GRAM-SCHMIDT ADAPTIVE ALGORITHMS

Technology Service Corporation

William C. Liles
James W. Demmel
Lawrence E. Brennan

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

80 5 14 071
This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-319 has been reviewed and is approved for publication.

APPROVED:  

VINCENT C. VANNICOLA  
Project Engineer

APPROVED:  

FRANK J. REHM  
Technical Director  
Surveillance Division

FOR THE COMMANDER:  

JOHN P. HUSS  
Acting Chief, Plans Office

If your address has been changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (OCTS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.
We consider solutions to the problem of computing adaptive weights in real time to maximize the signal-to-noise ratio of a multichannel input radar system with a large number of weights. The algorithms considered are based on the Gram-Schmidt orthogonalization process, and may be implemented on a modular multiprocessor computer using both parallel and pipeline techniques. The implementations are based on a simply and regularly connected processor array with simple and identical processors.
Item 20 (Cont'd)

(universal adaptive elements). The implementations can be made fast, fault tolerant, and easy to maintain.
EVALUATION

The significance of this contractual effort is the realization of generating a large number of adaptive weights simultaneously over more than one domain for radar systems. Such efforts will eventually render fully adaptive radar systems feasible. The effort supports RADC Technology Plan, TP0 R4B, by providing a technical basis for adaptive radar operation in a hostile and interfering electromagnetic environment. The results derived herein will serve as a basis for using universally designed adaptive modules to solve the large array multidomain problems.

VINCENT C. VANNICOLA
Project Engineer
## CONTENTS

### LIST OF FIGURES
- 3

### LIST OF TABLES
- 6

### Section

#### 1. INTRODUCTION
- 7
  - 1.1 STATEMENT OF THE PROBLEM
  - 1.2 RESTATEMENT OF THE PROBLEM IN MATHEMATICAL TERMS
  - 1.3 APPROACH

#### 2. DESCRIPTION OF THE ALGORITHM
- 13
  - 2.1 INTRODUCTION
  - 2.2 STATEMENT OF THE PROBLEM AND APPROACH
  - 2.3 IMPLEMENTATION OF TS
  - 2.4 COMPUTATION OF THE wij's
  - 2.5 IMPLEMENTATION OF M⁻¹(TS)
  - 2.6 IMPLEMENTATION OF T*(M⁻¹TS)
    - 2.6.1 Unit Vector Method
    - 2.6.2 Reverse Flow Method
    - 2.6.3 Forming the Filter Function in Transform Space
  - 2.7 THE SPECIAL CASE OF A SIDE-LOBE CANCELLER

#### 3. SYSTEM IMPLEMENTATION CONSIDERATIONS
- 45
  - 3.1 CALCULATION OF INNER PRODUCTS
  - 3.2 FAULT TOLERANCE AND NOISE SENSITIVITY
  - 3.3 GROWTH OF INTERMEDIATE RESULTS
  - 3.4 GRAM-SCHMIDT PROCESSING WITH O(n) PROCESSORS
    - 3.4.1 Advantages and Disadvantages
    - 3.4.2 Tradeoffs Between the Three O(n) Designs
  - 3.5 PROCESSOR INTERNAL PIPELINE/PARALLELISM
    - 3.5.1 Compute Inner Product
    - 3.5.2 Calculate Weights
    - 3.5.3 Apply Weight to Input Data
  - 3.6 DESIGN ALTERNATIVES AND TRADEOFFS
  - 3.7 UNIVERSAL ALGORITHM HARDWARE IMPLEMENTATION

#### 4. CONCLUSIONS
- 121

#### 5. RECOMMENDATIONS
- 127

### REFERENCES
- 129
CONTENTS (Cont'd)

Appendix

A Performance of the Sample Covariance Matrix Algorithm for Adaptive Arrays .......................................................... 131
B Derivation of Noise Sensitivity Results ........................................... 140
C Example of Using Gram-Schmidt to Solve A System of Simultaneous Equations .................................................. 146
D Reciprocal and Square-Root Calculation ........................................ 155
E Timing Equations for Block Average and O(n) Processors ............ 157
F Processing Timing for Block Average O(n^2) System .................. 167
G Proof of Results on Growth of Intermediate Results ................. 172
H Sample Voltage Vector Model ...................................................... 182
LIST OF FIGURES

Figure

1. Basic Gram-Schmidt array for n = 4 .................................................. 11
2. Processor array for computing TS ......................................................... 20
3. Timing diagram for computing TS ........................................................... 21
4a. Calculating $w_{ij}$ with the matrix M as input (without square roots) .............. 24
4b. Calculating $w_{ij}$ with the matrix M as input (with square roots) ... 24
5a. Calculating $w_{ij}$ with sample vectors as input (without square roots) .................. 25
5b. Calculating $w_{ij}$ with sample vectors as input (with square roots) .................. 26
6. Unit vector method array ........................................................................ 30
7. Timing diagram for the unit vector method ................................................ 32
8. Reverse flow method .............................................................................. 34
9. Timing diagram for the reverse flow method ............................................. 40
10. Computations for forming the filter function in transform space ..................... 42
11. Real data; 5 weights .............................................................................. 51
12. Real data; 5 weights .............................................................................. 52
13. Real data; 5 weights, 1 tap per channel ..................................................... 53
14. Simulated data ..................................................................................... 54
15. Simulated data ..................................................................................... 55
16. Simulated data ..................................................................................... 56
17. Simulated data ..................................................................................... 57
18. Simulated data ..................................................................................... 58
19. Simulated data ..................................................................................... 60
20. Simulated data ..................................................................................... 61
21. Simulated data ..................................................................................... 62
LIST OF FIGURES (Cont'd)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Effect of added noise on optimal SNR</td>
<td>68</td>
</tr>
<tr>
<td>23</td>
<td>Simulated data</td>
<td>71</td>
</tr>
<tr>
<td>24</td>
<td>Simulated data</td>
<td>72</td>
</tr>
<tr>
<td>25</td>
<td>Simulated data</td>
<td>73</td>
</tr>
<tr>
<td>26</td>
<td>Simulated data</td>
<td>74</td>
</tr>
<tr>
<td>27</td>
<td>Simulated data</td>
<td>75</td>
</tr>
<tr>
<td>28</td>
<td>Simulated data</td>
<td>76</td>
</tr>
<tr>
<td>29</td>
<td>Simulated data</td>
<td>77</td>
</tr>
<tr>
<td>30</td>
<td>Simulated data</td>
<td>78</td>
</tr>
<tr>
<td>31</td>
<td>Simulated data</td>
<td>79</td>
</tr>
<tr>
<td>32</td>
<td>Number of bits required to represent inner products</td>
<td>83</td>
</tr>
<tr>
<td>33</td>
<td>Collapsing an array into O(n) processors</td>
<td>88</td>
</tr>
<tr>
<td>34</td>
<td>Configuration using O(n)</td>
<td>89</td>
</tr>
<tr>
<td>35</td>
<td>Transposing internal weights</td>
<td>92</td>
</tr>
<tr>
<td>36</td>
<td>Inner-product computation time</td>
<td>100</td>
</tr>
<tr>
<td>37</td>
<td>Bus structure for pipelined (O(n^2)) configuration</td>
<td>110</td>
</tr>
<tr>
<td>38</td>
<td>Bus structure for recursive (O(n)) configuration</td>
<td>111</td>
</tr>
<tr>
<td>39</td>
<td>Variation combining O(n^2) and O(n) configurations</td>
<td>112</td>
</tr>
<tr>
<td>40</td>
<td>Gram-Schmidt (G-S) equations</td>
<td>114</td>
</tr>
<tr>
<td>41</td>
<td>Processor for the O(n^2) configuration</td>
<td>115</td>
</tr>
<tr>
<td>42</td>
<td>Processor for the O(n) configuration</td>
<td>116</td>
</tr>
<tr>
<td>43</td>
<td>Block diagram for node processing element (NPE)</td>
<td>117</td>
</tr>
<tr>
<td>44</td>
<td>Block diagram for diagonal processing element (DPE)</td>
<td>119</td>
</tr>
<tr>
<td>45</td>
<td>Timing curves for different Gram-Schmidt implementations</td>
<td>122</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (Cont'd)

Figure

46. Timing curves for various processors, showing banded region for Gram-Schmidt .............................................. 123
E-1. Model used for $O(n)$ timing measurements .......................... 158
E-2. $O(n)$ processor configuration with same speed to calculate internal weights as $O(n^2)$ processors using block averaging ......... 160
E-3. Recirculating pipelines ............................................. 162
E-4. Diagonal collapsing ................................................ 166
LIST OF TABLES

Table

1. Comparison of Inner-Product Numerator Implementation Techniques .................................................. 94

2. Comparison of Inner-Product Denominator Implementation Techniques .................................................. 101

3. Design Alternatives ................................................................. 105

4. Radar Engineering Considerations Affecting Choice of System Configuration ........................................ 106

5. Specific Radar Design Parameters ...................................................................................................... 108

6. Conclusions ................................................................................................................................. 124

E-1. Timings for Three O(n) Implementations .................................................................................... 163
1. INTRODUCTION

1.1 STATEMENT OF THE PROBLEM

The objective of this study, conducted under Contract No. F30602-78-C-0271, is to determine a modular architecture for a multiprocessor system to compute adaptive weights to maximize the signal-to-noise ratio (SNR) of a multichannel input radar system with a large number of weights. The radar system to be considered has channel inputs in all three signal domains—spatial, temporal, and polarization—with multiple inputs in each domain. The adaptive system must sense the environment and adjust at least 200 complex weights imposed on the array elements, on the delay taps off each element, and on the polarization sensors, thus maximizing the signal-to-noise ratio. This sensing must be accomplished on a time-varying basis, as the environment dictates, with minimum convergence time, minimum noise, and maximum stability.

For a multiprocessor to be able to calculate at least 200 complex weights quickly enough to adapt to the changing environment, not only must a fast adaptive algorithm and a fast processor system be chosen, but the algorithm chosen must itself be well suited to the processor system. The algorithms to be explored are based on the Gram-Schmidt orthogonalization process. A modular architecture to support these algorithms is designed.

1.2 RESTATEMENT OF THE PROBLEM IN MATHEMATICAL TERMS

Each antenna element in an adaptive array produces an input signal, which may be written

\[ X_j(t) = N_j(t) + S_j(t), \quad j=1, \ldots, n \]  

(1)
where \( X \) is the complex waveform as a function of time \( t \), \( N \) is the noise component, \( S \) is the signal component, the subscript \( j \) designates the particular antenna element, and \( n \) is the total number of antenna elements. \( X, N, \) and \( S \) may be thought of as complex column vectors of their components:

\[
X(t) = \begin{bmatrix}
X_1(t) \\
X_2(t) \\
\vdots \\
X_n(t)
\end{bmatrix}.
\]  

Instead of these continuous waveforms, we will usually deal with the sample vectors \( X^{(i)}, N^{(i)}, \) and \( S^{(i)} \), defined by:

\[
x^{(i)} = x(t_i),
\]

where the \( t_i \) are closely spaced points in time. Let \( W \) be a column vector of complex numbers. We form the filter function

\[
F = W^*X
\]

as the dot product of \( W \) and the input vector \( X \) (where \( W^* = W \) conjugate transpose).

Our problem is to choose the weights \( W \) to optimize the SNR of the system, and thereby maximize the probability of detection of the signal in the presence of jammers and clutter. These weights are a function of the sequence of sample input vectors \( X^{(i)} \). This report explores not only different algorithms for calculating the weights \( W \) through orthogonalizing the inputs \( X^{(i)} \) and filter function \( F \), but also the effects of different
modular parallel computer architectures on the implementations of these algorithms. These effects include execution speed, accuracy, convergence rates, and fault tolerance.

1.3 APPROACH

Let \( N \) be the column vector of noise inputs as, as in Section 1.2. Assuming the \( n \) components \( N_i \) have a multivariate Gaussian distribution, we can write their covariance matrix as

\[
M = E(\mathbf{N}\mathbf{N}^*)
\]  

(5)

It has been shown [Brennan and Reed 1973] that the optimal weights \( W \) are given by

\[
W = kM^{-1}\mathbf{S}
\]  

(6)

where \( \mathbf{S} \) is the expected signal vector (called steering vector) and \( k \) is any nonzero constant (normally 1).

\( M \) is not known, but must be estimated from sample data

\[
\hat{M} = \frac{1}{S} \sum_{j=1}^{S} x(j) x(j)^*
\]  

(7)

---

where $x^{(j)}$ are sample voltage vectors and $S$ is the number of samples. $\hat{M}$ is an $n \times n$ dense positive definite Hermitian matrix [Liles and Demmel 1978], and so forming $\hat{M}$ as in Eq. (7) and then computing $W$ from Eq. (6) with $\hat{M}$ in place of $M$ takes a great deal of computation.

Now let $T$ be a given linear transformation. Then the covariance matrix $M_T$ of the random vector $TN$ is given by

$$M_T + E[TN(TN)^*] = TE(NN^*)T^* = TMT^* \quad .$$

Thus, we have

$$W = M^{-1}S = T^*M^{-1}_TTS \quad .$$

If $T$ can be chosen so that $TS$ is easily computable, and $M_T$ is diagonal, we will have simplified our problem. Such a $T$ is given by the Gram-Schmidt orthogonalization process and produces a lower triangular matrix $T$ such that the entries of the random vector $TX$ are orthogonal.

The basic implementation of the processor for computing $T$ and $TX$ is shown in Figure 1. Each processor is very simple, and all are identical. The array operates with each row working in parallel on an intermediate step of $TX$, and with different rows working as stages in a pipeline to compute $TX$ for different $X$'s. The array overlaps its updates of its own internal coefficients $wij$ with computing $TX$ for different values of $X$.

---

Figure 1. Basic Gram-Schmidt array for $n = 4$. Arrows indicate direction of data flow.
Our approach is to analyze this basic scheme and its variations with respect to the following factors:

1. Variations on the basic mathematical algorithm, Section 2. We consider their different hardware implementations (e.g., cost and reliability properties) and stability (e.g., the number of bits required for intermediate steps).

2. Different ways of computing the internal coefficients, \( w_{ij} \), Section 3.1. The different algorithms are analyzed with respect to their hardware implementations, number of samples required for convergence, and sensitivity to the number of jammers. Simulations were performed.

3. Error sensitivity and fault tolerance, Section 3.2. Both theoretical and simulation results are presented on the effects of catastrophic and gradual failures of processors and receiver/antenna systems.

4. Different ways to perform arithmetic, Section 3.3. We consider both fixed- and floating-point implementations, and try to derive probabilistic bounds on the growth of intermediate results during the computations.

5. Implementation with fewer processors, Section 3.4. We consider speed, cost, and reliability tradeoffs when using only \( n \) processors instead of \( O(n^2) \) implementations of Figure 1.

6. Individual processor constructions, Section 3.5. The different ways of implementing each processor \( P_{ij} \) are considered with respect to speed and hardware complexity.

7. Radar engineering considerations, Section 3.6. We indicate how different system parameters might affect an engineer's decision of what version of the Gram-Schmidt algorithm to implement.

Finally, we select a typical set of radar system parameters and give a detailed hardware design for a Gram-Schmidt processor incorporating those parameters (see Section 3.7).
2. DESCRIPTION OF THE ALGORITHM

2.1 INTRODUCTION

An n-element radar system has inputs \( X_i, 1 \leq i \leq n \), where each component \( X_i \) is the sum of noise \( N_i \) and signal \( S_i \), expressed in column vector notation as

\[
X = N + S
\]  

(10)

The output of the system is the filter function, \( F \), given by

\[
F = W^*X
\]  

(11)

where \( W \) is a column vector of complex weights and \(*\) denotes conjugate transpose. Brennan and Reed [1973] have established that in order to maximize the signal-to-noise ratio (SNR) of the system, \( W \) should satisfy

\[
MW = S
\]  

(12)

where \( M \) is the covariance matrix of the noise

\[
M_{ij} = E(N_iN_j)
\]  

(13)

In a real system, \( M \) is not known and must be estimated from incoming samples \( X \). This section describes an algorithm based on Gram-Schmidt (G-S) orthogonalization to form \( M \), solve the system of simultaneous linear equations determining \( W \), (Eq. (12)), and compute \( F \).

The literature contains both practical and highly theoretical results on solving systems of equations. Csanky's algorithm [1976]* requires the

least number of operations of any algorithm \(O(\log^2 n)\) but is impractical because it requires too many processors connected in a complicated way, and is also numerically unstable. Gentleman [1978] has given bounds on the time spent routing data between processors during Gaussian Elimination. Several investigators have benchmarked various algorithms on different machines [Calahan et al. 1976; Liles and Demmel 1978; Blakely 1977; Berra and Singhania 1976; Sameh and Kuck 1975].** Sameh and Kuck [1975, 1977a, 1977b, 1978]*** have published survey articles. Kung and Leiserson [1978]† have produced algorithms emphasizing simple processors and simple interconnect structures.

The matrix \(M\) in our problem has two very nice properties which make solving Eq.(12) easier than in the case of a general matrix: It is Hermitian (i.e.,

---


M = M*) and positive definite (i.e., all its eigenvalues are positive, or, equivalently, Z*MZ > 0 for all nonzero vectors Z) [Liles and Demmel 1978].

Positive definite Hermitian matrices arise frequently in numerical work, often implicitly as the sample covariance matrix of a set of sample vectors. This is the situation not only in the adaptive signal processing problem which motivated this study, but in many regression and least squares problems. For example, solving the system AX = B in a least squares sense is equivalent to solving the normal equations (A*A)X = A*B, where A*A is positive definite and Hermitian (if A has full rank and where A* = A conjugate transpose). Positive definite Hermitian matrices also arise explicitly in variational problems, such as solving self-adjoint differential equations with finite element methods.

