R}(n-1)$ so that the leading element of $\mathbf{x}^T(t_n)$ is eliminated producing a reduced vector $\mathbf{x}^{T\prime}(t_n)$. The first row of $\mathbf{R}(n-1)$ will, of course, be modified in the process. Then rotate the $(N-2)$-element reduced vector $\mathbf{x}^{T\prime}(t_n)$ with the second row of $\beta\mathbf{R}(n-1)$ so that the leading element of $\mathbf{x}^{T\prime}(t_n)$ is eliminated and so on until every element has been eliminated. The resulting triangular matrix $\mathbf{R}(n)$ then corresponds to a complete triangularization of the matrix $\mathbf{X}(n)$ as defined in (22). The corresponding unitary matrix $\mathbf{Q}(n)$ is simply given by the recursive expression

$$\mathbf{Q}(n) = \hat{\mathbf{Q}}(n)\bar{\mathbf{Q}}(n-1) \tag{36}$$

where $\hat{\mathbf{Q}}(n)$ is a unitary matrix representing the sequence of Givens rotation operations described above, i.e.,

$$\hat{\mathbf{Q}}(n)\begin{pmatrix} \beta\mathbf{R}(n-1) \\ \hline \mathbf{0} \\ \hline \mathbf{x}^T(t_n) \end{pmatrix} = \begin{pmatrix} \mathbf{R}(n) \\ \hline \mathbf{0} \\ \hline \mathbf{0} \end{pmatrix}. \tag{37}$$

It is not difficult to deduce in addition that

$$\hat{\mathbf{Q}}(n)\begin{pmatrix} \beta\mathbf{u}(n-1) \\ \hline \beta v(n-1) \\ \hline y(t_n) \end{pmatrix} = \begin{pmatrix} \mathbf{u}(n) \\ \hline \beta v(n-1) \\ \hline \alpha(n) \end{pmatrix} = \begin{pmatrix} \mathbf{u}(n) \\ \hline v(n) \end{pmatrix} \tag{38}$$

and this shows how the vector $\mathbf{u}(n)$ can be updated recursively using the same sequence of Givens rotations. The least squares weight vector $\mathbf{w}(n)$ may then be derived by solving (26). The solution is not defined, of course, if $n < (N-1)$ but the recursive triangularization procedure may, nonetheless, be initialized by setting $\mathbf{R}(0) = \mathbf{0}$ and $\mathbf{u}(0) = \mathbf{0}$.

Direct Extraction of Residuals

In many least squares problems, and particularly in the adaptive antenna application, the main objective is to compute the least squares residual since the corresponding weight vector is not of direct interest. Previous work by McWhirter [11] has described a modified version of the $Q-R$ recursive least squares algorithm in which the least squares residual is produced directly at each stage of the recursive process without any need to derive the weight vector explicitly. The modified algorithm is much more robust since it avoids the solution of a system of linear equations which could be ill-conditioned. Furthermore, since the back-substitution circuit and the separate beamforming network are both eliminated, it offers a significant reduction in the complexity of the subsequent hardware implementation.

The derivation of this technique may be summarized as follows. Since

$$\mathbf{Q}(n)\mathbf{e}(n) = \begin{pmatrix} \mathbf{R}(n) \\ \hline \mathbf{0} \end{pmatrix}\mathbf{w}(n) + \begin{pmatrix} \mathbf{u}(n) \\ \hline \beta v(n-1) \\ \hline \alpha(n) \end{pmatrix} \tag{39}$$

and the weight vector must satisfy (26), it follows that the residual vector $\mathbf{e}(n)$ is given by

$$\mathbf{Q}(n)\mathbf{e}(n) = \hat{\mathbf{Q}}(n)\bar{\mathbf{Q}}(n-1)\mathbf{B}(n)\begin{pmatrix} e(t_1) \\ e(t_2) \\ \vdots \\ e(t_n) \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \hline \beta v(n-1) \\ \hline \alpha(n) \end{pmatrix}. \tag{40}$$

But $\hat{\mathbf{Q}}(n)$ is unitary and so we have

$$\bar{\mathbf{Q}}(n-1)\mathbf{B}(n)\begin{pmatrix} e(t_1) \\ e(t_2) \\ \vdots \\ e(t_n) \end{pmatrix} = \hat{\mathbf{Q}}^H(n)\begin{pmatrix} \mathbf{0} \\ \hline \beta v(n-1) \\ \hline \alpha(n) \end{pmatrix}. \tag{41}$$

Considering only the nth element of the vectors in (41) it is then possible to deduce that the current residual $e(t_n)$ is given by

$$e(t_n) = \gamma(n) \cdot \alpha(n) \tag{42}$$

where

$$\gamma(n) = \prod_{i=1}^{N-1} c_i \tag{43}$$

is the product of all cosine parameters generated during the sequence of Givens rotations used to eliminate the vector $\mathbf{x}^T(t_n)$. Equation (43) follows from the fact that $\hat{\mathbf{Q}}(n)$ is simply

the product of $(N-1)$ elementary rotations of the form

$$\hat{\mathbf{Q}}(n) = \hat{\mathbf{Q}}_{N-1}(n)\hat{\mathbf{Q}}_{N-2}(n) \cdots \hat{\mathbf{Q}}_1(n). \tag{44}$$

The ith elementary rotation is simply given by

$$\hat{\mathbf{Q}}_i(n) \begin{pmatrix} 1 & & & & & \\ & \ddots & & 0 & & \\ & & 1 & & & \\ & & & c_i & \cdots\cdots & s_i^* \\ & 0 & & \vdots & 1 \ddots & \vdots \\ & & & \vdots & \ddots\ 1 & \vdots \\ & & & -s_i & \cdots\cdots & c_i \end{pmatrix} \tag{45}$$

where the only nonzero off-diagonal elements occur in the ith row and the ith column. The result may be obtained by considering the effect of a reversed sequence of conjugate elementary rotations on the nth element of the right-hand vector in (41). The parameter $\gamma(n)$ may readily be computed during the recursive update of the matrix $\mathbf{R}(n)$ while the scalar quantity $\alpha(n)$ is available as a direct byproduct of the corresponding update for the vector $\mathbf{u}(n)$. The current residual $e(t_n)$ may therefore be evaluated in a very cost-effective manner.

In order to avoid complicating this discussion on adaptive beamforming, we have only considered the most direct form of the Givens rotation algorithm. However, it is important to point out that a very efficient "square root free" Givens algorithm has been derived by Gentleman. The square root free algorithm is equally applicable to the type of adaptive beamformer described in this paper and would almost certainly be used in any practical application. The essential details relating to its use may be found in [10] and [11].

Sensitivity to Arithmetic Precision

An important aspect of any signal processing algorithm is its sensitivity to limited arithmetic precision. We have recently carried out a detailed computer simulation study to compare the effect of limited precision on the performance of two adaptive cancellation processors—one based on sample matrix inversion and the other on the recursive $Q-R$ algorithm. The results indicate quite distinctly the improved performance offered by the data domain $Q-R$ method under conditions of finite resolution arithmetic compared with the sample matrix inversion technique. Fig. 3(a) shows a simple schematic representation of the two computer simulations. In both cases the sequence of data samples was generated and applied to the constraint preprocessor. The preprocessor applied a look direction constraint toward the desired signal and was implemented at full computer precision. The transformed data were then truncated to the chosen arithmetic precision, this word length being retained throughout subsequent $Q-R$ decomposition or sample matrix inversion computation. To ensure a fair comparison between the two basic approaches (i.e., covariance versus data domain), the effective sample matrix inversion solution was actually computed by performing a $Q-R$ decomposition on the covariance matrix estimate in (16). In both cases the back-substitution was performed at full computer precision.

Fig. 3(b) shows a typical comparative result which corresponds to a 24-bit floating point word length (16-bit mantissa and eight-bit exponent). Here, we plot the expected signal-to-noise ratio at the output of an eight-element array as an increasing number of data samples are used to compute the adapted weight vector solutions. In this example, we have modelled the effect of three equal power jamming signals received individually at levels of 0 dB relative to a thermal noise floor of -50 dB at the antenna array elements.

The complex envelope of each jammer was described by an independent, narrow-band Gaussian process. The model also incorporated a desired signal received by the array at a level of 15 dB above the thermal noise floor but approximately 40 dB below the total received jamming.

From Fig. 3(b) it can be seen that the initial rate of adaptation is extremely rapid for both sample matrix inversion and the data domain $Q-R$ algorithm. In both cases a good level of jamming cancellation is obtained after about ten to 20 data samples. However, with sample matrix inversion there is clear evidence of an unstable weight vector as reflected by extreme fluctuations in the adaptive response curve. In contrast, the data domain $Q-R$ method shows no sign of numerical instability and it is found that, over the timescale shown on these plots, the signal-to-noise ratio performance gets progressively better as the covariance information (in the form of the updated $\mathbf{R}$ matrix) gains more and more statistical accuracy with time.

For this scenario it was found that the sample matrix inversion technique required a floating point word length of 32 bits (24-bit mantissa and eight-bit exponent) to achieve comparable performance with the data domain $Q-R$ algorithm. It cannot be assumed, of course, that this word length would be sufficient for any arbitrary dynamic range environment. One should only conclude that the word length required by the sample matrix inversion approach will always be significantly greater than that for the data domain $Q-R$ method.

V. Systolic Array Implementation

Kung and Gentleman [12] have shown how the Givens rotation algorithm described above may be implemented in a very efficient pipelined manner using a triangular systolic array. The implementation of a five-channel adaptive beamforming network using this architecture is shown in Fig. 4. It may be considered to comprise three distinct sections—the basic triangular array labeled ABC, the right hand column of cells labeled DE and the final processing cell labeled F. The entire array is controlled by a single clock and comprises three types of processing cell. Each cell receives its input data from the directions indicated on one clock cycle, performs the specified function and delivers the appropriate output values to neighboring cells as indicated on the next clock cycle. Apart from the introduction of an extra parameter into the boundary cell, the function of the boundary and internal cells is precisely that required to implement the Givens rotations described above. Each cell within the basic triangular array stores one element of the recursively evolving triangular matrix $\mathbf{R}(n)$ which is initialized to zero at the outset of the least squares

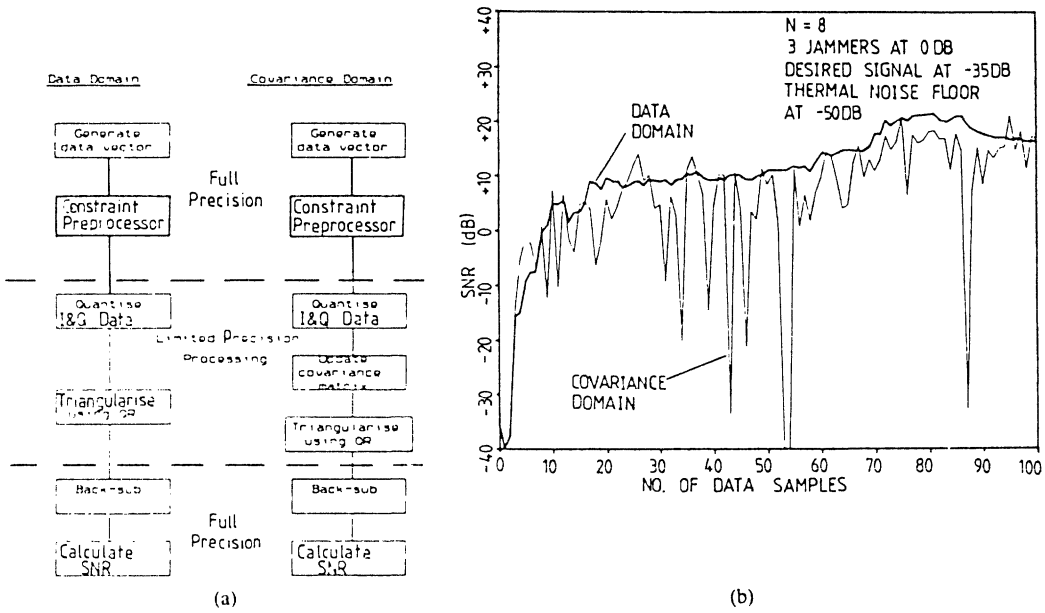


Fig. 3. Comparison of data and covariance domain algorithms. (a) Simulation flow diagrams. (b) Typical signal-to-noise ratio response.

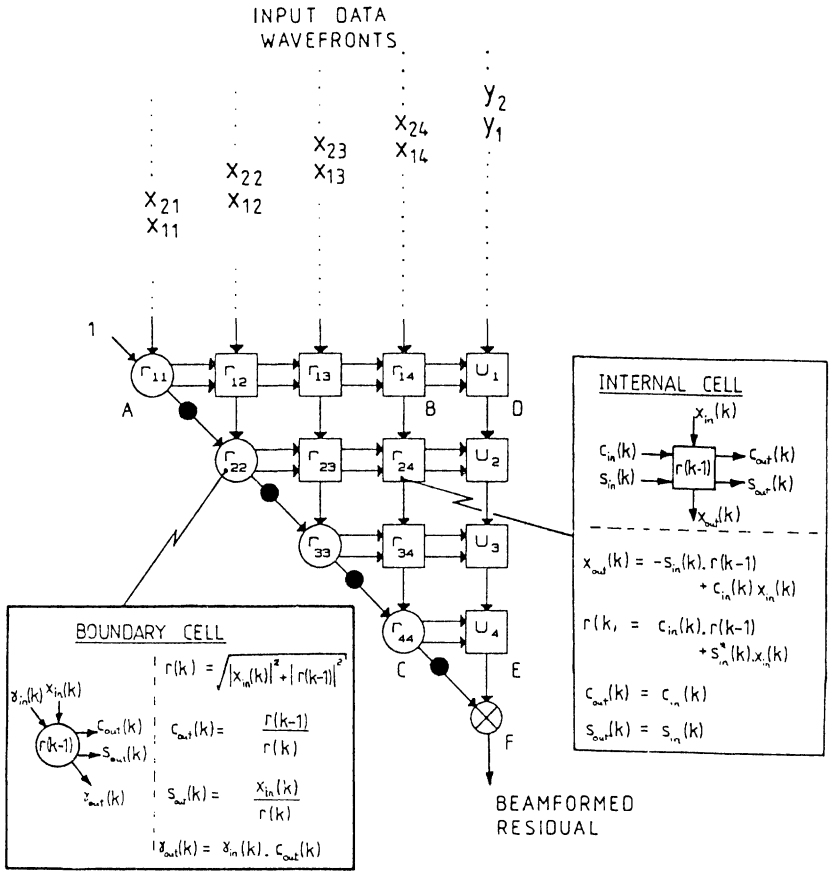


Fig. 4. Triangular systolic array for adaptive beamforming.

calculation and then updated every clock cycle. As a result of this initialization the value of r_{ii} within each boundary cell is entirely real. Cells in the right-hand column store one element of the evolving vector $\mathbf{u}(n)$ which is also initialized to zero and updated every clock cycle. Each row of cells within the array performs a basic Givens rotation between one row of the stored triangular matrix and a vector of data received from above so that the leading element of the received vector is eliminated as detailed in (29). The reduced data vector is then passed downwards through the array. This arrangement ensures that as each row $\mathbf{x}^T(t_n)$ of the matrix $\mathbf{X}$ moves down through the array it interacts with the previously stored

triangular matrix $\mathbf{R}(n-1)$ and undergoes the sequence of rotations $\hat{\mathbf{Q}}(n)$ described in the earlier analysis. All of its elements are thereby eliminated (one on each row of the array) and an updated triangular matrix $\mathbf{R}(n)$ is generated and stored in the process.

As each element of the vector $\mathbf{y}$ moves down through the right hand column of cells it undergoes the same sequence of Givens rotations interacting with the previously stored vector $\mathbf{u}(n-1)$ and generating an updated vector $\mathbf{u}(n)$ in the process. The resulting output, which emerges from the bottom cell in the right-hand column, is simply the value of the parameter $\alpha(n)$ in (42). The other value $\gamma(n)$ required for direct computation of the least squares residual $e(t_n)$ is generated recursively by the additional parameter γ which appears in the definition of the boundary cell function. The value of γ (initialized to one) is simply multiplied by the "cosine" parameter in each boundary cell and passed on to the boundary cell in the next row two clock cycles later. The extra delay, which is a direct consequence of the temporal data skew, may be achieved by using an additional storage element which is indicated by a black dot in Fig. 4 and would be incorporated within the boundary processor. The required value $\gamma(n)$ emerges from the final boundary cell and is simply multiplied by the corresponding output value $\alpha(n)$ to produce the desired residual. This operation takes place within the final processing cell F. A consequence of the highly pipelined nature of the systolic array and the need to impose a time-skew on the input data is the presence of an overall delay or latency in the system response. Each output residual $e(t_n)$ corresponds to a data vector whose first element was input to the network $2(N-1)$ clock periods previously.

The systolic array described in this section clearly exhibits many desirable properties such as regularity and local interconnections which render it comparatively simple to implement. Furthermore, the control overhead is extremely low since the processing cells operate synchronously and the only control required is a simple globally distributed clock. However, the need to distribute a common clock signal to every processor without incurring any appreciable clock skew is one possible disadvantage of the systolic array approach particularly in large multiprocessor systems. It is possible, however, to implement the same basic design as a wavefront array processor of the type proposed by S. Y. Kung *et al.* [13]. In a wavefront array processor, the required computation is distributed in exactly the same way over an array of elementary processors as it would be on the corresponding systolic array. Unlike its systolic counterpart, however, the wavefront array does not operate synchronously. Instead, the operation of each processor is controlled locally and depends on the necessary input data being available and on its previous outputs having been accepted by the appropriate neighboring processors. As a result, it is not necessary to impose a temporal skew on the data input to a wavefront processor. Instead the associated processing wavefront develops naturally within the array. In order to operate in the wavefront array mode, every processing element must incorporate some additional circuitry to implement a bidirectional handshake on each of its input/output links and thus ensure that the necessary communication protocol is observed. This represents an overhead which is not negligible but can easily be absorbed within the overall processing.

Obtaining the Weight Vector

It is worth pointing out that, as well as being capable of operating in the direct, beamforming mode, the triangular array in Fig. 4 can also be used in conjunction with some additional circuitry to compute the weight solution explicitly. The scheme which was originally proposed by Kung and Gentleman [12] uses the triangular systolic array in conjunction with a linear systolic array which solves for the weight vector by back-substitution. This method could clearly be used with the circuit in Fig. 4 by providing suitable means for extracting the triangular matrix $\mathbf{R}(n)$ from the array. However, the weight vector, if required, can be obtained in a much simpler way as a further byproduct of the direct residual extraction technique.

The method, which we refer to as "weight flushing" may be explained fairly simply as follows. As the nth data vector $\mathbf{x}(t_n)$ and the corresponding input $y(t_n)$ pass through the triangular array in Fig. 4 they update the parameters of the system from their state at time $n-1$ to the new state at time n. The vector $\mathbf{x}(t_n)$ also undergoes a simple linear projection with the implicit updated weight vector $\mathbf{w}(n)$ to produce the corresponding output residual

$$e(t_n) = \mathbf{x}^T(t_n)\mathbf{w}(n) + y(t_n). \tag{46}$$

Assume that the state of the system is subsequently "frozen" by preventing any further adaptation and define a simple $N-1$ element projection vector of the form

$$\boldsymbol{\phi}_i^T = (0 \cdots 010 \cdots 0) \tag{47}$$

with unit ith element. If the vector $\boldsymbol{\phi}_i$ is now input to the array as though it were another vector of auxiliary samples and the corresponding primary input is set equal to zero it follows from (46) that the associated output "residual" must be given by

$$\boldsymbol{\phi}_i^T \mathbf{w}(n) = w_i(n). \tag{48}$$

It is therefore possible to "flush" the entire weight vector $\mathbf{w}(n)$ out of the array by inputting to the $N-1$ auxiliary channels the sequence of vectors $\boldsymbol{\phi}_i$ $(i = 1, 2, \cdots, N-1)$ i.e., by inputting a simple unit diagonal matrix.

For the sake of brevity in this paper we have not explained in detail how the adaptive process may be "frozen" in practice. However, the technique is quite straightforward and may be implemented in a very direct manner. It is particularly simple when the square root free Givens rotation algorithm is being used.

VI. Conclusion

This paper has described a novel algorithm and associated systolic/wavefront array architecture for high performance, digital, adaptive beamforming. The adaptive beamformer enjoys all the desirable architectural features of a systolic or wavefront array. As each row of data moves down through the

array it is fully absorbed into the statistical estimation process, the triangular matrix $\mathbf{R}(n)$ is updated accordingly and the corresponding residual is produced automatically. The circuit architecture is greatly enhanced by avoiding the need to derive an explicit solution for the least squares weight vector $\mathbf{W}(n)$. This leads to a considerable reduction in the amount of computation and circuitry required since it is no longer necessary to clock out each triangular matrix $\mathbf{R}(n)$, carry out the back-substitution or form the vector product $\mathbf{x}^T(t_n)\ \mathbf{W}(n)$ explicitly.

The adaptive beamformer described in Sections IV and V is also based on a very stable and well-conditioned numerical algorithm. Indeed the method of $Q-R$ decomposition by Givens rotations is widely accepted as one of the very best techniques for solving linear least squares problems. However the final triangular linear system may, in general, be ill-conditioned and avoiding the back-substitution process also enhances the numerical properties of the adaptive combiner. In particular the systolic array implementation of the $Q-R$ algorithm produces the correct (zero) residual even if $n < (N - 1)$ and the matrix $\mathbf{X}$ is not of full rank. This sort of unconditional stability is most important in the design of real time signal processing systems.

As part of the United Kingdom's research program into advanced algorithms and architectures for adaptive antenna array signal processing, Standard Telecommunication Laboratories and the Royal Signals and Radar Establishment are developing jointly an experimental wavefront array processor. This digital processor will be configured primarily as an adaptive antenna test-bed and will have the ability to process six input channels of data in real-time. Each node of the wavefront array processor will be based on an existing digital signal processor chip and hence will provide a useful degree of programmability whilst maintaining a node throughput rate which will allow a comprehensive range of real-time tests and trials.

Eventually, the development of high performance processing nodes by VLSI design will permit the practical realization of such parallel processing architectures in extremely compact hardware form. In addition, the VLSI circuitry in conjunction with advanced technology will provide processing throughput rates far in excess of those obtainable by current DSP components and will therefore be matched to future wideband radar and communications applications.

Acknowledgment

The authors thank the Directors of Standard Telecommunication Laboratories Ltd. for permission to publish this paper.

References

[1] S. P. Applebaum, "Adaptive arrays," *IEEE Trans. Antennas Propagat.*, vol. AP-24, pp. 585-598, 1976.

[2] B. Widrow and J. M. McCool, "A comparison of adaptive algorithms based on the methods of steepest descent and random search," *IEEE Trans. Antennas Propagat.*, vol. AP-24, pp. 615-637, 1976.

[3] B. Widrow, P. E. Mantey, L. J. Griffiths, and B. B. Goode, "Adaptive antenna systems," *Proc. IEEE*, vol. 55, no. 12, pp. 2143-2159, Dec. 1967.

[4] O. L. Frost, "An algorithm for linearly constrained adaptive array processing," *Proc. IEEE*, vol. 60, pp. 661-675, 1971.

[5] G. L. Lawson and R. J. Hanson, *Solving Least-Squares Problems*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[6] I. S. Reed, J. D. Mallett, and L. E. Brennan, "Rapid convergence rate in adaptive arrays," *IEEE Trans. Aerospace Electron. Syst.*, vol. AES-10, pp. 853-863, 1974.

[7] G. H. Golub, "Numerical methods for solving linear least-squares problems," *Num. Math.*, no. 7, pp. 206-216, 1965.

[8] W. Givens, "Computation of plane unitary rotations transforming a general matrix to triangular form," *J. Soc. Ind. Appl. Math.*, no. 6, pp. 26-50, 1958.

[9] S. Hammarling, "A note on modifications to the Givens plane rotation," *J. Inst. Math. Appl.*, vol. 13, pp. 215-218, 1974.

[10] W. M. Gentleman, "Least-squares computations by Givens transformations without square-roots," *J. Inst. Math. Appl.*, vol. 12, pp. 329-336, 1973.

[11] J. G. McWhirter, "Recursive least-squares minimization using a systolic array," *Proc. SPIE*, 1983, p. 431, Real-Time Signal Processing VI, 2983.

[12] H. T. Kung and W. M. Gentleman, "Matrix triangularization by systolic arrays," *Proc. SPIE*, 1981, p. 298, Real-Time Signal Processing IV.

[13] S. Y. Kung, K. S. Arun, R. J. Gal-ezer, and D. V. Bhaskar Rao, "Wavefront array processor: Language, architecture and applications," *IEEE Trans. Comput.*, vol. C-31, no. 11, pp. 1054-1066, 1982.

Systolic array processor for MVDR beamforming

J.G. McWhirter
T.J. Shepherd

Indexing terms: Array processing, Adaptive antennas, Algorithms

Abstract: An efficient systolic array for computing the minimum variance distortionless response (MVDR) from an adaptive antenna array is described. It is fully pipelined and based on a numerically stable algorithm which requires $O(p^2 + Kp)$ arithmetic operations per sample time, where p is the number of antenna elements and K is the number of look direction constraints.

1 Introduction

In this paper, we describe a novel systolic array which can efficiently compute the minimum variance distortionless response (MVDR) from an array of p antenna receiver elements in a rapidly changing signal environment. The MVDR beamforming problem amounts to minimising, in a least squares sense, the combined output from an antenna array subject to K independent linear equality constraints, each of which corresponds to a chosen 'look direction'. The constraints are independent in the sense that, for each new vector of received data samples, it is necessary to compute the minimum array output subject to each constraint in turn. This involves the solution of K independent, but closely related, least squares minimisation problems.

In a previous publication, McWhirter and Shepherd [1] showed how a $p + 1 \times p + 1$ triangular systolic array of the type proposed by Gentleman and Kung [2] and adapted by McWhirter [3] could be applied to the problem of recursive least squares minimisation, subject to one or more simultaneous linear equality constraints. In effect, the top row or rows (one for each simultaneous constraint) of the triangular array are used to perform a constraint preprocessing operation. The remainder of the triangular array is used to perform a QR decomposition on the transformed data matrix produced by the 'constraint preprocessor'. The number of arithmetic operations performed by this array is $O((p + 1)^2)$ per sample time. Unfortunately, this type of systolic array is inefficient for computing the MVDR, which involves several independent constraints. As the constraint preprocessor will produce a different transformed data matrix for each of the constraints, the whole computation, including QR decomposition, must be repeated for each constraint, and the number of arithmetic operations required is $O(K(p + 1)^2)$ per sample time.

Paper 6490F (E16, E15, E11), first received 30th June and in revised form 25th October 1988

The authors are with the Royal Signals and Radar Establishment, St Andrews Road, Great Malvern, Worcs WR14 3PS, United Kingdom

In a recent paper, Bojanczyk and Luk [4] proposed the use of a different triangular systolic array for computing the MVDR. Their approach also requires a separate preprocessing operation to be applied to the received data vectors for each of the K independent constraints, and so it still requires $O(K(p + 1)^2)$ arithmetic operations per sample time. Compared to the constraint preprocessing technique of McWhirter and Shepherd, the method proposed by Bojanczyk and Luk for computing the MVDR requires more arithmetic operations to be performed and leads to a more complicated processor design. However, it involves much less data storage and is likely to have better numerical properties because the preprocessing operation is carried out using only orthogonal transformations.

A much more efficient algorithm for computing the MVDR has been developed by Schreiber [5]. His method only requires $O(p^2 + Kp)$ arithmetic operations per sample time, which corresponds to the minimum computational complexity; it is also known to have excellent numerical properties. Schreiber's algorithm involves a number of steps, each of which can be implemented efficiently on some form of systolic array. However, as pointed out by Bojanczyk and Luk [4], it seems to be difficult to realise the complete algorithm in a fully pipelined fashion as, for example, updating the Cholesky factor requires top-to-bottom processing, whereas the back substitution process requires bottom-to-top processing.

In this paper, we show how Schreiber's algorithm may be implemented very efficiently on a single systolic array by avoiding the need for an explicit back substitution processor. The resulting MVDR array is both fully efficient and fully pipelined. It is also shown how the initialisation stage required for Schreiber's algorithm may be carried out in a fully pipelined manner within the array.

2 Theory

The MVDR problem may be summarised as follows. At each sample time t_n, evaluate the *a posteriori* residuals

$$e^{(k)}(t_n) = \boldsymbol{x}^T(t_n)\boldsymbol{w}^{(k)}(n) \quad k = 1, 2, \ldots, K \tag{1}$$

where $\boldsymbol{x}(t_n)$ is the p-element vector of (complex) signal samples received by the array at time t_n, and $\boldsymbol{w}^{(k)}(n)$ is the p-element vector of (complex) weights which minimises the quantity

$$E^{(k)}(n) = \|\boldsymbol{e}^{(k)}(n)\| = \|\boldsymbol{X}(n)\boldsymbol{w}^{(k)}(n)\| \tag{2}$$

subject to a linear equality constraint of the form

$$\boldsymbol{c}^{(k)T}\boldsymbol{w}^{(k)}(n) = \mu^{(k)} \tag{3}$$

Reprinted with permission from *Proceedings of the IEE, Part F*, J. G. McWhirter and T. J. Shepherd, "Systolic Array Processor for MVDR Beamforming," Vol. 135, pp. 75-80, 1989. © Crown Copyright. Reprinted with the permission of the Controller of Her Majesty's Stationery Office.

In eqn. 2, we have assumed the notation

$$X(n) = B(n)\begin{bmatrix} x^T(t_1) \\ x^T(t_2) \\ \vdots \\ x^T(t_n) \end{bmatrix} \tag{4}$$

and

$$e^{(k)}(n) = B(n)\begin{bmatrix} e^{(k)}(t_1) \\ e^{(k)}(t_2) \\ \vdots \\ e^{(k)}(t_n) \end{bmatrix} \tag{5}$$

where the matrix

$$B(n) = \text{diag}\,\{\beta^{n-1}, \beta^{n-2}, \ldots, 1\} \tag{6}$$

has been included to represent the effect of applying an exponential 'forget' factor, which progressively scales down the statistical weight assigned to previously received data vectors in the least squares computation. $X(n)$ is simply the (weighted) matrix of all data received by the antenna array up to time t_n and $e^{(k)}(n)$ is the corresponding vector of least squares residuals for the kth look direction.

The solution to this constrained least squares minimisation problem is given by the well known formula

$$w^{(k)}(n) = \mu^{(k)} M^{-1}(n) c^{(k)*} / c^{(k)T} M^{-1}(n) c^{(k)*} \tag{7}$$

where $M(n)$ is the (weighted) covariance matrix defined by

$$M(n) = X^H(n) X(n) \tag{8}$$

Assuming that a QR decomposition has been carried out on the data matrix $X(n)$ so that

$$Q(n) X(n) = \begin{bmatrix} R(n) \\ 0 \end{bmatrix} \tag{9}$$

where $R(n)$ is a $p \times p$ upper triangular matrix, then it follows that

$$M(n) = R^H(n) R(n) \tag{10}$$

and so $R(n)$ is the Cholesky square root factor of the covariance matrix $M(n)$. Eqn. 7 may therefore be written in the form

$$\begin{aligned} w^{(k)}(n) &= \frac{\mu^{(k)} R^{-1}(n) R^{-H}(n) c^{(k)*}}{c^{(k)T} R^{-1}(n) R^{-H}(n) c^{(k)*}} \\ &= \frac{\mu^{(k)} R^{-1}(n) a^{(k)}(n)}{\| a^{(k)}(n) \|^2} \end{aligned} \tag{11}$$

where

$$a^{(k)}(n) = R^{-H}(n) c^{(k)*} \tag{12}$$

It follows that the *a posteriori* residual at time t_n is given by

$$e^{(k)}(t_n) = \frac{\mu^{(k)} \hat{e}^{(k)}(t_n)}{\| a^{(k)}(n) \|^2} \tag{13}$$

where

$$\begin{aligned} \hat{e}^{(k)}(t_n) &= x^T(t_n) R^{-1}(n) a^{(k)}(n) \\ &= b^T(n) a^{(k)}(n) \end{aligned} \tag{14}$$

and

$$b(n) = R^{-T}(n) x(t_n) \tag{15}$$

Gentleman and Kung [2] have shown how the QR decomposition of $X(n)$ may be implemented recursively on a triangular systolic array. The triangular matrix $R(n-1)$ is updated using an orthogonal transformation $\hat{Q}(n)$ of the form

$$\hat{Q}(n)\begin{bmatrix} \beta R(n-1) \\ 0 \\ x^T(t_n) \end{bmatrix} = \begin{bmatrix} R(n) \\ 0 \\ 0 \end{bmatrix} \tag{16}$$

where $\hat{Q}(n)$ represents a sequence of elementary Givens rotations used to annihilate each element of a new data vector $x^T(t_n)$ in turn. Schreiber's algorithm is based on the fact that the vector $a^{(k)}(n)$ can also be computed recursively. In the context of QR decomposition, this recursion may be derived as follows. From eqn. 12 it is clear that at time t_{n-1},

$$\begin{aligned} c^{(k)*} &= R^H(n-1) a^{(k)}(n-1) \\ &= \beta^{-2}[\beta R^H(n-1) \,|\, 0 \,|\, x^*(t_n)] \begin{bmatrix} \beta a^{(k)}(n-1) \\ \beta v^{(k)}(n-1) \\ 0 \end{bmatrix} \end{aligned} \tag{17}$$

where $v^{(k)}(n-1)$ is an arbitrary $n-p-1$ element vector. Now, as the matrix $\hat{Q}(n)$ is unitary, eqn. 17 may be expressed in the form

$$\begin{aligned} c^{(k)*} &= \beta^{-2}[\beta R^H(n-1) \,|\, 0 \,|\, x^*(t_n)] \\ &\quad \times \hat{Q}^H(n) \hat{Q}(n) \begin{bmatrix} \beta a^{(k)}(n-1) \\ \beta v^{(k)}(n-1) \\ 0 \end{bmatrix} \end{aligned} \tag{18}$$

and it follows from eqn. 16 that

$$c^{(k)*} = \beta^{-2}[R^H(n) \,|\, 0 \,|\, 0] \hat{Q}(n) \begin{bmatrix} \beta a^{(k)}(n-1) \\ \beta v^{(k)}(n-1) \\ 0 \end{bmatrix} \tag{19}$$

The structure of $\hat{Q}(n)$ is such that we may express

$$\hat{Q}(n)\begin{bmatrix} \beta a^{(k)}(n-1) \\ \beta v^{(k)}(n-1) \\ 0 \end{bmatrix} = \begin{bmatrix} d^{(k)}(n) \\ \beta v^{(k)}(n-1) \\ \alpha^{(k)}(n) \end{bmatrix} = \begin{bmatrix} d^{(k)}(n) \\ v^{(k)}(n) \end{bmatrix} \tag{20}$$

and so from eqns. 19 and 20 we obtain

$$c^{(k)*} = \beta^{-2} R^H(n) d^{(k)}(n) \tag{21}$$

It follows from the definition in eqn. 12 that

$$d^{(k)}(n) = \beta^2 a^{(k)}(n) \tag{22}$$

and hence eqn. 20 constitutes a straightforward recursion which may be used to compute the updated vector $a^{(k)}(n)$. It also follows from eqn. 20 that

$$\beta^2 \| a^{(k)}(n-1) \|^2 = \beta^4 \| a^{(k)}(n) \|^2 + |\alpha^{(k)}(n)|^2 \tag{23}$$

and this provides a simple recursive formula which could be used to update the denominator in eqn. 13. However, for reasons of numerical stability, it is preferred to compute the inner product $a^{(k)H}(n) a^{(k)}(n)$ directly [5].

The recursive update in eqn. 20 may be implemented using a simple extension to the triangular systolic array proposed by Gentleman and Kung [2], or to the type of systolic array described by Schreiber and Tang [6] for updating the Cholesky factor $R(n-1)$. Schreiber's algorithm proceeds by solving the triangular system of linear equations

$$R^T(n) b(n) = x(t_n) \tag{24}$$

forming the inner product $b^T(n) a^{(k)}(n)$, and multiplying the result by the normalisation factor $\mu^{(k)} / \| a^{(k)}(n) \|^2$. It is this part of the overall procedure which Bojanczyk and Luk [4] cite as being awkward from the point of view of pipelining Schreiber's algorithm. In the following Section

we will show how the entire algorithm may be pipelined on one triangular systolic array by avoiding the need for a separate processor to solve eqn. 24. A similar technique has recently been derived independently by Yang and Böhme [7] who propose the use of a single linear systolic array to perform the computation.

3 Systolic array implementation

In a previous publication [3], it was shown how a systolic array of the type illustrated in Fig. 1 could be used in

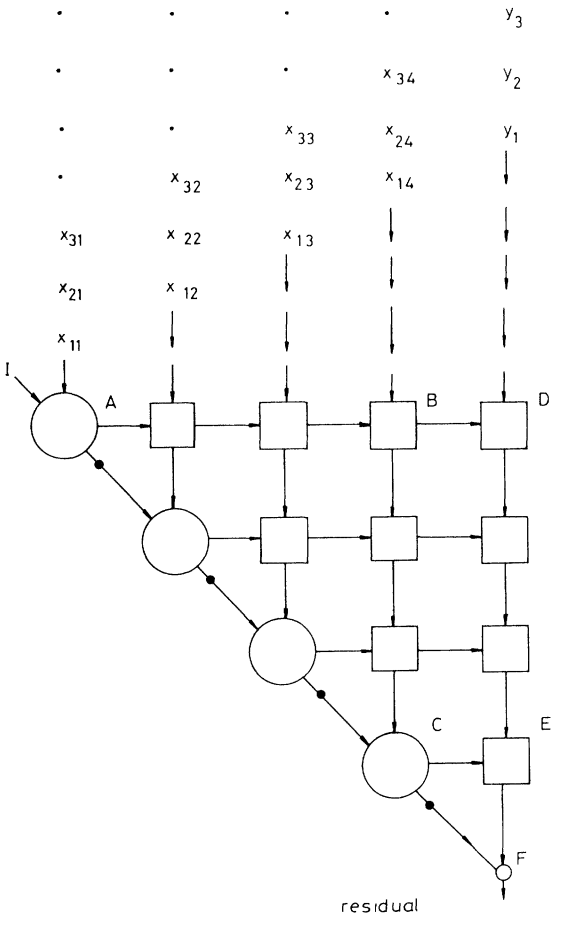


Fig. 1 *Systolic array for recursive least squares minimisation*

an efficient recursive manner, to evaluate the sequence of *a posteriori* least squares residuals

$$e(t_n) = \boldsymbol{x}^T(t_n)\boldsymbol{w}(n) + y(t_n) \tag{25}$$

where $y(t_n)$ denotes the value at time t_n of an additional 'reference' signal and $\boldsymbol{w}(n)$ is the (unconstrained) weight vector which minimises

$$E(n) = \|(\boldsymbol{X}(n)\boldsymbol{w}(n) + \boldsymbol{y}(n))\| \tag{26}$$

The vector

$$\boldsymbol{y}(n) = [y(t_1), y(t_2), \ldots, y(t_n)]^T \tag{27}$$

denotes the sequence of all reference signal samples obtained up to time t_n. The main triangular array ABC implements a sequence of Givens rotations in order to perform a recursive QR decomposition of the data matrix $\boldsymbol{X}(n)$, as described by eqns. 9 and 16. The right hand column of cells applies an identical sequence of rotations to the vector $\boldsymbol{y}(n)$ such that, at any time t_n,

$$\boldsymbol{Q}(n)\boldsymbol{y}(n) = \begin{bmatrix} \boldsymbol{u}(n) \\ \hline \boldsymbol{v}(n) \end{bmatrix} \tag{28}$$

The vector $\boldsymbol{u}(n)$ is stored in the right hand column of cells DE and updated using the recursive formula

$$\hat{\boldsymbol{Q}}(n)\begin{bmatrix} \beta\boldsymbol{u}(n-1) \\ \hline \beta\boldsymbol{v}(n-1) \\ \hline y(t_n) \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}(n) \\ \hline \beta\boldsymbol{v}(n-1) \\ \hline \alpha(n) \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}(n) \\ \hline \boldsymbol{v}(n) \end{bmatrix} \tag{29}$$

where $\hat{\boldsymbol{Q}}(n)$ is the sequence of Givens rotations defined in eqn. 16.

By applying the orthogonal transformation $\boldsymbol{Q}(n)$ to the term in brackets in eqn. 26, it is easy to show that the least squares weight vector is given by

$$\boldsymbol{R}(n)\boldsymbol{w}(n) + \boldsymbol{u}(n) = \boldsymbol{0} \tag{30}$$

and McWhirter [3] showed that the corresponding residual

$$e(t_n) = -\boldsymbol{x}^T(t_n)\boldsymbol{R}^{-1}(n)\boldsymbol{u}(n) + y(t_n) \tag{31}$$

may be obtained without any need to compute the weight vector $\boldsymbol{w}(n)$ explicitly. It is simply given by

$$e(t_n) = \gamma(n)\alpha(n) \tag{32}$$

where $\alpha(n)$ is the value of x_{out} produced by the internal cell E at time t_n, and $\gamma(n)$ is the corresponding value of γ_{out} produced by the boundary cell C. The product in eqn. 32 is computed by the final cell F. The function of each processing cell required for the systolic array in Fig. 1 was given by McWhirter [3], and is specified (for the more general case of complex data) by the first mode of operation of the corresponding cell in Fig. 2. By comparing eqns. 31 and 29 with eqns. 14 and 20, respectively, it

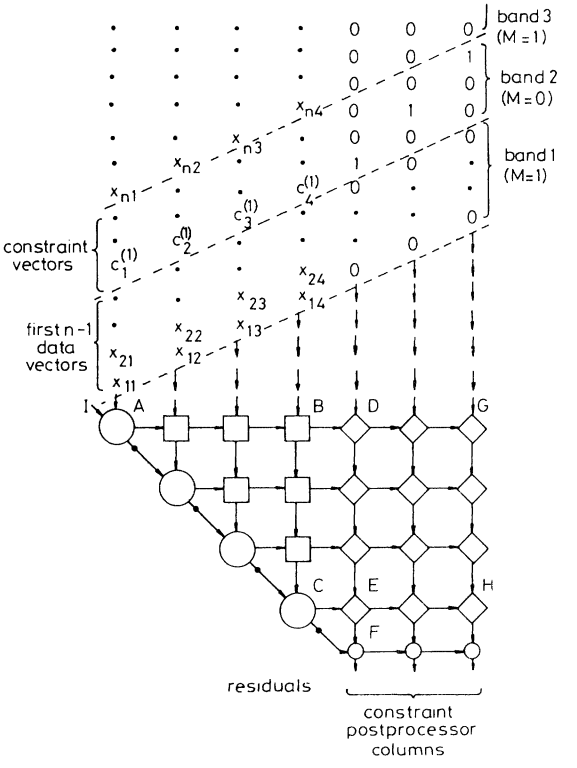


Fig. 2 *Systolic array for MVDR beamforming*

can be seen that if at time t_{n-1}, the vector $\boldsymbol{u}(n-1)$ generated by the right hand column of cells in Fig. 1 was replaced by the vector $\boldsymbol{a}^{(k)}(n-1)$, and the value of $y(t_n)$ was set equal to zero, then the output from the final cell F at time t_n would be identical to $-\beta^2\hat{e}^{(k)}(t_n)$. The vector $\beta^2\boldsymbol{a}^{(k)}(n)$ would also be generated and stored in the right hand column of cells as a result, and so the process may

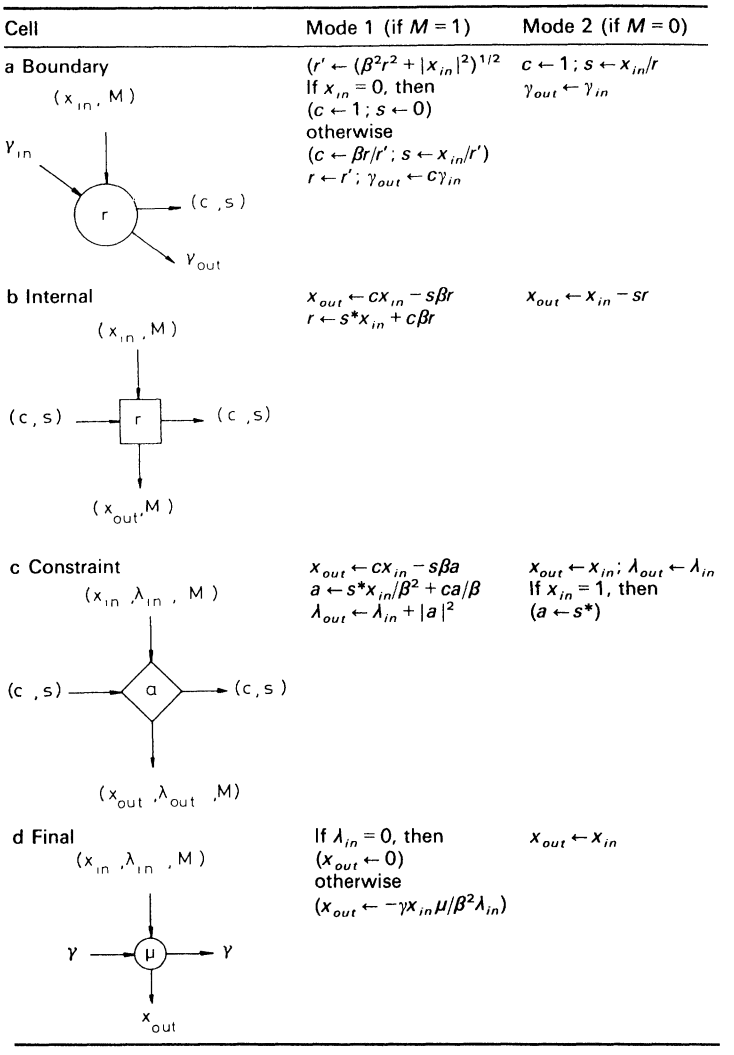


Fig. 3 *Processing cells required for MVDR systolic array (conventional Givens rotation algorithm)*

be continued in a very simple recursive manner to generate the sequence of output residuals $\hat{e}^{(k)}(t_i)$ $(i = n, n+1, \ldots)$. These considerations lead to the systolic array for MVDR beamforming, which is illustrated in Fig. 2. It incorporates four types of cell whose processing functions are detailed in Fig. 3. The array comprises a basic triangular array ABC and K columns of cells on the right hand side — one for each constraint. The MVDR computation is carried out in three distinct phases, the first two of which constitute a pipelined initialisation procedure.

During the first phase, which corresponds to the first band of input data in Fig. 2, the first $n-1$ rows of data $\boldsymbol{x}^T(t_i)$ $(i = 1, 2, \ldots, n-1$; where $n - 1 \geqslant p)$ are input to the main triangular array ABC in the usual time-staggered manner. The boundary and internal cells perform standard Givens rotations as defined by mode 1 of the procedures in Fig. 3. This mode is selected according to the value of a special control bit M associated with each input data sample, the mode control bit for all samples in the first band of input data being set to one. During the first processing phase, the basic triangular array ABC implements a QR decomposition of the initial data matrix $\boldsymbol{X}(n-1)$ to produce the upper triangular matrix $\boldsymbol{R}(n-1)$, which is stored within the array in the usual manner. A sequence of zeros is input to the rest of the array which performs no useful function during the first processing phase.

During the second processing phase, which corresponds to the second band of input data in Fig. 2, the associated mode control bits are set to zero, and all cells within the array operate in mode 2. For cells within the basic triangular array, mode 2 simply constitutes a non-adaptive or 'frozen' version of mode 1, in which the update of all stored values is suppressed. It can be shown that, when the triangular array operates in this mode, the effect of inputting an arbitrary p-element data vector $\boldsymbol{x}$ from above (in the usual time-staggered manner) is to produce a transformed vector

$$s(n-1) = \boldsymbol{R}^{-T}(n-1)\boldsymbol{x} \tag{33}$$

represented by the output values s which emerge from the array p cycles later (in a corresponding time-staggered fashion). This can be demonstrated quite simply by examining the operation of the 'frozen' network in detail, and expressing each input x_i as a function of the corresponding output values s_i. Alternatively, it may be considered as a corollary to the direct residual extraction technique defined in eqns. 31 and 32.

The input to the main triangular array during the second processing phase comprises the sequence of K time-staggered constraint vectors $\boldsymbol{c}^{(1)}, \boldsymbol{c}^{(2)}, \ldots, \boldsymbol{c}^{K}$, and so it serves to produce the corresponding sequence of output vectors $\boldsymbol{a}^{(k)*}(n-1)$ $(k = 1, 2, \ldots, K)$ as defined in eqn. 12. These emerge from the right hand boundary of the main triangular processor and continue to move to

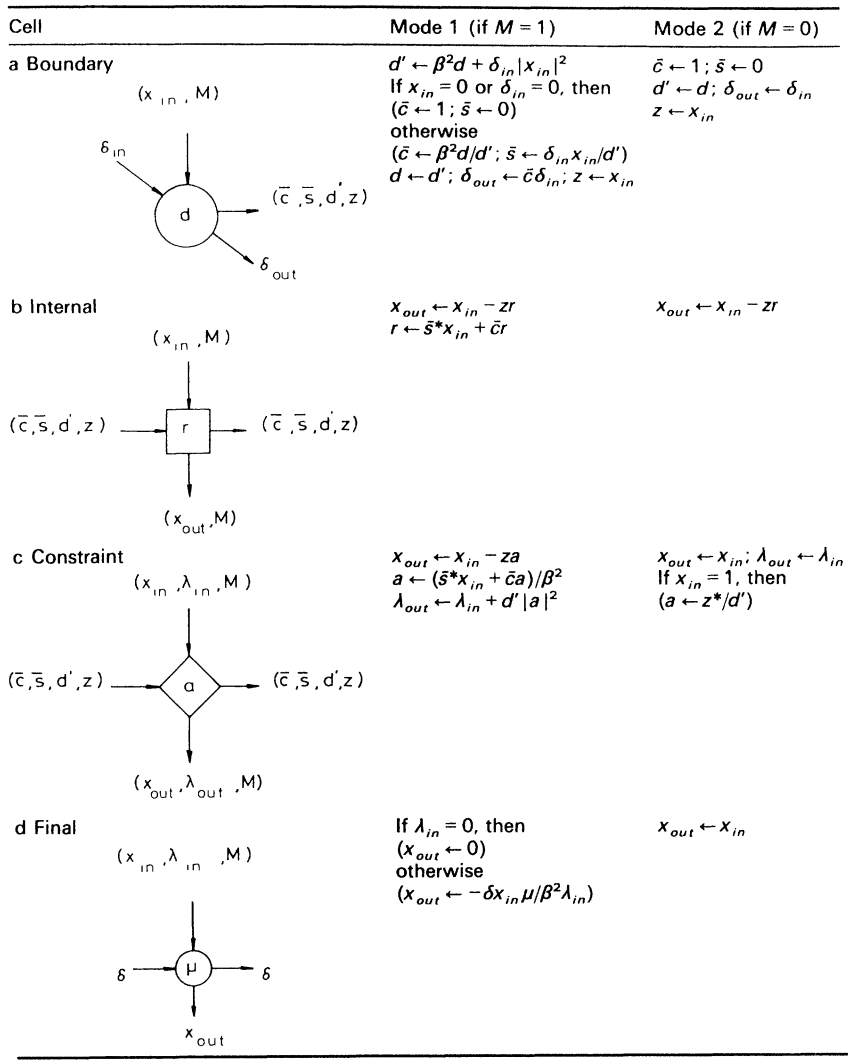


Fig. 4 *Processing cells required for MVDR systolic array (square-root-free Givens rotation algorithm)*

the right at the rate of one cell per clock cycle across the K constraint postprocessor columns. During the second phase of processing, the input to these columns from above takes the form of a unit diagonal matrix. The unit input associated with the kth column ($k = 1, 2, \ldots, K$) enters each cell within that column at the same time as the corresponding element of the vector $\boldsymbol{a}^{(k)*}(n-1)$. It serves to indicate that the complex conjugate of this element should be stored within the cell, where it remains unaltered during the remainder of the second processing phase. At the end of this phase, the systolic array in Fig. 2 has clearly been initialised as appropriate for the recursive MVDR computation described above.

During the third phase of processing, which corresponds to band 3 of the input data in Fig. 2, the sequence of data vectors $\boldsymbol{x}(t_i)$ ($i = n, n+1, \ldots$) is input to the main triangular processor, and zeros are fed into the remainder of the array as shown. All cells within the array receive a sequence of unit mode control bits and hence return to their first (adaptive) mode of operation. The mode 1 function for each cell in the constraint postprocessor section DGHE includes an additional computation associated with the parameter λ. This parameter is initialised to zero on entry to the array and serves to accumulate the normalisation term $\|\boldsymbol{a}^{(k)}(n)\|^2$ associated with each constraint column on each cycle. The value of μ in the final cell of the kth constraint column is set equal to $\mu^{(k)}$, and so it follows from eqn. 13 that the output produced by this cell during the third phase of processing is the required sequence of residual values $e^{(k)}(t_i)$ ($i = n, n+1, \ldots$). This concludes our description of the basic systolic array for MVDR beamforming. The following Section is devoted to discussing how it may be modified to perform the same computation using square-root-free Givens rotations.

4 Square-root-free algorithm

Fig. 4 details the cell functions which are required when the systolic array in Fig. 2 is used to compute the MVDR by means of square-root-free Givens rotations [8]. In their first mode of operation, the boundary and internal cells perform essentially the same functions (but generalised to the case of complex data) as those required for the square-root-free systolic least squares processor described by McWhirter [3]. The second mode of operation for these cells simply constitutes, as before, a non-adaptive or 'frozen' version of mode 1, in which the update of all stored quantities is suppressed. It is worth noting that for the square-root-free algorithm, this mode of operation can be selected very simply by setting the value of δ_{in} to zero (instead of 1) at the top boundary cell. For convenience, however, the use of explicit mode control bits has been assumed in Fig. 4.

To understand the operation of the constraint cells, it is important to realise that when square-root-free Givens rotations are used to perform a QR decomposition, the upper triangular matrix is represented in the form

$$\boldsymbol{R}(n-1) = \boldsymbol{D}^{1/2}(n-1)\boldsymbol{K}(n-1) \tag{34}$$

where $\boldsymbol{D}(n-1)$ is a diagonal matrix stored within the boundary cells, and $\boldsymbol{K}(n-1)$ is a unit upper triangular matrix stored within the internal cells of the systolic array in Fig. 1 or Fig. 2. Furthermore, the vector $\boldsymbol{u}(n-1)$ in eqn. 29 is represented in the form

$$\boldsymbol{u}(n-1) = \boldsymbol{D}^{1/2}(n-1)\bar{\boldsymbol{u}}(n-1) \tag{35}$$

where the vector $\bar{\boldsymbol{u}}(n-1)$ is stored in the right hand column of the array in Fig. 1. It follows that, to perform the recursive update in eqn. 20, the kth constraint column in Fig. 2 must be initialised to store the vector $\bar{\boldsymbol{a}}^{(k)}(n-1)$, rather than the vector $\boldsymbol{a}^{(k)}(n-1)$, where

$$\boldsymbol{a}^{(k)}(n-1) = \boldsymbol{D}^{1/2}(n-1)\bar{\boldsymbol{a}}^{(k)}(n-1) \tag{36}$$

Now it can readily be shown that during the second processing phase of the square-root-free algorithm, when the systolic array in Fig. 2 operates in mode 2, the effect of inputting an arbitrary p-element vector $\boldsymbol{x}$ from above (in the usual time-staggered manner) is to produce a transformed vector

$$\boldsymbol{z}(n-1) = \boldsymbol{K}^{-T}(n-1)\boldsymbol{x} \tag{37}$$

represented by the output values z which emerge from the array p cycles later (in a corresponding time-staggered manner). To initialise each of the K constraint columns, it is necessary, therefore, to divide the captured vector

$$\begin{aligned}\boldsymbol{z}^{(k)}(n-1) &= \boldsymbol{K}^{-T}(n-1)\boldsymbol{c}^{(k)} \\ &= \boldsymbol{D}^{1/2}(n-1)\boldsymbol{R}^{-T}(n-1)\boldsymbol{c}^{(k)} \\ &= \boldsymbol{D}(n-1)\bar{\boldsymbol{a}}^{(k)*}(n-1)\end{aligned} \tag{38}$$

by the diagonal matrix $\boldsymbol{D}(n-1)$, and as this is stored within the boundary cells, the associated parameter d' must be passed from left to right across each row of cells, together with the parameters $\bar{c}$, $\bar{s}$ and z as indicated in Fig. 4. Unfortunately, this places an additional communication overhead on every cell within the MVDR processor array.

To understand the operation of the square-root-free constraint cells, it is also important to appreciate that the recursion in eqn. 20 will produce an updated vector $\bar{\boldsymbol{a}}^{(k)}(n)$ stored in the kth constraint column and hence, to evaluate the normalisation term $\|\boldsymbol{a}^{(k)}(n)\|^2$, it is necessary to multiply this stored vector by the updated diagonal matrix $\boldsymbol{D}^{1/2}(n)$ in accordance with eqn. 36. This is achieved by making use once again of the associated parameter d' as indicated in Fig. 4. In all other respects, the operation of both the constraint cells and the final cells in Fig. 4 may be deduced quite readily from that of the internal and final cells derived by McWhirter [3] for the square-root-free algorithm.

5 Discussion

In this paper, we have described an efficient systolic array for computing the MVDR based on a recursive algorithm proposed by Schreiber. The numerical properties of this algorithm are obviously of vital importance in any application of the technique, and this aspect will now be discussed.

Schreiber showed by means of a simple error analysis [5], that the basic recursion is numerically stable, in the sense that the effect of an error introduced into the computation at any stage will not grow in time. This property may not seem to be consistent with the fact that, in mode 1, the operation of the constraint cell involves dividing the stored value a by the 'forget' factor β (or β^2 for the square-root-free algorithm). As β is a scalar generally close to, but less than one, this could obviously lead to an exponential growth in errors. However, this division in the constraint cells is, in effect, offset by the fact that, in mode 1, the operation of all cells within the main triangular array involves multiplying the stored value r by the same factor β (or β^2 for the square-root-free algorithm). As a result, the effect of an individual error is not magnified within the computation, but neither is it diminished by means of the 'forget' factor, as in the basic triangular array. In a sense, the constraint cells are simply counteracting the effect of the exponential forget factor on the stored matrix $\boldsymbol{R}$ in order to satisfy eqn. 12 at each stage of the recursion. The net effect is a steady accumulation of numerical errors within the algorithm, and so eqn. 12 is satisfied less accurately as the time index n increases. This difficulty may be overcome quite readily by switching the array to mode 2 and reinitialising the vectors $\boldsymbol{a}^{(k)}(n)$ periodically. Our numerical simulations have shown, for example, that for an exponential window of approximately 100 samples ($\beta = 0.99$) and a 24-bit floating point number representation (16-bit mantissa and 8-bit exponent), it is sufficient to reinitialise these vectors every 5000 samples to retain sufficient accuracy for most practical applications. We hope to discuss these and other numerical results more fully in a future publication.

An alternative initialisation scheme has been suggested, whereby at time $n = 0$, the triangular matrix $\boldsymbol{R}$ is set equal to $\delta\boldsymbol{I}$, δ being a small scalar and $\boldsymbol{I}$ representing the unit matrix. This simulates the effect of low-level thermal noise in the receiver channels prior to the input of any data vectors. The corresponding value of $\boldsymbol{a}^{(k)}$ is simply $\delta^{-1}\boldsymbol{c}^{(k)*}$, and so the overall initialisation procedure is very straightforward. This alternative technique has been found to work quite well in practice, but has the disadvantage that it is not suitable for reinitialising the vectors $\boldsymbol{a}^{(k)}(n)$ at a later stage of the adaptive process and does not, therefore, avoid the need for a mode 1 procedure to be defined within the array. The numerical implications of this initialisation technique will also be discussed in a future publication.

6 References

1 McWHIRTER, J.G., and SHEPHERD, T.J.: 'A systolic array for linearly constrained least-squares problems', *Proc. SPIE Int. Soc. Opt. Eng.*, Advanced algorithms and architectures for signal processing, 1986, **696**

2 GENTLEMAN, W.M., and KUNG, H.T.: 'Matrix triangularisation by systolic arrays', *Proc. SPIE*, Real time signal processing IV, 1981, **298**

3 McWHIRTER, J.G.: 'Recursive least-squares minimisation using a systolic array', *Proc. SPIE*, Real time signal processing VI, 1983, **431**

4 BOJANCZYK, A.W., and LUK, F.T.: 'A novel MVDR beamforming algorithm', *Proc. SPIE*, Advanced algorithms and architectures for signal processing II, 1987, **826**

5 SCHREIBER, R.: 'Implementation of adaptive array algorithms', *IEEE Trans.*, 1986, **ASSP-34**, (5), p. 1038

6 SCHREIBER, R., and TANG, W.: 'On systolic arrays for updating the Cholesky factorisation', *BIT*, 1986, **26**, p. 451

7 YANG, B., and BÖHME, J.F.: 'Systolic implementation of a general adaptive array processing algorithm', *IEEE Int. Conf. Acoust., Speech & Signal Process.*, New York, 1988

8 GENTLEMAN, W.M.: 'Least squares computations by Givens transformations without square roots', *J. Inst. Math. & Appl.*, 1973, **12**, p. 329

Systolic Block Householder Transformation for RLS Algorithm with Two-Level Pipelined Implementation

KuoJuey Ray Liu, *Member, IEEE,* Shih-Fu Hsieh, *Member, IEEE,* and Kung Yao, *Senior Member, IEEE*

***Abstract*—The *QR* decomposition, recursive least squares (QRD RLS) algorithm is one of the most promising RLS algorithms, due to its robust numerical stability and suitability for VLSI implementation based on a systolic array architecture. Up to now, among many techniques to implement the *QR* decomposition, only the Givens rotation and modified Gram–Schmidt methods have been successfully applied to the development of the QRD RLS systolic array. It is well known that Householder transformation (HT) outperforms the Givens rotation method under finite precision computations. Presently, there is no known technique to implement the HT on a systolic array architecture. In this paper, we propose a systolic block Householder transformation (SBHT) approach, to implement the HT on a systolic array as well as its application to the RLS algorithm. Since the data is fetched in a block manner, vector operations are in general required for the vectorized array. However, a modified HT algorithm permits a two-level pipelined implementation of the SBHT systolic array at both the vector and word levels. The throughput rate can be as fast as that of the Givens rotation method. Our approach makes the HT amenable for VLSI implementation as well as applicable to real-time high throughput applications of modern signal processing. The constrained RLS problem using the SBHT RLS systolic array is also considered in this paper.**

I. Introduction

LEAST squares (LS) technique constitutes an integral part of modern signal processing and communications methodology as used in adaptive filtering, beamforming, array signal processing, channel equalization, etc. [6]. Efficient implementation of the LS algorithm, particularly the recursive LS algorithm (RLS), is needed to meet the high throughput and speed requirements of modern signal processing. There are many possible approaches, such as the fast transversal method and the lattice method, which can perform RLS algorithm efficiently [1], [6]. Unfortunately, these methods can encounter numerical difficulties due to the accumulation of roundoff errors under a finite-precision implementation as summarized in [2]. This may lead to a divergence of the computations of the RLS algorithm [2]. A new type of systolic algorithm based on the *QR* decomposition (QRD), known as the QRD RLS, was first proposed by McWhirter in [18]. This algorithm is one of the most promising algorithms in that it is numerically stable [1], [12] as well as suitable for parallel processing implementation on a systolic array [6], [18].