The remainder of this section is organized as follows: Section 2.2 gives a statement of the problem and the approach to the problem; Section 2.3 presents an implementation of TS; Section 2.4 computes the w, j's; Section 2.5 gives an implementation of M⁻¹(S); Section 2.6 presents an implementation of T*(M⁻¹TS); and Section 2.7 describes possible simplification of the implementation in the case of a side-lobe canceller (SLC). Appendix C presents a numerical example of how to use the G-S array to solve a system of simultaneous linear equations.
2.2 STATEMENT OF THE PROBLEM AND APPROACH

Let the system to be solved be $MW = S$, where $M$ is a positive definite Hermitian $n \times n$ matrix, $S$ is an $n \times 1$ vector, and $W$ is an $n \times 1$ vector of unknowns. If $T$ is a nonsingular $n \times n$ matrix (to be specified later), define $M_T = TMT^*$, where $T^* = (T)^T$. Then $M_T$ is also positive definite Hermitian (by Sylvester's law of inertia) and $W = M^{-1}S = T^*M_T^{-1}TS$. By choosing $T$ so that $T$, $M_T^{-1}$, and $T^*$ are easily computable, we will have simplified our problem.

A good choice of $T$ is given by the Gram-Schmidt orthogonalization process set forth in Eq. (14) [Rice 1966]*, where $<a, b>_M$ denotes the complex inner product induced by the matrix $M$: $<a, b>_M$ denotes the complex inner product induced by the matrix $M$: $<a, b>_M = b^*Ma$.

\[
\begin{align*}
Z_j^1 &= S_j \quad \text{(input)} \\
Z_j^{i+1} &= Z_j^i - \frac{<Z_j^i, Z_i^i>_M}{<Z_i^i, Z_i^i>_M} Z_i^i, \quad 1 \leq i < n - 1, \quad i < j \leq n \\
Z_j &= Z_j^n \quad \text{(output: } Z = TS) 
\end{align*}
\]  

(14)

The coefficients $w_{ij} = <Z_i^i, Z_i^i>_M/<Z_i^i, Z_i^i>_M$ are obtained by performing Gram-Schmidt orthogonalization on the standard basis $e_j^i$, $1 \leq j \leq n$, where $e_i^i = \{1$ if $i = j$ and 0 otherwise) with the inner product $<. , .>_M$. Hence, the vectors $e_j^i$ are orthogonal with respect to the inner product induced by $M_T$, which implies $M_T$ is a diagonal matrix, since $<e_i^i, e_i^i>_M = M_T^{-1} = 0$ unless $i = j$. Also, $M_i^i = <e_i^i, e_i^i>_M = <Z_i^i, Z_i^i>_M$.

---

The $T$ induced by Eq. (14) is unit lower triangular (lower triangular with ones on the diagonal), since $Z_j = S_j$ plus a linear combination of $S_1$ through $S_{j-1}$. Hence $T^{-1}$ is also unit lower triangular, and both $T^*$ and $(T^*)^{-1}$ are unit upper triangular. Since $M = T^{-1}M_T(T^{-1})^*$, we have written $M$ as the product of a unit lower triangular matrix $L = T^{-1}$, a diagonal matrix $D = M_T$, and $L^* = (T^{-1})^*$: $M = LDL^*$. This is the same decomposition provided by the Cholesky algorithm (without square roots [Liles and Demmel 1978]), and because the Cholesky decomposition is unique, $(T^{-1})$ and $M_T$ are the matrices that would be produced by the Cholesky algorithm. In fact, we show in Subsection 2.6.2 that the coefficients $w_{ij}$ are the entries of the matrix $L$.

Viewing $M$ as the covariance matrix of a column vector $V$ of random variables with a multivariate normal distribution with zero means (which is computed as the average of the covariances of $s$ sample vectors $V_k, 1 \leq k \leq s$: $M_{ij} = 1/s \sum_{k=1}^{s} V_i^k V_j^k = E(V_i V_j)$), $M_T$ is then the covariance matrix of the random vector $Z = TV$:

$$M_T = E(ZZ^*)$$
$$= E(TV(TV)^*)$$
$$= E(TVV^*T^*)$$
$$= TE(VV^*)T^* \text{ since } T \text{ is constant}$$
$$= TMT^*$$

Thus $M_T$ diagonal means $Z_i$ and $Z_j$ are uncorrelated (orthogonal). This statistical interpretation helps us recognize that this method may be used to solve least squares and regression problems.

The variation on Eq. (14) given in Eq. (15) uses a nonunit lower triangular matrix $T$ and corresponds to the Cholesky decomposition $M = LL^*$. This variation
has a different implementation and different round off and stability properties, which will be discussed later.

\[
\begin{align*}
Z_{ij}' &\equiv S_j \quad \text{(input)} \\
\left\{ 
\begin{array}{l}
Z_{ij}' = \frac{Z_{ij}'}{<Z_{ij}',Z_{ij}'>_M} \\
Z_{ij}'^{i+1} = Z_{ij}' - \frac{<Z_{ij}',Z_{ij}'>_M}{Z_{ij}'^2}Z_{ij}'
\end{array}
\right. \\
1 \leq i \leq n, \quad i < j \leq n
\end{align*}
\]

(15)

\[
Z_j = Z_j^2 \quad \text{(output: } Z = TS)\]

Because of the normalization in the second line of Eq. (15), \( MT_{ij} = <Z_i,Z_i>_M = 1 \). Hence, \( M_T = I = \text{identity matrix} \). Writing \( L = T^{-1} \) as before, we now have \( M = LL^* \), the Cholesky decomposition (with square roots). As before, the coefficients \( w_{ij}' = <Z_{ij}',Z_{ij}'> \text{ if } i \neq j; 1/(<Z_{ij}',Z_{ij}'>)^{\frac{1}{2}} \text{ if } i=j \) are the entries of the Cholesky factor \( L \). The coefficients from Eq. (15) are related to the coefficients from Eq. (14) by \( w_{ij} = w_{ij}' \cdot w_{ij}' \) for \( i \neq j \). If \( V \) is a column vector of random variables as before, then the entries of \( Z = TV \) are independent Gaussian random variables with unit variances.

In summary, the algorithm to solve \( MW = S \) is:

1. Compute the \( w_{ij}' \)'s, either directly from the entries of the \( M \) matrix, or from sample vectors.
2. Compute \( M_T^{-1} \).
3. Compute \( TS \).
4. Compute \( M_T^{-1}(TS) \).
5. Compute \( T^*(M_T^{-1}TS) \).
2.3 IMPLEMENTATION OF TS

Our implementation of TS is shown in Figure 2, for n = 4. Arrows indicate directions of data flow. As in Eq. (14) (Section 2.2),

\[ w_{ij} = \frac{<Z_j^i, Z_i^i>}{<Z_i^i, Z_i^i>} \]

and processor \( P_{ij} \) performs the computation

\[ Z_{i+1}^j = Z_i^j - w_{ij} Z_i^i \]

The \( n(n - 1)/2 \) processors operate as follows: The processors in row \( i \) operate simultaneously on values passed to them by the row above. \( Z_{i+1}^j \) is passed vertically from \( P_{ij} \) to \( P_{i+1, j} \), and the rightmost processor \( P_{i, i+1} \) ("diagonal" processor) broadcasts \( Z_{i+1}^i \) to all the processors in the next row simultaneously. All processors operate simultaneously and in lockstep, each row performing its operations on the intermediate results of a different input vector in one unit time step. Different \( S \) vectors may be piped into the top of the array, and a new vector entered each unit time step.

A timing diagram is shown in Figure 3 for the case \( n=4 \). Superscripts indicate different vectors; subscripts, different components. The horizontal axis is labelled in unit time steps. The top four rows indicate when data should be submitted to the input ports at the top of the array, and the bottom four rows indicate when the components of the output vectors \( Z^j = TS^j \) are available at the output ports at the right of the array. One can see that the \( m \) products \( Z^j = TS^j, 1 \leq j \leq m \), can be formed in \( n+m \) time steps.

This implementation has the following advantages: It has a simple interconnect structure, with unidirectional and fixed-destination data flow; and the processors are very simple, identical, and independent of \( n \).
Figure 2. Processor array for computing TS.
<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1  2  3  4  5  6  7  8  9  10</td>
</tr>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>Port No. 1</td>
<td>$S_1^1$  $S_2^1$  $S_3^1$  $S_4^1$  $S_5^1$  -</td>
</tr>
<tr>
<td>Port No. 2</td>
<td>$S_1^2$  $S_2^2$  $S_3^2$  $S_4^2$  $S_5^2$  -</td>
</tr>
<tr>
<td>Port No. 3</td>
<td>$S_1^3$  $S_2^3$  $S_3^3$  $S_4^3$  $S_5^3$  -</td>
</tr>
<tr>
<td>Port No. 4</td>
<td>$S_1^4$  $S_2^4$  $S_3^4$  $S_4^4$  $S_5^4$  -</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td></td>
</tr>
<tr>
<td>Port No. 1</td>
<td>$Z_1^1$  $Z_2^1$  $Z_3^1$  $Z_4^1$  $Z_5^1$  -</td>
</tr>
<tr>
<td>Port No. 2</td>
<td>$Z_1^2$  $Z_2^2$  $Z_3^2$  $Z_4^2$  $Z_5^2$  -</td>
</tr>
<tr>
<td>Port No. 3</td>
<td>$Z_1^3$  $Z_2^3$  $Z_3^3$  $Z_4^3$  $Z_5^3$  -</td>
</tr>
<tr>
<td>Port No. 4</td>
<td>$Z_1^4$  $Z_2^4$  $Z_3^4$  $Z_4^4$  $Z_5^4$  -</td>
</tr>
</tbody>
</table>

Figure 3. Timing diagram for computing TS.
Also, the processor array works without modification on vectors of size less than $n$.

In the case of Eq. (15) (with square roots) of Section 2.2, the only difference in implementation is that there must be a processor to divide $Z_{i_1}^{f_1}$ by its norm $(<Z_{i_1}^{f_1}, Z_{i_1}^{f_1}>_M)^{1/2}$ before broadcasting it.
2.4 COMPUTATION OF THE $w_{ij}$'s

The $w_{ij}$'s are computed in two ways, depending on whether the matrix $M$ is known or must be estimated from vectors sampled from the underlying distribution. It turns out that the two methods require similar hardware. We first discuss the algorithm without square roots (Eq. (14)).

If the matrix $M$ is available, the $w_{ij}$'s are computed by pipelining the columns of $M$ into the top of the processor array, and performing the computations indicated in Figure 4. $Z_j(k)$ denotes the intermediate result from the $k$th column of the matrix $M$. The IF-THEN-ELSE in Figure 4 may be implemented by having a control line for each row of the array: During normal operation (computing $TS$, or the ELSE clause in Figure 4) the control signal is off, and when the data from column $i$ has finally reached row $i$ in the array, the control signal should be turned on to tell the processors to compute $w_{ij}$. A timing diagram for the complete implementation appears later, in Figures 7 and 9 (Section 2.6).

The configuration just described requires a divide unit in each processor. Because each processor in row $i$ computes the same reciprocal $1/Z_i(i)$, the diagonal processor, $P_{i-1,i}$, can also compute this value and broadcast it along with $Z_i(i)$ itself. This approach simplifies all but $n-1$ of the processors, but requires more data to be broadcast.

Either configuration can compute all the $w_{ij}$'s in $n+1$ unit time steps.

If sample vectors are known instead of the actual matrix, the method is similar. If $s$ sample vectors are given, all $s$ must be passed through the array $n-1$ times, performing the computations shown in Figure 5a. Each time the $s$ samples are passed through, another row of $w_{ij}$'s is calculated.
Figure 4a. Calculating $w_{ij}$ with the matrix $M$ as input (without square roots).

Figure 4b. Calculating $w_{ij}$ with the matrix $M$ as input (with square roots).
Figure 5a. Calculating $w_{ij}$ with sample vectors as input (without square roots).
if $k = 1$ then

if not last sample then

$$
\begin{align*}
\hat{w}_{ij}^1 &= \hat{w}_{ij}^1 + Z_{ij}(k) \cdot Z_{ij}(k) \\
\hat{w}_{ij}'' &= \hat{w}_{ij}'' + Z_{ij}(k) \cdot Z_{ij}(k)
\end{align*}
$$

else

$$
\begin{align*}
\hat{w}_{ij}^1 &= \hat{w}_{ij}^1 \\
\hat{w}_{ij}'' &= \hat{w}_{ij}''
\end{align*}
$$

else

$$
Z_j^{z+1}(k) = Z_j^z(k) - \hat{w}_{ij}^1 [\hat{w}_{ij}^1 \cdot Z_j^z(k)]
$$

Figure 5b. Calculating $\hat{w}_{ij}^1$ with sample vectors as input (with square roots).
In the expression $Z_j^i(k)$ the $k$ denotes which row of $w_{ij}$'s is being calculated. A total of $[(s + 1)(n - 1) + 1]$ time steps are required to compute all the $w_{ij}$'s.

As in the case where the matrix $M$ was input, an alternate configuration is to have only the diagonal processor compute $\tilde{w}_{ij} = \tilde{w}_{ij} + Z_j^i(k) \cdot Z_i^j(k)$ and $1/\tilde{w}_{ij}$ and broadcast $1/\tilde{w}_{ij}$.

The implementation of the algorithm with square roots (Eq. (15)) is similar and is shown in Figures 4b and 5b.

Since each processor $P_{ij}$ in row $i$ needs the same quantity $w_{ii}$ (or its reciprocal), it is possible to have a diagonal processor $P_{ii}$ to calculate this value only and broadcast it (along with $Z_i^j(k)$, in the case of Figure 5b). This possibility increases communication needs slightly but makes each $P_{ij}$ much simpler.

Other methods exist for computing the $w_{ij}$'s when sample vectors are used; they are variations on the general scheme shown above and are discussed in Section 3.1.
2.5 IMPLEMENTATION OF $M_T^{-1}(TS)$

Because $M_T$ is diagonal, $M_T^{-1} = \text{diag}(<Z_i^1, Z_i^1>_M^{-1})$, and these values are available in any processor (i.e., $<Z_i^1, Z_i^1>_M$ is in $P_j^i$ for all $j$), it is easy to compute $M_T^{-1}(TS)$: Just attach a multiply unit to each output port on the right of the array and multiply the output at port $i$ by the value of $<Z_i^1, Z_i^1>_M^{-1}$ borrowed from the processor just to the left, $P_n^i$. When the algorithm with square roots is used, $M_T$ is the identity matrix and so nothing needs to be done.
2.6 IMPLEMENTATION OF $T^*(M^{-1}TS)$

This step of the algorithm, the formation of $T^*(M^{-1}TS)$, has three different implementations. The first, the unit vector method, uses the array shown in Figure 2 (Section 2.3) with $n$ additional processors on the right. It can solve the $m$ problems $MW^i = S^i, 1 \leq i \leq m$, in time $O(mn)$ with high efficiency

$$E = \frac{\text{processor on-time}}{(\text{total time} \cdot \text{total processors})}$$

$$= \frac{m}{m+1} \frac{[(n+1)^2 - 2]/(n+1)^2}{(n+1)^2}$$

The second method, the reverse flow method, requires some changes to the array but preserves the simple interconnect scheme. It can solve the $m$ problems in time $O(m+n)$ but with somewhat lower efficiency

$$E = \frac{m}{m + 2(n+1)}$$

The third method, the transform space method, is essentially different from the other two in its use of the array to compute the filter function, $F$, directly. The first two methods compute the weights, $W$, which must then be used (by another piece of hardware) to compute $F$.

Overall timings of the different implementations are given. A more detailed comparison of the unit vector and reverse flow methods is given in Appendix F.

2.6.1 Unit Vector Method

The unit vector method forms $T^*(M^{-1}TS)$ with a vertically connected array of $n$ simple processors connected to the output ports of the array as in Figure 6. First, the vector $S$ is passed through the array, and the $n$ numbers $Y_j = (M^{-1}TS)_j$ are formed and stored in registers inside the processing elements (PE's). Then, the $n$ unit vector $E^j (E^j_i = 1 \text{ if } i = j \text{ and } 0 \text{ otherwise})$ are piped through the array to form the $n$ products $TE^j_i$. Given that
The image contains a diagram of a processor array with the following equations:

\[ D_j = D_{j-1} + (TE)^i_j \cdot Y_j \]

\[ Y_j = (M^{-1} TS)_j \]

\[ D_0 = 0, \quad D_n = \sum_{j=1}^{n} (TE)^i_j \cdot Y_j = \text{OUTPUT} \]

Figure 6. Unit vector method array.
\[(T*Y)_j = \sum_{k=1}^{n} T_k Y_k = \sum_{k=1}^{n} (T^{E^j})_k Y_k\]

and that terms \((T^{E^j})_k\) and \(Y_k\) are both available at the \(k^{th}\) output port, these terms may be multiplied, added to a partial sum computed by \(PE_{k-1}\) during the preceding time step, and passed to \(PE_{k+1}\) for the next time step, as shown in Figure 6.

A timing diagram for the entire process of solving \(MW^i = S^i\) for \(n = 3\) and \(1 \leq i \leq 2\) is shown in Figure 7. \(Y^i_j\) denotes \((M^iT^TS^i)_j\), and \(D^i_j\) denotes the output of \(PE_j\) with input \((T^{E^i})_j\).

A timing analysis may be made easily from the timing diagram. A total of \(n\) time steps are required to compute the \(w_{ij}\)'s. Not counting these time steps, a total of \(TUV = (m+1)(n+1)\) units of time are required to pass \(m\) vectors \(S^i\) and \(mn\) unit vectors \(E^j\) through the array. We may compute the efficiency of the array where \(T_p\) is the time processor \(p\) is on, \(P\) is the number of processors, and \(T\) is total time required to finish processing, as follows:

\[E_{UV} = \sum_{all\ processors} \frac{T_p}{(T*P)}\]

Each of the \(n(n-1)/2\) processors in the array has \(T_p = m(n+1)\), and each of the \(n\) outboard \(PE\)'s has \(T_p = mn\); therefore,

\[E_{UV} = \left( \frac{m}{m+1} \right) \left[ \frac{(n+1)^2 - 2}{(n+1)^2} \right] \]

Thus, \(E_{UV}\) approaches 1 as \(m\) and \(n\) grow. The unit vector method is thus an efficient implementation, and it is the same for both the with-square-roots and without-square-roots versions of the algorithm.
Figure 7. Timing diagram for the unit vector method.
2.6.2 Reverse Flow Method

The reverse flow method depends on the interesting fact that the array can compute \( T^*Y \) by inputting the components of the vector \( Y \) at the right side of the array, performing all the computations in the reverse order (using \( w_{ij} \) instead of \( w_{ji} \)), and extracting the outputs at the top of the array. This process is illustrated in Figure 8. Now the broadcasts take place in the vertical direction, whereas when computing \( TS \) the broadcasts were in the horizontal direction.