Up to now, most of the QRD RLS implementations were based on the Givens rotation method and modified Gram–Schmidt method, which are both rank-1 update approaches [2], [4], [7], [13], [16], [18], [9]. It is well known that the Householder transformation (HT), which is a rank-k update approach, is one of the most computationally efficient methods to compute QRD. The error analysis carried out by Wilkinson [26], [8] showed that the HT outperforms the Givens method under finite precision computations. Presently, there is no known technique to implement the HT on a systolic array parallel processing architecture, since there is a belief that nonlocal connections in the implementation are necessary due to the vector processing nature of the Householder transformation. One of the purposes of this paper is to show that we can implement the HT on a systolic array with only local connections. Thus, it is amenable to VLSI implementation and is applicable to real-time high throughput applications of modern signal processing.

In this paper, we first propose a systolic Householder algorithm called a systolic block Householder transformation (SBHT) to compute the QRD with an implementation on a vectorized systolic array. Then a RLS algorithm based on the SBHT called SBHT RLS algorithm is proposed to perform RLS operations on the array. We shall show that the SBHT array and the SBHT RLS array are generalizations of Gentleman–Kung's QRD array [4] and McWhirter's QRD RLS systolic array [18] (see Fig. 1), respectively. The difficulty in the applications of the above arrays is mainly due to the vectorized operations of the processing cells. This results in a high cell complexity as well as a high I/O bandwidth. By using a modified HT algorithm proposed by Tsao [25], a two-level pipelined implementation of the SBHT RLS algorithm can be achieved. That is, the algorithm is pipelined at the vector level as well as at the word level. The complexity of the processing cell and the I/O bandwidth are thus reduced.

Manuscript received April 18, 1990; revised March 12, 1991. This work was supported in part by a UC MICRO Grant and NSF Grants NCR-8814407 and ECD-8803012.

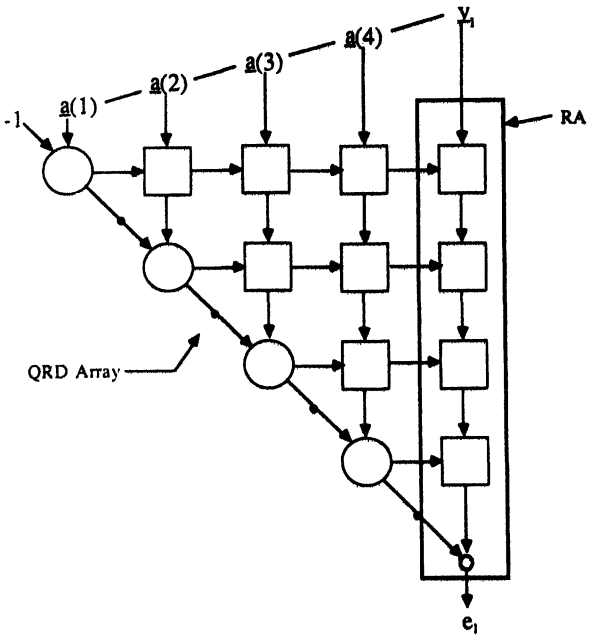
K. J. R. Liu is with the Department of Electrical Engineering, Systems Research Center, University of Maryland, College Park, MD 20742.

S. F. Hsieh is with the Department of Communication Engineering, National Chiao Tung University, Hsinchu, Taiwan 30039, Republic of China.

K. Yao is with the Department of Electrical Engineering, University of California, Los Angeles, CA 90024-1594.

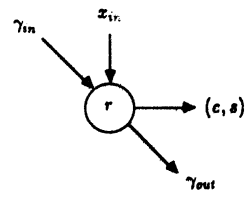
IEEE Log Number 9106033.

Reprinted from *IEEE Transactions on Signal Processing*, pp. 946-958, April 1992.



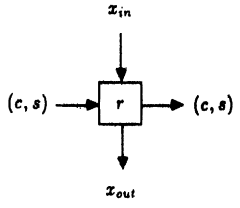
(a)

(1) Boundary Cell



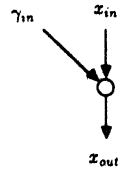
If $x_{in} = 0$ then
$c \leftarrow 1$; $s \leftarrow 0$; $\gamma_{out} \leftarrow \gamma_{in}$;
$r = \lambda r$,
otherwise
$r' = \sqrt{\lambda^2 r^2 + x_{in}^2}$;
$c \leftarrow \lambda r / r'$; $s \leftarrow x_{in} / r'$
$r \leftarrow r'$; $\gamma_{out} = c\gamma_{in}$
end

(2) Internal Cell



$x_{out} \leftarrow cx_{in} - s\lambda r$
$r \leftarrow sx_{in} + c\lambda r$

(3) Final Cell



$x_{out} \leftarrow \gamma_{in} x_{in}$

(b)

Fig. 1. (a) QRD RLS systolic array using Givens rotation method. (b) Processing cells of the Givens rotation method.

In general, the cell complexity of the SBHT array is higher and the system latency is longer than that of the conventional Givens rotation implementations. With the two-level pipelined implementation, the throughput of the *SBHT RLS systolic array* is as fast as that of McWhirter's Givens rotation array, and it offers better numerical stability than the Givens method. In addition, an extension of the SBHT RLS array to MVDR beamformation, which is a constrainted RLS problem, is also considered.

In Section II, a brief review of the QRD RLS algorithm is given. In Section III, the SBHT is presented, while the SBHT RLS algorithm is considered in Section IV. The two-level pipelined implementation of the SBHT RLS systolic array is discussed in Section V. In Section VI, the constrained RLS problem is applied to the MVDR beamformation, using an extension of the SBHT RLS array. Finally, the systolic array for the hyperbolic Householder transformation is considered in Section VII and a conclusion is given in Section VIII.

II. QRD RLS Algorithm

A full rank $m \times p$, $m > p$, rectangular matrix $\boldsymbol{X}$ can be uniquely factorized into two matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ such that $\boldsymbol{X} = \boldsymbol{QR}$, where $\boldsymbol{Q}$ is an $m \times p$ matrix with orthonormal columns and $\boldsymbol{R}$ is a $p \times p$ upper triangular matrix. Several different approaches of the QRD systolic arrays have been proposed by Gentleman and Kung [4], Heller and Ipsen [7], Luk [16], Ling *et al.* [13], and Kalson and Yao [9]. The first three approaches are based on the Givens rotations methods, while the last two are based on the modified Gram–Schmidt orthogonalization. Given an $m \times 1$ vector y, the LS problem is to minimize the norm of the residual vector ε

$$\|\varepsilon(m)\| = \|\boldsymbol{X}(m)\, w(m) - y(m)\|$$

by choosing an optimal weight vector w. If the matrix $\boldsymbol{X}$ and vector y grow in time, then the problem of minimizing the norm of the residual vector recursively becomes the RLS problem. Until recently, it appears that only Givens and modified Gram–Schmidt methods have been considered for RLS computations. Some recent RLS problems based upon the use of Householder transformation have appeared [3], [15]. In [18], McWhirter showed that a QRD RLS systolic array, which was based on the Gentleman–Kung array, can be designed without first computing the weight vector of the RLS problem. This approach is useful for high throughput applications in various modern signal processing problems such as adaptive filtering and beamforming since optimal residuals are of direct interest while the weight vector needs not be computed. The basic idea of the QRD RLS systolic array in [18] is to update the $p \times p$ matrix $\boldsymbol{R}$ using a sequence of Givens rotation matrices when a new row of data arrives. Suppose we have the QRD of the data matrix $\boldsymbol{X}$ at time m and expressed as $\boldsymbol{X}(m) = \boldsymbol{Q}(m)\boldsymbol{R}(m)$. Define

$$\overline{\boldsymbol{Q}}^T(m) = \left[\begin{array}{c:c} \boldsymbol{Q}^T(m) & \mathbf{0} \\ \hdashline 0^T & 1 \end{array}\right].$$

When a new row of data arrives, we then have

$$\overline{\boldsymbol{Q}}^T(m)\boldsymbol{X}(m+1) = \left[\begin{array}{c} \boldsymbol{R}(m) \\ \hline \mathbf{0} \\ \hline x_1, x_2, \cdots, x_p \end{array}\right].$$

This new row of data can be zeroed out by applying a sequence of Givens rotations

$$G = G_p \cdots G_2 G_1$$

where the $(m + 1) \times (m + 1)$ transformation matrix G_i is defined by

$$G_i = \left[\begin{array}{c:c:c:c} I_{i-1} & 0 & \mathbf{0} & 0 \\ \hline 0 & c_i & 0 & s_i \\ \hline \mathbf{0} & 0 & I_{m-i} & 0 \\ \hline 0 & -s_i & 0 & c_i \end{array}\right]$$

with

$$c_i = \frac{a}{\sqrt{a^2 + b^2}}, \qquad s_i = \frac{b}{\sqrt{a^2 + b^2}}$$

where a and b are elements of vectors in the ith and $(m + 1)$th rows under rotation.

Fig. 1(a) shows the systolic array proposed by McWhirter in [18]. It consists of a QRD triarray and a linear response array (RA). The rotation parameters are propagated from the boundary cells to the right for internal cells to update their contents, and the cosine parameters are also cumulated and propagated down diagonal boundary cells. Each skewed input row of data is zeroed out by the QRD triarray. The optimal residual is then obtained by the multiplication of the cumulated cosines and the rotated output of the desired response at the response array (see Fig. 1(a)) [18].

III. Systolic Block Householder Transformation

The Givens rotation method discussed above is a rank-1 update approach since each input consists of one row of data. For the systolic block Householder transformation (SBHT), we need a block data formulation. Denote the data matrix as

$$X(n) = \begin{bmatrix} X_1^T \\ X_2^T \\ \vdots \\ X_n^T \end{bmatrix} = \left[\begin{array}{c} X(n-1) \\ \hline X_n^T \end{array}\right] \in \Re^{nk \times p} \tag{1}$$

and the desired response vector as

$$y(n) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \left[\begin{array}{c} y(n-1) \\ \hline y_n \end{array}\right] \in \Re^{nk} \tag{2}$$

where X_i^T is the $k \times p$ ith data block matrix

$$X_i^T = \begin{bmatrix} x_{(i-1)k+1}^T \\ x_{(i-1)k+2}^T \\ \vdots \\ x_{ik}^T \end{bmatrix} = [x_{i,1} \quad x_{i,2} \quad \cdots \quad x_{i,p}] = \begin{bmatrix} x_{(i-1)k+1,1} & x_{(i-1)k+1,2} & \cdots & x_{(i-1)k+1,p} \\ x_{(i-1)k+2,1} & x_{(i-1)k+2,2} & \cdots & x_{(i-1)k+2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{ik,1} & x_{ik,2} & \cdots & x_{ik,p} \end{bmatrix} \in \Re^{k \times p} \tag{3}$$

and y_i is the $k \times 1$ ith desired response block vector

$$y_i = \begin{bmatrix} y_{(i-1)k+1} \\ y_{(i-1)k+2} \\ \vdots \\ y_{ik} \end{bmatrix} \in \Re^k \tag{5}$$

where k is the block size and p is the order (i.e., number of columns) of the system.

For a rank-k update QR decomposition, suppose we have

$$Q(n-1)X(n-1) = \left[\begin{array}{c} R(n-1) \\ \hline 0 \end{array}\right]. \tag{6}$$

Denote

$$\bar{Q}_k(n-1) = \left[\begin{array}{c:c} Q(n-1) & \mathbf{0} \\ \hline \mathbf{0}^T & I_k \end{array}\right] \tag{7}$$

then we have

$$\bar{Q}_k(n-1)X(n) = \left[\begin{array}{c} R(n-1) \\ \hline \mathbf{0} \\ \hline X_n^T \end{array}\right]. \tag{8}$$

If we can find a matrix $H(n)$ such that

$$H(n)\bar{Q}_k(n-1)X(n) = \left[\begin{array}{c} R(n) \\ \hline 0 \end{array}\right] \tag{9}$$

then the new $Q(n)$ is

$$Q(n) = H(n)\bar{Q}_k(n-1). \tag{10}$$

An $n \times n$ Householder transformation matrix T is of the form

$$T = I - \frac{2zz^T}{\|z\|^2} \tag{11}$$

where $z \in \Re^n$ [5]. When a vector x is multiplied by T, it is reflected in the hyperplane defined by $span\{z\}^\perp$. Choosing $z = x \pm \|x\|_2 e_1$, where $e_1 = [1, 0, 0, \cdots, 0] \in \Re^n$, then x is reflected onto e_1 by T as

$$Tx = \pm\|x\|_2 e_1. \tag{12}$$

That is, all of the energy of x is reflected onto the unit vector e_1 after the transformation. We can zero out X_n^T by applying successive Householder transformations as

follows:

$$H^{(i)}(n)\begin{bmatrix} R^{(i-1)}(n-1) \\ \hline 0 \\ \hline 0, \cdots, 0, x_{n,i}^{(i-1)}, \cdots, x_{n,p}^{(i-1)} \end{bmatrix} = \begin{bmatrix} R^{(i)}(n-1) \\ \hline 0 \\ \hline 0, \cdots, 0, 0, x_{n,i+1}^{(i)}, \cdots, x_{n,p}^{(i)} \end{bmatrix}$$

for $i = 1, \cdots, p$, where $x_{n,i}^{(0)} = x_{n,i}$, $R^{(0)}(n-1) = R(n-1)$, and the resultant matrix $H(n)$ is

$$H(n) = H^{(p)}(n)H^{(p-1)}(n) \cdots H^{(1)}(n) \tag{13}$$

where each $H^{(i)}(n)$ represents a Householder transformation which zeros out the ith column of the updated X_n^T, i.e., $x_{n,i}^{(i-1)}$.

To obtain $H^{(1)}(n)$, denote

$$z_1 = [r_{11} - \sigma_1 \vdots 0_{(n-1)k-1}^T \vdots x_{n,1}^T]^T$$

where r_{11} is the (1, 1) element of $R(n-1)$, $\sigma_1^2 = r_{11}^2 + \|x_{n,1}\|^2$. Then from (11)

$$H^{(1)}(n) = \begin{bmatrix} h_{11}^{(1)}(n) & \vdots & 0^T & \vdots & h_{12}^{(1)T}(n) \\ \hline 0 & \vdots & I_{(n-1)k-1} & \vdots & 0 \\ \hline h_{21}^{(1)}(n) & \vdots & 0 & \vdots & H_{22}^{(1)}(n) \end{bmatrix} \tag{14}$$

where $h_{11}^{(1)}(n)$ is a scalar, $h_{12}^{(1)}(n)$ is a $k \times 1$ vector, $h_{21}^{(1)}(n) = h_{12}^{(1)}(n)$, and $H_{22}^{(1)}(n)$ is a $k \times k$ matrix given by

$$H_{22}^{(1)}(n) = I_k - \frac{2x_{n,1}x_{n,1}^T}{\sigma_{z_1}^2} \tag{15}$$

with $\sigma_{z_1}^2 = \|z_1\|_2^2 = 2(\sigma_1^2 - \sigma_1 r_{11})$. Define $\psi_1 = \sigma_1^2 - \sigma_1 r_{11}$, (15) can be rewritten in a form without multiplication of the number 2 as

$$H_{22}^{(1)}(n) = I_k - \frac{x_{n,1}x_{n,1}^T}{\psi_1}.$$

In general,

$$H^{(m)}(n) = \begin{bmatrix} H_{11}^{(m)}(n) & \vdots & 0 & \vdots & H_{12}^{(m)}(n) \\ \cdots & \cdot & \cdots & \cdot & \cdots \\ 0 & \vdots & I_{(n-1)k-p} & \vdots & 0 \\ \cdots & \cdot & \cdots & \cdot & \cdots \\ H_{21}^{(m)}(n) & \vdots & 0 & \vdots & H_{22}^{(m)}(n) \end{bmatrix} \tag{16}$$

where $H_{11}^{(m)}(n) \in \Re^{p \times p}$ is an identity matrix except for the mth diagonal entry; $H_{12}^{(m)}(n) \in \Re^{p \times k}$ is a zero matrix except for the mth row; $H_{21}^{(m)}(n) = H_{12}^{(m)T}(n)$; and

$$H_{22}^{(m)}(n) = I_k - \frac{x_{n,m}^{(m-1)}x_{n,m}^{(m-1)T}}{\psi_m} \in \Re^{k \times k} \tag{17}$$

is symmetric with $\psi_m = \sigma_m^2 - \sigma_m r_{mm}$, where $\sigma_m^2 = r_{mm}^2 + \|x_{n,m}^{(m-1)}\|^2$. It can be easily seen that $H_{12}^{(i)}(n)H_{21}^{(j)}(n) = 0$, $H_{21}^{(i)}(n)H_{12}^{(j)}(n) = 0$, for $\forall i \neq j$. Thus we have the following lemma.

Lemma 1: The Householder transformation matrix, $H(n) \in \Re^{nk \times nk}$, is orthogonal and is of the form

$$H(n) = \begin{bmatrix} H_{11}(n) & \vdots & 0 & \vdots & H_{12}(n) \\ \cdots & \cdot & \cdots & \cdot & \cdots \\ 0 & \vdots & I_{(n-1)k-p} & \vdots & 0 \\ \cdots & \cdot & \cdots & \cdot & \cdots \\ H_{21}(n) & \vdots & 0 & \vdots & H_{22}(n) \end{bmatrix} = H^{(p)}(n)H^{(p-1)}(n) \cdots H^{(1)}(n) \tag{18}$$

with

$$H_{11}(n) = H_{11}^{(p)}(n) \cdots H_{11}^{(2)}(n)H_{11}^{(1)}(n)$$

$$H_{22}(n) = H_{22}^{(p)}(n) \cdots H_{22}^{(2)}(n)H_{22}^{(1)}(n). \quad \square \tag{19}$$

For the block size of $k = 1$ the Givens rotation method reduces to the special case of the rank-1 update Householder transformation [5], and the H matrix in Lemma 1 becomes a Givens rotation matrix G of the form [18]

$$G(n) = \begin{bmatrix} K(n) & 0 & h(n) \\ \hline 0 & I_{n-p-1} & 0 \\ \hline h^H(n) & 0 & \gamma(n) \end{bmatrix}$$

where $K(n)$ is a $p \times p$ matrix, $h(n)$ is a $p \times 1$ vector, and $\gamma(n)$ is a scalar given by $\gamma(n) = \Pi_{i=1}^{p} c_i(n)$, $n \geq p$ where $c_i(n)$ is the cosine parameter associated with the ith Givens rotation.

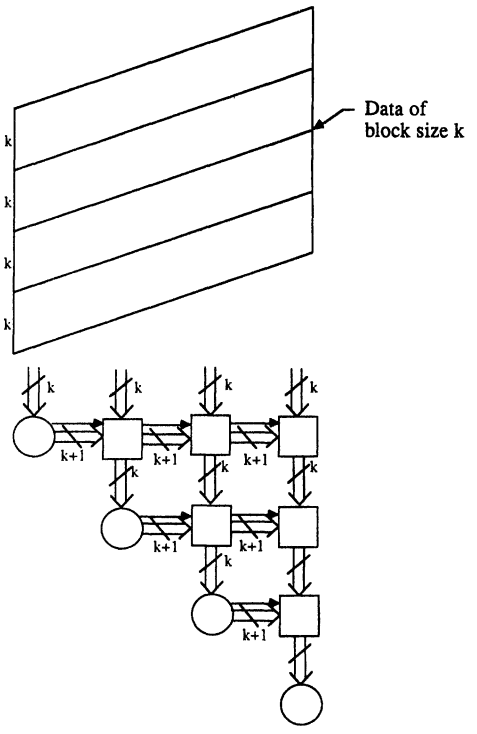
A. Vectorized SBHT QRD Systolic Array

Now we propose a vectorized systolic array to implement the QRD based on the SBHT. Similar to the *QR* triarray of Gentleman–Kung [4], this array has both boundary and internal cells. The boundary cell takes an input of block size k from the above internal processor or directly from the input port, updates its content and generates the reflection vector, and sends it to the right for the internal cell processing (see Fig. 2(a)). Define

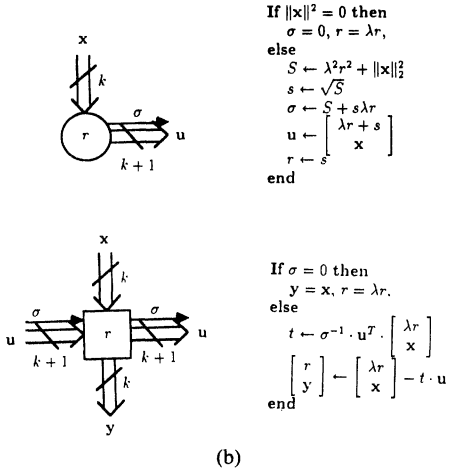
$$\bar{x}_{n,i}^{(i-1)^T} = [0_{i-1} \vdots r_{ii} \vdots 0_{(n-1)k-i} \vdots x_{n,i}^{(i-1)^T}], \quad i = 1, \cdots, p$$

and $z_i = \bar{x}_{n,i}^{(i-1)} - \sigma_i e_i$, where e_i is a zero vector except for a unity at the ith position. When an internal cell receives the reflection vector, instead of forming the matrix $z_i z_i^T$ and performing matrix arithmetics, it performs an inner product operation to update its content r_{ii} by doing

$$H^{(i)}(n)\bar{x}_{n,j}^{(i-1)} = \bar{x}_{n,j}^{(i-1)} - \frac{z_i}{\psi_i}(z_i^T \cdot \bar{x}_{n,j}^{(i-1)}) \quad j = i+1, \cdots, p \tag{20}$$



(a)



(b)

Fig. 2. (a) SBHT QRD systolic array. (b) Processing cells of the SBHT QRD systolic array.

and sends the reflected data $x_{n,j}$ downward for further processing. Fig. 2 shows the SBHT QRD array architecture and the processing cells. When the block size is $k = 1$, this vectorized array degenerates to the Gentleman–Kung's Givens rotation triarray.

IV. SBHT RLS Algorithm

The LS problem is to choose a weight vector $w(n) \in \Re^p$, such that the block-forgetting norm of

$$\varepsilon(n) = \begin{bmatrix} e_1(n) \\ e_2(n) \\ \vdots \\ e_n(n) \end{bmatrix} = X(n)w(n) - y(n) \tag{21}$$

is minimized. The optimal weight vector $\hat{w}(n)$ satisfies

$$\min_w \|\varepsilon(n)\|_{\Lambda_k} = \|X(n)\hat{w}(n) - y(n)\|_{\Lambda_k} \tag{22}$$

where

$$\|\varepsilon(n)\|_{\Lambda_k} = \|\Lambda_k(n)\varepsilon(n)\|_2 = \sqrt{\sum_{i=1}^{n} \lambda^{2(n-i)} \cdot \|e_i(n)\|_2^2}$$
$$0 < \lambda \le 1. \tag{23}$$

$\Lambda_k(n)$ is a block-diagonal exponential weighting matrix of the form

$$\Lambda_k(n) = \begin{bmatrix} \lambda^{n-1}I_k & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \lambda I_k & 0 \\ 0 & \cdots & 0 & I_k \end{bmatrix} \in \Re^{nk \times nk} \tag{24}$$

and $\|\cdot\|_2$ is the Euclidean norm,

$$\|e_i(n)\|_2^2 = \sum_{j=1}^{k} |e_{(i-1)k+j}(n)|^2. \tag{25}$$

The exponential forgetting weighting λ is incorporated in the RLS filtering scheme to avoid overflow in the processors as well as to facilitate nonstationary data updating.

The QRD of the weighted augmented data matrix at time n (in the block sense, it is equivalent to nk snapshots), is given by

$$\Lambda_k(n)[X(n) \vdots y(n)] = [Q_1^T(n) \vdots Q_2^T(n)] \begin{bmatrix} R(n) & \vdots & u(n) \\ 0 & \vdots & v(n) \end{bmatrix} \tag{26}$$

where

$$Q(n) = \begin{bmatrix} Q_1(n) \\ Q_2(n) \end{bmatrix}$$

constitutes an orthogonal transformation matrix with $Q_1(n) \in \Re^{p \times nk}$ spanning the column space of the weighted data matrix $\Lambda_k(n)X(n)$ and $Q_2(n) \in \Re^{(nk-p) \times nk}$ spanning the null space, $R(n) \in \Re^{p \times p}$ is an upper triangular matrix and

$$Q(n)y(n) = \begin{bmatrix} u(n) \\ v(n) \end{bmatrix}.$$

The optimal weight vector can be obtained by solving

$$R(n)\hat{w}(n) = u(n). \tag{27}$$

Obviously, $\Lambda_k(n)X(n) = Q_1^T(n)R(n)$. As a result, the weighted optimal residual of (21) is

$$\Lambda_k(n)\hat{\varepsilon}(n) = Q_1^T(n)R(n)\hat{w}(n) - Q_1^T(n)u(n) - Q_2^T(n)v(n)$$
$$= -Q_2^T(n)v(n) \tag{28}$$

which lies in the null space of the weighted data matrix.

Now, suppose we have the data matrix up to time $n-1$ and the QRD of $\Lambda_k(n-1)[X(n-1) \vdots y(n-1)]$. The recursive LS problem here is to compute efficiently the optimum residual at time n from the results we have at time $n-1$. In particular, we are interested in the new nth block of the optimal residual

$$\hat{e}_n(n) = X_n^T \hat{w}(n) - y_n. \tag{29}$$

From (8), (9), and (18), (26) can be expressed as

$$\left[\begin{array}{c:c:c} R(n) & u(n) \\ \hdashline 0 & v(n) \end{array}\right] = \left[\begin{array}{c:c:c} H_{11}(n) & 0 & H_{12}(n) \\ \hdashline 0 & I_{(n-1)k-p} & 0 \\ \hdashline H_{21}(n) & 0 & H_{22}(n) \end{array}\right] \cdot \left[\begin{array}{c:c} \lambda R(n-1) & \lambda u(n-1) \\ \hdashline 0 & \lambda v(n-1) \\ \hdashline X_n^T & y_n \end{array}\right].$$

$$\Lambda_k(n)\hat{\varepsilon}(n) = \left[\begin{array}{c} \hat{\varepsilon}(n-1|n) \\ \hdashline \hat{e}_n(n) \end{array}\right] = \left[\begin{array}{c} -\lambda Q_2^T(n-1)v(n-1) - Q_1^T(n-1)H_{21}^T(n)v_n \\ \hdashline -H_{22}^T(n)v_n \end{array}\right] \tag{34}$$

By recursion on n, we relate $Q(n)$ and $Q(n-1)$ using (10) and have

$$Q(n) = \left[\begin{array}{c:c:c} H_{11}(n) & 0 & H_{12}(n) \\ \hdashline 0 & I_{(n-1)k-p} & 0 \\ \hdashline H_{21}(n) & 0 & H_{22}(n) \end{array}\right] \cdot \left[\begin{array}{c:c} Q_1(n-1) & 0 \\ \hdashline Q_2(n-1) & 0 \\ \hdashline 0 & I_k \end{array}\right].$$

$$= \left[\begin{array}{c:c} H_{11}(n)Q_1(n-1) & H_{12}(n) \\ \hdashline Q_2(n-1) & 0 \\ \hdashline H_{21}(n)Q_1(n-1) & H_{22} \end{array}\right]. \tag{30}$$

We can see that $Q_2(n)$ is updated from $Q_1(n-1)$ and $Q_2(n-1)$ by

$$Q_2(n) = \left[\begin{array}{c:c} Q_2(n-1) & 0 \\ \hdashline H_{21}(n)Q_1(n-1) & H_{22} \end{array}\right]. \tag{31}$$

On the other hand, the updated $[u^T(n), v^T(n)]^T$ is

$$\left[\begin{array}{c} u(n) \\ \hdashline v(n) \end{array}\right] = Q(n)\Lambda_k(n) \left[\begin{array}{c} y(n-1) \\ \hdashline y_n \end{array}\right] = \left[\begin{array}{c} \lambda H_{11}(n)u(n-1) + H_{12}(n)y_n \\ \hdashline \lambda v(n-1) \\ \hdashline v_n \end{array}\right] \tag{32}$$

where

$$v_n = \lambda H_{21}(n)u(n-1) + H_{22}(n)y_n. \tag{33}$$

Therefore, from (28), (31), and (32), the weighted optimal residual vector can be obtained from parameters at time $n-1$ by

where $\hat{\varepsilon}(m|n)$ denotes the estimate of $\hat{\varepsilon}$ at time m, $m \le n$, given all of the data up to time n. The new nth block of the optimal residual is then obtained as

$$\hat{e}_n(n) = -H_{22}^T(n)v_n = -H_{22}^{(1)}(n)H_{22}^{(2)}(n) \cdots H_{22}^{(p)}(n)v_n. \tag{35}$$

For the block size of $k = 1$, all vector parameters in (35) become scalars and can be expressed as

$$e_n(n) = -\prod_{i=1}^{p} c_i v_n \tag{36}$$

which was first shown by McWhirter in [18]. Note that there are some differences between the optimal residuals estimated by SBHT and Givens rotation methods. To be specific, the optimal residual vector in (35) is given by

$$\hat{e}_n(n) = \left[\begin{array}{c} e_{(n-1)k+1}((n-1)k+1|nk) \\ \vdots \\ e_{nk-1}(nk-1|nk) \\ e_{nk}(nk|nk) \end{array}\right] \tag{37}$$

while, the optimal residual estimated by the Givens rotation method in (36) is

$$\hat{e}_n(n) = e_n(n|n). \tag{38}$$

In this sense, the SBHT RLS gives a better estimate of the residual since it uses more data samples to estimate the optimal residual. As an example, consider $k = 2$. Then the optimal residuals obtained from the SBHT RLS and Givens methods are $[e_{2n-1}((2n-1)|2n), e_{2n}(2n|2n)]$ and $[e_{2n-1}((2n-1)|(2n-1), e_{2n}(2n|2n)]$, respectively. It is clear now that the SBHT RLS method gives a better estimate for the previous residual than the Givens rotation method because the former makes use of the future data sample at time $2n$ to estimate the residual at time $2n-1$, while the latter does not.

A. Vectorized SBHT RLS Array

In order to obtain the RLS filtering residual vector in the systolic array, we can use two possible approaches. The first approach is to generalize the architecture of McWhirter's Givens rotation approach [18]. A SBHT QRD array with a RA based on this approach is shown in Fig. 3. Since the $\boldsymbol{v}_n$ in (33) results from the reflection computation in (32), therefore $\boldsymbol{v}_n$ is obtained naturally from the output of RA. Each boundary cell then forms the matrix $\boldsymbol{H}_{22}^{(\cdot)}$ and propagates it down the diagonal boundary cells. Since $\boldsymbol{H}_{22}^{(i)}$ is generated earlier than $\boldsymbol{H}_{22}^{(j)}$ for $i < j$, (35) has to be computed from left to right involving matrix–matrix multiplications. As a result, each boundary cell performs the matrix multiplication to accumulate $\boldsymbol{H}_{22}^{(\cdot)}$ when it is propagated down diagonal boundary cells. The matrix multiplications needed in the boundary cells in this approach are objectionable since they not only slow down the throughput but also increase the complexity of the boundary cells. We note, McWhirter's original approach based on Givens rotation worked well since only scalars need to be propagated down the diagonal boundary cells and the order of multiplications for the scalars is irrelevant.