A proof of this fact for the algorithm without square roots (illustrated here for \( n = 4 \)) depends on the fact that \( T \) can be written

\[
T = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -w_{34} & 1
\end{bmatrix}
\]

The \( i^{th} \) matrix in this product represents the computations performed by the \((n-i)^{th}\) row of the array. Hence,

\[
T^* = [(T^*)^{-1}]^{-1} = [(T^{-1})^*]^{-1}
\]

\[
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
w_{12} & 1 & 0 & 0 \\
w_{13} & 0 & 1 & 0 \\
w_{14} & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & w_{23} & 1 & 0 \\
0 & 0 & w_{24} & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & w_{34} & 1
\end{bmatrix}
\]
Figure 8. Reverse flow method.
The $i^{th}$ matrix in this product for $T^*$ represents the computations performed by the $(n-i)^{th}$ column of the array during reverse flow.

The same type of argument shows that the $w_{ij}$ are the entries of the matrix $L$ in the Cholesky factorization $M = LDL^*$. Recall from Section 2.2 that $L = T^{-1}$. 

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
The proof is similar for the algorithm with square roots:
\[ T = \begin{bmatrix}
1 & 1 & 1 \\
1 & w_{33} & -w_{34} \cdot w_{33}
\end{bmatrix}
\begin{bmatrix}
1 & w_{22} & -w_{23} \cdot w_{22} \\
-w_{23} \cdot w_{22} & -w_{34} \cdot w_{22}
\end{bmatrix} \]

\[ T^* = [(T^{-1})^*]^{-1} \]

\[ = \begin{bmatrix}
1/w_{11} & 1/w_{22} & 1/w_{33} \\
1/w_{12} & 1/w_{23} & 1/w_{34} \\
1/w_{13} & 1/w_{24} & 1/w_{44}
\end{bmatrix}^{-1} \]

\[ = \begin{bmatrix}
1/w_{11} & 1/w_{22} \\
1/w_{12} & 1/w_{23} & 1/w_{33} \\
1/w_{13} & 1/w_{24} & 1/w_{44}
\end{bmatrix}^{-1} \]
\[
\begin{align*}
&\left[\begin{array}{cccc}
\frac{1}{w_{11}^j} & \frac{w_{12}^j}{w_{11}^j} & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
\frac{1}{w_{22}^j} & \frac{w_{23}^j}{w_{22}^j} & \frac{w_{24}^j}{w_{22}^j} & \\
\frac{1}{w_{33}^j} & \frac{w_{34}^j}{w_{33}^j} & \\
\frac{1}{w_{44}^j} & \\
\end{array}\right]^{-1} \\
= & \left[\begin{array}{cccc}
1 & \frac{w_{12}^j}{w_{11}^j} & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
1 & \frac{w_{23}^j}{w_{22}^j} & \frac{w_{24}^j}{w_{22}^j} & \\
1 & \frac{w_{34}^j}{w_{33}^j} & \\
1 & \\
\end{array}\right]^{-1} \\
= & \left[\begin{array}{cccc}
1 & \frac{w_{12}^j}{w_{11}^j} & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
1 & \frac{w_{23}^j}{w_{22}^j} & \frac{w_{24}^j}{w_{22}^j} & \\
1 & \frac{w_{34}^j}{w_{33}^j} & \\
1 & \\
\end{array}\right]^{-1} \\
& \cdot \left[\begin{array}{cccc}
1 & \frac{w_{12}^j}{w_{11}^j} & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
1 & \frac{w_{23}^j}{w_{22}^j} & \frac{w_{24}^j}{w_{22}^j} & \\
1 & \frac{w_{34}^j}{w_{33}^j} & \\
1 & \\
\end{array}\right] \\
& \cdot \left[\begin{array}{cccc}
1 & \frac{w_{12}^j}{w_{11}^j} & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
1 & \frac{w_{23}^j}{w_{22}^j} & \frac{w_{24}^j}{w_{22}^j} & \\
1 & \frac{w_{34}^j}{w_{33}^j} & \\
1 & \\
\end{array}\right]^{-1} \\
= & \left[\begin{array}{cccc}
\frac{1}{w_{11}^j} & 1 & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
\frac{1}{w_{22}^j} & \frac{w_{23}^j}{w_{22}^j} & 1 & \frac{w_{24}^j}{w_{22}^j} \\
\frac{1}{w_{33}^j} & \frac{w_{34}^j}{w_{33}^j} & 1 & \\
\frac{1}{w_{44}^j} & 1 & \\
\end{array}\right]
\end{align*}
\]

Note that the quantity being broadcast vertically in column \( j \) gets multiplied by \( w_{jj}^j \) before broadcasting, since the processors in column \( j \) do not contain the number \( w_{jj}^j \). Also,

\[
T^{-1} = \left[\begin{array}{cccc}
\frac{1}{w_{11}^j} & 1 & \frac{w_{13}^j}{w_{11}^j} & \frac{w_{14}^j}{w_{11}^j} \\
\frac{1}{w_{22}^j} & \frac{w_{23}^j}{w_{22}^j} & 1 & \frac{w_{24}^j}{w_{22}^j} \\
\frac{1}{w_{33}^j} & \frac{w_{34}^j}{w_{33}^j} & 1 & \\
\frac{1}{w_{44}^j} & 1 & \\
\end{array}\right]
\]
so that the \( w_{ij} \)'s are the entries of the Cholesky factor \( L \) in \( M = LL^* \), except that the \( w_{ii} \)'s are the reciprocals of the diagonal elements of \( L \).

The same lockstep pipelined operation for data movement that was used for \( T \) is used for \( T^* \), except backwards. A timing diagram for the entire operation of solving \( MW^i = S^i \) for \( n = 3 \) and \( 1 \leq i \leq 4 \) is shown in Figure 9. A shift register of length \( n-j \) is required at output port \( j \) in order to resubmit the components of the vector \( M^{-1}_T S^i \) to the array simultaneously. In Figure 9, \( Y^i_j \) denotes \( M^{-1}_T S^i_j \) and \( W^i_j = M^{-1}_T S^i_j \), the solution.

As before, we analyze the timing excluding the time steps required to compute the \( w_{ij} \)'s. If \( m \) vectors \( S^i \) are to be input at the top, the total time required is \( m \) (to input the \( S^i \)'s) plus \( n \) (to compute \( TS^i \)) plus 1 (to compute \( M^{-1}_T TS^i \)) plus 1 (to resubmit \( Y^i \)) plus \( n \) (to compute \( T^*Y \)):

\[
T_{RF} = m + 2(n + 1) = O(m + n)
\]

This relation is to be contrasted with the \( T_{UV} = (m + 1)(n + 1) \) for the unit vector method. Analyzing efficiency as before, but interpreting \( P_{ij} \) as two processors, one for \( T \) and one for \( T^* \), we have \( n(n - 1) \) processors being used for \( m \) time units and \( n \) processors for \( m \) units, for an efficiency:

\[
E_{RF} = m/[m + 2(n + 1)]
\]

This efficiency function has a behavior different from \( E_{UV} \) for the unit vector method: \( E_{RF} \) is low for small \( m \) and large \( n \), whereas \( E_{UV} \) was an increasing
Figure 9. Timing diagram for the reverse flow method.
function of both m and n. This is a typical speed/efficiency design tradeoff often encountered when designing parallel systems [Chen 1971a; 1971b].

2.6.3 Forming the Filter Function in Transform Space

The ultimate output of the system under consideration is the filter function

\[ F = W^*X \]

where X is a column vector of receiver inputs. The algorithm discussed in the preceding two subsections (2.6.1 and 2.6.2) computes W, assuming another piece of hardware is available to form the inner product \( W^*X \). It is possible to avoid computing W itself, and to form the filter function as follows:

\[
F = W^*X \\
= (M^{-1}S)^*X \\
= S*M^{-1}X \\
= S*T*(T^*)^{-1}M^{-1}T^{-1}TX \\
= (TS)*(TMT^*)^{-1}TX \\
= (TS)M_T^{-1}(TX)
\]

Recalling that \( M_T \) is a diagonal matrix, we see that the vector \((TS)^*M_T^{-1}\) may be computed once and stored, and then its inner product with TX computed.

The hardware required to implement this approach is virtually identical to that of the unit vector method. After the \( w_{ij} \)'s have been calculated

(including the entries of $\mathbf{M}^{-1}_T$), the steering vector $\mathbf{S}$ is passed through the array, and the values $[(\mathbf{T}S)^*\mathbf{M}^{-1}_T]_i$ stored in a vertically connected array of processors to the right of the main array, as in Figure 10. Then, as the $\mathbf{X}$ vectors are pipelined into the top of the array, their results $(\mathbf{T}X)_i$ will be available at $\text{PE}_i$ where $[(\mathbf{T}S)^*\mathbf{M}^{-1}_T]_i$ is stored, and inner products of the vectors will be formed just as in Figure 10. The timing diagram is similar to that in Figure 7, with the value of $\mathbf{F} = \mathbf{W}^*\mathbf{X}$ being available from $\text{PE}_n$ $n+1$ time steps after $\mathbf{X}$ is input.

$$D_j = D_{j-1} + (TX)_j \cdot Q_j$$
$$Q_j = [(TS)^*\mathbf{M}^{-1}_T]_j$$
$$D_0 = 0$$
$$D_n = \sum_{j=1}^{n} Q_j (TX)_j$$
$$= (TS)^*\mathbf{M}^{-1}_T \mathbf{X} \equiv \text{OUTPUT}$$

Figure 10. Computations for forming the filter function in transform space.

This approach has two advantages over approaches described in the preceding two subsections. First, the array produces the desired output directly instead of just the weights $\mathbf{W}$. The hardware is as simple or simpler than the unit vector method implementation and much simpler than the reverse flow method. Second, it is possible to continuously update the $w_{ij}$'s without temporarily halting computation of the weights; in this way, the $w_{ij}$'s are always computed from the most recent information. This point will be discussed further in Section 3.
The disadvantage of the transform space method is that in a real system, the X's may be sampled at a far faster rate than the array is able to process them. The X's are accepted one per time step of the array (and one $F = W^\times X$ value is computed once per time step), whereas the sampling rate of the radar may be many times faster. (The unit vector method and reverse flow method do not have this disadvantage because, given $W$, $F = W^\times X$ can be calculated extremely fast.) Whether this downsampling would adversely affect the overall system performance depends on the individual system.
2.7 THE SPECIAL CASE OF A SIDE-LOBE CANCELLER

It is possible to simplify our proposed implementation in the case of a side-lobe canceller (SLC), because of the special form of the steering signal (all zeroes, except for a one on the component corresponding to the main beam). It turns out that the best way to exploit this special form is to have the one in the steering signal occur in the last \(n^{th}\) component. With this \(S\), \(TS = S\) (since \(T\) is unit lower triangular); thus, no work is required to pass \(S\) through the array.

The effect on the unit vector method is to reduce the sum defining the output,

\[
(T^*Y)_j = \sum_{k=1}^{n} T_{k,j} Y_k = \sum_{k=1}^{n} (TE^j)_k Y_k
\]

to

\[
W_j = (TE^j)_n M^{-1}_{T_{nn}}
\]

where \(M_T = \text{diag}(MT_{jj})\). Thus, only one outboard processor is needed at output port \(n\) to compute \(W_j\). The overall timing is the same.

The reverse flow method also simplifies because the vector to be passed backwards through the array is known a priori, \((0, \ldots, 0, M^{-1}_{T_{nn}})^T\), eliminating the need for shift registers and reducing the overall length of the pipe by \(n\) steps.

The transform space method becomes exceedingly simple, with only one outboard processor required at output port \(n\), and since

\[
F = (TS)^* M^{-1}_{T}(TX) = (M^{-1}_{T}TX)_n
\]

its output is simply the filter function.
3. SYSTEM IMPLEMENTATION CONSIDERATIONS

This section describes the characteristics of different versions of the basic implementation discussed in Section 2, and various radar engineering considerations and how they affect the choice of implementation. We consider different ways of computing the internal coefficients (Section 3.1); error sensitivity and fault tolerance (Section 3.2); different ways to perform arithmetic and growth bounds for intermediate results (Section 3.3); implementation with $O(n)$ processors instead of $O(n^2)$ (Section 3.4); and construction of individual processors (Section 3.5). In Section 3.6 we summarize the variety of system implementation choices discussed in Sections 3.1 through 3.5, and show how different radar engineering system parameters affect implementation choice. Finally, in Section 3.7, we select a typical set of system parameters and give a detailed design for a Gram-Schmidt processor incorporating those parameters.

3.1 CALCULATION OF INNER PRODUCTS

The algorithm of Section 2.1 requires the calculation of many inner products $\langle Z^j_i, Z^i_1 \rangle_M$. The method given, which computes the sample average $\sum_k Z^j_i(k)Z^i_1(k)$ (see Figure 5), is one of several possible methods. All the methods involve averaging over samples, as in Figure 5, but can weight different samples differently:

$$E^m = \sum_{k=1}^{m} w_k E_k$$  \hspace{1cm} (16)

Here $E_k$ is one sample value (say, $Z^j_i(k)Z^i_1(k)$), $w_k$ is its weight, and $E^m$ is the estimated average using $m$ samples. For $E^m$ to be an unbiased estimator of the true average, it is necessary that $w_k \geq 0$ and $\sum w_k = 1$. Subject to this
mild restriction, we are free to choose the \( w_k \)'s so that our estimate \( E \) has desirable statistical properties, such as being able to closely follow the true average as it changes in time without being confused by noise spikes.

It is well known that if the underlying distribution from which the \( E_k \)'s are sampled is stationary in time, then the best unbiased estimator of their mean is an equal weighting of all available samples:

\[
E^m = \frac{1}{m} \sum_{k=1}^{m} E_k .
\]  

When the underlying distribution is changing, as it will in any real system, then Eq. (17) introduces a bias, because it weights old samples as much as new ones. A good choice of \( w_k \)'s must satisfy two conditions: It must average over enough samples to compute a statistically significant result (at least \( n \) samples \( (n=\text{number of weights}) \) are required just for the matrix \( M \) to be nonsingular), and it must weight the recent values heavily enough to follow the average quickly, but not so heavily that noise spikes confuse it.

The optimal choice of an averaging method probably would vary from system to system, but there are certain schemes which work well in many situations and are worth analyzing. The methods have different implementations, depending on the rest of the system (e.g., implementations of Sections 2.6.2 and 2.6.1 versus that of Section 2.6.3). After the discussion of each method, therefore, we analyze various implementations. Finally, we present test results using actual radar data. These results allow us to compare how well the methods maximize the signal-to-noise ratio (SNR) of the system.
Block Averaging

First, we have block averaging, the method described in Section 2. In this method, the $w_{ij}$'s are periodically reinitialized to zero, data are passed through the array, and the $w_{ij}$'s are formed, giving equal weight to each product in the sum (as in Eq. (17)). This form of averaging may be accomplished using the same set of $K$ samples to compute the $w_{ij}$'s in each row, or different sets. If the same set of samples is used, buffering is required to save the data. Buffering occurs just before the inputs of the array, in order to pass the data through the array $n-1$ times. If buffering is not used, a total of $K(n-1)$ samples is required, and the method is called cascaded block averaging. The first time through is to compute the first-row averages ($\bar{w}_{ij}$ and $\bar{w}_{j}$) in the “if $k=i$” clause in Figure 5. The second pass performs the outer “else” clause of Figure 5. $Z_j^2(2) = Z_j^1(2) - w_{ij}Z_j^1(2)$ in row 1 and the “if $k=i$” clause in row 2. In general, the copy of the data input to the array between time steps $(m-1)K+1$ and $mK$ finally arrives at row $m$ and is used to compute $w_{mj}$ between time steps $(2n-2)K+1$ and $(2m-1)K$. Steering vectors $S$ (in $MW=S$) can be input starting at time $mK+1$, and all the values of $TS$ are available at time $(2n-3)K+1$.

If no buffer is to be used, then a different set of $K$ samples is input in each time period $(m-1)K+1$ to $mK$. This method, called cascaded block averaging, will compute values of $w_{ij}$ close to their true values if the underlying distribution determining the input values changes slowly and enough samples are used. Recall the result of Reed, Mallett, and Brennan [1974] that says approximately $2n$ samples are required for an expected SNR of 3 dB within optimum.

---

Exponential Averaging

Second, we have exponential averaging. In this case, we take a weighted sum of the old average and the new data:

\[ E_{m+1} = SE_{m+1} + (1-S)E_m \quad 0 < S < 1 \]  

(18)

If \( S \) is close to 1, then the new average \( E_{m+1} \) is very responsive to new data \( E_{m+1} \). If \( S \) is close to zero, \( E_{m+1} \) changes more slowly, being determined mainly by the old average. Exponential averaging can be implemented either with or without a buffer, as in block averaging, or, once the system has been started up and \( w_{ij} \) values are available, the \( w_{ij} \)'s can be updated by inputting just a few sample voltage vectors and averaging them in. This is an advantage over block averaging: With block averaging, each time the \( w_{ij} \)'s are to be updated, a lag time of \((2n-3)K\) is needed, where \( K \) is at least \( n \) and preferably \( 2n \); \( K \) may be significantly smaller with exponential averaging.

Note that it is possible either to exponentially smooth \( w_{ij} \) itself, or to smooth its numerator and denominator separately, before dividing them. (If, in addition to this separate smoothing, the denominator is approximated by its closest power of 2, the problem of doing division is reduced to shifting).

Window Averaging

Finally, we have window averaging. In this case, the last \( K \) samples are always used to compute the \( w_{ij} \)'s, thus using the most recently available information at all times. This method requires saving a buffer of the last \( K \) values used. When a new value (e.g., \( Z_i^1 \cdot Z_j^1 \)) is computed, it is added to the window average, put on top of the buffer, and the oldest value removed from the bottom of the buffer and subtracted from the average. The buffer can conveniently be implemented as a shift register. In addition to always using the most recent data, window averaging has the same advantage as ex-
ponential over block averaging: It does not always require a lag of \((2n-3)K\) samples to update the \(w_{ij}\)'s. The disadvantage, of course, is the need for a long shift register (at least \(n\) and probably \(2n\) words) for each average \((n(n-1)/2\) or \(n(n-1)\), depending on the implementation).

The transform space method of Section 2 can be used as well as any of the above methods, but its real advantage lies in the fact that since samples (in contrast to steering vectors) are constantly being passed through the array, the \(w_{ij}\)'s can be updated simultaneously with the calculation of filter functions. Hence, when exponential or window averaging is used, no delay at all (after startup) is needed to update the weights. Each filter function is always a function of the most recent information.

Reed, Mallett, and Brennan [1974] have shown that the expected value of the achieved SNR is \(10 \log_{10}[(K+2-n)/(K+1)]\) dB below the SNR achieved with optimal weights. This expected loss is about 3 dB when the number of samples, \(K\), equals \(2n\). Of course, if a great deal of clutter is present, more samples may be required to average it out. Also, Brennan [1974] has recently shown that if the weights determined by \(K\) samples are in turn used to form filter functions from those same samples instead of a different set of samples (as in nontransform space operation), then the expected SNR is actually higher (see Appendix A).

We now present some graphs of system performance versus sample size for both real and simulated data, varying numbers of weights, and varying averaging schemes (Figures 11 through 21). System performance is measured by how many decibels down the achieved SNR is from the optimal SNR:

Performance = $10 \log_{10}\left[\left(|S^*W|^2/W^*MW\right)/S*M^{-1}S\right]\]

The real data is taken from an operational 5-weight adaptive system, and the simulated data model is described in Appendix H. The true matrix $M$ for the real data is estimated by averaging over all available samples (100). The different algorithms used are Cholesky, Gram-Schmidt using blocked averaging, Gram-Schmidt using exponential averaging (for various exponential weights, i.e., the factor $S$ in Eq. (18)), and Gram-Schmidt using cascaded block averaging. In the case of simulated data, the various model parameters are the number of weights, number of jammers, location and power of the jammers, and the ratio of strongest jammer power to receiver noise power (per receiver). This last quantity is the spectral condition number of the matrix. The use of tapped delay lines (one-sample long) is also indicated.

For the real data (Figures 11 and 13) we see that the results for Cholesky and blocked G-S almost overlap. They are mathematically identical algorithms, and any small difference is attributable to roundoff error (all arithmetic is 32-bit floating point with 24-bit mantissas). Cascaded blocked G-S has essentially the same performance as blocked G-S; but exponential G-S lags behind, performance improving, but still poor as the weight decreases (which means the new samples are weighted less than the old).

For the simulated data with 5 weights (Figures 14 through 17), Cholesky, blocked G-S, and cascaded blocked G-S all have virtually identical performance, but exponential becomes poorer as the number of jammers increases (from 1 to 4).

With 10-channel simulated data using 5 jammers (Figure 18), cascaded G-S, blocked G-S, and Cholesky are again virtually identical. In fact,
Figure 11. Real data; 5 weights.
Figure 12. Real data; 5 weights.
Figure 13. Real data; 5 weights, 1 tap per channel.
**Figure 14.** Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 1
4. Location and power of jammer: (-36.9°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 50 dB.
Figure 15. Simulated data:
1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 2
4. Location and power of jammers:
   #1 (-36.9°, 5.0 dB), #2 (-11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver power = 40 dB.
Figure 16. Simulated data:
1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (-36.9°, 6.67 dB)
   #2 (-11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB.
Figure 17. Simulated data:
1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 4
4. Location and power of jammers:
   #1 (-36.9°, 7.5 dB)  
   #2 (-11.5°, 5.0 dB)  
   #3 (+11.5°, 2.5 dB)  
   #4 (+36.9°, 0 dB) 
5. Ratio of power of strongest jammer to receiver noise power = 50 dB.
Figure 18. Simulated data:
1. Number of weights = 10
2. No tapped delays
3. Number of jammers = 5
4. Location and power of jammers:
   #1 (-53.1°, 8.0 dB)
   #2 (-36.9°, 6.0 dB)
   #3 (-23.6°, 4.0 dB)
   #4 (-11.5°, 2.0 dB)
   #5 ( 0.0°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB.

58
cascaded G-S performs somewhat better, because it uses \( n-1 \) times as much data as the other methods. Again, exponential performs the poorest, with a weight of 0.25 being best in this case (Figure 19).

With 35- and 50-channel simulated data, using 10 and 20 jammers, respectively, blocked and cascaded G-S and Cholesky are again virtually identical, with exponential G-S performing poorly (Figures 20 and 21, respectively).

Seven of the nine cases achieve an SNR within 3 dB of optimal after 2n samples; the other two cases are within 5 dB.

The location of the jammers (first number in parentheses in item 4 of the simulated-data captions) is given in degrees from boresight, and, jammer power (second number in parentheses) is expressed as decibels below the strongest jammer.

We spent a great deal of effort analyzing cases with low numbers of weights because the effects of changes in the number of jammers and exponential weights show up more quickly.
Figure 19. Simulated data:

1. Number of weights = 10
2. No tapped delays
3. Number of jammers = 5
4. Location and power of jammers:
   #1 (-53.1°, 8.0 dB)
   #2 (-36.9°, 6.0 dB)
   #3 (-23.6°, 4.0 dB)
   #4 (-11.5°, 2.0 dB)
   #5 (0.0°, 0 dB)

5. Ratio of power of strongest jammer to receiver noise power = 40 dB.
Figure 20. Simulated data:

1. Number of weights = 35
2. No tapped delays
3. Number of jammers = 10
4. Location and power of jammers:
   #1 (-70.5°, 9.0 dB)   #6 (-41.1°, 4.0 dB)
   #2 (-62.3°, 8.0 dB)   #7 (-36.9°, 3.0 dB)
   #3 (-55.9°, 7.0 dB)   #8 (-32.9°, 2.0 dB)
   #4 (-50.5°, 6.0 dB)   #9 (-29.1°, 1.0 dB)
   #5 (-45.6°, 5.0 dB)   #10 (-25.4°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 27 dB.
Figure 21. Simulated data:
1. Number of weights = 50
2. No tapped delays
3. Number of jammers = 20
4. Location and power of jammers:
   #1 (-73.7°, 9.5 dB)  #11 (-34.05°, 4.5 dB)
   #2 (-66.9°, 9.0 dB)  #12 (-31.3°, 4.0 dB)
   #3 (-61.6°, 8.5 dB)  #13 (-28.7°, 3.5 dB)
   #4 (-57.1°, 8.0 dB)  #14 (-26.1°, 3.0 dB)
   #5 (-53.1°, 7.5 dB)  #15 (-23.6°, 2.5 dB)
   #6 (-49.5°, 7.0 dB)  #16 (-21.1°, 2.0 dB)
   #7 (-46.0°, 6.5 dB)  #17 (-18.7°, 1.5 dB)
   #8 (-42.8°, 6.0 dB)  #18 (-16.3°, 1.0 dB)
   #9 (-39.8°, 5.5 dB)  #19 (-13.9°, 0.5 dB)
   #10 (-36.9°, 5.0 dB) #20 (-11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 20 dB.
3.2 FAULT TOLERANCE AND NOISE SENSITIVITY

In this section we analyze the effect of the system's different failure modes on performance, as measured by the SNR.

External Failure Mode

The first category of failure mode is external, which means that although the processor array is functioning normally, something is wrong with the antenna/receiver subsystem supplying inputs. The possible failures we examine are 1) input \( j \) is zero (others normal), 2) input \( j \) is a nonzero constant (others normal), and 3) Gaussian noise of power \( \nu \) is added to input \( j \) (others normal).

When the input \( j \) is zero, the system continues to function, computing the correct weights for an \( n-1 \) weight system and a zero weight for input \( j \). To verify this phenomenon, note that all the \( w_{ij} \) values, \( 1 \leq i \leq j-1 \), are zero, because (in the notation of Figure 5) \( w_{ij} = w_{ij} + z_i^j(k) \cdot z_j^j(k) \) and \( z_j^j \) is zero, so \( w_{ij} \) is zero, \( z_j^j \) is zero, \( w_{2j} \) is zero, and so on. Thus, \( z_j^j = 0 \), and (with the odd convention that \( 1/0 = 0 \)) we have \( w_{jj} = 0 \), \( j+1 \leq i \leq n \), since \( w_{jj} \) is \( w_{jj}/\sum z_j^j \cdot z_j^j = w_{jj} \cdot (1/0) = 0 \). Thus the values computed in columns \( j+1 \) to \( n \) of the array have nothing subtracted from them in row \( j \) of the array: They are independent of input \( j \). Since the values computed in columns 1 to \( j-1 \) are obviously independent of column \( j \), all the other computed values are computed correctly as though input \( j \) were missing. Since \( 1/w_{jj} \) is taken to be zero, and the output at port \( j \) on the right of the array is multiplied by \( 1/w_{jj} \), we see the system operates as claimed for the transform space and unit vector methods, because they get a zero contribution from output \( j \) and only operate the array in the forward direction discussed above in this paragraph. To see that reverse flow also operates normally, note that the zero input at port \( j \) on the right remains zero as it moves left across the array (since the
w_ji's in row j are zero) and when it is broadcast vertically by P_j,j+1, it has no effect on the values computed by rows 1 to j-1. Since rows j+1 to n are similarly unaffected, we see all weights except j are computed normally, and weight j is zero as desired.

The optimal SNR of the n-1 weight system can be expressed in terms of the old optimal SNR with n weights, the j^{th} optimal weight w_j (when all n are available) and m_{jj}^{j} = j^{th} diagonal element of the inverse of the true n x n covariance matrix:

$$SNR_{NEWOPT} = SNR_{OLDOPT} - \frac{|w_j|^2}{m_{jj}}$$

(19)

This result will be derived later.

When input j is a nonzero constant, a similar analysis shows incorrect weights are computed. Approximately 0 values for w_{ij} in column j are computed, since Z_j is Gaussian with zero mean. Thus Z_j is constant and nonzero, and 1/w_{jj} is nonzero. The values of w_{ji} in row j are again approximately zero, but output j, even after multiplication by 1/w_{jj}, remains approximately constant and nonzero. Thus incorrect values are computed by all three methods.

To examine the case of Gaussian noise, we need some general results on the effects of perturbations in the weights on the SNR. These results are derived in Appendix B and presented below.

The SNR is given by:

$$SNR_W = \frac{|S*W|^2}{N*MW}$$

(20)

where SNR_W is the SNR achieved using W as weights. When using the optimal weights, W_{OPT} = M^{-1}S, we have

$$SNR_{OPT} = S^*W = W*MW = S*M^{-1}S$$

(21)
Suppose that $W = W_{\text{OPT}} + E$, where $E$ is an error vector. Then we may write $\text{SNR}_W$ in two equivalent forms:

$$
\text{SNR}_W = (1 - \frac{E^*M^*}{W^*W}) \cdot \text{SNR}_{\text{OPT}} + \frac{E^*M^*}{W^*W} \cdot \text{SNR}_E
$$

$$
= \text{SNR}_{\text{OPT}} - E^* \left( \frac{SS^*}{\text{SNR}_{\text{OPT}}} \right) E + \frac{E^*M^*}{W^*W} \cdot \text{SNR}_{\text{OPT}} \quad \text{(22)}
$$

The first expression for $\text{SNR}_W$ shows it is a weighted linear combination of $\text{SNR}_{\text{OPT}}$ and $\text{SNR}_E$, and the second expression shows how the degradation in SNR depends on the eigenvalues and eigenvectors of the positive semidefinite Hermitian matrix $(M - SS^*/\text{SNR}_{\text{OPT}})$. In particular, a lower bound for $\text{SNR}_W$ is

$$
\text{SNR}_W \geq \text{SNR}_{\text{OPT}} - \lambda_{\max}(M) \cdot ||E||^2 + O(||E||^3) \quad \text{(23)}
$$

which shows that the $\text{SNR}_W$ depends quadratically on $E$, and is worse if the largest eigenvalue of $M$, $\lambda_{\max}(M)$, is large. The bound on Eq. (23) can be attained if $S$ is orthogonal to the eigenvector of $M$ belonging to $\lambda_{\max}(M)$, and if $E$ is proportional to it. Physically, since eigenvectors of the covariance matrix $M$ are the expected signal of noise sources and the eigenvalue is the power of the noise source, the worst degradation in SNR should occur when the steering signal $S$ is aiming the array in a direction orthogonal to the strongest noise source but $E$ is aiming exactly in the direction of the strongest noise source.

We may also examine the effects of perturbations on arbitrary (nonoptimal) weights $W$. Suppose that instead of $W$ weights, $\hat{W} = W + E$ are used. Then the new SNR using $\hat{W}$ is

$$
\text{SNR}_{\hat{W}} = \text{SNR}_W - 2 \text{Re} \left[ \left( \frac{SS^*}{\text{SNR}_W} \right) E \right] + O(||E||^2) \quad \text{(24)}
$$
where the quadratic and higher-order terms are indicated by \( O(\|E\|^2) \).

Now the dependence on \( E \) is linear, with the effect on \( SNR_W \) being largest when \( E \) is proportional to \((SS^*-SNR_W^*\cdot M)W \) (the SNR can increase or decrease), and to a linear approximation lies in the range

\[
SNR_W \pm \frac{2SNR_W}{W^*MW} \left\| \left( M - \frac{SS^*}{SNR_W^*} \right) W \right\| \cdot \|E\| \leq SNR_W \leq \left[ SNR_W + \frac{2SNR_W}{W^*MW} \left\| \left( M - \frac{SS^*}{SNR_W^*} \right) W \right\| \cdot \|E\| \right]
\]

When \( E \) is orthogonal to \((M - SS^*/SNR_W^*)W \), the dependence of \( SNR_W \) on \( E \) is quadratic. Thus,

\[
SNR_W = \text{linear terms in Eq. (20)} + \frac{SNR_W}{W^*MW} \text{Re} \left( 4\text{Re} \left( \frac{W^*ME}{W^*MW} \right)^* - E^* \right) \left( M - \frac{SS^*}{SNR_W^*} \right) E \\
+ \text{higher-order terms in } \|E\|.
\]

When \( W = W_{\text{OPT}} \), Eq. (26) simplifies to Eq. (23).

Now let us return to the specific problems of Gaussian noise of power \( \rho \) being added to input \( j \). (The following results are also derived in Appendix B). Adding this noise is equivalent to adding \( \rho \) to the \( j \)th diagonal elements of the true covariance matrix \( M \) to get the new matrix \( \hat{M} \), \( \hat{M} = M + A \), where \( A_{k\ell} = \rho \) if \( k = \ell = j \), 0 otherwise. We may express \( \hat{M}^{-1} \) in terms of \( M^{-1} \)

\[
\hat{M}^{-1} = M^{-1} + M^{-1} \cdot \left( \frac{-\rho}{1 + \rho \cdot m_j} \right) \begin{bmatrix} 0 & m_j \cdot \ldots \cdot m_j \end{bmatrix} \cdot j^{\text{th}} \text{ row}.
\]
where \( M = \{m_{ij}\} \) and \( M^{-1} = \{m^{-1}_{ij}\} \). Thus the new weights \( \hat{w} = M^{-1}s \) can be expressed in terms of the old weights

\[
\hat{w} = w + M^{-1} \begin{bmatrix} -\rho \\ \frac{-\rho}{1 + \rho m_{jj}} \end{bmatrix} \begin{bmatrix} w_j \\ 0 \end{bmatrix} + j^{th} \text{ row} .
\]

Hence the new weights are the sum of the old weights and the \( j^{th} \) column of \( M^{-1} \) times \( -\rho w_j / (1 + \rho m_{jj}) \). In particular, the new \( j^{th} \) weight

\[
\hat{w}_j = \frac{w_j}{1 + \rho m_{jj}} .
\]

So as \( \rho \) increases, \( \hat{w}_j \) decreases, approximately in proportion to \( 1/\rho \) for \( \rho \) large. As \( \rho \) approaches infinity, \( \hat{w}_j \) approaches zero and the system operates as though there were only \( n-1 \) weights, because all the information in channel \( j \) is being obscured by the large noise.

We must now distinguish between two subcases, depending on when the noise is added to the system. Case 1 occurs when the noise is added by the antenna/receiver subsystem, so that the sample's voltage vectors used to compute the filter functions also contain noise, and case 2 occurs when the noise is added later, thus affecting the weights only and not the sample voltages for the filter function. In the first case, where the noise is present from the beginning, it is as though the true covariance matrix of the noise process changes (by adding \( \rho \) to the diagonal); in the second case, the true covariance matrix remains the same, but the weights change.

In the first case, we can compute the optimal SNR of the new system with noise \( \text{SNR}_{\text{NEWOPT}} = S \times M^{-1}S \) using Eq. (23):

\[
\text{SNR}_{\text{NEWOPT}} = \text{SNR}_{\text{OLD OPT}} - \left[ \rho |w_j|^2 / (1 + \rho m_{jj}) \right] .
\]
Thus, the optimal SNR achievable with noise $\rho$ added varies as shown in Figure 22. For $\rho$ small it decreases approximately linearly with a slope of $-|w_j|^2$. If $|w_j|$ is large, then system performance depends heavily on the $j$th channel; hence, adding noise to that channel is particularly bad. Also, if $m_j$ is small, then the lower bound for $SNR_{NEWOPT}$,

$$SNR_{OLDOPT} - \left(\frac{|w_j|^2}{m_j}\right),$$

is small.

$$SNR_{NEWOPT} = \frac{|w_j|^2}{m_j}$$

In fact, if $S = e_j = \{0, \ldots, 0, 1, 0, \ldots, 0\} (1 \text{ in } j^{th} \text{ place})$, then $SNR_{OPT} = m_j^j$ and $w_j = m_j^j$. Then $SNR_{NEWOPT} = 1/(\rho + 1/\text{SNR}_{OLDOPT})$, which approaches 0 as $\rho$ approaches infinity.