Obviously, we prefer to compute (35) from right to left such that only inner product computations are performed. Instead of forming the matrix $\boldsymbol{H}_{22}^{(\cdot)}$ and propagating it down, another approach is to use the facts that $\boldsymbol{H}_{22}^{(\cdot)}$ can be expressed by using (17) and the reflection vectors are sent to the right from boundary cells as described in Section III-A. From these observations, (35) can be computed in a manner similar to the internal cell operation. A new architecture shown in Fig. 4 is thus introduced to circumvent this problem. A column array of internal cells called backward propagation array (BPA) is added at the right-hand side to perform the backward propagation of $\boldsymbol{v}_n$. Each row, say the ith one, needs $2(p-i)$ delayed buffers as shown in Fig. 4. The $\boldsymbol{v}_n$ obtained at the output of RA is then backward propagated through the BPA. From (17), each cell of this array performs the operation

$$\boldsymbol{H}_{22}^{(i)}(n)\tilde{\boldsymbol{v}}_n = \tilde{\boldsymbol{v}}_n - \frac{\boldsymbol{x}_{n,i}^{(i-1)}}{\psi_i}\left(\boldsymbol{x}_{n,i}^{(i-1)T}\tilde{\boldsymbol{v}}_n\right)$$

$$i = p, \cdots, 2, 1 \qquad (39)$$

where $\tilde{\boldsymbol{v}}_n$ is an updated $\boldsymbol{v}_n$. This is a subset of the operations performed by the internal cell shown in (20). The residual vector is obtained from the top of the newly appended column array.

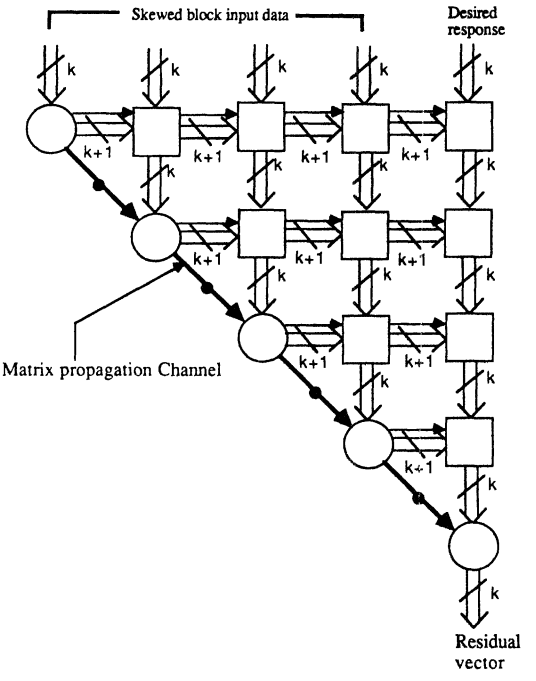


Fig. 3. SBHT RLS systolic array obtained by direct generalization of the Givens rotation array.

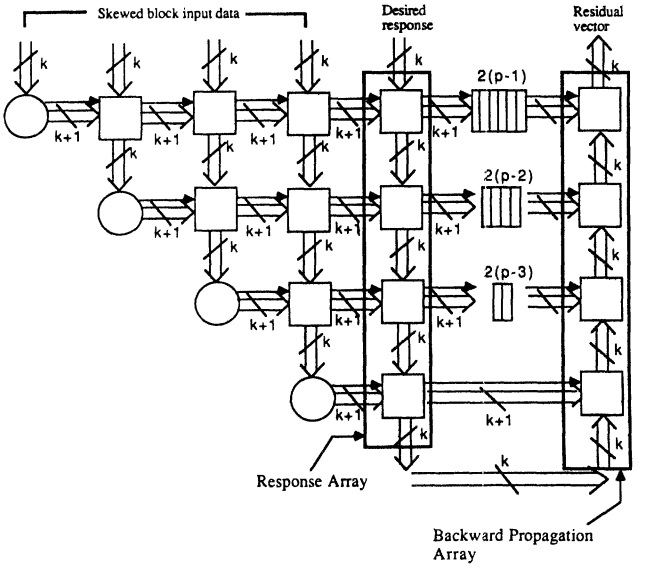


Fig. 4. New matrix-multiplication-free SBHT RLS systolic array.

The costs for this proposed architecture are: an increased latency time from $(2p+1)t_s$ of McWhirter's Givens method to $3pt_v$, where t_s represents the processing time for the scalar operations used in the Givens rotation method and t_v is the processing time for vector operations used in the SBHT method; the number of delay elements needed increases from p to $\Sigma_1^p 2(p-i) = p(p-1)$; and p additional internal processing cells. The operations of the boundary and internal cells are still given in Fig. 2(b).

These results clearly show that HT can be implemented simply on a systolic array to achieve massive parallel processing with vector operations. This provides an efficient method to obtain a high throughput rate for recursive LS filtering by using the HT method.

V. Two-Level Pipelined Implementations

The SBHT QRD array and the RLS array discussed in the above sections are derived using the conventional Householder transformation as shown in (11). Due to the vector processing nature of the conventional method, the cells of both arrays perform vector operations such as inner products. This means the complexity of each cell is high and the I/O bandwidth is large in order to achieve an effective vector data communication. Each cell, due to the complexity of vector processing, may require a large processor. Clearly, this is not desirable for VLSI implementation. Thus, we are motivated to find a suitable algorithm to pipeline the data down to the word level such that the I/O bandwidth as well as the complexity can be reduced. In addition, we still wish to achieve a high throughput which is needed in many modern signal processing applications.

The conventional approach in computing Householder transformation, $\boldsymbol{y} = \boldsymbol{Tq}$, based on (11) is to form first z and $\|z\|^2$ from $\boldsymbol{x}$ and then $z^T\boldsymbol{q}/\|z\|^2$ and $\boldsymbol{q} - 2z(z^T\boldsymbol{q}/\|z\|^2)$ as considered before. It can be stated in the following form:

HT Algorithm (Conventional):

Step 1. $S_{xx} = \|\boldsymbol{x}\|^2$.

Step 2. If $S_{xx} = 0$, then $\boldsymbol{y} = \boldsymbol{q}$.

Step 3. If $S_{xx} \neq 0$ then

(1) $s = \sqrt{S_{xx}}$, $z = \boldsymbol{x} + [s, 0, 0, \cdots, 0]^T$,

(2) $\phi = S_{xx} + sx_1$, $S_{zq} = \boldsymbol{z}^T\boldsymbol{q}$,

(3) $d = S_{zq}/\phi$, $\boldsymbol{y} = \boldsymbol{q} - dz$.

In [25], Tsao pointed out that by skipping the computation of ϕ and avoiding the cumbersome intermediate steps of forming vector z for further computations, a modified algorithm with smaller round off error and fewer operations can be obtained. Only step 3 of the conventional algorithm is modified as follows.

Modified HT Algorithm [25]:

Step 3. If $S_{xx} \neq 0$ then

(1) $s = \sqrt{S_{xx}}$, $\sigma = x_1 + s$,

(2) $S_{xq} = \boldsymbol{x}^T\boldsymbol{q}$,

(3) $y_1 = -S_{xq}/s$, $d = (q_1 - y_1)/\sigma$, $y_i = q_i - dx_i$, $i = 2, \cdots, n$.

With this algorithm, the operations of the cells of the vectorized systolic arrays can be modified as shown in Fig. 5. As we can see, for the boundary cell, the vector $\boldsymbol{u}$, which consists of the weighted diagonal element of the upper triangular matrix and one column of the input data block (updated or not), can be sent out immediately when the input $\boldsymbol{x}$ is available, without waiting for any computations as required in the implementation using the conventional algorithm. Due to this advantage and the modified operations in the internal cells, we can then pipeline the vector operations down to the word level such that each cell only performs scalar operations, which will significantly reduce its complexity.

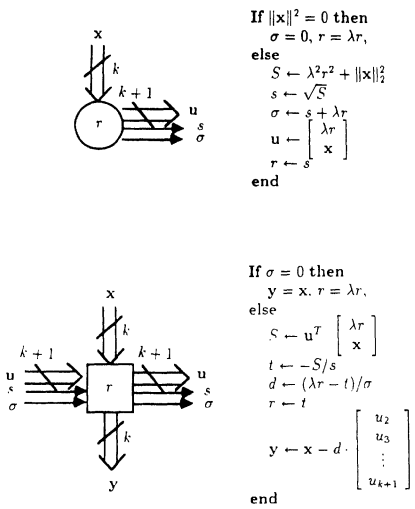
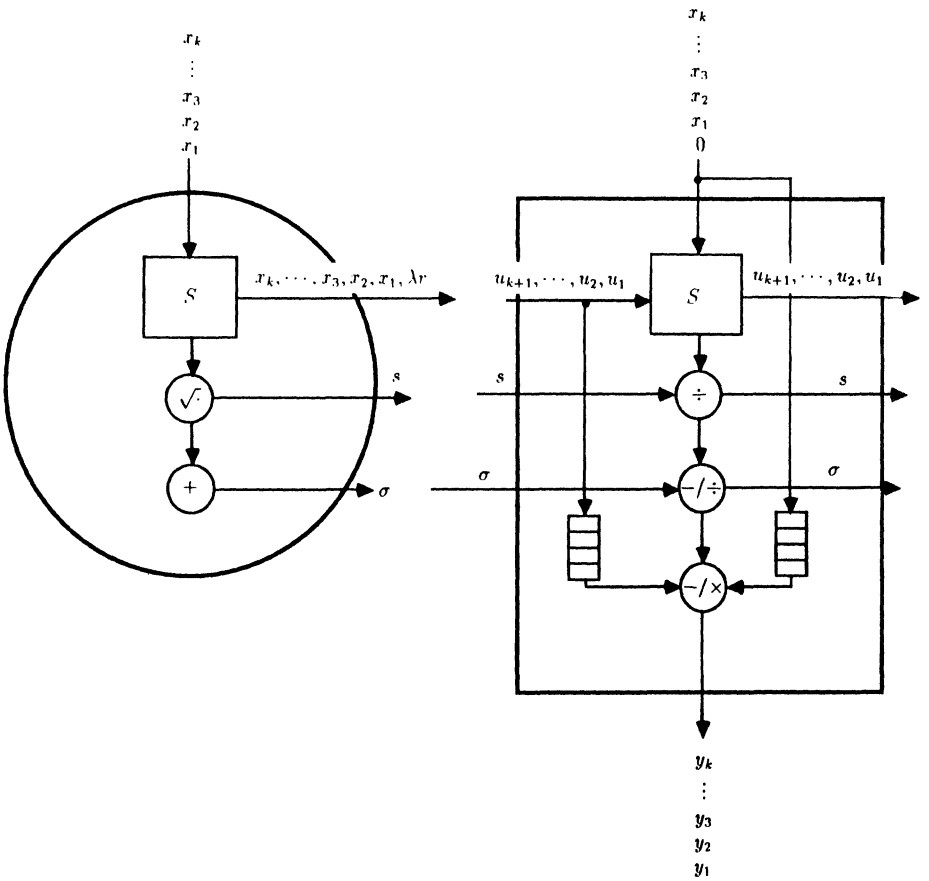
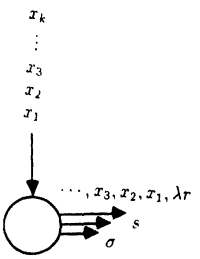


Fig. 5. Operations of the processing cells by using the modified Householder transformation.

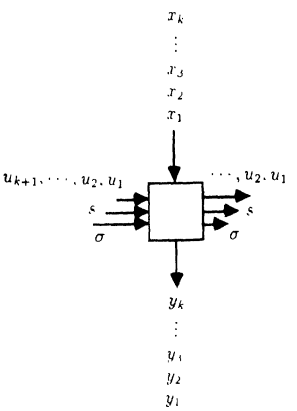
A two-level pipelined implementation of the modified HT algorithm is given in Fig. 6(a). The boundary cell performs three major functions: square and accumulate, square root, and addition. For each data block, the boundary cell fetches one data sample, accumulates the squares of the samples, and sends the data to the right for internal cell. When all the data of the block are processed, the content of S is then sent down for square-root operation. The resultant s is sent to the right for internal cell as well as sent down to obtain σ, which is then sent to the right when available. At the same time, when an internal cell receives a u_i, it multiplies u_i with an input x_i and accumulates all these products to obtain S. When S is available, it is sent down for division operation with s, which arrives at the same time, to obtain t; then t is sent down and σ again arrives at the same time to compute d. To compute y_i of (3) in step 3, we need registers to store u_i and x_i temporarily. Since data from the next block are continuously being sent into the system, each internal cell needs $2(k + 3)$ registers to store u_i and x_i as indicated in Fig. 6(a). When d is available, the y_i are then obtained one by one and sent down for further processing. Data from the next block undergo the same processing. When a new d is available in the internal cell, the corresponding x_i and u_i are already waiting in the registers. Therefore the vector operations are successfully pipelined down to the word level. This means that by using the modified HT algorithm, we have not only pipelined the SBHT arrays at the vector level but also at the word level. The input data is now skewed in the word level as shown in Fig. 6(a) rather than in the vector level as shown in Fig. 4. The functional descriptions of the processing cells for two-level pipelined implementation are given in Fig. 6(b).



(a)



send_right λr
$S \leftarrow \lambda^2 r^2$
do $i = 1, k$
fetch x_i; $S \leftarrow S + x_i^2$; send_right x_i,
end do
$r' \leftarrow \lambda r$; $s \leftarrow \sqrt{S}$; send_right s,
$r \leftarrow s$; $\sigma \leftarrow r' + s$; send_right σ.



$S = 0$.
do $i = 1, K + 1$
fetch u_i.
if $i = 1$ **then**
$S \leftarrow S + u_i \lambda r$; send_right u_i,
else
fetch x_{i-1}; $S = S + u_i x_{i-1}$; send_right u_i.
end if
fetch s; $r' \leftarrow \lambda r$; $t \leftarrow -S/s$,
fetch σ; $r \leftarrow t$; $d \leftarrow (r' - t)/\sigma$.
end do
do $i = 1, k$
$y_i = x_i - d u_{i+1}$; send_down y_i.
end do

(b)

Fig. 6. (a) Architectures of the processing cells for two-level pipelined implementation. (b) Functional descriptions of processing cells for two-level pipelined implementation.

TABLE I
COMPARISONS OF THE SBHT AND GIVENS ROTATION METHODS

	Givens Rotation	SBHT
Number of cells	$(p^2 + 3p)/2$	$(p^2 + 5p)/2$
Number of delay elements	p	$p(p - 1)$
Number of registers	0	$(p^2 + 3p)(k + 3)$
System latency	$2p + 1$	$2p(k + 4)$
Cell complexity	less	higher
Numerical stability	good	better

Since the most time consuming operation of this two-level pipelined implementation is the square root operation which is also the critical operation in McWhirter's Givens rotation implementation, the throughput of this two-level pipelined implementation is as fast as that of the McWhirter's Givens array. However, with a longer pipeline, a longer system latency for the SBHT method is obtained. This is due to the fact that the registers of the internal cells have to be filled before we can obtain the residual vector. For the SBHT RLS systolic array of order p, we have $(p^2 + 3p)/2$ internal cells, including the BPA. Thus, there are a total of $(p^2 + 3p)(k + 3)$ registers for the whole system. The system latency is given by $t_s = 2p(k + 4)$, which is linearly proportional to p and k. However, for the Givens rotation method, the system latency is only $t_s = 2p + 1$. Comparisons of both RLS arrays based on the SBHT and Givens rotation are summarized in Table I. In general, the throughput of the SBHT RLS systolic array is as fast as the Givens rotation method. Of course, while the cell complexity of the SBHT array is higher, it does offer better numerical stability [26]. A detailed backward error analysis carried out by Wilkinson showed that for an $n \times n$ matrix A, after $n(n - 1)/2$ Givens rotations, the roundoff error in the upper triangular matrix is in the order of $\mathcal{O}(\kappa_g n^{3/2} \mu \|A\|)$ [26, p. 138], while a series of $(n - 1)$ HT gives $\mathcal{O}(\kappa_h n \mu \|A\|)$ [26, p. 160], with κ_g and κ_h being constants and μ a machine floating point computation precision.

VI. CONSTRAINED RLS PROBLEMS

In the above sections, we have dealt with an unconstrained RLS problem. The RLS systolic array considered there was motivated originally by the sidelobe canceller beamformation problem [18]. Other practical motivation could have come from the adaptive filtering problem [6]. However, there are other signal processing applications which are modeled by a constrained RLS problem. The MVDR beamformation constitutes such an example [19], [20], [23]. It is interesting to determine whether a systolic array for an unconstrained RLS problem can also be used for a constrained RLS problem. In [19], McWhirter and Shepherd showed an extension of the unconstrained RLS array to the MVDR beamforming problem. Based on their approach, we shall also demonstrate the implementation of a MVDR beamformation problem using a SBHT RLS array.

The MVDR beamforming problem is to minimize

$$\xi^{(l)}(n) = \|X(n)w^{(l)}(n)\|_\Lambda, \qquad l = 1, \cdots, L \tag{40}$$

subject to the linear constaints of

$$c^{(l)T}w^{(l)}(n) = \beta^{(l)}, \qquad l = 1, \cdots, L \tag{41}$$

where L is the number of constraints. We are interested in the *a posteriori* residual vector

$$\hat{e}_n^{(l)}(n) = X_n^T \hat{w}^{(l)}(n). \tag{42}$$

The optimal solution of the weight vector is known [19] to be given by

$$\hat{w}^{(l)}(n) = \frac{\beta^{(l)} M^{-1}(n) c^{(l)}}{c^{(l)T} M^{-1}(n) c^{(l)}} = \frac{\beta^{(l)} R^{-1}(n) a^{(l)}(n)}{\|a^{(l)}(n)\|^2} \tag{43}$$

where $M = X^T(n)\Lambda_k^2(n)X(n)$ is the weighted covariance matrix, $R(n)$ is the upper triangular matrix resulted from the QRD of the weighted data matrix $\Lambda_k X(n)$, and

$$a^{(l)}(n) = R^{-T}(n)c^{(l)}. \tag{44}$$

Therefore the optimal residual vector at time n is

$$\hat{e}_n^{(l)}(n) = \frac{\beta^{(l)}}{\|a^{(l)}(n)\|^2} \cdot X_n^T R^{-1}(n) a^{(l)}(n). \tag{45}$$

A crucial step needed is for the efficient recursive updating of $a^{(l)}(n)$. A novel approach was proposed for performing this updating [19]. Specifically, from (8), (9), and (44)

$$\begin{aligned} c^{(l)} &= R^T(n-1)a^{(l)}(n-1) \\ &= \lambda^{-2}[\lambda R^T(n-1) \vdots 0^T \vdots X_n] \begin{bmatrix} \lambda a^{(l)}(n-1) \\ \hline \lambda b^{(l)}(n-1) \\ \hline 0 \end{bmatrix} \end{aligned} \tag{46}$$

where $b^{(l)}(n - 1)$ is an arbitrary $((n - 1)k - p) \times 1$ vector. Then from Lemma 1, (8), and (9), we have

$$\begin{aligned} c^{(l)} &= \lambda^{-2}[\lambda R^T(n-1) \vdots 0^T \vdots X_n] H^T(n) H(n) \\ &\quad \cdot \begin{bmatrix} \lambda a^{(l)}(n-1) \\ \hline \lambda b^{(l)}(n-1) \\ \hline 0 \end{bmatrix} \\ &= R^T(n) \cdot \lambda^{-2}(\lambda H_{11}(n) a^{(l)}(n-1)). \end{aligned} \tag{47}$$

Thus, $a^{(l)}(n) = \lambda^{-2}(\lambda H_{11}(n)a^{(l)}(n - 1))$ can be obtained by updating $a^{(l)}(n - 1)$ in a way similar to that $u(n)$ is obtained by updating $u(n - 1)$ using (32). The only differences are the input for updating $a^{(l)}(n - 1)$ is a zero vector and a scaling factor λ^{-2}. Due to the structure of H in Lemma 1, the vector $b^{(l)}(\cdot)$ plays no role in the updating of $a^{(l)}(\cdot)$. Furthermore, from (27) and (29), we have

$$\hat{e}_n(n) = X_n^T R^{-1}(n) u(n) - y_n. \tag{48}$$

From (32), we see that $u(n)$ results from the update of

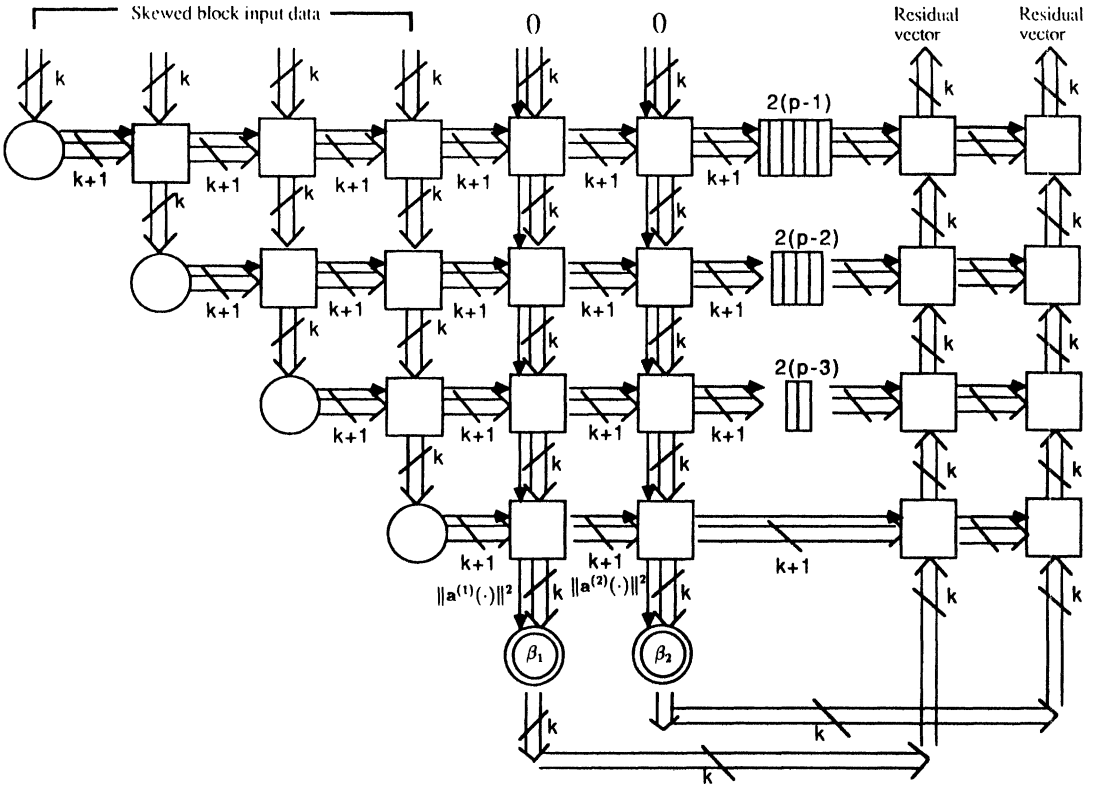


Fig. 7. MVDR beamforming using the SBHT systolic array.

$[y(n-1) \vdots y_n]$, where y_n is the new input. Now replacing $u(n)$ with $a^{(l)}(n)$ and y_n with a zero vector, we have

$$\hat{e}_n(n) = X_n^T R^{-1}(n) a^{(l)}(n) \tag{49}$$

and from (45), we then obtain

$$\hat{e}_n^{(l)}(n) = \frac{\beta^{(l)}}{\|a^{(l)}(n)\|^2} \cdot \hat{e}_n(n). \tag{50}$$

This equation reveals that by the proper scaling of $\hat{e}_n(n)$, which can be obtained from the SBHT RLS systolic array, we can obtain the *a posteriori* residual vector, $\hat{e}_n^{(l)}(n)$, of the MVDR beamformation. Fig. 7 shows an extension of the SBHT RLS array for the new problem. Now one more data channel is needed for the RA to pipeline cumulation of $\|a^{(l)}(n)\|^2$, and the scaling of the residual vector is done at the bottom of the RA when a new $\|a^{(l)}(\cdot)\|^2$ is available. Each RA/BPA pair in Fig. 7 represents one of the K constraints. The optimal *a posteriori* residual vector of each linear constraint is obtained at the output of the corresponding backward propagation array.

As pointed out in [19], there are two ways to initialize the array. One method is to set $R(0) = \delta I$, where δ is a small scalar, and thus from (44), $a^{(l)}(0) = \delta^{-1} c^{(l)}$, $l = 1, \cdots, L$. Another method is to obtain $R(n)$ to some time n, then use (44) to obtain $a^{(l)}(n)$. The details of a two-mode operation required for this initialization procedure are also considered in [19].

VII. Systolic Array for Hyperbolic Householder Transformation

In some applications, the fixed window approach is preferred to the exponentially weighted window approach. The updating of new data and downdating of old data must then be considered. Rader and Steinhardt [22] in 1986 proposed a hyperbolic Householder transformation (HHT) to simultaneously perform up/downdating. Let us define a J-hyperbolic Householder matrix H_J as follows:

$$H_J = J - 2hh^T / \|h\|_J^2 \tag{51}$$

where h is a column vector, J is a pseudoidentity matrix

$$J = \begin{bmatrix} I_p & 0 & 0 \\ 0 & I_k & 0 \\ 0 & 0 & -I_k \end{bmatrix}$$

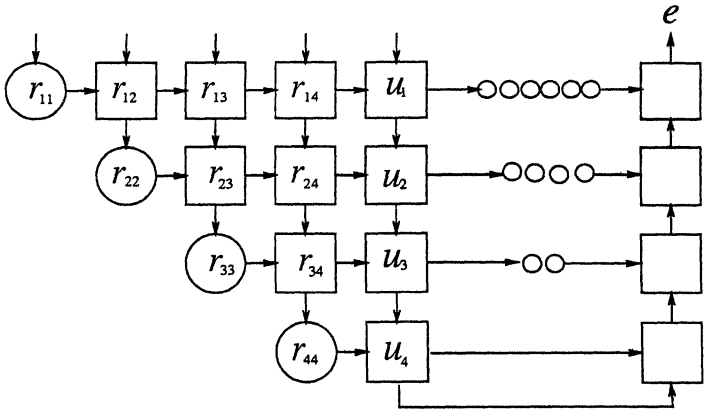
with I_p representing preserving the previous Cholesky factor, I_k incorporating the new data for updating, $-I_k$ discarding the old data for downdating, and

$$\|h\|_J^2 = \sum_{i=1}^{p} h_i^2 + \sum_{i=p+1}^{p+k} h_i^2 - \sum_{i=p+k+1}^{p+2k} h_i^2$$

is the J-pseudo vector norm.

We note that H_J is Hermitian and J-pseudo orthogonal, namely,

$$H_J = H_J^T \tag{52}$$



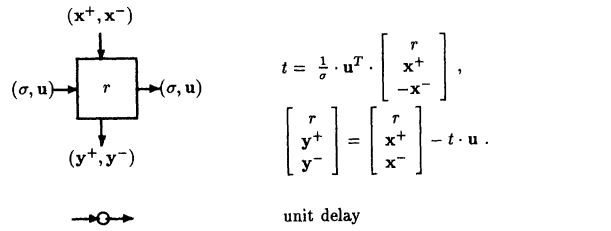


Fig. 8. Block hyperbolic Householder systolic array.

and

$$H_J^T J H_J = J. \tag{53}$$

We can compress all of the J-pseudo energy of a vector $\boldsymbol{a}$ into its jth entry by premultiplying it (performing pseudoorthogonal transformation) by H_J and choosing

$$\boldsymbol{h} = J\boldsymbol{a} + \alpha \boldsymbol{u}_j \tag{54}$$

with

$$\alpha = (\pm a_j / \|a_j\|)\|\boldsymbol{a}\|_J. \tag{55}$$

Here $\boldsymbol{u}_j$ is a unit vector with all zeros except for its jth entry. Then we have

$$H_J \boldsymbol{a} = -\alpha \boldsymbol{u}_j. \tag{56}$$

An algorithm using HHT to update the block data matrix $A = [\boldsymbol{a}_1, \cdots, \boldsymbol{a}_p]$ and downdate the matrix $B = [\boldsymbol{b}_1, \cdots, \boldsymbol{b}_p]$ from the Cholesky factor R is given below:

The HHT Up/Downdating Algorithm

For $i = 1, \cdots, p$, do
  $\tilde{r}_{ii} = \sqrt{r_{ii}^2 + \boldsymbol{a}_i^T \boldsymbol{a}_i - \boldsymbol{b}_i^T \boldsymbol{b}_i}$;
  if $r_{ii} < 0$, $\tilde{r}_{ii} = -\tilde{r}_{ii}$;
  For $j = (i + 1), \cdots, p$, do
    $\tilde{r}_{ij} = (r_{ii} r_{ij} + \boldsymbol{a}_i^T \boldsymbol{a}_j - \boldsymbol{b}_i^T \boldsymbol{b}_j)/\tilde{r}_{ii}$;
    $\boldsymbol{a}_j = \boldsymbol{a}_j - (\tilde{r}_{ij} + r_{ij}/\tilde{r}_{ii} + r_{ii})\boldsymbol{a}_i$;
    $\boldsymbol{b}_j = \boldsymbol{b}_j - (\tilde{r}_{ij} + r_{ij}/\tilde{r}_{ii} + r_{ii})\boldsymbol{b}_i$;
    if $r_{ii} < 0$, $\boldsymbol{a}_j = -\boldsymbol{a}_j$; $\boldsymbol{b}_j = -\boldsymbol{b}_j$;
  End;
End.

Same as previous sections, it can be shown that $\hat{Q}^{\perp} = \hat{H}_J^{(1)\perp} \cdots \hat{H}_J^{(p)\perp}$, where $\hat{H}_J^{(j)\perp} \in \mathcal{R}^{2k \times 2k}$ is the lower right submatrix of the hyperbolic Householder reflection matrix

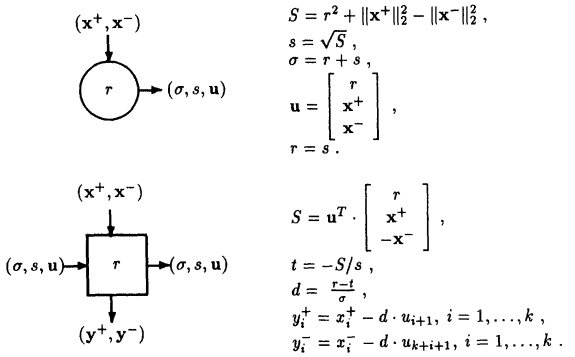


Fig. 9. Modified processing cells for the block HHT systolic array.

in zeroing out the jth column of appended data

$$\begin{bmatrix} X^+ & y^+ \\ X^- & y^- \end{bmatrix}$$

with $[X^+ \quad y^+]$ and $[X^- \quad y^-]$ representing the new and old data block to be up/downdated, respectively. The residual vector $\boldsymbol{e}$ therefore can be written as $\boldsymbol{e} = -\hat{H}_J^{(1)\perp} \cdots \hat{H}_J^{(p)\perp} \boldsymbol{v}$, which can be computed by a series of backward matrix-vector multiplications as given in previous sections. A block HHT systolic array for RLS filtering is given in Fig. 8. Fig. 9 depicts the modified boundary and regular processors based on Tsao's algorithm.