It is possible to make a physical interpretation of $m_j^j$ being small. Mathematically, it means the covariance matrix $M^{n \times M}$ has an eigenvector approximately equal to $e_j = \{0, \ldots, 0, 1, 0, \ldots, 0\} (1 \text{ in } j^{th} \text{ entry})$ whose
corresponding eigenvalue is large, which the matrix $M_{n-1} = M$ with $j^{th}$ row and $j^{th}$ column removed does not have. Physically, this means the $n$ weight system with matrix $M_n$ is capable of adapting to a strong noise source, with steering signal $e_j$, whereas the $n-1$ weight system with matrix $M_{n-1}$ cannot. Since

$$m_{jj} = \frac{n-1}{\prod_{j=1}^{n-1} \lambda_j(M_{n-1})} = \frac{n}{\prod_{j=1}^{n} \lambda_j(M_n)}$$

($\lambda_j(x) = j^{th}$ eigenvalue of matrix $x$), $m_{jj}$ will be small if this relationship of eigenvalues and vectors of $M_n$ and $M_{n-1}$ holds. In other words, $m_{jj}$ is small if the $j^{th}$ channel is important for the system to be able to adapt to some large noise source. The system will then be especially sensitive to noise in that channel.

In the second case, where noise is added late so that it affects the weight computations but not the true covariance matrix $M$, we may use Eqs. (22), (24), and (25) to analyze system performance. Assuming that the weights used after perturbations are the optimal weights for the noise field seen by the array, i.e., $\hat{W} = \hat{M}^{-1}S$, we may use Eqs. (21) and (26) to compute the new $SNR^c_W$:

$$SNR^c_W = SNR^{OPT} - \rho^2 |w_j|^2 m_{jj}^{1/2} \left[ SNR^{OPT}(1 + \rho m_{jj})^2 - \rho^2 |w_j|^2 (2 + \rho m_{jj}) \right] \left[ SNR^{OPT} - |w_j|^2/m_{jj} \right].$$

(30)

Note that $|w_j|^2/m_{jj} = SNR_E$, where $E = \hat{W} - \hat{W}^{OPT}$. $SNR^c_W$ is a decreasing function of $\rho$, with a lower bound of

$$SNR^c_W \geq SNR^{OPT} - \left( |w_j|^2/m_{jj} \right).$$

(31)
which is the same lower bound as for SNR\textsubscript{NEWOPT}, given in Eq. (25).
Thus, if an infinite amount of noise is added to channel \(j\), effectively turning the system into an \(n-1\) weight system, the degradation in SNR can be bounded independent of where in the system the noise is added. In both cases, a large \(j^{th}\) weight \(|w_j|\) or a small \(m_{jj}\) indicates the system is particularly sensitive to noise in that channel. This analysis also proves Eq. (16).

We now present some simulation results of adding noise to a 5-weight system with 3 jammers. Figure 23 is a control case with no noise added. The total noise power (summed over all inputs to which noise is added) is 4 dB above the strongest jammer. These results correspond to case 2 of the above discussion, where the noise affects the weight calculation only.

Apparently, inputs 2 and 3 (Figures 25 and 26) are very insensitive to noise; inputs 1 and 4 (Figures 24 and 27) are somewhat sensitive; and input 5 (Figure 28) is very sensitive. This sensitivity was determined by looking at the steady-state cancellation for either G-S or Cholesky (i.e., performance degradation for 20 samples). Since the steering signal for these latter simulations is \((0,0,0,0,1)\), i.e., an SLC with channel 5 as the main, channel 5 should, by far, be the most sensitive. The other inputs are not as sensitive. Since there are only 3 jammers, as long as only one input is affected enough degrees of freedom remain to effectively cancel the jammers (unless, of course, channel 5 is affected). When 3, 4, or 5 channels are affected (as in the last three plots, Figures 29 through 31), so that there are not enough degrees of freedom to cancel the jammers and the extra noise, performance degrades markedly.
Figure 23. Simulated data:
1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (-36.9°, 6.67 dB)
   #2 (-11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. No noise added.
Figure 24. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (--36.9°, 6.67 dB)
   #2 (--11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channel 1 of total power 4 dB above strongest jammer.
Figure 25. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (-36.9°, 6.67 dB)
   #2 (-11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB.
6. Noise added to channel 2 of total power 4 dB above the strongest jammer.
Figure 26. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (-36.9°, 6.67 dB)
   #2 (-11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channel 3 of total power 4 dB above strongest jammer.
Figure 27. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   - #1 (-36.9°, 6.67 dB)
   - #2 (-115.6°, 3.33 dB)
   - #3 (+115.6°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channel 4 of total power 4 dB above strongest jammer.
Figure 28. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (-36.9°, 6.67 dB)
   #2 (-11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channel 5 of total power 4 dB above strongest jammer.
Figure 29. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   - #1 (-36.9°, 6.67 dB)
   - #2 (-11.5°, 3.33 dB)
   - #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channels 1, 2, 3 of total power 4 dB above strongest jammer.
Figure 30. Simulated data:
1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   #1 (-36.9°, 6.67 dB)
   #2 (-11.5°, 3.33 dB)
   #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channels 1, 2, 3, and 4 of total power 4 dB above strongest jammer.
Figure 31. Simulated data:

1. Number of weights = 5
2. No tapped delays
3. Number of jammers = 3
4. Location and power of jammers:
   - #1 (-36.9°, 6.67 dB)
   - #2 (-11.5°, 3.33 dB)
   - #3 (+11.5°, 0 dB)
5. Ratio of power of strongest jammer to receiver noise power = 40 dB
6. Noise added to channels 1, 2, 3, 4, and 5 of total power 4 dB above strongest jammer.
Internal Failure Mode

In the internal failure mode, processor $P_{ij}$ outputs only constants. This case is similar to the external failure mode where input $j$ is constant.

Consider first the case where the output of $P_{ij}$ is identically zero. Outputs $1$ to $j-1$ remain unaffected. As in the external case, all the $w_{kj}$'s and outputs of $P_{kj}$ for $k > i$ are zero; so, $w_{jj}$ is zero, and the rest of the array (rows $j$ to $n-1$) operate as though the zero had been input at the very top of column $j$, and we have normal $n-1$ weight operation. The reverse flow situation is similar, so that if $P_{ij}$ fails by outputting only zero, it is equivalent to the external failure situation, where the array operates correctly as an $n-1$ weight system. When $P_{ij}$ puts out a nonzero constant, incorrect weights are calculated, as with the external failure mode.
3.3 GROWTH OF INTERMEDIATE RESULTS

In this section, we derive probabilistic bounds on the growth of intermediate results during the computations.

We first consider the algorithm without square roots and using block averaging. We further subdivide the case into the situation where 1) the actual covariance matrix $M$ is known (Figure 4a) and 2) where only samples are known (Figure 5a). When $M$ is known, we have the following bounds on the $Z_j^k(k)$'s:

$$\lambda_{\text{min}} \leq Z_j^k(k) \leq m_{\text{max}} \quad (32a)$$

$$|Z_j^k(k)| < 2 \cdot m_{\text{max}}; \text{sharp within a factor of 4} \quad (32b)$$

$$|Z_j^k(k)| \leq m_{\text{max}}; \text{sharp within a factor of 2} \quad (32c)$$

$$|w_{ij}| \leq \sqrt{\frac{m_{\text{max}}}{\lambda_{\text{min}}}}; \text{sharp within a factor of 2} \quad (32d)$$

where

$$m_{\text{max}} = \max_{ij} |m_{ij}| \quad (M = \{m_{ij}\})$$

$$\lambda_{\text{min}} = \text{smallest eigenvalue of } M, \text{ and}$$

$$\lambda_{\text{max}} = \text{largest eigenvalue of } M.$$ 

The value $m_{\text{max}}$ can be bounded simply by using the number of bits and normalization used for each sample, and the number of samples used. If the largest sample value is $X$ (in absolute value), and $S$ samples are used, $m_{\text{max}} \leq SX^2$. Note that the sample matrix is $S$ times as large as the true matrix ($S = \text{number of samples}$); because we do not divide the inner products, we calculate by $S$. The $m_{\text{max}}$ used here is the maximum possible entry of the sample matrix and is $S$ times as large as the maximum possible entry of the actual matrix.
Figure 32 plots the number of bits required to represent $\sum_{r=1}^{S} |x_i|^2$ where $|X| \leq 127$ (8-bit 2's complement representation). This plot puts an upper bound on the number of bits required to represent inner products. The simplest a priori bound on $\lambda_{\text{max}}$ is $n \cdot m_{\text{max}}$; once the matrix is computed, the norms $||M||_1$, $||M||_\infty$, and $\text{tr}(M)$ (trace $M$) (see Isaacson and Keller [1966]) also provide upper bounds to $\lambda_{\text{max}}$. A lower bound for $\lambda_{\text{min}}$ is provided by the level of receiver noise. Typically, ADC's (analog-to-digital converters, used to digitize the voltage inputs) are adjusted so that the receiver noise causes the least significant bit of the converted signal to be random; the receiver noise is therefore close to $S^2 t^2 x^2$, where $t$ is the number of bits used in the ADC (this approximation may be off by a system-dependent factor close to 1). Thus $|w_{ij}| \leq \sqrt{S^2/2} - 2t^2 = 2t$.

When using samples to compute the $w_{ij}$'s, we use the statistical interpretation of the receiver inputs being complex Gaussian random variables with zero mean and covariance matrix $M$. The intermediate quantities $z_{ij}^{(k)}$ are then also Gaussian random variables with zero means and computable variances. The quantity $\tilde{w}_{ij}$ has a complicated distribution, but for many samples we can approximate it by a Gaussian distribution, and $\tilde{w}_{ij}$ is similarly very close to a $\chi^2$ variable. The quotient of the two quantities, $\tilde{W}_{ij} = \tilde{w}_{ij}/\tilde{w}_{ij}$, also has a complicated distribution, and for this analysis we replace it by its expected value. The results are as follows:

$Z_{ij}^{(k)}$ is a Gaussian random variable with zero means and variance $\text{var}[Z_{ij}^{(k)}] = Z_{ij}^{(j)}$, where $Z_{ij}^{(j)}$ is computed as though the actual

---

Figure 32. Number of bits required to represent inner products.
sample covariance matrix were being passed through. Thus
\[ \lambda_{\text{min}} \leq \text{var}[Z_j^i(k)] \leq m_{\text{max}}, \]
using the results of Eq. (32).

2. \( \tilde{w}_{ij} \) is approximately Gaussian with mean \( E(\tilde{w}_{ij}) = Z_j^i(i) \) (again as though the actual matrix were being passed);
so \( |E(\tilde{w}_{ij})| \leq m_{\text{max}} \), and variance \( \text{var}(\tilde{w}_{ij}) \leq 2 \cdot m_{\text{max}}^2 / S \) (\( S \) = number of samples).

3. \( \tilde{w}_{ij} \) is approximately \( \chi^2 \) with \( S \) degrees of freedom, a mean of \( E(\tilde{w}_{ij}) = Z_j^i(i) \); so \( |E(\tilde{w}_{ij})| \leq m_{\text{max}} \) and variance
\[ \text{var}(\tilde{w}_{ij}) \leq 2 \cdot m_{\text{max}}^2 / S. \]

Thus, it is a simple matter to calculate the probability of overflow of the \( Z_j^i(k) \)'s as a function of the number of bits, \( b \), used to represent the \( Z_j^i(k) \)'s in fixed point (where \( N(0,1) \) is a standard normal random variable):

\[ \text{Probability of overflow} \leq P\left(|N(0,1)| > 2^{b-\log_2 m_{\text{max}}} \right). \]

If we let \( C = b - \log_2 m_{\text{max}} \) = number of extra bits used beyond those needed to represent \( \sqrt{m_{\text{max}}} \), we can make the following table:

<table>
<thead>
<tr>
<th>( C )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability of overflow</td>
<td>\leq 31.8%</td>
<td>\leq 4.55%</td>
<td>\leq 6.34 \times 10^{-3}%</td>
<td>\leq 1.24 \times 10^{-13}%</td>
</tr>
</tbody>
</table>

We see that as \( C \) increases, the chances for overflow become negligible quickly.

Similarly, if \( C \) is the number of extra bits beyond those needed to represent \( m_{\text{max}} \), then the probability of overflow when representing \( \tilde{w}_{ij} \) is

\[ \text{Probability of overflow} \leq P\left(|N(0,1)| > \sqrt{\frac{C}{2}} \right). \]

The following table evaluates the above function for various values of \( C \) and \( S \).

84
Again, the probability of overflow becomes negligible quickly.

If C is the number of extra bits needed beyond those to represent $m_{\text{max}}$, then for the probability of overflow when computing $w_{ij}$ to be $< 10^{-4}$, we need the following number of extra bits:

<table>
<thead>
<tr>
<th>C</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.0%</td>
<td>3.6%</td>
<td>7.42x10^{-5}%</td>
</tr>
<tr>
<td>2</td>
<td>31.7%</td>
<td>0.27%</td>
<td>2.58x10^{-10}%</td>
</tr>
<tr>
<td>4</td>
<td>15.7%</td>
<td>2.14x10^{-3}%</td>
<td>-2.10^{-21}%</td>
</tr>
<tr>
<td>32</td>
<td>6.33x10^{-3}%</td>
<td>3.56x10^{-31}%</td>
<td>-10^{-170}%</td>
</tr>
<tr>
<td>400</td>
<td>1.59x10^{-42}%</td>
<td>-10^{-383}%</td>
<td>-10^{-2171}%</td>
</tr>
</tbody>
</table>

This may be derived using $[P(w_{ij} > 2^C m_{\text{max}}) = P(x_S^2 > 2^{CS})]$.

Finally, when passing a steering vector through the array, it must be remembered that if $S$ is an eigenvector of $M$ corresponding to $\lambda_{\text{min}}$, then $W = M^{-1}S = (1/\lambda_{\text{min}})S$, so that quantities on the order of $1/\lambda_{\text{min}}$ must be introduced in the computation. These numbers of magnitudes potentially much larger than those already discussed can be introduced either during division by $<z_{i,1}^1, z_{i,1}^j>$ at the right of the array or during back substitution. Thus the unit-vector and transform-space methods might have an advantage over the reverse flow method because they do not require passing numbers as large.
as $1/\lambda_{\min}$ back through the array. The unit vector method, of course, passes
numbers as large as $\left( m_{\max}/\lambda_{\min} \right)_i^j = \max |w_{ij}|$ through the array. Still, the
transform space method and unit vector method might be able to use the simpler
fixed-point arithmetic in the triangular array and then only have floating
points in the outboard processors.

The analysis of the method with square roots is very similar to the
above analysis. The $Z_{ij}^i(k)'s$, $\tilde{w}_{ij}$, and $\tilde{w}_{ii}$ have the same distributions as
before, but the $1/\sqrt{\tilde{w}_{ii}}$ quantity broadcast is new. It is easy to derive its
distribution from that of $\tilde{w}_{ij}$. In fact, since $\tilde{w}_{ii}$ is $x^2$ with $S$ degrees of
freedom and mean between $\lambda_{\min}$ and $m_{\max}$, we have the probability that the most
significant digit of $1/\sqrt{\tilde{w}_{ii}}$ is more than $C$ bits to the right of the most
significant bit of $1/\sqrt{m_{\max}}$ is

$$P\left( 1/\sqrt{\tilde{w}_{ii}} \leq 2^{-C} \cdot 1/\sqrt{m_{\max}} \right) \leq P\left( x_S^2 \geq S^2 2^C \right),$$

which is very small, as can be seen from the last table. Also, the probability
that $1/\sqrt{\tilde{w}_{ii}}$ will need more than $C$ bits more than $1/\sqrt{\lambda_{\min}}$ is

$$P\left( 1/\sqrt{\tilde{w}_{ii}} \geq 2^C \cdot 2^{t}/\sqrt{m_{\max}} \right) \leq P\left( x_S^2 \leq 2^{-2C_S} \right),$$

which is also very small for $C=1$ and $S \geq 20$ (probability $\leq 2.8 \times 10^{-2}$).

The results presented in this section are proved in Appendix G.
3.4 GRAM-SCHMIDT PROCESSING WITH O(n) PROCESSORS

The Gram-Schmidt array with O(n^2) processors can also be implemented with only O(n) processors. This implementation can be performed by collapsing the array in the vertical, horizontal, or diagonal dimension (Figure 33); i.e., letting the work of all processors in a column, in a row, or along a diagonal be performed by a single processor.

Each processor performs the same calculations as the processors in the O(n^2) array but on multiple data sets. Three main differences between the three O(n) implementations are: 1) work is not equally shared by the processors; 2) they have different internal-memory requirements; and 3) the output may be routed internally to the same processor or externally to another processor. All of these will be discussed in more detail. The processor diagram for O(n) processors is shown in Figure 34.

As can be seen in Figure 33 for the case of four inputs, one processor must perform the work of from one to three processors of the original O(n^2) implementation. This mismatch is increased as n increases. If W is the amount of work that one processor in an O(n^2) array must perform, then the amount of work for a processor in an O(n) configuration can range from W to (n-1)W. Since the total time in a parallel system is determined by the slowest processor, then, on the average, half of the processors are idle.

The internal memory for each processor must be increased because it must store all of the information required for the calculations to be performed. This memory increase should, however, be small, mainly the inputs and internal weights, which amount to approximately 4(n-1) complex words.

In an O(n^2) array, each processor is independent of its position. In an O(n) array, however, the processors must be "aware" of the interconnect
a. Vertical collapsing.

b. Horizontal collapsing.

c. Diagonal collapsing.

Figure 33. Collapsing an array into $O(n)$ processors. The boxes represent work performed by a processor.
a. Standard configuration.

b. Alternative configuration for symmetry.

Figure 34. Configuration using $O(n)$ processors.
structure so that they will know where to transfer the processor data. The controller can handle this requirement by specifying the routing and, as Figure 34 shows, all routing is regular. The routing problem is not serious and should not prevent consideration of this method.