VIII. Conclusions

In this paper, we have shown that the Householder transformation can be implemented on a systolic array. By using a two-level pipelined implementation, the throughput of the SBHT RLS systolic array can be as fast as that of the original Givens array in [18]. While, the system latency is longer for the SBHT, it provides a better numerical stability than the Givens method. Clearly, the Givens array is a special case of the SBHT array with a block size of one. In general, the block size is an important variable. A larger block size results in a better numerical stability, while the system latency is increased. Many known properties of the Givens array are also applicable to the SBHT array. For example, the real-time algorithm-based fault-tolerant scheme proposed in [14] can also be easily incorporated into the SBHT RLS array. From the results described in this paper, it shows that the Householder transformation method is useful in real-time high throughput applications of modern signal processing as well as in VLSI implementation.

References

[1] M. G. Bellanger, "Computational complexity and accuracy issues in fast least squares algorithms for adaptive filtering," in *Proc. IEEE ISCAS* (Finland), 1988, pp. 2635-2639.
[2] J. M. Cioffi, "Limited-precision effects in adaptive filtering," *IEEE Trans. Circuits Syst.*, vol. CAS-34, pp. 821-833, July 1987.
[3] J. M. Cioffi, "The fast Householder filters RLS adaptive filter," in *Proc. IEEE ICASSP* (Albuquerque, NM), Apr. 1990, pp. 1619-1622.
[4] W. M. Gentleman and H. T. Kung, "Matrix triangularization by systolic arrays," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 298, pp. 19-26, 1981.

[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed. Baltimore, MD: Johns Hopkins University Press, 1989.
[6] S. Haykin, *Adaptive Filter Theory.* Englewood Cliffs, NJ: Prentice-Hall, 1986.
[7] D. E. Heller and I. C. F. Ipsen, "Systolic networks for orthogonal decomposition," *SIAM J. Sci. Stat. Comput.*, vol. 4, pp. 261–269, June 1983.
[8] L. Johnsson, "A computational array for the *QR* method," in *Proc. 1982 Conf. Advanced Res. VLSI* (M.I.T., Cambridge, MA), pp. 123–129.
[9] S. Kalson and K. Yao, "Systolic array processing for order and time recursive generalized least-squares estimation," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 564, pp. 28–38, 1985.
[10] H. T. Kung and M. S. Lam, "Wafer-scale integration and two-level pipelined implementation of systolic array," *J. Parallel Distrib. Comput.*, vol. 1, pp. 32–63, 1984.
[11] S. Y. Kung, *VLSI Array Processors.* Englewood Cliffs, NJ: Prentice-Hall, 1988.
[12] H. Leung and S. Haykin, "Stability of recursive QRD LS algorithms using finite-precision systolic array implementation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 760–763, May 1989.
[13] F. Ling, D. Manolakis, and J. G. Proakis, "A recursive modified Gram–Schmidt algorithm for least squares estimation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 829–836, Aug. 1986.
[14] K. J. R. Liu and K. Yao, "Gracefully degradable real-time algorithm-based fault-tolerant method for *QR* recursive least-squares systolic array," in *Proc. Int. Conf. Systolic Array* (Killarney, Ireland), May 1989, pp. 401–410.
[15] K. J. R. Liu, S. F. Hsieh, and K. Yao, "Recursive LS filtering using block Householder transformations," in *Proc. IEEE ICASSP* (Albuquerque, NM), Apr. 1990, pp. 1631–1634.
[16] F. T. Luk, "A rotation method for computing the *QR* decomposition," *SIAM J. Sci. Stat. Comput.*, vol. 7, pp. 452–459, Apr. 1986.
[17] F. T. Luk and S. Qiao, "Analysis of a recursive least squares signal-processing algorithm," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 3, pp. 407–418, May 1989.
[18] J. G. McWhirter, "Recursive least-squares minimization using a systolic array," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 431, pp. 105–112, 1983.
[19] J. G. McWhirter and T. J. Shepherd, "Systolic array processor for MVDR beamforming," *Proc. Inst. Elec. Eng.*, vol. 136, pt. F, no. 2, pp. 75–80, 1989.
[20] N. L. Owsley, "Sonar array processing," in *Array Signal Processing*, Haykin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 115–193.
[21] B. N. Parlett, "Analysis of algorithms for reflections in bisectors," *SIAM Rev.*, vol. 13, no. 2, pp. 197–208, Apr. 1971.
[22] C. M. Rader and A. O. Steinhardt, "Hyperbolic Householder transformations," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 34, pp. 1589–1602, Dec. 1986.
[23] R. Schreiber, "Implementation of adaptive array algorithms," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 1038–1045, Oct. 1986.
[24] A. Steinhardt, "Householder transformation," *IEEE ASSP Mag.*, vol. 15, pp. 4–12, 1988.
[25] N.-K. Tsao, "A note on implementing the Householder transformation," *SIAM J. Numer. Anal.*, vol. 12, no. 1, pp. 53–58, Mar. 1975.
[26] J. H. Wilkinson, *The Algebraic Eigenvalue Problem.* Oxford, 1965.
[27] J. H. Wilkinson, "Modern error analysis," *SIAM Rev.*, vol. 13, no. 4, pp. 548–568, Oct. 1971.

A Class of Square Root and Division Free Algorithms and Architectures for QRD-Based Adaptive Signal Processing

E. N. Frantzeskakis, *Member, IEEE,* and K. J. R. Liu, *Senior Member, IEEE*

Abstract— **The least squares (LS) minimization problem constitutes the core of many real-time signal processing problems, such as adaptive filtering, system identification and adaptive beamforming. Recently efficient implementations of the recursive least squares (RLS) algorithm and the constrained recursive least squares (CRLS) algorithm based on the numerically stable QR decomposition (QRD) have been of great interest. Several papers have proposed modifications to the rotation algorithm that circumvent the square root operations and minimize the number of divisions that are involved in the Givens rotation. It has also been shown that all the known square root free algorithms are instances of one parametric algorithm. Recently, a square root free and division free algorithm has also been proposed. In this paper, we propose a family of square root and division free algorithms and examine its relationship with the square root free parametric family. We choose a specific instance for each one of the two parametric algorithms and make a comparative study of the systolic structures based on these two instances, as well as the standard Givens rotation. We consider the architectures for both the optimal residual computation and the optimal weight vector extraction. The dynamic range of the newly proposed algorithm for QRD-RLS optimal residual computation and the wordlength lower bounds that guarantee no overflow are presented. The numerical stability of the algorithm is also considered. A number of obscure points relevant to the realization of the QRD-RLS and the QRD-CRLS algorithms are clarified. Some systolic structures that are described in this paper are very promising, since they require less computational complexity (in various aspects) than the structures known to date and they make the VLSI implementation easier.**

I. Introduction

THE least squares (LS) minimization problem constitutes the core of many real-time signal processing problems, such as adaptive filtering, system identification and beamforming [6]. There are two common variations of the LS problem for adaptive signal processing:

1) Solve the minimization problem

$$w(n) = \arg\min_{w(n)} \|\beta(n)(X(n)w(n) - y(n))\|^2 \qquad (1)$$

where $X(n)$ is a matrix of size $n \times p$, $w(n)$ is a vector of length p, $y(n)$ is a vector of length n and $\beta(n) = \text{diag}\{\beta^{n-1}, \beta^{n-2}, \cdots, 1\}, 0 < \beta < 1$, that is, β is a forgetting factor.

2) Solve the minimization problem in (1) subject to the linear constraints

$$c^{iT} w(n) = r^i, \qquad i = 1, 2, \cdots, N \qquad (2)$$

where c^i is a vector of length p and r^i is a scalar. In this paper, we consider only the special case of the minimum variance distortionless response (MVDR) beamforming problem [15] for which $y(n) = 0$ for all n and (1) is solved by subjecting to each linear constraint, i.e., there are N linear-constrained LS problems.

There are two different pieces of information that may be required as the result of this minimization [6]:

1) The optimizing weight vector $w(n)$ and/or
2) the optimal residual at the time instant n

$$e(t_n) = X(t_n)w(n) - y(t_n) \qquad (3)$$

where $X(t_n)$ is the last row of the matrix $X(n)$ and $y(t_n)$ is the last element of the vector $y(n)$.

Efficient implementations of the recursive least squares (RLS) algorithms and the constrained recursive least squares (CRLS) algorithms based on the QR decomposition (QRD) were first introduced by McWhirter [14], [15]. A comprehensive description of the algorithms and the architectural implementations of these algorithms is given in [6, chap.14]. It has been proved that the QRD-based algorithms have good numerical properties [6]. However, they are not very appropriate for VLSI implementation, because of the square root and the division operations that are involved in the Givens rotation and the backprintingsubstitution required for the case of weight extraction.

Several papers have proposed modifications in order to reduce the computational load involved in the original Givens rotation [2], [5], [4], [8]. These rotation-based algorithms are not rotations any more, since they do not exhibit the normalization property of the Givens rotation. Nevertheless, they can substitute for the Givens rotation as the building block of the QRD algorithm and thus they can be treated as rotation algorithms in a wider sense:

Manuscript received June 14, 1992; revised January 14, 1994. This work was supported in part by the ONR grant N00014-93-1-0566, the AASERT/ONR grant N00014-93-11028, and the NSF grant MIP9309506. The associate editor coordinating the review of this paper and approving it for publication was Prof. Monty Hayes.

The authors are with the Electrical Engineering Department and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA.

IEEE Log Number 9403256.

Reprinted from *IEEE Transactions on Signal Processing*, pp. 2455-2469, September 1994.

Definition 1: A Givens-rotation-based algorithm that can be used as the building block of the QRD algorithm will be called a $\Re$otation algorithm.

A number of square-root-free $\Re$otations have appeared in the literature [2], [5], [8], [10]. It has been shown that a square-root-free and division-free $\Re$otation does exist [4]. Recently, a parametric family of square-root-free $\Re$otation algorithms was proposed in [8]; it was also shown that all the known square-root-free $\Re$otation algorithms belong to this family, which is called the "$\mu\nu$-family." In this paper we will refer to the $\mu\nu$-family of $\Re$otation algorithms with the name *parametric $\mu\nu$ $\Re$otation*. We will also say that a $\Re$otation algorithm is a*$\mu\nu$ $\Re$otation* if this algorithm belongs to the $\mu\nu$-family. Several QRD-based algorithms have made use of these $\Re$otation algorithms. McWhirter has been able to compute the optimal residual of the RLS algorithm without square root operations [14]. He also employed an argument for the similarity of the RLS and the CRLS algorithms to obtain a square-root-free computation for the optimal residual of the CRLS algorithm [15]. A fully-pipelined structure for weight extraction that circumvents the back-substitution divisions was also derived independently in [17] and in [19]. Finally, an algorithm for computing the RLS optimal residual based on the parametric $\mu\nu$ $\Re$otation was derived in [8].

In this paper, we introduce a parametric family of square-root-free and division-free $\Re$otations. We will refer to this family of algorithms with the name *parametric $\kappa\lambda$ $\Re$otation*. We will also say that a $\Re$otation algorithm is *a $\kappa\lambda$ $\Re$otation* if this algorithm is obtained by the parametric $\kappa\lambda$ $\Re$otation with a choice of specific values for the parameters κ and λ. We employ the arguments in [8], [14], [15] and [17] in order to design novel architectures for the RLS and the CRLS algorithms that have less computation and circuit complexity. Some systolic structures that are described here are very promising, since they require the minimum computational complexity (in various aspects) known to date, and they can be easily implemented in VLSI.

In Section II, we introduce the parametric $\kappa\lambda$ $\Re$otation. In Section III, we derive the RLS algorithms that are based on the parametric $\kappa\lambda$ $\Re$otation and we consider the architectural implementations for a specific $\kappa\lambda$ $\Re$otation. In Section IV, we follow the same procedure for the CRLS algorithms. In Section V, we address the issues of dynamic range, lower bounds for the wordlength, stability and error bounds. We conclude with Section VI. In the Appendix we give the proofs of some lemmas that are stated in the course of the paper.

II. Square Root and Division Free Algorithms

In this section, we introduce a new parametric family of Givens-rotation-based algorithms that require neither square root nor division operations. This modification to the Givens rotation provides a better insight on the computational complexity optimization issues of the QR decomposition and makes the VLSI implementation easier.

A. The Parametric $\kappa\lambda$ $\Re$otation

The standard Givens rotation operates (for real-valued data) as follows:

$$\begin{bmatrix} r_1' & r_2' & \cdots & r_m' \\ 0 & x_2' & \cdots & x_m' \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} \beta r_1 & \beta r_2 & \cdots & \beta r_m \\ x_1 & x_2 & \cdots & x_m \end{bmatrix} \tag{4}$$

where

$$c = \frac{\beta r_1}{\sqrt{\beta^2 r_1^2 + x_1^2}}, \qquad s = \frac{x_1}{\sqrt{\beta^2 r_1^2 + x_1^2}} \tag{5}$$

$$r_1' = \sqrt{\beta^2 r_1^2 + x_1^2} \tag{6}$$

$$r_j' = c\beta r_j + s x_j, \qquad j = 1, 2, \cdots, m \tag{7}$$

$$x_j' = -s\beta r_j + c x_j, \qquad j = 2, 3, \cdots, m. \tag{8}$$

We introduce the following data transformation

$$r_j = \frac{1}{\sqrt{l_a}} a_j, \qquad x_j = \frac{1}{\sqrt{l_b}} b_j, \qquad r_j' = \frac{1}{\sqrt{l_a'}} a_j', \quad j = 1, 2, \cdots, m$$

$$x_j' = \frac{1}{\sqrt{l_b'}} b_j', \qquad j = 2, 3, \cdots, m. \tag{9}$$

We seek the square root and division-free expressions for the transformed data $a_j', j = 1, 2, \cdots, m, b_j', j = 2, 3, \cdots, m$, in (6) and solving for a_1', we get

$$a_1' = \sqrt{\frac{l_a'}{l_a l_b}(l_b \beta^2 a_1^2 + l_a b_1^2)}. \tag{10}$$

By substituting (5) and (9) in (7) and (8) and solving for a_j' and b_j', we get

$$a_j' = \frac{l_b \beta^2 a_1 a_j + l_a b_1 b_j}{\sqrt{l_a l_b (l_b \beta^2 a_1^2 + l_a b_1^2)/l_a'}} \quad \text{and}$$

$$b_j' = \frac{-b_1 \beta a_j + \beta a_1 b_j}{\sqrt{(l_b \beta^2 a_1^2 + l_a b_1^2)/l_b'}} \qquad j = 2, 3, \cdots, m. \tag{11}$$

We will let l_a' and l_b' be equal to

$$l_a' = l_a l_b (l_b \beta^2 a_1^2 + l_a b_1^2)\kappa^2, \qquad l_b' = (l_b \beta^2 a_1^2 + l_a b_1^2)\lambda^2 \tag{12}$$

where κ and λ are two parameters. By substituting (12) in (10)–(11) we obtain the expressions

$$a_1' = \kappa(l_b \beta^2 a_1^2 + l_a b_1^2) \tag{13}$$

$$a_j' = \kappa(l_b \beta^2 a_1 a_j + l_a b_1 b_j), \qquad j = 2, 3, \cdots, m \text{ and} \tag{14}$$

$$b_j' = \lambda\beta(-b_1 a_j + a_1 b_j), \qquad j = 2, 3, \cdots, m. \tag{15}$$

If the evaluation of the parameters κ and λ does not involve any square root or division operations, the update equations (12)–(15) will be square root and division-free. In other words, every such choice of the parameters κ and λ specifies a square root and division-free $\Re$otation algorithm.

Definition 2: Equations (12)–(15) specify the parametric $\kappa\lambda$ $\Re$otation algorithm. Furthermore, a $\Re$otation algorithm will be called a $\kappa\lambda$ $\Re$otation if it is specified by (12)–(15) for specific square-root-free and division-free expressions of the parameters κ and λ.

One can easily verify that the only one square root and division-free $\Re$otation in the literature to date [4] is a $\kappa\lambda$ $\Re$otation and is obtained by choosing $\kappa = \lambda = 1$.

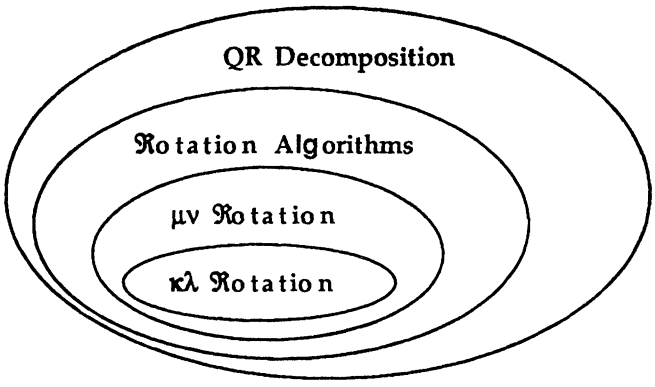


Fig. 1. The relations among the classes of algorithms based on QR decomposition, a Rotation algorithm, a $\mu\nu$ Rotation, and a $\kappa\lambda$ Rotation.

B. The Relation between the Parametric $\kappa\lambda$ and the Parametric $\mu\nu$ Rotation

Let

$$k_a = \frac{1}{l_a}, \qquad k_b = \frac{1}{l_b}, \qquad k'_a = \frac{1}{l'_a}, \qquad k'_b = \frac{1}{l'_b}. \tag{16}$$

We can express k'_a and k'_b in terms of k_a and k_b as follows [8]

$$k'_a = \left(k_a\beta^2 a_1^2 + k_b b_1^2\right)/\mu^2, \qquad k'_b = \frac{k_a k_b}{\mu^2\nu^2}\frac{1}{k'_a}. \tag{17}$$

If we substitute (16) and (17) in (12) and solve for μ and ν we obtain

$$\mu = \frac{\kappa(k_a\beta^2 a_1^2 + k_b b_1^2)}{k_a k_b}, \qquad \nu = \lambda. \tag{18}$$

The above provides a proof for the following Lemma:

Lemma 2.1: For each square root and division-free pair of parameters (κ, λ) that specifies a $\kappa\lambda$ Rotation algorithm $A1$, we can find square-root-free parameters $(\mu(\kappa), \nu(\lambda))$ with two properties: first, the pair $(\mu(\kappa), \nu(\lambda))$ specifies a $\mu\nu$ Rotation algorithm $A2$ and second, both $A1$ and $A2$ are mathematically equivalent[1]. □

Consequently, the set of $\kappa\lambda$ Rotation algorithms can be thought of as a subset of the set of the $\mu\nu$ Rotations. Furthermore, (18) provides a means of mapping a $\kappa\lambda$ Rotation onto a $\mu\nu$ Rotation. For example, one can verify that the square root and division-free algorithm in [4] is a $\mu\nu$ Rotation and is obtained for

$$\mu = \frac{k_a\beta^2 a_1^2 + k_b b_1^2}{k_a k_b}, \qquad \nu = 1.$$

In Fig. 1, we draw a graph that summarizes the relations among the classes of algorithms based on QR decomposition, a Rotation algorithm, a $\mu\nu$ Rotation and a $\kappa\lambda$ Rotation.

III. RLS Algorithm and Architecture

In this section, we consider the $\kappa\lambda$ Rotation for optimal residual and weight extraction using systolic array implementation. Detailed comparisons with existing approaches are presented.

[1] They evaluate logically equivalent equations.

A. A Novel Fast Algorithm for the RLS Optimal Residual Computation

The QR-decomposition of the data at time instant n is as follows

$$\begin{bmatrix} R(n) & u(n) \\ 0^T & v(t_n) \end{bmatrix} = T(n)\begin{bmatrix} \beta R(n-1) & \beta u(n-1) \\ X(t_n) & y(t_n) \end{bmatrix} \tag{19}$$

where $T(n)$ is a unitary matrix of size $(p+1)\times(p+1)$ that performs a sequence of p Givens rotations. This can be written symbolically as

$$\begin{bmatrix} L(n-1)^{-\frac{1}{2}} & 0 \\ 0^T & l_q(n)^{-\frac{1}{2}} \end{bmatrix}\begin{bmatrix} \beta\bar{R}(n-1) & \beta\bar{u}(n-1) \\ \bar{X}(t_n) & \bar{y}(t_n) \end{bmatrix}$$
$$\xrightarrow{T(n)}\begin{bmatrix} L(n)^{-\frac{1}{2}} & 0 \\ 0^T & l_q(n+1)^{-\frac{1}{2}} \end{bmatrix}\begin{bmatrix} \bar{R}(n) & \bar{u}(n) \\ 0^T & b_{p+1}^{(p)} \end{bmatrix} \tag{20}$$

where

$$L(\cdot)^{-\frac{1}{2}}\bar{R}(\cdot) = R(\cdot), \qquad L(\cdot)^{-\frac{1}{2}}\bar{u}(\cdot) = u(\cdot),$$
$$l_q(n)^{-\frac{1}{2}}\bar{X}(t_n) = X(t_n), \qquad l_q(n)^{-\frac{1}{2}}\bar{y}(t_n) = y(t_n) \tag{21}$$

and

$$L(n-1) = \mathrm{diag}\{l_1, l_2, \cdots, l_p\}$$
$$L(n) = \mathrm{diag}\{l'_1, l'_2, \cdots, l'_p\},$$
$$\bar{R}(n-1) = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ & a_{22} & \cdots & a_{2p} \\ & & \ddots & \vdots \\ & & & a_{pp} \end{bmatrix}$$
$$\bar{R}(n) = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1p} \\ & a'_{22} & \cdots & a'_{2p} \\ & & \ddots & \vdots \\ & & & a'_{pp} \end{bmatrix}$$
$$\bar{u}(n-1) = \left[a_{1,p+1}\ a_{2,p+1}\cdots\ a_{p,p+1}\right]^T$$
$$\bar{u}(n) = \left[a'_{1,p+1}\ a'_{2,p+1}\cdots\ a'_{p,p+1}\right]^T$$
$$\left[\bar{X}(t_n)\ \bar{y}(t_n)\right] = [b_1\ b_2\cdots\ b_p\ b_{p+1}].$$

Equations (12)–(15) imply that the ith Rotation is specified as follows

$$l'_i = l_i l_q^{(i-1)}(l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2})\kappa_i^2 \tag{22}$$
$$l_q^{(i)} = (l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2})\lambda_i^2 \tag{23}$$
$$a'_{ij} = \kappa_i(l_q^{(i-1)}\beta^2 a_{ii}a_{ij} + l_i b_i^{(i-1)}b_j^{(i-1)}),$$
$$j = i, i+1, \cdots, p+1 \tag{24}$$
$$b_j^{(i)} = \lambda_i\beta(-b_i^{(i-1)}a_{ij} + a_{ii}b_j^{(i-1)}),$$
$$j = i+1, i+2, \cdots, p+1 \tag{25}$$

where $i = 1, 2, \cdots, p$, $b_j^{(0)} = b_j, j = 1, \cdots, p+1$ and $l_q^{(0)} = l_q$. For the optimal residual we have:

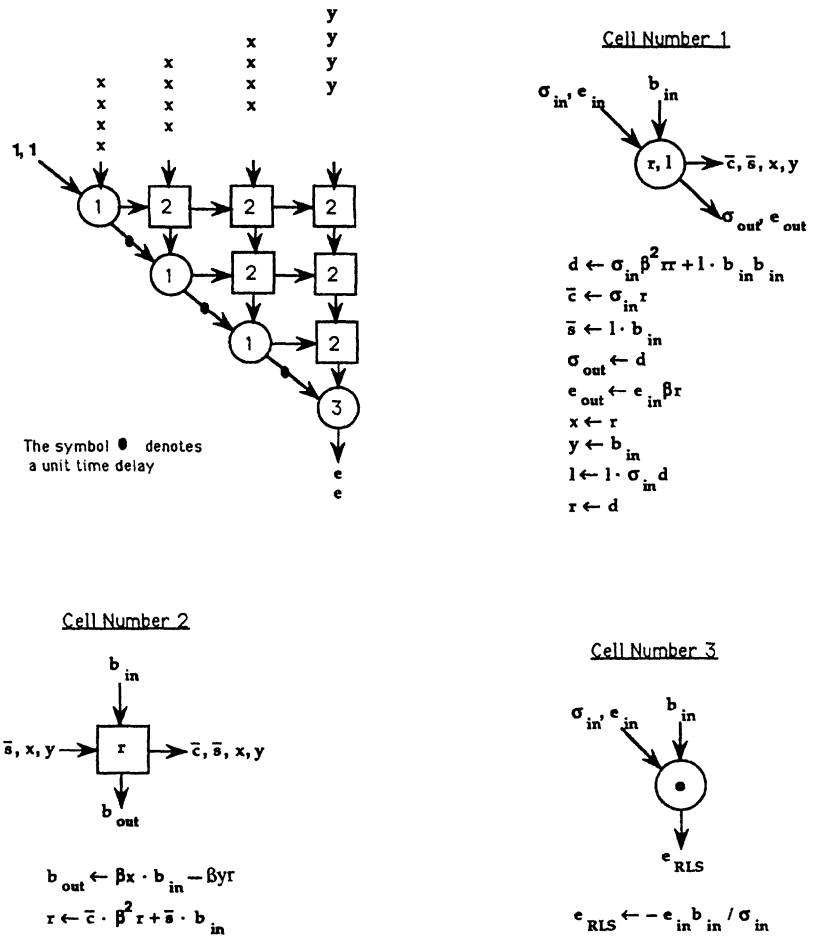


Fig. 2. $S1.1$: Systolic array that computes the RLS optimal residual. It implements the algorithm that is based on the $\kappa\lambda$ Rotation for which $\kappa = \lambda = 1$.

Lemma 3.1: If the parametric $\kappa\lambda$ Rotation is used in the QRD-RLS algorithm, the optimal residual is given by the expression

$$e_{RLS}(t_n) = -\left(\prod_{i=1}^{p-1} \lambda_i \beta a_{ii}\right) \frac{\kappa_p \beta a_{pp}}{\lambda_p a'_{pp}} b_{p+1}^{(p)} \sqrt{l_q}. \qquad (26)$$

□

The proof is given in the Appendix.

Here, l_q is a free variable. If we choose $l_q = 1$, we can avoid the square root operation. We can see that for a recursive computation of (26) only one division operation is needed at the last step of the recursion. This compares very favorably with *the square root free fast algorithms* that require one division for every recursion step, as well as with the original approach (63), which involves one division and one square root operation for every recursion step.

The division operation in (26) cannot be avoided by proper choice of expressions for the parameters κ and λ. This is restated by the following Lemma, which is proved in the Appendix:

Lemma 3.2: If a $\kappa\lambda$ Rotation is used, the RLS optimal residual evaluation will require at least one division evaluation. □

Note that the proper choice of the expression for the parameter λ_p, along with the rest of the parameters, is an open question, since the minimization of the multiplication operations, as well as communication and stability issues have to be considered.

B. A Systolic Architecture for the Optimal RLS Residual Evaluation

McWhirter has used a systolic architecture for the implementation of the QR decomposition [14]. This architecture is modified, so that equations (22)–(26) be evaluated for the special case of $\kappa_i = \lambda_i = 1, i = 1, 2, \cdots, p$ and $l_q = 1$. The systolic array, as well as the memory and the communication links of its components, are depicted in Fig. 2[2]. The boundary cells (cell number 1) are responsible for evaluating (22) and (23), as well as the coefficients $\bar{c}_i = l_q^{(i-1)} a_{ii}$ and $\bar{s}_i = l_i b_i^{(i-1)}$ and the partial products $e_i = \prod_{j=1}^{i} (\beta a_{jj})$. The internal cells (cell number 2) are responsible for evaluating (24) and (25). Finally, the output cell (cell number 3) evaluates (26). The functionality of each one of the cells is described in Fig. 2. We will call this systolic array $S1.1$.

[2]Note the aliases $l_q^{(i-1)} \equiv \sigma_{in}, l_q^{(i)} \equiv \sigma_{out}, l_i \equiv l, a_{ij} \equiv r, b_j^{(i-1)} \equiv b_{in}, b_j^{(i)} \equiv b_{out}, e_{i-1} \equiv e_{in}, e_i \equiv e_{out}$.

TABLE I
RLS RESIDUAL COMPUTATIONAL COMPLEXITY

	$S1.1:\kappa\lambda$			$S1.2:\mu\nu$			$S1.3$: Givens rotation		
cell	1	2	3	1	2	3	1	2	3
number of	p	$\frac{p(p+1)}{2}$	1	p	$\frac{p(p+1)}{2}$	1	p	$\frac{p(p+1)}{2}$	1
sq.rt	-	-	-	-	-	-	1	-	-
div.	-	-	1	1	-	-	1	-	-
mult.	9	4	1	5	3	1	4	4	1
i/o	9	10	4	6	8	3	5	6	3

On Table I, we collect some features of the systolic structure $S1.1$ and the two structures, $S1.2$ and $S1.3$, in [14] that are pertinent to the circuit complexity. The $S1.2$ implements the square-root-free QRD-RLS algorithm with $\mu = \nu = 1$, while $S1.3$ is the systolic implementation based on the original Givens rotation. In Table I, the complexity per processor cell and the number of required processor cells are indicated for each one of the three different cells[3]. One can easily observe that $S1.1$ requires only one division operator and no square root operator, $S1.2$ requires p division operators and no square root operator, while $S1.3$ requires p division and p square root operators. This reduction of the complexity in terms of division and square root operators is penalized with the increase of the number of the multiplications and the communication links that are required.

Apart from the circuit complexity that is involved in the implementation of the systolic structures, another feature of the computational complexity is the number of *operations-per-cycle*. This number determines the minimum required delay between two consecutive sets of input data. For the structures $S1.2$ and $S1.3$ the boundary cell (cell number 1) constitutes the bottleneck of the computation and therefore it determines the operations-per-cycle that are shown on Table V. For the structure $S1.1$ either the boundary cell or the output cell are the bottleneck of the computation.

C. A Systolic Architecture for the Optimal RLS Weight Extraction

Shepherd *et al.* [17] and Tang *et al.* [19] have independently shown that the optimal weight vector can be evaluated in a recursive way. More specifically, one can compute recursively the term $R^{-T}(n)$ by

$$\begin{bmatrix} R^{-T}(n) \\ \# \end{bmatrix} = T(n) \begin{bmatrix} \frac{1}{\beta} R^{-T}(n-1) \\ 0^T \end{bmatrix} \tag{27}$$

and then use parallel multiplication for computing $w^T(n)$ by

$$w^T(n) = u^T(n) R^{-T}(n). \tag{28}$$

The symbol # denotes a term of no interest. The above algorithm can be implemented by a fully pipelined systolic array that can operate in two distinct modes, 0 and 1. The initialization phase consists of $2p$ steps for each processor. During the first p steps the processors operate in mode 0 in order to calculate a full rank matrix R. During the following p steps, the processors operate in mode 1 in order to compute R^{-T}, by performing a task equivalent to forward substitution.

[3]The multiplications with the constants β and β^2 are not encountered.

After the initialization phase the processors operate in mode 0. In [17] one can find the systolic array implementations based both on the original Givens rotation and the Gentleman's variation of the square-root-free Rotation, that is, the $\mu\nu$ Rotation for $\mu = \nu = 1$. We will call these two structures $S2.3$ and $S2.2$, respectively.

In Fig. 3, we present the systolic structure $S2.1$ based on the $\kappa\lambda$ Rotation with $\kappa_i = \lambda_i = 1, i = 1, 2, \cdots, p$. This is a square-root-free and division-free implementation. The boundary cells (cell number 1) are slightly simpler than the corresponding ones of the array $S1.1$. More specifically, they do not compute the partial products e_i. The internal cells (cell number 2), that compute the elements of the matrix R, are identical to the corresponding ones of the array $S1.1$. The cells that are responsible for computing the vector u (cell number 3) differ from the other internal cells only in the fact that they communicate their memory value with their right neighbors. The latter (cell number 4) are responsible for evaluating (28) and (27). The functionality of the processing cells, as well as their communication links and their memory contents, are given in Fig. 3. The mode of operation of each cell is controlled by the *mode bit* provided from the input. For a more detailed description of the operation of the mode bit one can see [15] and [17].

On Tables II and V, we collect some computational complexity metrics for the systolic arrays $S2.1$, $S2.2$ and $S2.3$, when they operate in mode 0[4]. The conclusions we can draw are similar to the ones we had for the circuits that calculate the optimal residual: the square root operations and the division operations can be eliminated with the cost of an increased number of multiplication operations and communication links. We should also note that $S2.1$ does require the implementation of division operators in the boundary cells, since these operators are used during the initialization phase. Nevertheless, after the initialization phase the circuit will not suffer from any time delay caused by division operations. The computational bottleneck of all three structures, $S2.1$, $S2.2$ and $S2.3$, is the boundary cell, thus it determines the operations-per-cycle metric.

As a conclusion for the RLS architectures, we observe that the figures on Tables I, II, and V favor the architectures based on the $\kappa\lambda$ Rotation, $\kappa = \lambda = 1$ versus the ones that are based on the $\mu\nu$ rotation with $\mu = \nu = 1$ and the standard Givens rotation. This claim is clearly substantiated by the delay times on Table V, associated to the DSP implementation of the QRD-RLS algorithm. These delay times are calculated on the basis of the manufacturers benchmark speeds for floating point operations [stewart]. Due to the way of updating R^{-1}, such a weight extraction scheme will have a numerical stability problem if the weight vector at each time instant is required.

IV. CRLS ALGORITHM AND ARCHITECTURE

The optimal weight vector $w^i(n)$ and the optimal residual $e^i_{CRLS}(t_n)$ that correspond to the ith constraint vector c^i are given by the expressions [15]

[4]The multiplications with the constants β, β^2, $1/\beta$ and $1/\beta^2$, as well as the communication links that drive the mode bit, are not encountered.

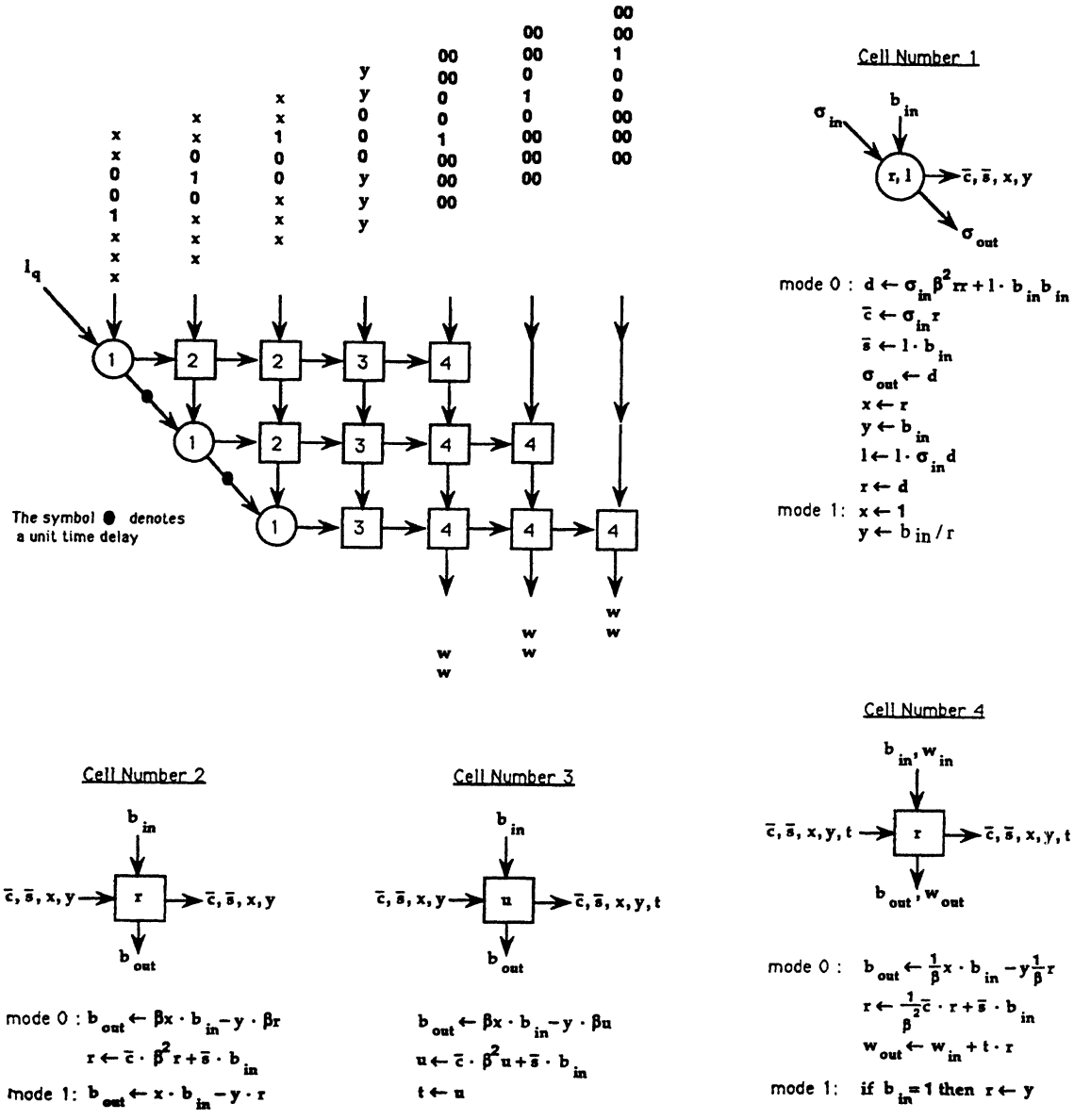


Fig. 3. $S2.1$: Systolic array that computes the RLS optimal weight vector. It implements the algorithm that is based on the $\kappa\lambda$ ℜotation for which $\kappa = \lambda = 1$.

TABLE II
RLS WEIGHT EXTRACTION COMPUTATIONAL COMPLEXITY (MODE 0)

	$S2.1 : \kappa\lambda$				$S2.2 : \mu\nu$				$S2.3$: Givens rotation			
cell	1	2	3	4	1	2	3	4	1	2	3	4
number of	p	$\frac{(p-1)p}{2}$	p	$\frac{p(p+1)}{2}$	p	$\frac{(p-1)p}{2}$	p	$\frac{p(p+1)}{2}$	p	$\frac{(p-1)p}{2}$	p	$\frac{p(p+1)}{2}$
sq.rt	-	-	-	-	-	-	-	-	1	-	-	-
div.	-	-	-	-	1	-	-	-	1	-	-	-
mult.	8	4	4	5	5	3	3	4	4	4	4	5
i/o	7	10	11	14	6	8	9	12	3	6	7	10

$$w^i(n) = \frac{r^i}{\|z^i(n)\|^2} R^{-1}(n) z^i(n) \qquad (29)$$

and

$$e^i_{CRLS}(t_n) = \frac{r^i}{\|z^i(n)\|^2} \hat{e}^i_{CRLS}(t_n) \qquad (30)$$

where

$$\hat{e}^i_{CRLS}(t_n) = X(t_n) R^{-1}(n) z^i(n). \qquad (31)$$

The term $z^i(n)$ is defined as follows

$$z^i(n) = R^{-T}(n) c^i \qquad (32)$$

and it is computed with the recursion [15]

$$\begin{bmatrix} z^i(n) \\ \# \end{bmatrix} = T(n) \begin{bmatrix} \frac{1}{\beta} z^i(n-1) \\ 0^T \end{bmatrix} \tag{33}$$

where the symbol # denotes a term of no interest. In this section, we derive a variation of the recursion that is based on the parametric $\kappa\lambda$ $\Re$otation. Then, we design the systolic arrays that implement this recursion for $\kappa = \lambda = 1$. We also make a comparison of these systolic structures with those based on the Givens rotation and the $\mu\nu$ $\Re$otation introduced by Gentleman [6], [2], [15], [17].

From (32) and (21) we have $z^i(n) = \left(L(n)^{-1/2}\bar{R}(n)\right)^{-T} c^i$ and since $L(n)$ is a diagonal real valued matrix we get $z^i(n) = L(n)^{1/2}\bar{R}(n)^{-T}c^i$, where c^i is the constraint direction. If we let

$$\bar{z}^i(n) = L(n)\bar{R}(n)^{-T}c^i \tag{34}$$

we obtain

$$z^i(n) = L(n)^{-1/2}\bar{z}^i(n). \tag{35}$$

From (35) we get $\|z^i(n)\|^2 = \bar{z}^{i^T}(n)L^{-1}(n)\bar{z}^i(n)$. Also, from (21) and (35) we get $R^{-1}(n)z^i(n) = \bar{R}^{-1}(n)\bar{z}^i(n)$. Consequently, from (29), (30), (31) we have

$$e^i_{CRLS}(n) = \frac{r^i}{\bar{z}^{i^T}(n)L^{-1}(n)\bar{z}^i(n)} \hat{e}^i_{CRLS}(n) \tag{36}$$

and

$$w^i(n) = \frac{r^i}{\bar{z}^{i^T}(n)L^{-1}(n)\bar{z}^i(n)} \bar{R}^{-1}(n)\bar{z}^i(n) \tag{37}$$

where

$$\hat{e}^i_{CRLS}(n) = X(n)\bar{R}^{-1}(n)\bar{z}^i(n). \tag{38}$$

Because of the similarity of (31) with (38) and (29) with (37) we are able to use a variation of the systolic arrays that are based on the Givens rotation [15], [17] in order to evaluate (36)–(37).

A. Systolic Architecture for the Optimal CRLS Residual Evaluation

From (26) and (36), if $l_q = 1$, we get the optimal residual

$$e^i_{CRLS}(n) = -\frac{r^i}{\bar{z}^{i^T}(n)L^{-1}(n)\bar{z}^i(n)} \left(\prod_{j=1}^{p-1} \lambda_j a_{jj} \right) \frac{\kappa_p a_{pp}}{\lambda_p a'_{pp}} b^{(p)}_{p+1}. \tag{39}$$

In Fig. 4, we present the systolic array $S3.1$, that evaluates the optimal residual for $\kappa_j = \lambda_j = 1, j = 1, 2, \cdots, p$, and the number of constraints is $N = 2$. This systolic array is based on the design proposed by McWhirter [15]. It operates in two modes and is in a way very similar to the operation of the systolic structure $S2.1$ (see Section III). The recursive equations for the data of the matrix $\bar{R}$ are given in (22)–(25). They are evaluated by the boundary cells (cell number 1) and the internal cells (cell number 2). These internal cells are identical to the ones of the array $S2.1$. The boundary cells have a very important difference from the corresponding ones of $S2.1$: while they operate in mode 0, they make use of their division operators in order to evaluate the elements of the diagonal matrix $L^{-1}(n)$, i.e., the quantities $1/l_i, i = 1, 2, \cdots, p$. These quantities are needed for the evaluation of the term $\bar{z}^{i^T}(n)L^{-1}(n)\bar{z}^i(n)$ in (39). The elements of the vectors $\bar{z}^1$ and $\bar{z}^2$ are updated by a variation of (24) and (25), for which the constant β is replaced by $1/\beta$. The two columns of the internal cells (cell number 3) are responsible for these computations. They initialize their memory value during the second phase of the initialization (mode 1) according to (34). While they operate in mode 0, they are responsible for evaluating the partial sums

$$\eta_k = \sum_{j=1}^{k} \|\bar{z}^i_j\|^2 / l_j. \tag{40}$$

The output cells (cell number 4) are responsible for the final evaluation of the residual[5].

McWhirter has designed the systolic arrays that evaluate the optimal residual, based on either the Givens rotation or the square-root-free variation that was introduced by Gentleman [2], [15]. We will call these systolic arrays $S3.3$ and $S3.2$, respectively. On Tables III and V we collect some computational complexity metrics for the systolic arrays $S3.1$, $S3.2$ and $S3.3$, when they operate in mode 0[6]. We observe that the $\mu\nu$ $\Re$otation-based $S3.2$, outperforms the $\kappa\lambda$ $\Re$otation-based $S3.1$. The two structures require the same number of division operators, while $S3.2$ needs less multipliers and also it has less communication overhead.

B. A Systolic Architecture for the Optimal CRLS Weight Vector Extraction

In Fig. 5, we present the systolic array that evaluates (37) for $\kappa_j = \lambda_j = 1, j = 1, 2, \cdots, p$ and the number of constraints equal to $N = 2$. This systolic array operates in two modes, just as the arrays $S2.1$ and $S3.1$ do. The boundary cell (cell number 1) is responsible for evaluating the diagonal elements of the matrices R and L, the variable l_q, as well as all the coefficients that will be needed in the computations of the internal cells. In mode 0 its operation is almost identical to the operation of the boundary cell in $S2.1$ (except for t), while in mode 1 it behaves like the corresponding cell of $S3.1$. The internal cells in the left triangular part of the systolic structure (cell number 2) evaluate the nondiagonal elements of the matrix R and they are identical to the corresponding cells of $S3.1$. The remaining part of the systolic structure is a 2-layer array. The cells in the first column of each layer (cell number 3) are responsible for the calculation of the vector z^i and the partial summations (40). They also communicate their memory values to their right neighbors. The latter (cell number 4) evaluate the elements of the matrix R^{-T} and they are identical to the corresponding elements of $S2.1$. The output elements (cell number 5) are responsible for the normalization of the weight vectors and they compute the final result.

[5]Note the alias $r^i \equiv \tau$.

[6]The multiplications with the constants $\beta, \beta^2, 1/\beta$ and $1/\beta^2$, as well as the communication links that drive the mode bit, are not encountered.

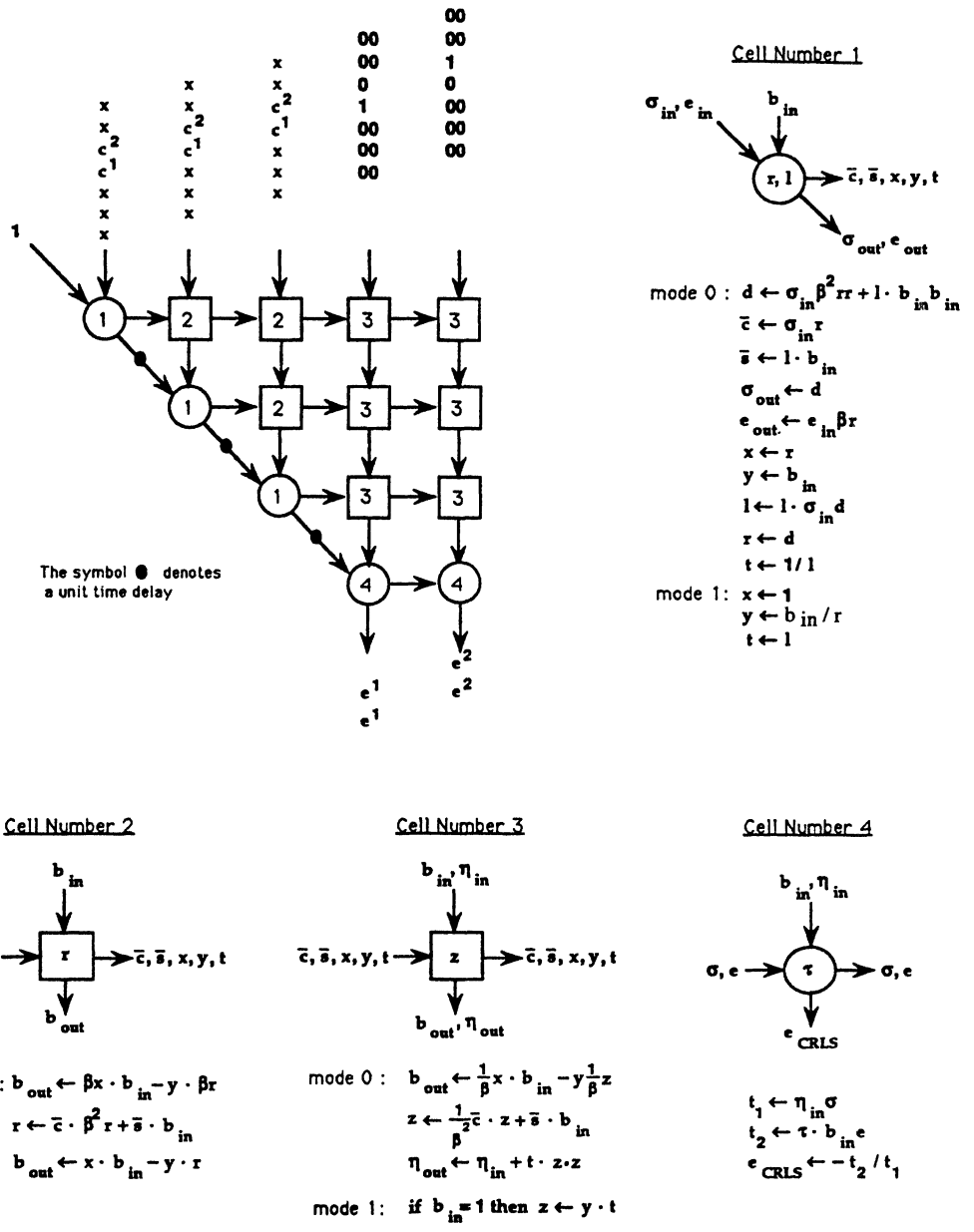


Fig. 4. $S3.1$: Systolic array that computes the CRLS optimal residual. It implements the algorithm that is based on the $\kappa\lambda$ $\Re$otations for which $\kappa = \lambda = 1$.

TABLE III
CRLS OPTIMAL RESIDUAL COMPUTATIONAL COMPLEXITY (MODE 0)

	$S3.1 : \kappa\lambda$				$S3.2 : \mu\nu$				$S3.3$: Givens rotation			
cell	1	2	3	4	1	2	3	4	1	2	3	4
number of	p	$\frac{(p-1)p}{2}$	Np	N	p	$\frac{(p-1)p}{2}$	Np	N	p	$\frac{(p-1)p}{2}$	Np	N
sq.rt	-	-	-	-	-	-	-	-	1	-	-	-
div.	1	-	-	1	1	-	-	1	1	-	-	1
mult.	9	4	6	3	6	3	5	2	5	4	5	2
i/o	10	12	14	7	7	10	12	5	5	6	8	5

Shepherd *et al.* [17] and Tang *et al.* [19] have designed systolic structures for the weight vector extraction based on the Givens rotation and the square-root-free $\Re$otation of Gentleman [2]. We will call these two arrays $S4.3$ and $S4.2$, respectively. On Tables IV and V, we show the computational complexity metrics for the systolic arrays $S4.1$, $S4.2$ and $S4.3$, when they operate in mode 0. The observations we make are similar to the ones we have for the systolic arrays that evaluate the RLS weight vector (see Section III).

Note that each part of the 2-layer structure computes the terms relevant to one of the two constraints. In the same way, a problem with N constraints will require an N-layer structure. With this arrangement of the multiple layers we obtain a unit time delay between the evaluation of the weight

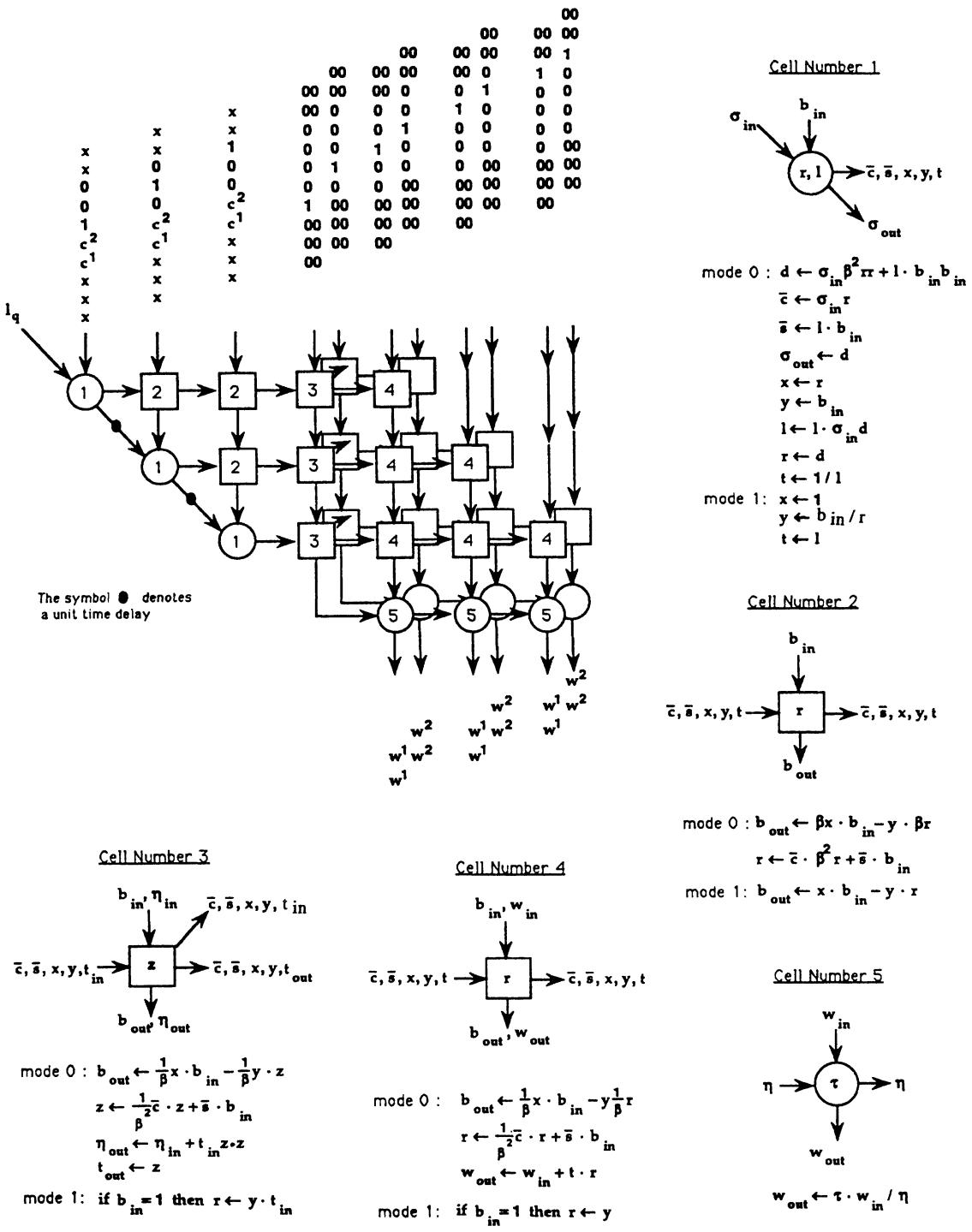


Fig. 5. $S4.1$: Systolic array that computes the CRLS optimal weight vector. It implements the algorithm that is based on the $\kappa\lambda$ Rotation for which $\kappa = \lambda = 1$.

vectors for the different constraints. The price we have to pay is the global wiring for some of the communication links of cell 3. A different approach can also be considered: we may place the multiple layers side by side, one on the right of the other. In this way, not only the global wiring will be avoided, but also the number of communication links of cell 3, will be considerably reduced. The price we will pay with this approach is a time delay of p units between consequent evaluations of the weight vectors for different constraints.

As a conclusion for the CRLS architectures, we observe that the figures on Tables III, IV and V favor the architectures based on the $\mu\nu$ Rotation, $\mu = \nu = 1$ versus the ones that are based on the $\kappa\lambda$ rotation with $\kappa = \lambda = 1$.

V. Dynamic Range, Stability, and Error Bounds

Both the $\kappa\lambda$ and $\mu\nu$ Rotation algorithms enjoy computational complexity advantages compared to the standard Givens rotation with the cost of the denormalization of the latter.

TABLE IV
CRLS WEIGHT VECTOR EXTRACTION COMP COMPLEXITY (MODE 0)

	$S4.1:\kappa\lambda$					$S4.2:\mu\nu$				
cell	1	2	3	4	5	1	2	3	4	5
number of	p	$\frac{(p-1)p}{2}$	Np	$\frac{Np(p+1)}{2}$	Np	p	$\frac{(p-1)p}{2}$	Np	$\frac{Np(p+1)}{2}$	Np
sq.rt	-	-	-	-	-	-	-	-	-	-
div.	1	-	-	-	1	1	-	-	-	1
mult.	8	4	6	5	1	5	3	5	4	-
i/o	8	12	19	14	4	6	8	14	10	4
	$S4.3$: Givens rotation									
cell	1	2	3	4	5					
number of	p	$\frac{(p-1)p}{2}$	Np	$\frac{Np(p+1)}{2}$	Np					
sq.rt	1	-	-	-	-					
div.	1	-	-	-	1					
mult.	4	4	5	5	-					
i/o	4	8	13	10	4					

TABLE V
MINIMUM REQUIRED DELAY BETWEEN TWO CONSEQUENT SETS OF INPUT DATA

	operations-per-cycle	DSP96000 (ns)	IMS T800 (ns)	WEITEK 3164 (ns)	ADSP-3201/2 (ns)
S1.1	max{1 div. + 1 mult. , 9 mult. }	900	3150	1800	2700
S1.2	1 div. + 5 mult.	1020	2300	2700	3675
S1.3	1 sq.rt. + 1 div. + 4 mult.	1810	4500	5300	7175
S2.1	8 mult.	800	2800	1600	2400
S2.2	1 div. + 5 mult.	1020	2300	2700	3675
S2.3	1 sq.rt. + 1 div. + 4 mult.	1810	4500	5300	7175
S3.1	1 div. + 9 mult.	1420	3700	3500	4875
S3.2	1 div. + 6 mult.	1120	2650	2900	3975
S3.3	1 sq.rt. + 1 div. + 5 mult.	1810	4500	5300	7175
S4.1	1 div. + 8 mult.	1320	3350	3300	4575
S4.2	1 div. + 5 mult.	1020	2300	2700	3675
S4.3	1 sq.rt. + 1 div. + 4 mult.	1810	4500	5300	7175

Consequently, the numerical stability of the QRD architectures based on these algorithms can be questioned. Furthermore, a crucial piece of information in the circuit design is the wordlength, that is the number of bits per word required to ensure correct operations of the algorithm without overflow. At the same time, the wordlength has large impact on the complexity and the speed of the hardware implementation. In this section, we address issues on stability, error bounds and lower bounds for the wordlength by means of dynamic range analysis. We focus on the algorithm for RLS optimal residual extraction based on a $\kappa\lambda$ $\Re$otation. The dynamic range of the variables involved in the other newly introduced algorithms can be computed in a similar way.

In [13], Liu *et al.* study the dynamic range of the QRD-RLS algorithm that utilizes the standard Givens rotation. This study is based on the fact that the rotation parameters generated by the boundary cells of the systolic QRD-RLS structure eventually reach a *quasi-steady-state* regardless of the input data statistics, provided that the forgetting factor β is close to one. A worst case analysis of the steady state dynamic range reveals the bound [13]

$$\lim_{n\to\infty}|r_{ij}(n)| \le \frac{(2\beta)^{i-1}}{\sqrt{1-\beta^2}}|x_{max}| \triangleq \mathcal{R}_i^r, \quad j=i,i+1,\cdots,p+1 \tag{41}$$

for the contents of the processing elements of the ith row in the systolic structure, $i=1,2,\cdots,p$, where $|x_{max}|$ is the largest value in the input data. Similarly, at the steady state the output of the ith row $x_j^{(i)}, j=i,i+1,\cdots,p+1$ is bounded by [13]

$$\lim_{n\to\infty}\left|x_j^{(i)}(n)\right| \le (2\beta)^{i-1}|x_{max}| \triangleq \mathcal{R}_i^x, \quad j=i+1,i+2,\cdots,p+1. \tag{42}$$

Furthermore, the optimal residual e_{RLS} is bounded by [13]

$$\lim_{n\to\infty}|e_{RLS}(n)| \le (2\beta)^{p-1}|x_{max}| \triangleq \mathcal{R}_i^{err}.$$

The latter is a BIBO stability result that applies also for the QRD-RLS algorithm based on a $\kappa\lambda$ $\Re$otation. Nevertheless, the internal stability is not guaranteed. More concretely, the terms involved in the QRD-RLS algorithm may not be upper bounded.

In view of the internal stability problem, a proper choice of the parameters κ and λ should be made. A correct choice will compensate for the denormalization of the type

$$r'_{ij}=\frac{1}{\sqrt{l'_i}}a'_{ij}, \qquad x_j^{(i)}=\frac{1}{\sqrt{l_q^{(i)}}}b_j^{(i)} \tag{43}$$

where l'_i and $l_q^{(i)}$ are given in (22) and (23), respectively. The terms κ_i^2 and λ_i^2 in (22) and (23) can be used as shift operators by choosing

$$\kappa_i=2^{-\rho_i} \text{ and } \lambda_i=2^{-\tau_i}, \quad i=1,2,\cdots,p \tag{44}$$

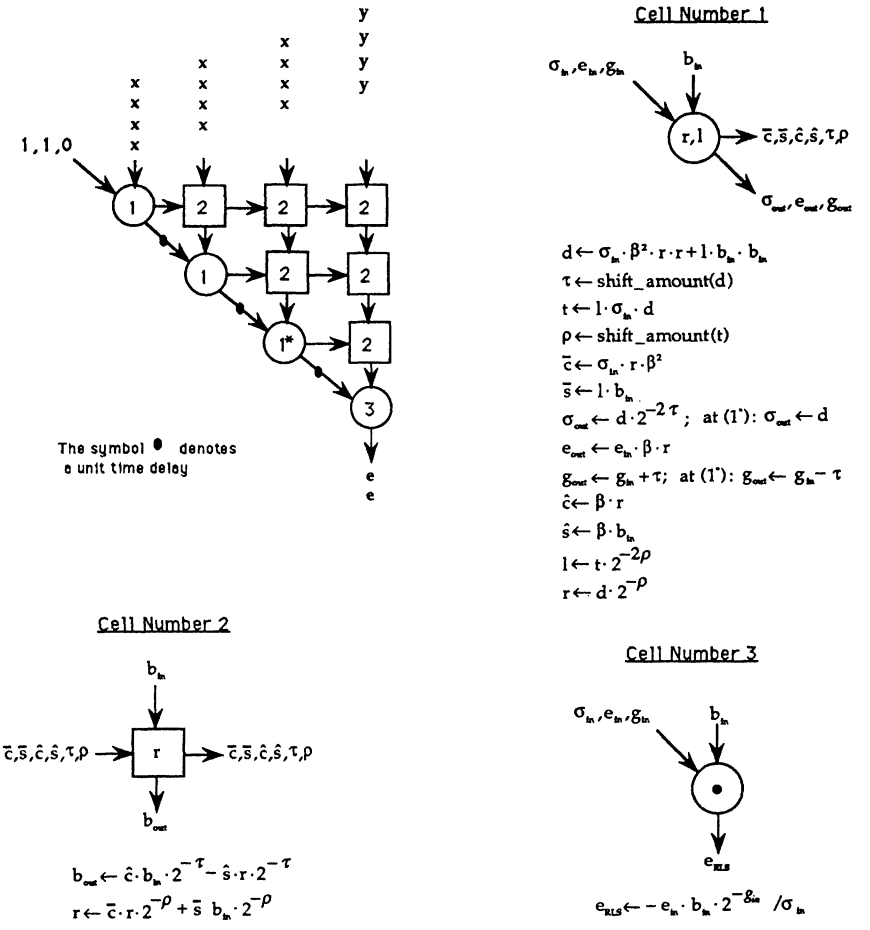


Fig. 6. Systolic array that computes the RLS optimal residual based on the scaled square root free and division free $\Re$otation.

where ρ_i and τ_i take integer values. For instance, in (23), if $\tau_i > 0$ the effect of λ_i^2 on $(l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2})$ will be a right shift of $2\tau_i$ bits. We can ensure that

$$0.5 \leq l_i' < 2 \text{ and } 0.5 \leq l_q^{(i)} < 2, \qquad i = 1, 2, \cdots, p \quad (45)$$

by forcing the most significant bit (MSB) of the binary representation to be either at position 2^0 or 2^{-1} after the shift operation. This normalization task has been introduced in [1] and further used in [4]. It can be described in analytic terms by the expression

$$\begin{aligned} &\text{shift\_amount}(\text{unnormalized\_quantity}) \\ &= \lfloor \{\log_2(\text{unnormalized\_quantity}) + 1\}/2 \rfloor \end{aligned}$$

and it can be implemented very easily in hardware.