3.4.1 Advantages and Disadvantages

The obvious disadvantage of \( O(n) \) processors is speed. The actual timing models are developed in Appendix E, but time is a function of many variables. In a block average system, most of the time is spent calculating the internal weights; in this case, an \( O(n) \) system is comparable to an \( O(n^2) \) system (in actual practice, \( O(n) \) and \( O(n^2) \) systems can be equal). The \( O(n^2) \) system is advantageous when there are a large number of steering vectors. The determination of which system to use should be based on the relationship between the number of samples, \( S \), and the number of steering vectors, \( K \).

Fault tolerance and reliability offer potential advantages and disadvantages. The \( O(n) \) system can be much more reliable because of fewer parts; however, because of the increased demand on each processor, faster hardware operating in critical conditions may be required.

3.4.2 Tradeoffs Between the Three \( O(n) \) Designs

Which system, from Figure 33, should be used in a radar installation depends on many factors. We discuss briefly the major tradeoffs for each array.

Since Section 3.2 showed that if any column of the array became error-prone or stopped working, the system could still operate, the system in Figure 33a would be chosen for fault tolerance because one processor corresponds to one column. In the system in Figure 33b, if the first processor went bad, the entire system would become ineffective which is also true of that in Figure 33c.
Diagonal collapsing, Figure 33c, has the advantage of being able to implement reverse flow without requiring backward busses. This implementation is accomplished by transposing the internal weights and using forward flow. The transposition is simple because each processor contains all the internal weights for the transposition (Figure 35). This transposition is more difficult for the other configurations, but reverse-flow busses are cheaper to implement for $O(n)$ systems because of fewer processors.
Figure 35. Transposing internal weights.

Transpose $i + n + 2 - j$

$j + n + 2 - i$
3.5 PROCESSOR INTERNAL PIPELINE/PARALLELISM

Each processor must compute the inner product of its two inputs in order to compute the internal weights. Once the internal weight has been computed, it is applied against the processor inputs to compute the outputs. In this section we discuss different methods of implementing these computations. Block averaging is assumed. Complex multiplication is performed with 4 multiplies and 2 adds. Complex addition is performed with 2 adds.

3.5.1 Compute Inner Product

The mathematical formula for calculating the numerator to determine the weights is

\[ \sum_{i=1}^{S} X^* Y \]

where \( X \) and \( Y \) are input samples and \( S \) is the number of samples. Typically \( S \geq 2n \), where \( n \) is the number of adaptive weights. The primitive operations consist of multiply and add/sub. Let the time per operation be \( t \). Input time is assumed to be \( 1/2t \).

The techniques examined are:

- Sequential
- Sequential with overlap
- Full pipeline
- Parallel/2 multipliers
- Parallel/4 multipliers
- Pipeline/2 parallel multipliers
- Pipeline/4 parallel multipliers
- Optimal

Table 1 provides a comparison of the numerators computed by the above implementation techniques.
## TABLE 1. COMPARISON OF INNER-PRODUCT NUMERATOR IMPLEMENTATION TECHNIQUES

<table>
<thead>
<tr>
<th>Technique</th>
<th>No. of Multipliers</th>
<th>No. of Adders</th>
<th>Time for S Samples</th>
<th>Time Between Samples</th>
<th>Increase in Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1</td>
<td>1</td>
<td>10St</td>
<td>10t</td>
<td></td>
</tr>
<tr>
<td>Sequential with Overlap</td>
<td>1</td>
<td>1</td>
<td>(6S + 2)t</td>
<td>6t</td>
<td></td>
</tr>
<tr>
<td>Full Pipeline</td>
<td>4</td>
<td>4</td>
<td>10t + (S-1)t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>Parallel/2 Multipliers</td>
<td>2</td>
<td>1</td>
<td>8St</td>
<td>8t</td>
<td></td>
</tr>
<tr>
<td>Parallel/4 Multipliers</td>
<td>4</td>
<td>4</td>
<td>7St</td>
<td>7t</td>
<td></td>
</tr>
<tr>
<td>Pipeline/2 Parallel Multipliers</td>
<td>4</td>
<td>4</td>
<td>8t + (S-1)t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>Pipeline/4 Parallel Multipliers</td>
<td>4</td>
<td>4</td>
<td>5t + (S-1)t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>Optimal</td>
<td>4</td>
<td>4</td>
<td>4t + (S-1)t</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>
Sequential

In this case we have one multiplier and one adder with no overlap:

read X real
read X imaginary
read Y real
read Y imaginary
multiply $X_R Y_R$
multiply $X_I Y_I$
add
add add to running sum real
multiply $X_R Y_I$
multiply $X_I Y_R$
add
add add to running sum imaginary

Number of multipliers: 1
Number of adders: 1
Time for 5 samples: 10 $St$
Time between samples: 10 $t$

Sequential with Overlap

This is a slight modification of the sequential method, but the adder and multiplier are separate functional units, which enables both to be used simultaneously:

read X real add $X_I Y_R + X_R Y_I$ from previous sample
read X imaginary
read Y real add to running sum imaginary
read Y imaginary
multiply $X_R Y_R$
multiply $X_I Y_I$
multiply $X_R Y_I$ add $X_R Y_R + X_I Y_I$
multiply $X_I Y_R$ add to running sum real

Number of multipliers: 1
Number of adders: 1
Time for 5 samples: $(6S + 2)t$
Time between samples: 6 $t$
Full Pipeline

This method is similar to sequential, except that each stage is implemented so that multiple data streams can be executed simultaneously. We must also change the I/O either by transferring 2 words in parallel on each bus or making the bus operate at time $1/2t$. We chose the latter for this analysis.

read $X$ real
read $X$ imaginary
read $Y$ real
read $Y$ imaginary
multiply $X_R Y_R$
multiply $X_I Y_I$
add
add
multiply $X_R Y_I$
multiply $X_I Y_R$
add
add

No. of multipliers: 4
No. of adders: 4
Time for $S$ samples: $10t + (S-1)t$
Time between samples: $t$

Parallel/2 Multipliers

Since complex multiply can be done in parallel, we connect two multipliers in parallel:

read $X_R$
read $X_I$
read $Y_R$
read $Y_I$
multiply
add
add
multiply
add
add

No. of multipliers: 2
No. of adders: 1
Time for $S$ samples: $8St$
Time between samples: $8t$
Parallel/4 Multipliers

The next modification is to implement a 4-word-wide multiplier and a 2-word-wide adder.

read XR  
read XI  
read YR  
read YI  
multiply  
add  
multiply  
add  

No. of multipliers: 4  
No. of adders: 4  
Time for S samples: 7St  
Time between samples: 7t

Pipeline/2 Parallel Multipliers

This is the pipeline method with 2 parallel multipliers at the multiplier stages:

read XR  
read XI  
read YR  
read YI  
multiply  
add  
multiply  
add  

No. of multipliers: 4  
No. of adders: 4  
Time for S samples: 8t + (S-1)t  
Time between samples: t

Note that in this case the number of multipliers has not been increased, and time saved is only 2t, both with added processor complexity.
Pipeline/4 Parallel Multipliers

This method has a full complex multiplier (4 multipliers) and complex adder (2 adders) implemented in parallel.

\[
\begin{align*}
&\text{read } X_R \\
&\text{read } X_I \\
&\text{read } Y_R \\
&\text{read } Y_I \\
&multiply \\
&\text{add} \\
&\text{add}
\end{align*}
\]

No. of multipliers: 4  
No. of adders: 4  
Time for \( S \) samples: \( 5t + (S-1)t \)  
Time between samples: \( t \)

For the large additional complexity in the processor, this method is only \( 5t \) faster than full pipelined and \( 3t \) faster than the pipeline/2 parallel multiplier method.

Optimal

The parallel and pipeline methods explained above can take advantage of overlap simultaneously, as in the sequential method. The minimum system would use overlap in a full parallel sense. This system would look as follows for time slice \( t_i \):

\[
\begin{align*}
&\text{read } X_R(t_i) \\
&\text{read } Y_R(t_i) \\
&multiply \text{ for } t_{i-1} \\
&\text{add} \text{ for } t_{i-2} \\
&\text{add} \text{ for } t_{i-3} \\
&\text{read } X_I(t_i) \\
&\text{read } Y_I(t_i)
\end{align*}
\]

No. of multipliers: 4  
No. of adders: 4
Time for $S$ samples: $4t + (S-1)t$
Time between samples: $t$

The time for $S$ samples is only a savings of $t$ over the previous method.

A graphical comparison of the times required by all the above methods versus the number of samples is given in Figure 36.

3.5.2 Calculate Weights

In Section 3.5.1 we discuss the calculation of the numerator to determine the weights. We now discuss the calculation of the denominator and the division.

The denominator is calculated using the formula $\sum_{i=1}^{S} x_i^2 x_i$. Since $x_i$ times its conjugate is a real number, only the real part needs to be computed. This calculation can be performed either in parallel with the computation of the numerator or in sequential order. We have the following table for parallel computation (Table 2). The effective rate shown in Table 1 is not always attained. If the denominator calculation is carried out in parallel with the numerator calculation by separate processors, then the sampling rate will be governed by the slowest of the two processes because the data will be placed on the bus once for all processors. In the case of the sequential processing we have a time between samples of $10t$ and $5t$ for the denominator and numerator calculations, respectively. The actual system time between samples will therefore be $10t$. Because of this dependency on the numerator calculation, we can use the same processor to compute the denominator without serious time penalties. If the denominator is calculated sequentially in the same processor as the numerator, time $t$ can be saved on inputting the value $x$. 
<table>
<thead>
<tr>
<th>Technique</th>
<th>No. of Multipliers</th>
<th>No. of Adders</th>
<th>Time for S Samples</th>
<th>Time Between Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1</td>
<td>1</td>
<td>5St</td>
<td>5t</td>
</tr>
<tr>
<td>Sequential with Overlap</td>
<td>1</td>
<td>1</td>
<td>(2S + 2)t</td>
<td>2t</td>
</tr>
<tr>
<td>Full Pipeline</td>
<td>2</td>
<td>2</td>
<td>5t + (S-1)t</td>
<td>t</td>
</tr>
<tr>
<td>Parallel/2 Multipliers</td>
<td>2</td>
<td>1</td>
<td>4St</td>
<td>4t</td>
</tr>
<tr>
<td>Parallel/4 Multipliers</td>
<td>Not Applicable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipeline/2 Parallel</td>
<td>2</td>
<td>2</td>
<td>4t + (S-1)t</td>
<td>t</td>
</tr>
<tr>
<td>Multipliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipeline/4 Parallel</td>
<td>Not Applicable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multipliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimal</td>
<td>2</td>
<td>2</td>
<td>4t + (S-1)t</td>
<td>t</td>
</tr>
</tbody>
</table>
The division can be performed by reciprocal calculation and broadcasting the value in the horizontal direction. Assuming no overlap, the time to compute the reciprocal is the time required to perform one table lookup and two iterations of Newton-Raphson (see Appendix D). The program to find $1/C$ would be as follows:

<table>
<thead>
<tr>
<th>Table lookup</th>
<th>$X_1$ estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>$CX_1$</td>
</tr>
<tr>
<td>Sub</td>
<td>$2-CX_1$</td>
</tr>
<tr>
<td>Multiply</td>
<td>$(2-CX_1)X_1$</td>
</tr>
<tr>
<td>Multiply</td>
<td>$CX_2$</td>
</tr>
<tr>
<td>Sub</td>
<td>$2-CX_2$</td>
</tr>
<tr>
<td>Multiply</td>
<td>$(2-CX_2)X_2$</td>
</tr>
</tbody>
</table>

$X_2$ new estimate

$X_3 = 1/C$

Time = $7t$

To broadcast the value and multiply by the reciprocal in each processor requires the following code in each processor:

Read reciprocal
Multiply

Time = $2t$

3.5.3 Apply Weight to Input Data

The formula to apply weights, $w$, is output = $X - WY$. This is the same form as the inner-product calculation, which can be expressed as

\[ \text{sum} = \text{sum} + XY \]

Because of this similarity the programs are similar, as shown by the sequential method:

read $X_R$
read $X_I$
read $Y_R$
read $Y_I$
multiply \( W_R Y_R \)
multiply \( W_I Y_I \)
sub
\[ \text{sub} \quad X_R - (W_R Y_R - W_I Y_I) \]
multiply \( W_R Y_I \)
multiply \( W_I Y_R \)
add
\[ \text{add} \quad X_I - (W_R Y_I + W_I Y_R) \]
out \( X_R \)
out \( X_I \)

Assuming output takes time \( 1/2t \), all of the previous formulas are applicable if a step of \( t \) is added; i.e., \( 8t + (S-1)t \) becomes
\[ 8t + (S-1)t + t = 9t + (S-1)t. \]
3.6 DESIGN ALTERNATIVES AND TRADEOFFS

This section summarizes the variety of system configurations described in detail in the first part of this report (Sections 2.6 and 3.1 through 3.5), and indicates how various radar engineering considerations might influence the choice of configuration. Choosing the best configuration is a complicated task and depends heavily on the details of a given radar system; therefore, we only summarize some of the tradeoffs. In particular, we conclude that while a universal element may exist, it might be far from optimal for any individual system, since it would have to be designed for all possible worst-case situations, and hence be very expensive, large, and power-consuming.

The different design alternatives are summarized in Table 3. (For more details, see the indicated sections.)

The unit vector method requires simpler hardware than the reverse flow method, but is slower, solving $m \times n \times n$ systems in time $O(mn)$ instead of $O(m+n)$. The transform space method requires the simplest hardware, but every sample from which filter functions are to be computed must be passed through the array, which means there can be fewer filter function evaluations per unit time than with the other two methods.

The different means of calculating the weights not only require different amounts of hardware (most for window averaging, least for cascaded block) but have different statistical properties, requiring different numbers of samples to converge or update old estimates, and with different lag times.

The higher performance desired, or the smaller probability of overflow desired, the more bits are needed; and depending on the basic implementation chosen, it may be possible to have fewer bits in one part of the processor...
<table>
<thead>
<tr>
<th>1) Basic Implementation (See Section 2.6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Unit Vector Method</td>
</tr>
<tr>
<td>b) Reverse Flow Method</td>
</tr>
<tr>
<td>c) Transform Space Method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2) How to Compute Inner Products (See Section 3.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Block Averaging</td>
</tr>
<tr>
<td>b) Cascaded Block Averaging</td>
</tr>
<tr>
<td>c) Exponential Averaging</td>
</tr>
<tr>
<td>d) Window Averaging</td>
</tr>
<tr>
<td>e) Number of Samples</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3) Arithmetic Used (See Section 3.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Fixed, Floating Point, Block Floating Point</td>
</tr>
<tr>
<td>b) Number of Bits</td>
</tr>
<tr>
<td>c) With or Without Square Roots</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4) n Versus n² Processor Implementation (See Section 3.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Horizontal, Vertical or Diagonal Collapsing</td>
</tr>
<tr>
<td>b) Unit Vector, Reverse Flow, or Transform Space Method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5) How to Implement Each Individual Processor (See Section 3.5)</th>
</tr>
</thead>
</table>
than another, and even have different kinds of arithmetic (fixed versus floating point) in different parts of the processor.

The choice between $n$ and $n^2$ processors depends on the speed desired and cost considerations. The different $O(n)$ implementations have different fault-tolerance properties, require processors of differing complexities, and differ in how compatible they are with the three basic implementations.

How to implement each individual processor also depends on the speed required and cost limits, and is certainly dependent on the number of bits and type of arithmetic chosen.

Some of the most important radar engineering factors affecting the choice of system configuration are given in Table 4. We discuss how they affect the choice, but remind the reader that the list is not meant to be exhaustive, nor is there necessarily a single-best configuration for all operating modes of a given radar system.

**TABLE 4. RADAR ENGINEERING CONSIDERATIONS AFFECTING CHOICE OF SYSTEM CONFIGURATION**

1) Number of Weights  
2) Sampling Rate  
3) Electronic Versus Mechanical Scan  
4) Operating Modes  
5) Size of a PRI; Amount of Dead Time  
6) Fully Adaptive Versus SLC  
7) Number of Bits in the ADC  
8) Amount of Clutter  
9) Performance Required (in Terms of SNR Achieved)  
10) Cost Requirements  
11) Reliability Requirements  
12) Environment
How fast the array must process depends on the number of weights, sampling rate, type of scanning used, system mode, and the amount of dead time. A higher sampling rate requires either a faster array or downsampling. If the sampling rate is slow enough, the transform space method might be sufficiently fast. If there are many steering signals for a given covariance matrix (e.g., electronic scan), then reverse flow might be preferable to the unit vector method. If there is a small sampling window and large amount of dead time in each PRI, simple block averaging, with its long startup, might be sufficient instead of a more expensive implementation like window averaging. If overall speed requirements are low, n processors instead of $n^2$ might suffice.

If the system is a SLC instead of fully adaptive, some of the arithmetic can be simplified.

More bits on the ADC means more expensive hardware to retain accuracy. If there is so much clutter in the system that there are not enough weights to adapt to it, then more than the usual 2n samples may be required to get good performance. In general, the higher performance desired, the more bits of accuracy and the more samples required.

The many design alternatives discussed have widely varying costs ($n$ versus $n^2$ processors, and number of bits carried for example), so there are many speed/accuracy/performance versus cost tradeoffs.

The different designs have different fault-tolerance properties, which may also vary with the type of fabrication techniques used.

The radar environment will influence the design greatly. A land-based system with large amounts of power, cooling, and spare parts available will
certainly have less stringent packaging, power, and reliability constraints than one that is used in the field.

The dimension of the design space can be reduced for problems of interest so that a universal adaptive algorithm (UAA) element does exist. One such breakdown is shown in Table 5. For this radar system, the UAA element would be near optimal for most configurations. For the 30-weight system, 21 bits are required; for the 50-weight system, 22 bits are required for inner-product calculation. But due to MSI and LSI integration sizes, either a 22-bit or even a 24-bit system would be implemented.

The system designer must decide which subset of all possible adaptive systems the processors must support.