In the sequel, we consider the $\kappa\lambda$ $\Re$otation by choosing

$$\begin{aligned} \rho_i &= \text{shift\_amount}\left[l_i l_q^{(i-1)}(l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2})\right] \\ \tau_i &= \text{shift\_amount}\left[(l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2})\right] \end{aligned} \quad (46)$$

for $i = 1, 2, \cdots, p$ [4]. Note that (46) along with (44) should precede (22)–(25) in the rotation algorithm. In conformity to [1] and [4] we will refer to the resulting rotation algorithm with the name *scaled rotation*.

The systolic array that implements the QRD-RLS algorithm for the optimal residual extraction is depicted in Fig. 6. A comparison of this systolic array with the one in Fig. 2 is summarized by the following points: The boundary cells generate the shift quantities ρ and τ associated with the parameters κ and λ, respectively, and they communicate them horizontally with the internal cells. This yields two additional links for the boundary cells and four additional ones for the internal cells. In the dynamic range study that follows, we show that the number of bits these links occupy is close to the logarithm of the number of bits required by the rest of the links. The boundary cells are also responsible for computing the quantities $\prod_{i=1}^{p} \beta a_{ii}$ and $\prod_{i=1}^{p-1} \lambda_i$ in (26). In this case, λ_i is an exponential term according to (44), so the above product can be computed as the running sum of the exponents

$$g_i = \sum_{k=1}^{i} \tau_k, \qquad i = 1, 2, \cdots, p-1 \quad (47)$$

yielding an additional adder for the boundary cells. Finally, as far as the boundary cells are concerned, we observe that the cell at position (p, p) of the systolic array is not identical to the rest of the boundary cells. This is a direct consequence of (26). On the other hand, the shift operators constitute the only overhead of the internal and the output cells compared with the corresponding ones in Fig. 2. Overall, the computational complexity (in terms of operator counts) is slightly higher than that of the systolic array with $\kappa = \lambda = 1$.

Let us focus now on the dynamic range of the variables in the systolic array. By solving (43) for a_{ij}' and using (41) and

(45) one can compute an upper bound for a_{ij} at the steady state, thus one can specify the dynamic range of the ith row cell content. A similar result can be obtained for the output of the ith row by using (42), (43) and (45). The results are summarized by the following Lemma:

Lemma 5.1: The steady state dynamic range of the cell content $\mathcal{R}_i^a$ and output range $\mathcal{R}_i^b$ in the ith row are given by

$$\lim_{n\to\infty} |a_{ij}(n)| \le \mathcal{R}_i^a \triangleq \sqrt{2}\mathcal{R}_i^r \text{ and}$$
$$\lim_{n\to\infty} \left|b_j^{(i)}(n)\right| \le \mathcal{R}_i^b \triangleq \sqrt{2}\mathcal{R}_i^x \tag{48}$$

respectively.

The lower bounds in the wordlength come as a direct consequence of Lemma 4: The wordlength of the cell content β_i^a and output β_i^b in the ith row must be lower bounded by

$$\beta_i^a \ge \lceil \beta_i^r + 0.5 \rceil \text{ and } \beta_i^b \ge \lceil \beta_i^x + 0.5 \rceil \tag{49}$$

respectively, where $\beta_i^r = \lceil \log_2 \mathcal{R}_i^r \rceil$ and $\beta_i^x = \lceil \log_2 \mathcal{R}_i^x \rceil$ are the corresponding wordlength lower bounds for the QRD-RLS implementation based on the standard Givens rotation.

The parameters κ_i, λ_i are communicated via their exponents ρ_i and τ_i. The dynamic ranges of these exponents are given by Lemma 5 which is proved in the Appendix.

Lemma 5.2: The steady state dynamic range of the terms ρ_i and τ_i at the ith row $\mathcal{R}_i^\rho$ and $\mathcal{R}_i^\tau$ are given by

$$\lim_{n\to\infty} \rho_i \le \mathcal{R}_i^\rho \triangleq \beta_i^a + 2.5$$
$$\lim_{n\to\infty} \tau_i \le \mathcal{R}_i^\tau \triangleq \beta_i^a + 1.5 \tag{50}$$

respectively[7], if both ρ_i and τ_i are nonnegative.

Obviously, if both ρ_i and τ_i take negative values, (50) will also satisfy. But there is no guarantee in its dynamic bound. Notice that taking negative value means a left shift. Uncontrolled arbitrary left shift may end up losing the MSB, an equivalence of overflow. Thus, it will be wise to also limit the magnitudes of negative ρ_i and τ_i to the bounds in (50), i.e.

$$\lim_{n\to\infty} |\rho_i| \le \mathcal{R}_i^\rho$$
$$\lim_{n\to\infty} |\tau_i| \le \mathcal{R}_i^\tau \tag{51}$$

Consequently, the lower bounds on the wordlength β_i^ρ and β_i^τ of ρ_i and τ_i are

$$\beta_i^\rho \ge \lceil \log(\beta_i^a + 2.5) \rceil \text{ and}$$
$$\beta_i^\tau \ge \lceil \log(\beta_i^a + 1.5) \rceil \tag{52}$$

respectively.

For the computation of the optimal residual the boundary cells need to evaluate both the running product $e_i = \prod_{k=1}^{i} \beta a_{kk}$ and the running sum in (47). The dynamic ranges for these terms are given by the following Lemma:

[7] For the sake of simplicity in notation we have dropped the time parameter n from the expression in the limit argument.

Lemma 5.3: The steady state dynamic range of the terms e_i and g_i at the ith row $\mathcal{R}_i^e$ and $\mathcal{R}_i^g$ are given by

$$\lim_{n\to\infty} |e_i| \le \mathcal{R}_i^e \triangleq \prod_{k=1}^{i} \mathcal{R}_k^a$$
$$\lim_{n\to\infty} |g_i| \le \mathcal{R}_i^g \triangleq i\beta_1^a + \frac{i(i+2)}{2} \tag{53}$$

respectively.

The proof is given in the Appendix. With simple algebraic manipulations one can show that the corresponding lower bounds on wordlength β_i^e and β_i^g of e_i and g_i are

$$\beta_i^e \ge \sum_{k=1}^{i} \beta_k^a \quad \text{and}$$
$$\beta_i^g \ge \max\{\lceil \log \beta_1^a + \log i + 1 \rceil, \lceil \log i + \log(i+2) \rceil\} \tag{54}$$

respectively.

Finally, consider the coefficients defined as

$$\bar{c}_i = l_q^{(i-1)} \beta^2 a_{ii} \quad \bar{s}_i = l_i b_i^{(i-1)}$$
$$\hat{c}_i = \beta a_{ii} \quad \hat{s}_i = \beta b_i^{(i-1)},$$

that describe the information exchanged by the remaining horizontal links in the systolic array (cf. Fig. 6). One can easily show that the steady state dynamic range of these coefficients, denoted by $\mathcal{R}_i^{\bar{c}}, \mathcal{R}_i^{\bar{s}}, \mathcal{R}_i^{\hat{c}}$ and $\mathcal{R}_i^{\hat{s}}$, respectively are

$$\lim_{n\to\infty} |\bar{c}_i| \le \mathcal{R}_i^{\bar{c}} \triangleq 2\mathcal{R}_i^a$$
$$\lim_{n\to\infty} |\bar{s}_i| \le \mathcal{R}_i^{\bar{s}} \triangleq 2\mathcal{R}_i^b$$
$$\lim_{n\to\infty} |\hat{c}_i| \le \mathcal{R}_i^{\hat{c}} \triangleq \mathcal{R}_i^a$$
$$\lim_{n\to\infty} |\hat{s}_i| \le \mathcal{R}_i^{\hat{s}} \triangleq \mathcal{R}_i^b. \tag{55}$$

The implied wordlength lower bounds are $\beta_i^{\bar{c}} \ge \beta_i^a + 1$, $\beta_i^{\bar{s}} \ge \beta_i^b + 1, \beta_i^{\hat{c}} \ge \beta_i^a$ and $\beta_i^{\hat{s}} \ge \beta_i^b$, respectively.

In summary, (45), (48), (50), (53), and (55) show that all the internal parameters are bounded and therefore the algorithm is stable. Furthermore, the lower bounds on the wordlength provide the guidelines for an inexpensive, functionally correct realization.

The error bound of the whole QRD to a given matrix $A \in \Re^{m\times n}$ due to floating point operations is given by [1], [4]

$$\|\delta A\| \le \tau(m+n-3)(1+\tau)^{m+n-4}\|A\| + O(\epsilon^2), \tag{56}$$

where τ is the upper bound and ϵ is the largest number such that $1+\epsilon$ is computed as 1. If (44) and (45) are satisfied, for $\kappa = \lambda = 1$, then it follows that $\tau = 6.5\epsilon$ [4]. This is fairly close to the standard Givens rotation which has $\tau = 6.0\epsilon$ [4].

VI. Conclusion

We introduced *the parametric $\kappa\lambda$ $\Re$otation*, which is a square-root-free and division-free algorithm, and showed that the parametric $\kappa\lambda$ $\Re$otation describes a subset of the $\mu\nu$ $\Re$otation algorithms [8]. We then derived novel architectures based on the $\kappa\lambda$ $\Re$otation for $\kappa = \lambda = 1$ and made a

comparative study with the standard Givens rotation and the $\mu\nu$ $\Re$otation with $\mu = \nu = 1$. Finally, a dynamic range study is pursued. It is observed that considerable improvements can be obtained for the implementation of some QRD-based algorithms.

We pointed out the tradeoffs between the architectures based on the above $\Re$otations. Our analysis suggests the following decision rule for selecting between the architectures that are based on the $\mu\nu$ $\Re$otation and the $\kappa\lambda$ $\Re$otation: *Use the $\mu\nu$ $\Re$otation – based architectures, with $\mu = \nu = 1$, for the constrained minimization problems and the $\kappa\lambda$ $\Re$otation – based architectures, with $\kappa = \lambda = 1$, for the unconstrained minimization problems.* Table V shows the benchmark comparisons of different algorithms using different DSP processors and it confirms the properties claimed in this paper.

A number of obscure points relevant to the realization of the QRD-RLS and the QRD-CRLS algorithms are clarified. Some systolic structures that are described in this paper are very promising, since they require less computational complexity (in various aspects) from the structures known to date and they make the VLSI implementation easier.

APPENDIX

Proof of Lemma 3.1: First, we derive some equations that will be used in the course of the optimal residual computation.

If we solve (24), case $i = j = 1$, for $l_q\beta^2 a_{11}^2 + l_1 b_1^2$ and substitute in (22) we get

$$l_1' = l_1 l_q \frac{a_{11}'}{\kappa_1}\kappa_1^2$$

and therefore

$$\frac{l_1'}{l_1} = l_q a_{11}' \kappa_1. \tag{57}$$

If we solve (24), case $j = i$, for $l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2}$ and substitute in (23) we get

$$l_q^{(i)} = \frac{\lambda_i^2 a_{ii}'}{\kappa_i}. \tag{58}$$

If we substitute the same expression in (22) we get

$$l_i' = l_i l_q^{(i-1)} a_{ii}' \kappa_i.$$

(58), and solve for l_i'/l_i to obtain

$$\frac{l_i'}{l_i} = \frac{\lambda_{i-1}^2 \kappa_i}{\kappa_{i-1}} a_{i-1,i-1}' a_{ii}'. \tag{59}$$

If we solve (22) for $l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2}$ and substitute in (23) we get

$$l_q^{(i)} = \frac{\lambda_i^2}{\kappa_i^2}\frac{l_i'}{l_i}\frac{1}{l_q^{(i-1)}}. \tag{60}$$

Also, we note that (4) implies that

$$c_i = \beta r_{ii}/r_{ii}'$$

and by substituting (9) we obtain

$$c_i = \frac{\beta a_{ii}}{a_{ii}'}\sqrt{\frac{l_i'}{l_i}} \qquad i = 1, 2, \cdots, p. \tag{61}$$

Similarly, from (4) and (9), we get

$$s_i = \frac{b_i^{(i-1)}}{a_{ii}'}\sqrt{\frac{l_i'}{l_q^{(i-1)}}} \qquad i = 1, 2, \cdots, p. \tag{62}$$

The optimal residual for the RLS problem is [6]

$$e_{RLS}(t_n) = -\left(\prod_{k=1}^{p} c_k\right) v(t_n). \tag{63}$$

The expressions in (20) and (19) imply

$$v(t_n) = \frac{1}{\sqrt{l_q^{(p)}}} b_{p+1}^{(p)}.$$

If we substitute the above expressions of $v(t_n)$ and c_i in (63) we obtain

$$e_{RLS}(t_n) = -\prod_{i=1}^{p}\left(\frac{\beta a_{ii}}{a_{ii}'}\sqrt{\frac{l_i'}{l_i}}\right)\frac{1}{\sqrt{l_q^{(p)}}} b_{p+1}^{(p)}. \tag{64}$$

From (60) we get

$$\begin{aligned} l_q^{(p)} &= \frac{\lambda_p^2}{\kappa_p^2}\frac{l_p'}{l_p}\frac{1}{l_q^{(p-1)}} = \frac{\lambda_p^2}{\kappa_p^2}\frac{l_p'}{l_p}\frac{\kappa_{p-1}^2}{\lambda_{p-1}^2}\frac{l_{p-1}}{l_{p-1}'} l_q^{(p-2)} \\ &= \begin{cases} \prod_{j=1}^{k}\left(\frac{\lambda_{2j}^2}{\kappa_{2j}^2}\frac{l_{2j}'}{l_{2j}}\frac{\kappa_{2j-1}^2}{\lambda_{2j-1}^2}\frac{l_{2j-1}}{l_{2j-1}'}\right) l_q, & p = 2k \\ \prod_{j=1}^{k-1}\left(\frac{\kappa_{2j}^2}{\lambda_{2j}^2}\frac{l_{2j}}{l_{2j}'}\frac{\lambda_{2j-1}^2}{\kappa_{2j-1}^2}\frac{l_{2j-1}'}{l_{2j-1}}\right)\frac{\lambda_p^2}{\kappa_p^2}\frac{l_p'}{l_p}\frac{1}{l_q}, & p = 2k-1 \end{cases} \end{aligned} \tag{65}$$

Thus, from (64) and (65), for the case of $p = 2k$, we have the first equation at the top of the next page. By doing the appropriate term cancelations and by substituting the expressions of $l_i'/l_i, i = 1, 2, \cdots, 2k$ from (57) and (59) we obtain the expression (26) for the optimal residual. Similarly, for the case of $p = 2k - 1$, from (64) and (65) we obtain the second equation at the top of the next page and by substituting (59), we get (26).

Proof of Lemma 3.2: The question is whether we can avoid the division in the evaluation of the residual. Obviously we should $\lambda_p a_{pp}'$ or

$$\lambda_p = \kappa_p / a_{pp}'$$

holds. But, from (24), for $j = i$, we get

$$\kappa_p / a_{pp}' = \frac{1}{l_q^{(p-1)}\beta^2 a_{pp}^2 + l_p b_p^{(p-1)^2}}.$$

Therefore, if we choose to avoid the division operation in the expression of the residual, we will need to perform another division in order to evaluate the parameter λ_p.

$$e_{RLS}(t_n) = -\prod_{j=1}^{k}\left(\frac{\beta a_{2j,2j}}{a'_{2j,2j}}\sqrt{\frac{l'_{2j}}{l_{2j}}}\frac{\beta a_{2j-1,2j-1}}{a'_{2j-1,2j-1}}\sqrt{\frac{l'_{2j-1}}{l_{2j-1}}}\left(\frac{\lambda_{2j}^2}{\kappa_{2j}^2}\frac{l'_{2j}}{l_{2j}}\frac{\kappa_{2j-1}^2}{\lambda_{2j-1}^2}\frac{l_{2j-1}}{l'_{2j-1}}\right)^{-\frac{1}{2}}\right)\cdot\frac{1}{\sqrt{l_q}}\cdot b_{p+1}^{(p)}$$

$$e_{RLS}(t_n) = -\prod_{j=1}^{k-1}\left(\frac{\beta a_{2j,2j}}{a'_{2j,2j}}\sqrt{\frac{l'_{2j}}{l_{2j}}}\frac{\beta a_{2j-1,2j-1}}{a'_{2j-1,2j-1}}\sqrt{\frac{l'_{2j-1}}{l_{2j-1}}}\left(\frac{\lambda_{2j-1}^2}{\kappa_{2j-1}^2}\frac{l'_{2j-1}}{l_{2j-1}}\frac{\kappa_{2j}^2}{\lambda_{2j}^2}\frac{l_{2j}}{l'_{2j}}\right)^{-\frac{1}{2}}\right)\cdot\frac{\beta a_{pp}}{a'_{pp}}\sqrt{\frac{l'_p}{l_p}}\sqrt{\frac{\kappa_p^2 l_p l_q}{\lambda_p^2 l'_p}}b_{p+1}^{(p)}$$

Proof of Lemma 5.2: From (45) and the fact that $0 < \beta < 1$ we get

$$l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2} < 2a_{ii}(n)^2 + 2b_i^{(i-1)^2}$$

Consequently, at the steady state we have

$$\lim_{n\to\infty}\left|l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2}\right| < 2(\mathcal{R}_i^a)^2 + 2(\mathcal{R}_i^b)^2.$$

Also, (41), (42), and (48) imply that $\mathcal{R}_i^a > \mathcal{R}_i^b$. Therefore, we obtain the bound

$$\lim_{n\to\infty}\left|l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2}\right| < 4(\mathcal{R}_i^a)^2$$

and by utilizing (23) and the fact that $l_q^{(i)} \geq 0.5$we get

$$\lim_{n\to\infty}|\lambda_i^{-2}| \leq 2\lim_{n\to\infty}\left|l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2}\right| < 8(\mathcal{R}_i^a)^2. \tag{66}$$

By substituting the expression $\lambda_i = 2^{-\tau_i}$, using (66) and solving the resulting inequality for τ_i

$$\lim_{n\to\infty}\tau_i \leq \log\mathcal{R}_i^a + 1.5$$

if $\tau_i$is nonnegative. The expression for the dynamic range of τ_i in (50) is a direct consequence of the above inequality.

Similarly, for the computation of the dynamic range of the term ρ_i first one can prove that

$$\lim_{n\to\infty}\left|l_i l_q^{(i)}(l_q^{(i-1)}\beta^2 a_{ii}^2 + l_i b_i^{(i-1)^2})\right| < 16(\mathcal{R}_i^a)^2$$

and then compute an upper bound for $\rho_i$at the steady state based on (22), (44) and the fact that $l'_i \geq 0.5$.

Proof of Lemma 5.3: Since $0 < \beta < 1$, for the term e_i we have

$$\lim_{n\to\infty}|e_i| = \beta^i\prod_{k=1}^{i}\lim_{n\to\infty}|a_{kk}| \leq \prod_{k=1}^{i}\mathcal{R}_k^a.$$

Similarly, for the term $g_i$we have

$$\lim_{n\to\infty}|g_i| = \sum_{k=1}^{i}\lim_{n\to\infty}|\tau_k|$$

and from (50)

$$\lim_{n\to\infty}|g_i| \leq \sum_{k=1}^{i}\mathcal{R}_k^r = \sum_{k=1}^{i}(\beta_k^a + 1.5). \tag{67}$$

Equation (41) implies that the wordlength for the variable r should satisfy the inequality

$$\beta_i^r \geq \lceil (i-1)(1+\log\beta) + C\rceil$$

where C is constant with respect to i. Since $\beta < 1$, it is sufficient to have

$$\beta_i^r \geq \lceil i - 1 + C\rceil$$

or

$$\beta_i^r \geq i - 1 + \beta_1^r. \tag{68}$$

A similar formula can be derived for the wordlength of the contents of the the array that utilizes the scaled rotation, based on (49) and (68). More specifically, we have

$$\beta_i^a \geq \beta_1^a + i - 1.$$

From this inequality and (67) we get

$$\lim_{n\to\infty}|g_i| \leq i\beta_i^a + \frac{i(i-1)}{2} + 1.5i.$$

The dynamic range expression in (53) follows directly.

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.*, vol. 8, no. 5, pp. 716–733, Sept. 1987.

[2] W. M. Gentleman, "Least squares computations by Givens transformations without square roots," *J. Inst. Math. Applicat.*, vol. 12, pp. 329–336, 1973.

[3] W. M. Gentleman and H. T. Kung, "Matrix triangularization by systolic arrays," in *Proc. SPIE 298, Real-Time Signal Processing IV*, pp. 19–26, 1981.

[4] J. Gotze and U. Schwiegelshohn, "A square root and division free Givens rotation for solving least squares problems on systolic arrays," *SIAM J. Sci., Statist. Comput.*, vol. 12, no. 4, pp. 800–807, July 1991.

[5] S. Hammarling, "A note on modifications to the Givens plane rotation," *J. Inst. Math. Applicat.*, vol. 13, pp. 215–218, 1974.

[6] S. Haykin, *Adaptive Filter Theory*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[7] S. F. Hsieh, K. J. R. Liu, and K. Yao, "Dual-state systolic architectures for up/downdating RLS adaptive filtering," *IEEE Trans. Circuits, Syst. II*, vol. 39, no. 6, pp. 382–385, June 1992.

[8] ——, "A unified approach for QRD-based recursive least-squares estimation without square roots," *IEEE Trans. Signal Processing*, vol. 41, no. 3, pp. 1405–1409, March 1993.

[9] S. Kalson and K. Yao, "Systolic array processing for order and time recursive generalized least-squares estimation," in *Proc. SPIE 564, Real-Time Signal Processing VIII*, 1985, pp. 28–38.

[10] F. Ling, "Efficient least squares lattice algorithm based on Givens rotations with systolic array implementation," *IEEE Trans. Signal Processing,* vol. 39, pp. 1541–1551, July 1991.
[11] F. Ling, D. Manolakis, and J. G. Proakis, "A recursive modified Gram-Schmidt algorithm for least-squares estimation, *IEEE Trans. Acoust., Speech, Signal Processing,* vol. ASSP-34, no. 4, pp. 829–836, Aug. 1986.
[12] K. J. R. Liu, S. F. Hsieh, and K. Yao, "Systolic block Housholder transformation for RLS algorithm with two-level pipelined implementation," *IEEE Trans. Signal Processing,* vol. 40, no. 4, pp. 946–958, Apr. 1992.
[13] K. J. R. Liu, S. F. Hsieh, K. Yao, and C. T. chiu, "Dynamic range, stability and fault-tolerant capability of finite precision RLS systolic array based on Givens rotation," *IEEE Trans. Circuits, Syst.,* vol. 38, no. 6, pp. 625–636, June 1991.
[14] J. G. McWhirter, "Recursive least-squares minimization using a systolic array," in *Proc. SPIE, Real Time Signal Processing VI,* vol. 431, 1983, pp. 105–112.
[15] J. G. McWhirter and T. J. Shepherd, "Systolic array processor for MVDR beamforming," in *IEE Proc.,* pt. F, vol. 136, no. 2, Apr. 1989, pp. 75–80.
[16] I. K. Proulder, J. G. McWhirter, and T. J. Shepherd, "The QRD-based least squares lattice algorithm: Some computer simulations using finite wordlength," in *Proc. IEEE ISCAS* (New Orleans, LA), May 1990, pp. 258–261
[17] T. J. Shepherd, J. G. McWirter, and J. E. Hudson, "Parallel weight extraction from a systolic adaptive beamforming," *Math. Signal Processing II,* 1990.
[18] R. W. Stewart, R. Chapman, and T. S. Durrani, "Arithmetic implementation of the Givens QR tiarray," in *Proc. IEEE/ICASSP,* 1989, pp. V-2405–2408.
[19] C. F. T. Tang, K. J. R. Liu, and S. Tretter, "Optimal weight extraction for adaptive beamforming using systolic arrays," *IEEE Trans. Aerosp., Electron. Syst.*, vol. 30, no. 2, pp. 367–385, Apr. 1994.

Algorithmic engineering in adaptive signal processing

J.G. McWhirter

Indexing terms: Signal processing, Algorithms

Abstract: The concept of algorithmic engineering is introduced and discussed in the context of parallel digital signal processing. The main points are illustrated by means of some fairly simple worked examples. Most of these relate to the use of QR decomposition by square-root-free Givens rotations as applied to adaptive filtering and beamforming.

1 Introduction

Achieving the performance required from a modern digital signal processing (DSP) system often necessitates the real-time application of reliable numerical algorithms for least squares estimation, solving linear systems, performing singular value decomposition and so on. To perform such computations at the required data rate, it is often necessary to introduce a high degree of parallel processing. For some applications, it may be sufficient to exploit a general purpose parallel computer, in which case the signal processing designer must map the relevant numerical algorithms as efficiently as possible onto the architecture of that particular machine. For many real-time applications however, it is necessary to design a highly dedicated parallel processor that can be implemented using advanced VLSI technology. It is in this context that the concept of algorithmic engineering has started to emerge. It describes the hybrid discipline of deriving stable numerical algorithms, which are suitable for parallel computation, and then mapping them onto parallel-processing architectures capable of performing the computation efficiently at the required throughput rate. Both aspects are extremely important and cannot be treated in isolation. For real-time DSP, it is obviously essential to ensure that an algorithm may be implemented at the required data rate. However, there is no point in designing a high-performance parallel processor if the underlying algorithm is numerically unstable or grossly inefficient.

The most significant advance in algorithmic engineering was undoubtedly the pioneering work of Kung and Leiserson [1]. They introduced the concept of a systolic array and showed how a number of important linear algebra computations, such as matrix multiplication, triangularisation and back substitution, could be mapped onto this very efficient type of parallel-processing architecture. A particularly important development, which features in this paper, was the design by Gentleman and Kung [2] of a triangular systolic array for QR decomposition. This constitutes a relatively simple and highly regular architecture whereby the essential least squares process required for many adaptive filtering and beamforming operations may be implemented in a numberically stable and efficient recursive manner [3]. It has since been generalised to include linear constraints and applied to a wide range of problems in signal processing. These include narrowband and broadband adaptive filtering, Kalman filtering and nonlinear adaptive filtering as applied to neural networks and pattern recognition.

Now that a number of kernel processing architectures, such as the Gentleman and Kung array, have become widely understood and accepted, many signal-processing engineers are willing to accept them as high-level building blocks on which the design of other hybrid processing structures may be based. The use of high-level building blocks can be a valuable aid in the design of more complicated processing architectures, but, if the technique is to gain wider acceptance, a more rigorous methodology is essential. The approach adopted to date is much too *ad hoc* and ill-defined. In this paper, I hope to illustrate, by means of suitable worked examples, a more methodical procedure which the author has developed in the context of research on adaptive filtering.

A well-defined block diagrammatic representation of any high-level building block is essential for the purposes of algorithmic engineering, and the concept of a signal flow graph (SFG) proves to be very useful in this context [4]. A signal flow graph hides the detailed timing features associated with a synchronous systolic array, but, when required, this information can easily be re-inserted or established from first principles, using the cut theorem and retiming techniques (assuming that systolic operation of the hybrid architecture is possible). Postponing the detailed timing issues in this way gives the DSP designer more freedom to establish the optimum level of granularity for the processing cells in each block. More importantly, it allows the high-level building blocks to be represented functionally as well-defined mathematical operators. It is then possible to manipulate the blocks in a rigorous manner, determined by the type of matrix (or other) algebra associated with those operators. The author and his colleagues have found this to be a very powerful approach and have used it in the past to derive novel signal-processing architectures, such as the systolic array for MVDR beamforming described in Reference 5.

2 Fixed matrix operators

In this section, I will discuss some relatively simple matrix operators, which may be implemented using an array of the basic processing cells defined in Fig. 1. These are referred to as fixed matrix operators because their

Paper 8792F (E5), first received 1st October 1991 and in revised form 17th February 1992

The author is with DRA, Electronics Division, RSRE, Malvern, Worcestershire WR14 3PS, United Kingdom

Reprinted with permission from *Proceedings of the IEE, Part F*, J. G. McWhirter, "Algorithmic Engineering in Adaptive Signal Processsing," Vol. 139, No. 3, pp. 226-232, June 1992. © Crown Copyright.
Reprinted with the permission of the Controller of Her Majesty's Stationery Office.

function remains constant and is not affected by the data that they process. The parameter r in each case represents a value stored within the cell.

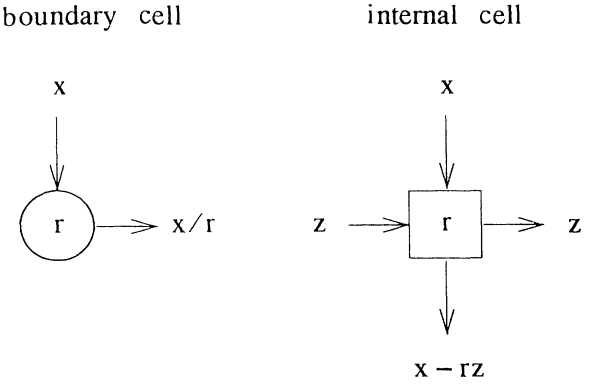


Fig. 1 *Processing cells required for fixed matrix operators*

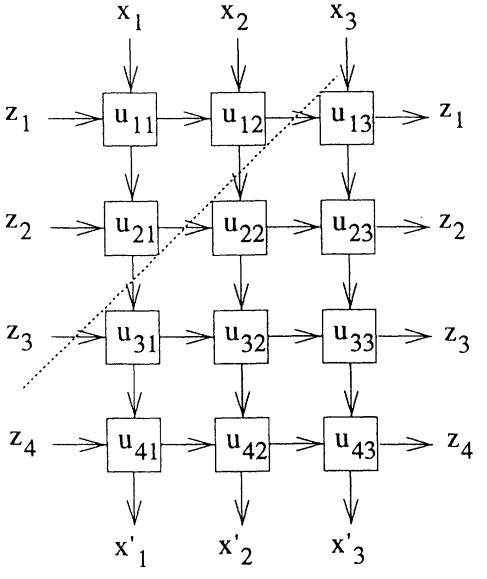


Fig. 2 *Rectangular fixed matrix operator*

Consider first a $p \times q$ rectangular array of internal cells of the type illustrated in Fig. 2. It is well known and can readily be shown that, if the vectors x^T and z are input to such a processing network as indicated in Fig. 2, the resulting output vector from below is given by

$$x'^T = x^T - z^T U \tag{1}$$

where U denotes the matrix of values stored within the cells. The array may therefore be regarded as a matrix multiplication operator. Fig. 2 should be regarded as a signal flow graph, which is only intended to provide an abstract description of the algorithm mapping. For ease of understanding, it may therefore be assumed that the output of each cell, and hence the entire array, is generated instantaneously. In practice, of course, the processing time will not be negligible, and it may be necessary to introduce some form of pipelining into the computation. Accordingly, Fig. 2 may be used to define a fully pipelined, systolic array by generating a pipeline 'cut' between each diagonal row of processors. The dashed line in Fig. 2 indicates one such cut, the others being drawn parallel to this. Where each pipeline cut crosses a data interconnection line, the systolic array will require a corresponding data storage or delay element. Note that the triangular wedge of delay elements required to skew/deskew each input/output vector is also specified by the complete set of pipeline cuts.

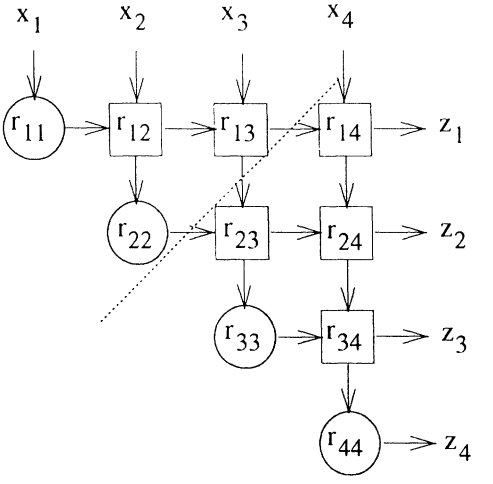


Fig. 3 *Triangular fixed matrix operator*

Now consider a $p \times p$ triangular array of processing cells of the type represented by the signal flow graph in Fig. 3. It is easy to show that the input vector x^T may be expressed in terms of the output vector z, as follows:

$$x = R^T z \tag{2}$$

where R denotes the triangular matrix of stored values. For example, it is clear that

$$z_1 = x_1/r_{11} \quad \text{and} \quad z_2 = (x_2 - r_{12}z_1)/r_{22} \tag{3}$$

i.e.

$$x_1 = r_{11}z_1 \quad \text{and} \quad x_2 = r_{12}z_1 + r_{22}z_2 \tag{4}$$

Hence, if R is nonsingular (i.e. no diagonal element of R is zero), the output vector z is given by

$$z = R^{-T}x \tag{5}$$

and so the processor array represented by Fig. 3 constitutes an inverse triangular matrix operator. As before, a fully systolic processor array may be defined by introducing a set of diagonal pipeline cuts parallel to the one indicated by a dashed line in Fig. 3.

The matrix operators in Figs. 2 and 3 may be represented quite compactly by the simplified schematic diagrams in Figs. 4*a* and *b*, respectively. In effect, each of these diagrams constitutes a vector-level signal flow graph which provides no information about the underlying algorithm and architecture. Now, as a very simple example of algorithmic engineering, consider the trapezoidal processor array represented by the diagram in Fig. 4*c*. It is formed by combining a $p \times q$ rectangular array and a $p \times p$ triangular array of the type illustrated in Figs. 2 and 3, respectively. From the discussion above, it follows that the effect of inputting a p-element vector x_a^T and a q-element vector x_b^T from the top as shown, is to generate the p-element output vector $R^{-T}x_a$, which emerges from the right-hand edge, and the q-element vector $x_b^T - x_a^T R^{-1}U$, which is output from the bottom edge of the trapezoidal network. This output vector is, by definition, the Schur complement [6] of R in the compound matrix

$$S = \begin{bmatrix} R & U \\ x_a^T & x_b^T \end{bmatrix} \tag{6}$$

and it is interesting to note that the processor array represented by Fig. 4c may also be viewed as a parallel implementation of the Fadeev algorithm [7]. In effect, the triangular array serves to null the input vector x_a^T by forming a linear combination of the form $x_a^T + MR$,

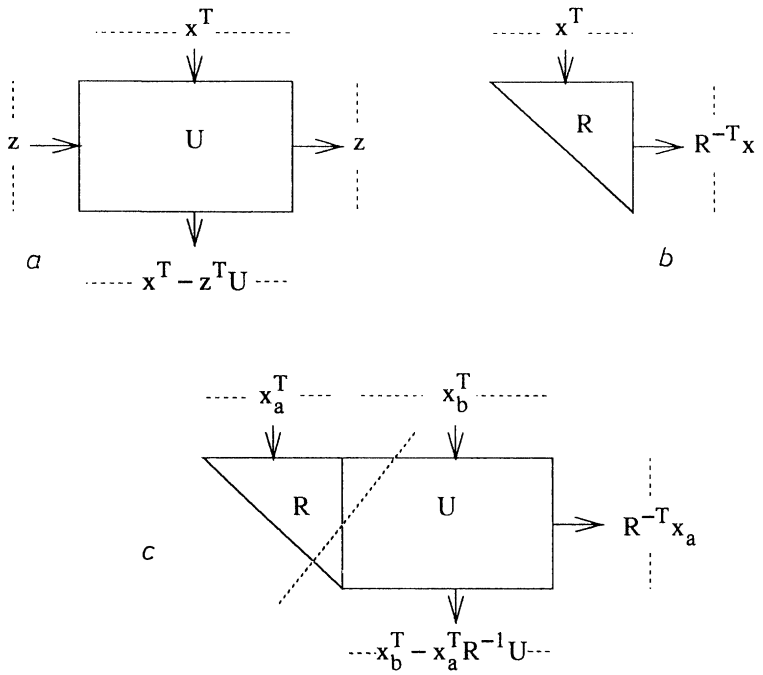


Fig. 4 *Schematic representation of fixed networks*
a Rectangular fixed network *b* Triangular fixed network *c* Trapezoidal fixed network

where

$$M = -x_a^T R^{-1} \tag{7}$$

An identical linear combination is formed by the rectangular array to produce the output vector

$$x'^T = x_b^T + MU \tag{8}$$

which emerges from below, and combining eqns. 7 and 8 leads immediately to the expression in Fig. 4c. Note how easily this result was obtained by considering the constituent blocks in Fig. 4c as simple matrix operators.

The dashed line in Fig. 4c is intended to demonstrate how the type of pipeline cut shown in Figs. 2 and 3 may be applied to the combined array. In particular, it serves to illustrate the fact that there is no need for any delay elements to deskew/skew the data between the individual subarrays to define the corresponding systolic processor. It is for reasons like this, that algorithmic engineering is best carried out in terms of the basic signal flow graph associated with each elementary operator.

3 Recursive least squares

Fig. 5 depicts a parallel processing network of the type proposed by Gentleman and Kung [2] for linear least squares estimation. Its purpose at each sample time t_n is to compute the complex weight vector $w(n)$, which minimises the norm of the residual vector

$$e(n) = X(n)w(n) + y(n) \tag{9}$$

where $X(n)$ is an $n \times p$ complex matrix defined by

$$X(n) = [x(t_1), x(t_2), \ldots, x(t_n)]^T \tag{10}$$

and $y(n)$ is the complex n-element vector

$$y(n) = [y(t_1), y(t_2), \ldots, y(t_n)]^T \tag{11}$$

The main triangular array ABC transforms the data matrix $X(n)$ into a $p \times p$ upper triangular matrix $R(n)$ by performing a QR decomposition of the form

$$Q(n)X(n) = \begin{bmatrix} R(n) \\ O \end{bmatrix} \tag{12}$$

$Q(n)$ denotes an $n \times n$ unitary matrix which is generated as a sequence of complex square-root-free Givens rotations. The array operates in the following recursive manner: The triangular matrix $R(n-1)$ corresponding to the data matrix $X(n-1)$ is assumed to be known and represented in the form

$$R(n-1) = D^{1/2}(n-1)\bar{R}(n-1) \tag{13}$$

$D(n-1)$ denotes a real, positive diagonal matrix stored within the boundary cells of the array, and $\bar{R}(n-1)$ is a unit upper triangular matrix stored within the internal cells. The data vector $x^T(t_n)$ is input from the top, as indicated in Fig. 5, and progressively eliminated by rotating it with each row of the stored triangular matrix $R(n-1)$ in turn. The appropriate square-root-free rotation parameters are computed within each boundary cell and passed on to the internal cells in the same row to complete the rotation process. The updated triangular matrix $R(n)$ and diagonal matrix $D(n)$ are computed in the course of eliminating the vector $x^T(t_n)$ and subsequently stored within the array. In a similar manner, the right-hand column of cells DE in the least squares processor array evaluates and stores the p-element vector $\bar{u}(n)$ defined by

$$Q(n)y(n) = \begin{bmatrix} u(n) \\ v(n) \end{bmatrix} \tag{14}$$

and

$$u(n) = D^{1/2}(n)\bar{u}(n) \tag{15}$$

The optimum weight vector is then given by the equation

$$\boldsymbol{R}(n)\boldsymbol{w}(n) + \boldsymbol{u}(n) = \boldsymbol{o} \tag{16}$$

i.e.

$$\boldsymbol{w}(n) = -\boldsymbol{R}^{-1}(n)\boldsymbol{u}(n) = -\bar{\boldsymbol{R}}^{-1}(n)\bar{\boldsymbol{u}}(n) \tag{17}$$

Note that each cell of the array in Fig. 5 must wait until the elimination of $x^T(t_n)$ has been completed before updating its stored value. The procedure may, of course, be pipelined, and the familiar systolic array is obtained by introducing a complete set of diagonal cuts similar to the one represented by the dashed line. Note that each diagonal interconnection will be cut twice, and so it is necessary to impose two delays on the output of the boundary cells.

Since the function of the main QR decomposition array in Fig. 5 is modified by the data that it processes, it constitutes an adaptive matrix operator. The adaptation may be frozen very simply by setting $\delta = 0$ within each boundary cell and thereby suppressing the update of all stored quantities. A close inspection reveals that, in frozen mode, the operation of the triangular array ABC is equivalent to that of the fixed matrix operator illustrated in Fig. 3, assuming that the latter stores the unit upper triangular matrix $\bar{\boldsymbol{R}}(n-1)$. Hence, the effect of inputting a vector $x^T(t_n)$ from the top is to produce the output vector $\bar{\boldsymbol{R}}^{-T}(n-1)\boldsymbol{x}(t_n)$, which emerges from the right. It can also be seen that, in its fully adaptive mode, the triangular array ABC performs the same matrix operation before updating the stored matrices $\boldsymbol{D}(n-1)$ and $\bar{\boldsymbol{R}}(n-1)$. As a consequence, it may be represented by the simplified block diagram in Fig. 6*a*, where the adaptive nature is denoted by the legend $\boldsymbol{D}, \bar{\boldsymbol{R}}(n-1) \rightarrow \boldsymbol{D}, \bar{\boldsymbol{R}}(n)$.

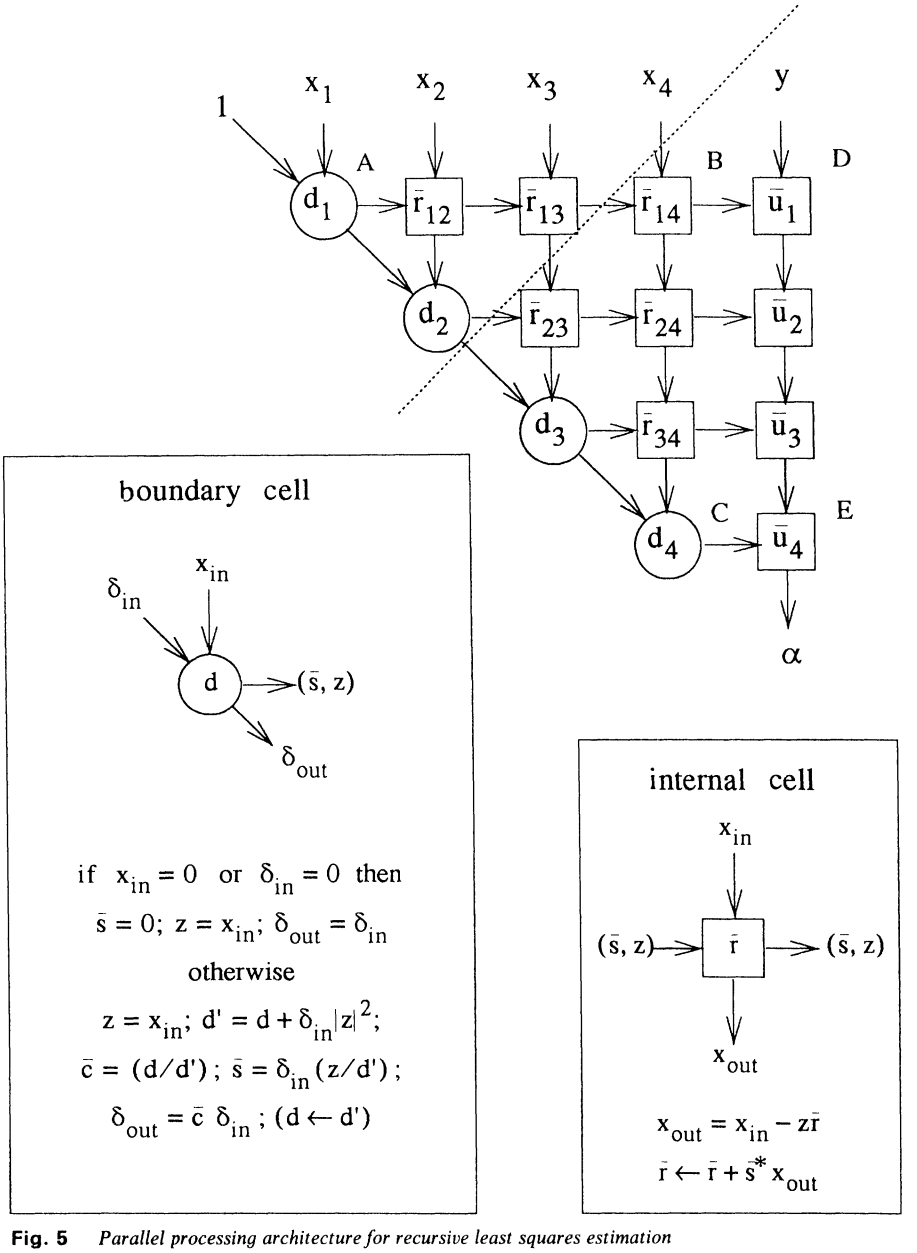


Fig. 5 *Parallel processing architecture for recursive least squares estimation*

The right-hand column of cells may be analysed in a similar manner. In the frozen mode, it is functionally equivalent to a single column of the rectangular matrix operator in Fig. 2. In the adaptive mode, it performs the same operation before updating the stored vector $\bar{\boldsymbol{u}}(n-1)$. The least squares processor array may therefore be represented by the simplified block diagram in Fig. 6*b* and, from the description above, it follows that the

output $\alpha(t_n)$ from the bottom cell in the right-hand column is given by the expression

$$\alpha(t_n) = y(t_n) - \boldsymbol{x}^T(t_n)\bar{\boldsymbol{R}}^{-1}(n-1)\bar{\boldsymbol{u}}(n-1) \tag{18}$$

Hence, from eqn. 17, we have

$$\alpha(t_n) = y(t_n) + \boldsymbol{x}^T(t_n)\boldsymbol{w}(n-1) \tag{19}$$

which is, by definition, the *a priori* least squares residual. In effect, by means of some fairly straightforward algorithmic engineering, it has been shown how the *a priori* residual may be extracted from the square-root-free least squares processor array without computing the optimum weight vector $\boldsymbol{w}(n-1)$ explicitly. It should be noted, however, that direct extraction of the *a posteriori* residual cannot be proved so easily. A more detailed mathematical analysis reveals that the output scalar $\alpha(t_n)$ must be multiplied by an auxiliary variable [8]. This comment applies to direct extraction of either type of residual, when conventional (as opposed to square-root-free) Givens rotations are applied. In this case, the adaptive algorithm cannot be viewed as a fixed matrix operation followed by an update procedure (or *vice versa*). Nonetheless, a simplified block schematic representation of the corresponding frozen processing network provides a very useful insight to the residual extraction techniques that have been developed.

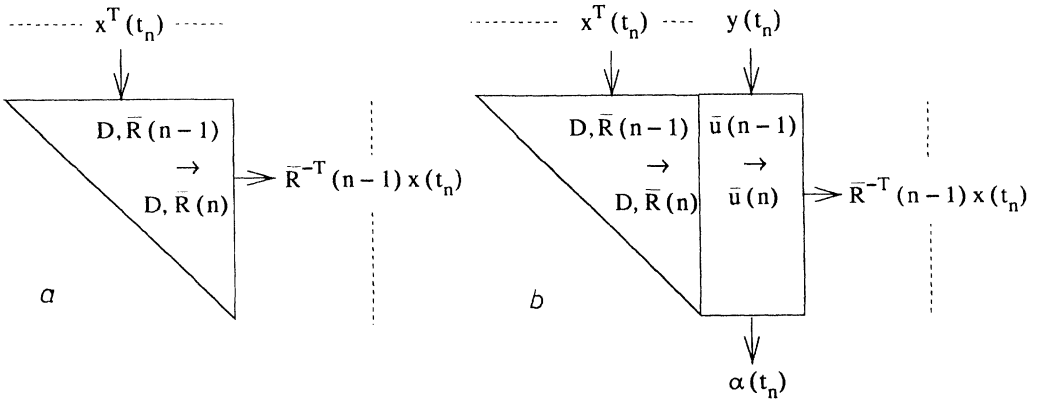


Fig. 6 *Schematic representation of arrays*
a Triangular *QR* decomposition array *b* Triangular least squares processor array

4 Triangular post-processor

The updating process performed by the triangular *QR* decomposition array in Fig. 5 may be described by an equation of the form

$$\hat{\boldsymbol{Q}}(n)\begin{bmatrix}\boldsymbol{R}(n-1)\\ \boldsymbol{x}^T(t_n)\end{bmatrix} = \begin{bmatrix}\boldsymbol{R}(n)\\ \boldsymbol{O}\end{bmatrix} \tag{20}$$

where $\hat{\boldsymbol{Q}}(n)$ is a unitary matrix representing the sequence of elementary Givens rotations which eliminates $\boldsymbol{x}^T(t_n)$. Now, it can easily be shown that

$$\hat{\boldsymbol{Q}}(n)\begin{bmatrix}\boldsymbol{R}^{-H}(n-1)\\ \boldsymbol{o}\end{bmatrix} = \begin{bmatrix}\boldsymbol{R}^{-H}(n)\\ *\end{bmatrix} \tag{21}$$

where $*$ denotes a vector of no specific interest. In other words, the inverse matrix $\boldsymbol{R}^{-H}(n)$ can be updated by applying the same sequence of Givens rotations as used to update the matrix $\boldsymbol{R}(n)$. This is readily proved by taking the Hermitian conjugate of eqn. 20 and post-multiplying the left-hand side by the term on the left-hand side of eqn. 21. The update procedure defined in eqn. 21 may be carried out, in practice, by extending the basic triangular array *ABC* in Fig. 5 to include an additional triangle of internal cells, as illustrated schematically in Fig. 7. If the matrix $\boldsymbol{K}(n-1)$, defined by

$$\boldsymbol{R}^{-H}(n-1) = \boldsymbol{D}^{1/2}(n-1)\boldsymbol{K}(n-1) \tag{22}$$

is stored within this additional triangular array at time t_{n-1}, the combined processing network will automatically update $\boldsymbol{D}(n-1)$, $\bar{\boldsymbol{R}}(n-1)$ and $\boldsymbol{K}(n-1)$ in response to the next input vector $\boldsymbol{x}^T(t_n)$, and so on. The additional triangular array, often referred to as the post-processor, may be initialised by freezing the main triangular array at time t_{n-1} so that it stores the matrix $\bar{\boldsymbol{R}}(n-1)$ and acts as a fixed matrix operator of the type illustrated in Fig. 3. If a unit matrix $\boldsymbol{I}$ is then input from the top, the matrix

$$\bar{\boldsymbol{R}}^{-T}(n-1) = \boldsymbol{D}(n-1)\boldsymbol{K}^*(n-1) \tag{23}$$

will emerge from the right-hand edge and enter the post-processor, where it can easily be captured, divided by the diagonal matrix $\boldsymbol{D}(n-1)$ and stored in the required complex conjugate form.

Now consider the effect of inputting the vectors $\boldsymbol{x}^T(t_n)$ and $\boldsymbol{o}$ to the combined adaptive processing network, as indicated in Fig. 7. The output vector $\bar{\boldsymbol{R}}^{-T}(n-1)\boldsymbol{x}(t_n)$ produced by the main triangular array also serves as an input to the post-processor along its left-hand edge. Prior to updating the matrix $\boldsymbol{K}(n-1)$, the post-processor acts as a (triangular) matrix multiplier of the type specified in Fig. 2 and, since its input from the top is $\boldsymbol{o}$, the vector $\boldsymbol{\phi}^T$,

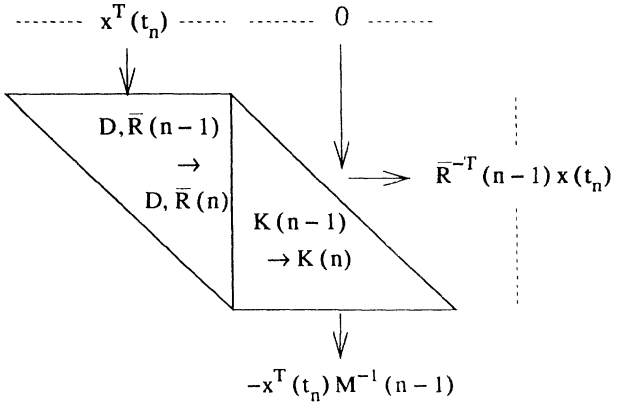


Fig. 7 *Schematic representation of triangular post-processor*

which emerges from below, is given by

$$\phi^T = -x^T(t_n)\bar{R}^{-1}(n-1)K(n-1)$$
$$= -x^T(t_n)R^{-1}(n-1)R^{-H}(n-1) \quad (24)$$

But, from eqn. 12, it follows that

$$M(n-1) = X^H(n-1)X(n-1)$$
$$= R^H(n-1)R(n-1) \quad (25)$$

and so

$$\phi^T = -x^T(t_n)M^{-1}(n-1) \quad (26)$$

where $M(n-1)$ is, by definition, an unnormalised estimate of the input covariance matrix, based on all data samples up to time t_{n-1}. The combined array therefore acts as an inverse covariance matrix operator, which is particularly useful, for example, in the design of parallel Kalman filters. It also serves, in effect, to update the underlying covariance matrix and its inverse in terms of the Cholesky square-root factors $R(n)$ and $R^{-H}(n)$. The orthogonal updating of $R(n)$ is known to be numerically stable. The procedure for updating $R^{-H}(n)$ has also been shown to be stable [9], although a steady build-up of errors can occur, and care must be taken in any practical application of the technique. Moonen [10] has recently shown how the numerical accuracy may be improved by performing additional Jacobi rotations, but his method is beyond the scope of this paper.

5 Multichannel lattice filter

The triangular QR decomposition array in Fig. 5 has now been adopted as the key building block for a number of higher-level processing structures, such as the square root information Kalman filter proposed by Chen and Yao [11], the square root covariance Kalman filter described by Gaston *et al.* [12] and the orthogonal multichannel lattice filter depicted in Fig. 8. The latter provides a particularly rich example of algorithmic engineering. Each of the triangles represents a $p \times p$ QR decomposition array of the type illustrated in Fig. 5, where p is the number of auxiliary input channels, i.e. the number of elements in the input vector x. The squares and rectangles represent $p \times p$ arrays and p-element columns of internal cells, respectively. The symbol Δ is used to denote a unit time delay.

The orthogonal multichannel lattice architecture can be derived from the conventional multichannel least squares lattice equations by expressing the covariance matrix, at each stage of the filter, in terms of its Cholesky square root factorisation. The Cholesky square root factor may, of course, be obtained directly from the underlying data by performing a recursive QR decomposition and exploiting the processing structure in Fig. 5. Incorporating the type of direct residual extraction process described in eqn. 19 then leads to the processing architecture in Fig. 8. This approach was adopted by Lewis [13] and also by Yang and Bohme [14]. By construction, the underlying algorithm must be orthogonal, as the only mathematical operations required are elementary Givens rotations. Accordingly, it has excellent numerical properties and is proving to be one of the most stable lattice algorithms available.

Proudler *et al.* [15] have recently shown how the orthogonal least squares lattice filter in Fig. 8 may be derived explicitly by applying a recursive QR decomposition to the type of data matrix generated by a multichannel tapped delay line, taking account of its block Toeplitz structure and thereby achieving the improved computational efficiency associated with least squares lattice algorithms. The requirement for a number of processing modules based on the triangular QR decomposition array arises quite naturally in this case.

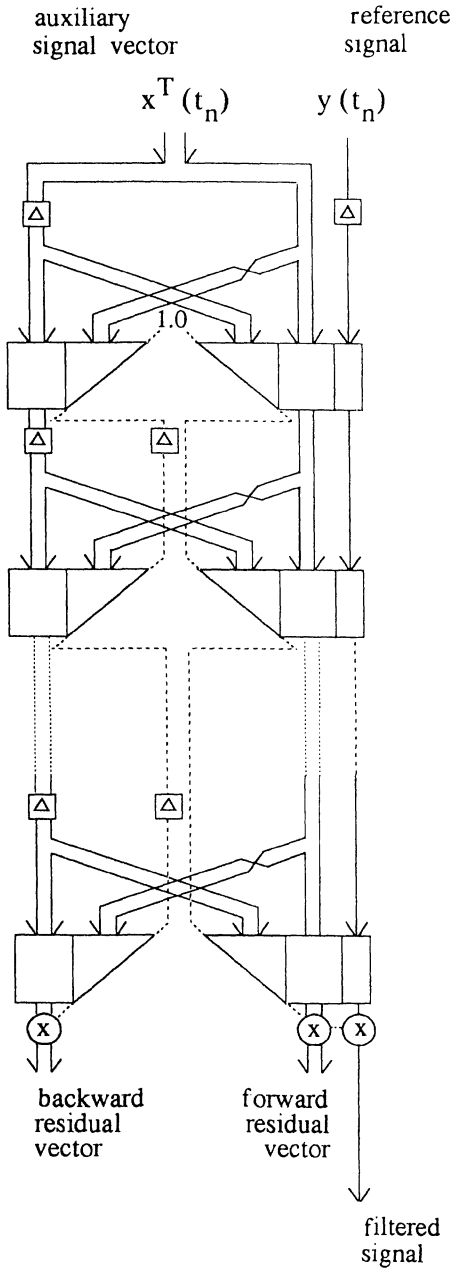


Fig. 8 *Multichannel least squares lattice filter based on Givens rotations*

The triangle denotes a $p \times p$ processor array, as defined in Fig. 5; the square and rectangle denote a $p \times p$ array and a p-element column of internal cells

By virtue of its modular structure, the diagram in Fig. 8 constitutes more than a highly regular parallel-processing architecture. It also provides a very compact and well-structured representation of the underlying algorithm. The only mathematics required to describe this very complex algorithm is that of the elementary square-root-free Givens rotation cells. As previously discussed, all cells within this multichannel lattice architecture are assumed to operate instantaneously, but, for any

practical application, the processor array may be partitioned and pipelined as appropriate. For examle, in the context of a recent application to acoustic adaptive beamforming, it was found that the computation required for each stage of the multichannel lattice filter could be performed in real time using two floating-point transputers, one for each of the extended QR decomposition modules [16].

6 Conclusions

In this paper, I have attempted to introduce and define the emerging concept of algorithmic engineering. The examples chosen serve to illustrate how various parallel algorithms and processing architectures may be represented very effectively in terms of simple mathematical operator diagrams. Section 2 introduces some fairly simple fixed matrix operators and shows how they may be combined in a very precise way to generate more powerful compound processors. This theme is generalised to adaptive processors in Sections 3 and 4, which consider the application of QR decomposition to least squares adaptive filtering. In this context, for example, the application of algorithmic engineering allows the (*a priori*) direct residual extraction technique to be proved in an extremely simply yet fully rigorous manner. Section 5 briefly illustrates how the simplified operator representations discussed in Section 3 may be used to describe and specify accurately a novel multichannel lattice filter based on QR decomposition. This is a much more complicated processing structure, whose derivation is beyond the scope of this paper, but the example serves to illustrate how powerful the use of formal diagrammatic techniques can be.

Throughout the paper, with the possible exception of Section 5, I have chosen to illustrate the concept of algorithmic engineering with reference to some fairly simple, well-known examples. It might appear, therefore, that the method is only useful for describing existing algorithms and architectures. However, the approach has already proved useful in the derivation of some novel processing structures. This is partly due to the fact that a simple but accurate representation of the basic building blocks helps to clarify the design process, and partly as a result of rigorous diagrammatic manipulation. Most recently, an original architecture for linearly constrained adaptive beamforming has been derived, by starting with an established processor design and applying a sequence of formal transformations to the basic operator diagrams, to produce an entirely different structure [17]. This example clearly suggests that the concept of algorithmic engineering, as introduced in this paper, could provide the basis of a powerful formal method for designing future parallel-processing architectures, at least in the context of digital signal processing.

7 References

1 KUNG, H.T., and LEISERSON, C.E.: 'Algorithms for VLSI processor arrays', *in* MEAD, C., and CONWAY, L. (Eds.): 'Introduction to VLSI systems' (Addison-Wesley, Reading, Mass., 1980)

2 GENTLEMAN, W.M., and KUNG, H.T.: 'Matrix triangularisation by systolic arrays', *Proc. SPIE*, 1981, Real Time Signal Processing IV, **298**, pp. 19–26

3 WARD, C.R., HARGRAVE, P.J., and McWHIRTER, J.G.: 'A novel algorithm and architecture for adaptive digital beamforming', *IEEE Trans.*, 1986, **AP-34**, (3), pp. 338–346

4 KUNG, S.Y.: 'VLSI array processors' (Prentice Hall Information & Systems Science Series, 1988)

5 McWHIRTER, J.G., and SHEPHERD, T.J.: 'Systolic array processor for MVDR beamforming', *IEE Proc. F, Commun., Radar & Signal Process.*, 1989, **136**, (2), pp. 75–80

6 KAILATH, T.: 'Linear systems' (Prentice-Hall Information & System Series, 1980), p. 656

7 FADEEV, D.K., and FADEEVA, V.N.: 'Computational methods of linear algebra' (Freeman and Co., 1963)

8 McWHIRTER, J.G.: 'Recursive least squares minimisation using a systolic array', *Proc. SPIE*, 1983, Real Time Signal Processing VI, **431**, p. 105

9 SCHREIBER, R.: 'Implementation of adaptive array algorithms', *IEEE Trans.*, 1986, **ASSP-34**, (5), p. 1038

10 MOONEN, M.: 'Jacobi-type updating algorithms'. PhD Thesis, Katholieke Universiteit Leuven, 1990

11 CHEN, M.J., and YAO, K.: 'Systolic Kalman filtering based on QR decomposition', *Proc. SPIE*, 1987, Advanced Algorithms and Architectures for Signal Processing II, **826**, pp. 25–32

12 GASTON, F.M.F., IRWIN, G.W., and McWHIRTER, J.G.: 'Systolic square root covariance Kalman Filtering', *J. VLSI Signal Process.*, 1990, **2**, (1), pp. 37–49

13 LEWIS, P.S.: 'Systolic architectures for adaptive multichannel least squares lattice filters', *J. VLSI Signal Process.*, 1990, **2**, (1), pp. 29–36

14 YANG, B., and BOHME, J.F.: 'On a parallel implementation of the adaptive multi-channel least squares lattice filter'. Proc. Int. Symp. on Signals Systems & Electronics, Erlangen, Sept. 1989

15 PROUDLER, I.K., McWHIRTER, J.G., and SHEPHERD, T.J.: 'QRD-based lattice algorithm for wide-band beamforming', *in* TORRES, L., MASGRAU, E., and LAGUNA, M.A. (Eds.): 'Signal processing V: Theories and applications' (Elsevier Science Publishers B.V., 1990)

16 PROUDLER, I.K.: 'Adaptive acoustic beamformer design study'. Interim Report, DRA, Malvern, Oct. 1991

17 McWHIRTER, J.G., and PROUDLER, I.K.: 'Algorithmic engineering: A worked example'. Proc. EUSIPCO Conference, Brussels, April 1992