**TABLE 5. SPECIFIC RADAR DESIGN PARAMETERS**

1) Number of Weights - 30 to 50
2) Sampling Rate - 20 MHz
3) Electronic Scan
4) Search and Track Modes
5) Typical Times for Land-Based Surveillance Systems
6) Fully Adaptive
7) 8-Bit 2 Complément Complex ADC
8) Low Center
9) 3 to 8 dB Down from Optimal SNR
10) Low Cost
11) 24-Hour Continuous Duty Every Day
12) Land-Based Stationary Environment Constructed to Support Processor
3.7 UNIVERSAL ALGORITHM HARDWARE IMPLEMENTATION

The Gram-Schmidt processor has a very simple interconnect structure regardless of implementation. The bus structure for the pipelined \(O(n^2)\) configuration is shown in Figure 37. All buses have a single source, making this configuration's the simplest interconnect protocol. The recursive \(O(n)\) configuration is shown in Figure 38. It has a slightly more complicated interconnect protocol because one interconnect bus has several sources, which implies that each module must know when it should place its data on the bus. This bus can be implemented with either open-collector or tri-state logic. A comparison of the two configurations shows that each processing element (PE) has the same inputs and outputs; thus, with the addition of bus-sharing logic, a processor can be made to run in either configuration.

The bus interconnects may be either one-half of a complex word wide or a full complex word wide. The latter provides the potential for a higher processing rate.

A variation combining the \(O(n^2)\) and \(O(n)\) configurations may be implemented. This variation is shown in Figure 39 for the five-input case. It enables the system designer to meet the speed requirements without using more hardware than is really needed, a good alternative solution for large \(n\) when the \(O(n^2)\) configuration would contain a prohibitive number of processors. The processor control is no more difficult than for the \(O(n)\) configuration.

The configuration chosen by the system designer is a function of 1) the processor speed; 2) the number of degrees of freedom required; and 3) the required processing rate, which, in general, will be slower than the sampling rate.
Figure 37. Bus structure for pipelined (O(n^2)) configuration.
Figure 38. Bus structure for recursive $O(n)$ configuration.
Figure 39. Variation combining $O(n^2)$ and $O(n)$ configurations. Five-input case is shown.
The processor configuration is governed by the Gram-Schmidt equations, shown in Figure 40. They have been broken down into their real and imaginary components. The majority of the operations performed are add, subtract, and multiply. Only one division is required after all the terms have been summed. The divisor in all processors in a row will be identical. If this process is moved to a dedicated module in each row, n-2 sets of redundant logic have been removed from each row. This removal will reduce the amount of logic per PE, reducing each PE's cost and mean-time-between-failures (MTBF). For a system with a large number of degrees of freedom, this could result in considerable savings in system cost and greatly improve the system's MTBF. The control becomes no more difficult and the bus structure does not change for either the $O(n^2)$ or $O(n)$ configurations (Figures 41 and 42).

The configuration discussed in the paragraph above is expanded here to show a possible hardware implementation. Two generic processing elements are shown. First, the node processing element (NPE) computes all equations except the divisor equation. This processor is present at all the nodes in the Gram-Schmidt array. The second processor, the diagonal processing element (DPE), computes the divisor and its inverse, and then broadcasts the inverse to the NPEs in its row. There is one DPE for each row of the Gram-Schmidt array.

A block diagram for a node processing element (NPE) is shown in Figure 43. This processor does not include the ability to calculate the divisor. This NPE can perform block and exponential averaging. In addition, it can operate in either the $O(n^2)$ or $O(n)$ configurations. Because of its generality, the NPE would not be practical to implement for use in a real
Block G-S Equations

\[ \text{Re}(\tilde{w}) = \text{Re}(\tilde{w}) + \text{Re}(Z^1_1)\text{Re}(Z^1_3) + \text{Im}(Z^1_1)\text{Im}(Z^1_3) \]

\[ \text{Im}(\tilde{w}) = \text{Im}(\tilde{w}) + \text{Re}(Z^1_1)\text{Im}(Z^1_3) - \text{Im}(Z^1_1)\text{Re}(Z^1_3) \]

\[ \tilde{w} = \tilde{w} + \text{Re}(Z^1_1)\text{Re}(Z^1_3) + \text{Im}(Z^1_1)\text{Im}(Z^1_3) \]

\[ \text{Re}(w) = \text{Re}(\tilde{w})(1/\tilde{w}) \]

\[ \text{Im}(w) = \text{Im}(\tilde{w})(1/\tilde{w}) \]

\[ \text{Re}(Z^1_{j+1}) = \text{Re}(Z^1_j) - [\text{Re}(w)\text{Re}(Z^1_j) - \text{Im}(w)\text{Im}(Z^1_j)] \]

\[ \text{Im}(Z^1_{j+1}) = \text{Im}(Z^1_j) - [\text{Re}(w)\text{Im}(Z^1_j) + \text{Im}(w)\text{Re}(Z^1_j)] \]

Exponential G-S Equations

\[ \text{Re}(w_n) = \text{Re}(w_{n-1})S + [\text{Re}(Z^1_1)\text{Re}(Z^1_3) + \text{Im}(Z^1_1)\text{Im}(Z^1_3)](1/\tilde{w}_{n-1})(1 - S) \]

\[ \text{Im}(w_n) = \text{Im}(w_{n-1})S + [\text{Re}(Z^1_1)\text{Im}(Z^1_3) - \text{Im}(Z^1_1)\text{Re}(Z^1_3)](1/\tilde{w}_{n-1})(1 - S) \]

\[ \tilde{w}_n = \tilde{w}_{n-1}S + [\text{Re}(Z^1_1)\text{Re}(Z^1_3) + \text{Im}(Z^1_1)\text{Im}(Z^1_3)](1 - S) \]

\[ \text{Re}(Z^1_{j+1}) = \text{Re}(Z^1_j) - [\text{Re}(w_{n-1})\text{Re}(Z^1_j) - \text{Im}(w_{n-1})\text{Im}(Z^1_j)] \]

\[ \text{Im}(Z^1_{j+1}) = \text{Im}(Z^1_j) - [\text{Re}(w_{n-1})\text{Im}(Z^1_j) + \text{Im}(w_{n-1})\text{Re}(Z^1_j)] \]

Figure 40. Gram-Schmidt (G-S) equations
Figure 41. Processor for the $O(n^2)$ configuration.
Figure 42. Processor for the $O(n)$ configuration.
Figure 43. Block diagram for node processing element (NPE).
system. The inherent cost advantages of using Gram-Schmidt would be can-
celled out.

The arithmetic operations performed within the NPE shown are serial in
nature. The processing speed of the NPE could be increased by adding parallel
computation abilities to the processing element. This parallelism could be
expanded to the point where every equation is implemented in parallel hardware.
Some middle ground between these two extremes, one that meets the processor
speed requirements, will usually be chosen.

The block diagram for the diagonal processing element (DPE) that com-
putes the common divisor is shown in Figure 44. This processor will operate
in the same modes as the NPE and would not be practical to implement because
of its generality. Examination of the block diagram reveals no functional
block representing division. This block is unnecessary because of the
following equation to calculate the inverse of a number:

\[ C = 2 - AB_n \]

\[ B_n = B_{n-1}C \]

where

A is the value whose inverse is to be found,
B is the inverse, and
C is the correction factor to the inverse.

This equation is computed recursively, approximately doubling the
accuracy of the inverse on each iteration. The first value of the inverse is
Figure 44. Block diagram for diagonal processing element (DPE).
a coarse value found in a lookup table. These two processing elements form the basis for the Gram-Schmidt processor. Their physical implementations depend on the system requirements.
4. CONCLUSIONS

We have studied in detail the Gram-Schmidt orthogonalization technique and modular architecture for its implementation. The analysis and simulations show that for blocked average Gram-Schmidt, the Gram-Schmidt system is identical in performance to sample covariance matrix techniques, in particular, Cholesky decomposition. This result shows that a universal adaptive algorithm does exist with a modular architecture since the Cholesky method is universal.

Also, because the architecture is tailored to the Gram-Schmidt algorithm, each processor performs simple operations of the form $z = x + Ay$. The complexity of each processor is low, so that high throughput paths can be achieved.

Figure 45 shows the timing curves for different Gram-Schmidt implementations. Figure 46 shows a banded region for Gram-Schmidt compared against the timing curves determined under a previous RADC contract [Liles et al. 1978]. As can be seen from these figures, systems capable of real-time operation are feasible.

Table 6 shows a summary of the conclusions of this study. We have already discussed the first conclusion, the existence of a good universal adaptive algorithm (UAA) which uses a minimum number of samples and has a modular architecture.

The importance of the second conclusion is that while a UAA may exist, it would have to be large, expensive, and power-hungry to satisfy the performance requirements of all possible radar systems. For example, a land-based system with 200 weights and a 20-MHz sampling rate, a stable, air-conditioned

---

Figure 45. Timing curves for different Gram-Schmidt implementations. One steering vector; two times the number of weights equals number of samples.
Figure 46. Timing curves for various processors, showing banded region for Gram-Schmidt.
| 1. Universal adaptive algorithm with modular architecture exists. |
| 2. Universal adaptive algorithm implementation for subset of all adaptive space exists. |
| 3. Fast implementations are possible (Figure 45). |
| 4. Processors are simple. |
| 5. Processors can be improved as hardware advances are made. |
| 6. The processor array can be designed to have good fault-tolerance properties. |
| 7. Possible to get a priori probability bounds on growth of intermediate results. |
environment with ample space puts far different requirements on a UAA than a 5-weight system small, cool, and light enough to carry on someone's back. But given a reasonably uniform subset of system requirements, a UAA does exist for that subset. A subset of interest is ground-based, permanent installations, 30 to 100 weights, high reliability, and adequate power supply. As advances in digital electronics occur, the size of subset areas increases until a universal implementation does exist.

One must remember that the algorithm, the processor interconnect structure, and the general processor design are all universal. The only thing which can change is particular implementations (number of bits, etc.) of the processor.

The third conclusion dealing with speed has been discussed above and is demonstrated by Figures 45 and 46.

The fourth conclusion deals with the complexity of a processor. Each processor is very simple as shown in Sections 2 and 3.5. Because of this simplicity and the advances of VLSI, shortly a processor on a chip will be possible. Since the processor functions are well specified, as new hardware becomes available, processors using the new components can be intermixed with other processors without system degradation. This intermixing is due to functional replacement on a processor level (conclusion 5).

Conclusion 6 deals with fault tolerance of the system. The Gram-Schmidt array is no more susceptible to noise caused by misaligned or broken converters, receivers, antennas, than any other technique. This was an unexpected, but welcome, conclusion, since Gram-Schmidt implementations have been shown to be prone to noise on digital computers. But due to our implementation of Gram-Schmidt and the nature of radar signals, the noise problem
is not a concern. Also, if any column of the array becomes bad, the system loses one degree of freedom and remains operative—a condition known as graceful degradation.

Conclusion 7 was the most difficult to arrive at, and more work should be performed to sharpen the results. Conclusion 7 states that a priori probability bounds on the internal Gram-Schmidt weights and interprocessor communication can be determined. These bounds are used to determine the number of bits used at different parts of the processing system, and what kind of arithmetic (fixed point or floating point) needs to be performed in order to avoid overflow/underflow and bound effects of roundoff error. In Section 3.3 we discuss in more detail the formulation of these bounds and actual number of bits.
5. RECOMMENDATIONS

Given the results of this study, we feel confident that the modular approach has benefits for the U.S. Air Force in the following areas: 1) logistics, 2) maintainability, 3) ease on new system design, and 4) low life cycle cost.

We believe that a Gram-Schmidt system of $O(n)$ processors should be designed for high-speed updating of weights. The processor should be designed to be integrated into a 30- to 100-weight system. This system would demonstrate proof of concept and would serve as a test bed for further research. The processor designed for an $O(n)$ configuration can also be used to study $O(n^2)$ array architectures; the inverse is not true. Thus, one processor can be used to study both possible configurations.

The processor should accumulate internally at least 24 bits (preferably 32) and transmit to other processors 16 bits. The internal memory for I & Q inputs should be at least 512 complex words.
REFERENCES


REFERENCES (Cont'd)


Sameh, A. H., and D. J. Kuck, Linear Systems Solvers for Parallel Computers, Department of Computer Sciences, Report NO. UIUCDCS-R-75-701, University of Illinois at Urbana-Champaign, February 1975.


Appendix A

PERFORMANCE OF THE SAMPLE COVARIANCE MATRIX ALGORITHM FOR ADAPTIVE ARRAYS

A.1 INTRODUCTION

When an adaptive array is implemented digitally, the sample covariance matrix algorithm provides a direct method of computing the adaptive weights and rapid convergence independent of the eigenvalues of the covariance matrix. Previous analyses of this algorithm have assumed that the weights are computed using one set of array element outputs and these weights are applied to later array outputs. This report considers the case where the adaptive weights are tested against the same set of data used in the weight computation.

For many applications, the multiple channel sidelobe canceller is the preferred adaptive configuration. It can be shown that the sidelobe canceller is a special case of the more general adaptive array. In the next section it is shown that the general adaptive array problem can be transformed to an equivalent sidelobe canceller problem, a form which is more convenient for some analyses. It is also shown that the array performance is independent of this transformation, and that the effective weights, output S/N, etc., can be computed in any convenient coordinate system provided the transformation of coordinates is non-singular.


A. COORDINATE TRANSFORMATIONS

Let \( X \) denote the column vector of array element outputs, 
\( X_T = (x_1, x_2, \ldots, x_N) \), and \( S_X \) denote the corresponding signal vector. The noise covariance matrix of the array outputs is

\[
M_X = E X X^\dagger,
\]

where \( M_X \) is a \( N \times N \) Hermitian matrix for an \( N \) element array, \( E \) denotes the expectation, \( \dagger \) the complex transpose, and all noise components (but no signal) are included in \( M_X \).

The weights which maximize the \( S/N \) ratio are

\[
W_X = M_X^{-1} S_X
\]

and the corresponding array output is

\[
Z = W_X^\dagger X = S_X^\dagger M_X^{-1} X
\]

With optimum weights, the output \( S/N \) ratio is

\[
r = \frac{(W_X^\dagger S_X)^2}{W_X^\dagger M_X W_X} = S_X^\dagger M_X^{-1} S_X
\]

Let \( T \) denote any non-singular transformation, and

\[
Y = T X
\]
In the new coordinate system,

\[ \mathbf{M}_y = \mathbf{E} \mathbf{Y} \mathbf{Y}^\dagger = \mathbf{T} \mathbf{M}_x \mathbf{T}^\dagger \quad \text{(A-6)} \]

\[ \mathbf{S}_y = \mathbf{T} \mathbf{S}_x \quad \text{(A-7)} \]

\[ \mathbf{W}_y = \mathbf{M}_y^{-1} \mathbf{S}_y \quad \text{(A-8)} \]

Combining Eqs. (A-5) through (A-8),

\[ \mathbf{W}_y = (\mathbf{T}^\dagger)^{-1} \mathbf{W}_x \quad \text{(A-9)} \]

and

\[ \mathbf{z} = \mathbf{W}_y^\dagger \mathbf{Y} = \mathbf{W}_x^\dagger \mathbf{X} \quad \text{(A-10)} \]

i.e., the output of the array is unchanged by the transformation.

The sample covariance matrix in \( \mathbf{X} \) coordinates is

\[ \mathbf{\hat{M}}_x = \frac{1}{K} \sum_{k=1}^{K} \mathbf{X}_k \mathbf{X}_k^\dagger \quad \text{(A-11)} \]

where \( \mathbf{X}_k \) denotes the \( k \text{th} \) independent sample of array element outputs and \( K \) is the number of samples in the estimator of \( \mathbf{M}_x \). The weights based on a sample covariance matrix are

\[ \mathbf{\hat{W}}_x = \mathbf{\hat{M}}_x^{-1} \mathbf{S}_x \quad \text{(A-12)} \]

Replacing \( \mathbf{M}_x \) and \( \mathbf{M}_y \) with the corresponding estimators, \( \mathbf{\hat{M}}_x \) and \( \mathbf{\hat{M}}_y \), and following the analysis of the preceding paragraph, it can easily be shown
the array output with weights based on a sample covariance matrix is also independent of the transformation $T$. Hence, the analysis of adaptive array performance, including the sample covariance matrix algorithm, can be performed in any convenient coordinate system provided the required transformation is non-singular.

A.3 PROBABILITY DISTRIBUTION OF S/N

For any arbitrary adaptive array, the input vector $X$ can be transformed to a new set of coordinates in which the signal is present in only one component and the noise covariance matrix is diagonal. First, let $V = M^{-1/2} X$ to diagonalize the noise covariance matrix. Then,

$$M_V = E M_X^{-1/2} X \ X^\dagger M_X^{-1/2} = I$$

(A-13)

$$S_V = M_X^{-1/2} S_X$$

Next, rotate the coordinates by a unitary transformation $U$ so that the $S$ vector is non-zero only in the first component, and normalize its amplitude to unity

$$Y = (S_X^\dagger M_X^{-1} S_X)^{-1/2} U V$$

(A-14)

$$= r_0^{-1/2} U M_X^{-1/2} X$$
where \( r_0 \) is the S/N ratio with optimum weights, given by (A-4). Then,

\[
M_y = r_0^{-1} U M_x^{-1/2} M_x M_x^{-1/2} U^+ = \frac{1}{r_0} I
\]

(A-15)

and

\[
S_y = r_0^{-1/2} U M_x^{-1/2} S_x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}
\]

(A-16)

Note that the output S/N ratio in the new coordinate system is

\[
S_y M_y^{-1} S_y = (M_y)^{-1}_{11}
\]

(A-17)

i.e., the \((1,1)\) element of the matrix \(M_y^{-1}\).

The sample covariance matrix algorithm can be analyzed conveniently in the new coordinate system. The subscript \( y \) will be dropped in the following equations. Again, the sample covariance matrix is

\[
\hat{M} = \frac{1}{K} \sum_{k=1}^{K} y_k y_k^+
\]

(A-18)

The weights based on this estimator are

\[
\hat{W} = \hat{M}^{-1} S
\]

(A-19)
When these weights are tested on a different set of samples than those used in estimating \( \hat{W} \), the S/N ratio \( r_1 \) is

\[
r_1 = \left[ \frac{\hat{W}^+ S}{\hat{W}^+ M \hat{W}} \right]^2
\]

(A-20)

The ratio of \( r_1 \) to the S/N with optimum weights is

\[
\rho_1 = \frac{r_1}{r_0} = \frac{(S^+ \hat{M}^{-1} S)^2}{(S^+ M^{-1} S)(S^+ \hat{M}^{-1} M \hat{M}^{-1} S)}
\]

(A-21)

The probability density of this variable \( \rho_1 \) was derived in [1].

In some cases of interest, the weights may be applied to the same set of samples used in computing \( \hat{W} \). In this case, the output S/N ratio is

\[
r = S^+ \hat{M}^{-1} S = (M^{-1})_1
\]

(A-22)

The analysis of [1] can be extended to obtain an expression for the probability density of \( r \).

The sample covariance matrix has a complex Wishart distribution[1,3], i.e.,

\[
P(A) = \frac{|A|^{K-N}}{I(M)} \exp \left\{ -\text{tr}(M^{-1} A) \right\}
\]

(A-23)

where \(|A|\) denotes the determinant of \(A\), \(N\) is the number of elements in the array, \(\text{tr}\) denotes the trace of the matrix, and \(A = K \hat{M}\). The constant \(I(M)\) is a function of \(K\), \(N\), and the covariance matrix \(M\). In Eq. (A-23), \(P(A)\) is the joint probability density of the elements of \(A\), and is restricted to those matrices \(A\) which are positive definite. It assumes that the underlying noise process is complex Gaussian.

Consider the following representation of the matrix \(A\).

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad (A-24)
\]

where \(A_{11}\) is a scalar, \(A_{21}\) is a \((N-1)\times 1\) column vector equal to \(A_{12}^\dagger\), and \(A_{22}\) is a \((N-1)\times(N-1)\) matrix. As in [1], \(A\) can be factored as follows

\[
A = \begin{pmatrix} 1 & A_{21}^\dagger \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} A_{11} - A_{12} A_{22}^{-1} A_{21} & 0 \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} A_{22}^{-1} & A_{21} \\ 0 & I \end{pmatrix}, \quad (A-25)
\]

and

\[
|A| = |A_{11} - A_{12} A_{22}^{-1} A_{21}| |A_{22}|
\]

(A-26)
Let

\[ D_{11} = A_{11} - A_{12} \ A_{22}^{-1} A_{12} \]
\[ D_{12} = A_{12} = A_{21}^+ \]  \hspace{1cm} (A-27)
\[ D_{22} = A_{22} \]

The Jacobian of the transformation from \((A_{11}, A_{12}, A_{22})\) to \((D_{11}, D_{12}, D_{22})\) is one, so

\[ P(D_{11}, D_{12}, D_{22}) = D_{11}^{K-N} \left| D_{22} \right|^{K-N} \frac{1}{I(M)} \exp \left\{ -(D_{11} + D_{12} D_{22}^{-1} D_{12}^+ + \text{tr} \ D_{22}) r_o \right\} \]

\[ = P(D_{11}) \ P(D_{12}, D_{22}) \]  \hspace{1cm} (A-28)

where

\[ P(D_{11}) = C_1 \ D_{11}^{K-N} \exp \left\{ -r_o D_{11} \right\} \]  \hspace{1cm} (A-29)

and \(C_1\) is a constant.

Representing \(A^{-1}\) in the same form as Eq. (A-24),

\[ A^{-1} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \]  \hspace{1cm} (A-30)

it can easily be shown that \(A_{11}^{-1} = \frac{1}{D_{11}}\). Since \(A = K \hat{M}\) and, from Eq. (A-22)
the output S/N ratio is

\[ r' = (\hat{M}^{-1})_{11} = K A_{11}^{11} = \frac{K}{D_{11}} \]  

(A-31)

Let \( \rho = \frac{r}{r_0} = -\frac{k}{r_0 D_{11}} \). From Eq. (A-29)

\[ P(\rho) = \frac{C_2}{\rho K-N+1} \exp \left\{ -\frac{K}{\rho} \right\} \]  

(A-32)

Normalizing this distribution, \( C_2 = \frac{1}{(K-N)^1} \), and

\[ P(\rho) = \frac{K-N+1}{(K-N)! \rho K-N+2} \exp \left[ -\frac{K}{\rho} \right] \]  

(A-33)

This is the probability density function for the normalized output S/N ratio, \( \rho = r/r_0 \), when the same samples are used in \( \hat{M} \) for computing the weights and for testing the weights.

From Eq. (A-33),

\[ \overline{\rho} = \frac{K}{K-N} \]  

(A-34)

The mean output S/N ratio for the same samples is greater than with optimum weights, \( W = M^{-1} S \). That is, \( \overline{\rho} > 1 \).
Appendix B
DERIVATION OF NOISE SENSITIVITY RESULTS

This appendix derives the results used in Section 3.2. We know [Brennan and Reed 1973] that the SNR of an adaptive system with true covariance matrix \( M \), steering signal \( S \), and weights \( W \) is

\[
\text{SNR}_W = \frac{|S^*W|^2}{W \cdot MW} \quad \text{(B-1)}
\]

When \( W = W_{\text{OPT}} = M^{-1}S \), we get

\[
S^*W = S^*(M^{-1}S) = S^*(M^{-1}M)^{-1}S = (M^{-1}S)^*(M^{-1}S) = WMW
\]

so

\[
\text{SNR}_{\text{OPT}} = \frac{|S^*W|^2}{W \cdot MW} = S^*W = WMW = S^*M^{-1}S
\]

When \( W = W_{\text{OPT}} + E \), where \( E \) is an error vector, we get

\[
\text{SNR}_W = \frac{|S^*(W_{\text{OPT}} + E)|^2}{(W_{\text{OPT}} + E)^*M(W_{\text{OPT}} + E)}
\]

\[
= \frac{[|S^*W_{\text{OPT}}|^2 + 2\text{Re}(S^*W_{\text{OPT}})(S^*E) + |S^*E|^2]}{(W_{\text{OPT}}^*MW_{\text{OPT}} + 2\text{Re}W_{\text{OPT}}^*ME + E^*ME)}
\]

\[
= \frac{(\text{SNR}_{\text{OPT}}^2 + 2\text{SNR}_{\text{OPT}}\text{Re}(S^*E) + |S^*E|^2)/(\text{SNR}_{\text{OPT}}^2 + 2\text{Re}S^*E + E^*ME)}
\]

\[
= \text{SNR}_{\text{OPT}} + \frac{(|S^*E|^2 - \text{SNR}_{\text{OPT}}E^*ME)}{W \cdot MW}
\]

140
\[ \begin{align*}
&= [1 - (E*ME/W*MW)]SNR_{OPT} + (E*ME/W*MW)(|S*E|^2/E*ME) \\
&= [1 - (E*ME/W*MW)]SNR_{OPT} + (E*ME/W*MW)SNR_E \\
&= SNR_{OPT} + (E*S*SE - SNR_{OPT}E*ME)/W*MW \\
&= SNR_{OPT} - E*[M - (S*S/SNR_{OPT})]E \cdot (SNR_{OPT}/W*MW) \quad \text{(B-2)}
\end{align*} \]

Since \( M \) is positive definite Hermitian, \( S*S \) is positive semidefinite Hermitian of rank 1, and

\[ x*[M - (S*S/SNR_{OPT})]x = x*Mx/SNR_{OPT} \cdot [SNR_{OPT} - (x*S*Sx/x*Mx)] = (x*Mx/SNR_{OPT})(SNR_{OPT} - SNR_x) \geq 0 \]

we have \( M - (S*S/SNR_{OPT}) \) is positive semidefinite Hermitian of rank \( n-1 \) (\( W = M^{-1}S \) is in the null space) and

\[ 0 \leq \lambda[M - (S*S/SNR_{OPT})] \leq \lambda_{max}(M) - \lambda_{min}S*S = \lambda_{max}(M) \]

Also

\[ W*MW = (W_{OPT} + E)*M(W_{OPT} + E) = W_{OPT}^*MW_{OPT} + 2ReW_{OPT}*ME + E \cdot ME \]

\[ \geq SNR_{OPT} - 2||S|| \cdot ||E|| + \lambda_{min} \cdot ||E||^2 ; \]

so

\[ SNR_{OPT}/W*MW \leq 1 + (2||S|| \cdot ||E||/SNR_{OPT}) + O(||E||^2) = 1 + O(||E||). \]
When $\hat{W} = W + E$, W not necessarily optimal, we get

$$\text{SNR}_W^* = \left[ |S^*W|^2 + 2\text{Re}(S^*W)(S^*E) + |S^*E|^2 \right] / (W*MW + 2\text{Re}W*ME + E*ME)$$

$$= (\text{assuming } |2\text{Re}W*ME - E*ME| < 1 \text{ for small } E)$$

$$= \left[ |S^*W|^2 + 2\text{Re}(W*SS*E) + |S^*E|^2 \right] \cdot (1/W*MW)$$

$$\cdot \left[ 1 - (2\text{Re}W*ME/W*MW) - (E*ME/W*MW) + 4\text{Re}^2(W*ME)/(W*MW)^2 + O(||E||^3) \right]$$

$$= |S^*W|^2/W*MW + \left( |S^*W|^2/W*MW \right) \cdot (-2\text{Re}W*ME/W*MW) + 2\text{Re}W*SS*E/W*MW$$

$$+ \left( |S^*W|^2/W*MW \right) (-E*ME/W*MW) + \left( S^*W \right)^2/W*MW \cdot \left[ 4\text{Re}^2W*ME/(W*MW)^2 \right]$$

$$- \left[ 4\text{Re}(W*SS*E)\text{Re}(W*ME)/(W*MW)^2 \right] + |S^*E|^2/W*MW + O(||E||^3)$$

$$= \text{SNR}_W + \text{SNR}_W(-2/W*MW) \cdot \text{Re}[W*(M - SS*/SNR W)E]$$

$$+ (\text{SNR}_W/W*MW) \cdot \left\{ (E*SS*E)(1/\text{SNR}_W) - E*ME \right\}$$

$$+ 4\text{Re}W*ME/(W*MW) \cdot \left[ \text{Re}W*ME - \text{Re}(W*SS*E)/\text{SNR}_W \right] \} + O(||E||^3)$$

$$= \text{SNR}_W - 2(\text{SNR}_W/W*MW) \cdot \text{Re}\{[M - SS*/\text{SNR}_W]E\}$$

$$+ \text{SNR}_W/W*MW \text{ Re}\{[4(\text{Ke}W*ME/W*MW)W* - E\} (M - SS*/\text{SNR}_W)E\}$$

$$+ O(||E||^3) \quad . \quad \text{(B-3)}$$

Now we turn to the problem of inverting $\hat{M} = M + A$, where $M$ is a covariance matrix and $A_{k\ell} = \{\rho \text{ if } k=\ell=j, \ 0 \text{ otherwise}\}$. We may write
\[
\hat{M}^{-1} = (M + A)^{-1} = M^{-1}M(M + A)^{-1} = M^{-1}[(M + A)M^{-1}]^{-1} = M^{-1}(I + AM^{-1})^{-1} .
\]  

(B-4)

Since \(AM^{-1}\) is all zeroes except for row \(j\), which is the \(j\)th row of \(M^{-1}\) multiplied by \(\rho\), we can write

\[
\hat{M}^{-1} = M^{-1}\left(I + \rho\begin{bmatrix} 0 & \cdots & 0 \\ m_{j1} & \cdots & m_{jn} \\ 0 & \cdots & 0 \end{bmatrix}\right)^{-1} .
\]  

(B-5)

where \(M = \{m_{ij}\}\) and \(M^{-1} = \{m_{ij}^{-1}\}\). The inverse of the expression in parentheses is given simply by

\[
\left(I + \rho\begin{bmatrix} 0 & \cdots & 0 \\ m_{j1} & \cdots & m_{jn} \\ 0 & \cdots & 0 \end{bmatrix}\right)^{-1} = I - \frac{\rho}{1 + \rho m_{jj}} \begin{bmatrix} 0 & \cdots & 0 \\ m_{j1} & \cdots & m_{jn} \\ 0 & \cdots & 0 \end{bmatrix}
\]  

(B-6)

so that \(\hat{M}^{-1}\) is

\[
\hat{M}^{-1} = M^{-1} + M^{-1}\left(\frac{-\rho}{1 + \rho m_{jj}}\begin{bmatrix} 0 & \cdots & 0 \\ m_{j1} & \cdots & m_{jn} \\ 0 & \cdots & 0 \end{bmatrix}\right) .
\]  

(B-7)

This formula is a special case of the Sherman-Morrison formula [Dahlquist and Björck 1974, p. 161].

The new weights \(\hat{W} = \hat{M}^{-1}S\) are given by

\[
\hat{W} = W + M^{-1}\left(\frac{-\rho}{1 + \rho m_{jj}}\right)\begin{bmatrix} 0 \\ w_{j} \\ 0 \end{bmatrix} + \text{row } j
\]  

where $W = \{w_j\} = M^{-1}S$ are the original weights. Thus the new weights equal the old weights plus the $j$th column of $M^{-1}$ times $[-\rho/(1 + \rho m_{jj})]$. We can express $S^*M^{-1}S$, the optimal SNR of the system with noise, in terms of the old optimal SNR $= S^*M^{-1}S$, using Eq. (B-8):

$$\begin{align*}
SNR_{NEWOPT} &= S^*M^{-1}S = S^*w = S^*w + S^*M^{-1} \cdot \left[ -\rho/(1 + \rho m_{jj}) \right] w_j \\
&= SNR_{OLDOPT} + W \left[ \begin{array}{c} 0 \\ w_j \\ 0 \end{array} \right] \frac{-\rho}{(1 + \rho m_{jj})} \\
&= SNR_{OLDOPT} - \left[ \rho |w_j|^2/(1 + \rho m_{jj}) \right]. 
\end{align*}$$

(B-9)

If the covariance matrix is still $M$, but weights $\hat{W} = \hat{M}^{-1}S$ are used, we may compute the new SNR using $E = \hat{W} - W$ and Eq. (B-2). We need to know

$$E^*M^*E = \left[ M^{-1}(-\rho)(1 + \rho m_{jj}) \right]^* \left[ \begin{array}{c} 0 \\ w_j \\ 0 \end{array} \right] \left[ M^{-1}(-\rho)(1 + \rho m_{jj}) \right] \left[ \begin{array}{c} 0 \\ w_j \\ 0 \end{array} \right]$$

and

$$E^*M^*/(1 + \rho m_{jj})^2 = \left[ \begin{array}{c} 0 \\ w_j \\ 0 \end{array} \right]^* \left[ \begin{array}{c} 0 \\ w_j \\ 0 \end{array} \right] = \frac{\rho^2 |w_j|^2 m_{jj}}{(1 + \rho m_{jj})^2}.$$
\( S^*E = W^*ME = W^* \begin{bmatrix} -\frac{J}{1 + \rho m^3 j} & 0 \\ \frac{J}{1 + \rho m^3 j} & 0 \end{bmatrix} \begin{bmatrix} 0 \\ w_j \end{bmatrix} = \frac{\rho |w_j|^2}{1 + \rho m^3 j} \) \hspace{1cm} (B-11)

and

\[
W^*MW = \text{SNR}_{\text{OPT}} + 2\text{Re}W^*_{\text{OPT}}ME + E^*ME
\]
\[
= \text{SNR}_{\text{OPT}} + 2\rho |w_j|^2/(1 + \rho m^3 j) + \frac{\rho^2 |w_j|^2 m^3 j^2}{(1 + \rho m^3 j)^2} \hspace{1cm} (B-12)
\]

Then the SNR \( W \) is

\[
\text{SNR}_W = \text{SNR}_{\text{OPT}} - \frac{\text{SNR}_{\text{OPT}} - 2\rho |w_j|^2/(1 + \rho m^3 j) + \rho^2 |w_j|^2 m^3 j^2/(1 + \rho m^3 j)^2}{\text{SNR}_{\text{OPT}} - \rho |w_j|^2/(1 + \rho m^3 j)[2 - \rho m^3 j/(1 + \rho m^3 j)]}
\]
\[
= \text{SNR}_{\text{OPT}} - \frac{\rho^2 |w_j|^2 m^3 j^2}{\text{SNR}_{\text{OPT}} - \rho |w_j|^2/(1 + \rho m^3 j)[2 - \rho m^3 j/(1 + \rho m^3 j)]}
\]
\[
\cdot \left[ \text{SNR}_{\text{OPT}} - \frac{|w_j|^2}{m^3 j^2} \right]
\]
\[
= \text{SNR}_{\text{OPT}} - \rho^2 |w_j|^2 m^3 j^2/[\text{SNR}_{\text{OPT}}(1 + \rho m^3 j)^2 - \rho^2 |w_j|^2 (2 + \rho m^3 j)]
\]
\[
\cdot \left[ \text{SNR}_{\text{OPT}} - \frac{|w_j|^2}{m^3 j^2} \right] \hspace{1cm} (B-13)
\]

By differentiating Eq. (B-13) with respect to \( \rho \), we can show SNR \( W \) is a monotonic decreasing function of \( \rho \), and that

\[
\lim_{\rho \to \infty} \text{SNR}_W = \text{SNR}_{\text{OPT}} - |w_j|^2/m^3 j^2,
\]

the same lower bound as for Eq. (B-9).
Appendix C

EXAMPLE OF USING GRAM-SCHMIDT TO SOLVE A SYSTEM OF SIMULTANEOUS EQUATIONS

Assume the matrix equation

\[ Ax = b, \]

where \( A \) is real positive definite Hermitian and

\[
A = \begin{bmatrix}
2 & 4 & 6 \\
4 & 12 & 20 \\
6 & 20 & 36
\end{bmatrix}
\]

\[ x = \text{unknown} = \begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
\]

\[ b = \begin{bmatrix}
4 \\
16 \\
30
\end{bmatrix}
\]

The network used is shown below.

![Network Diagram]
Here,

\[ I_1 \quad I_2 \]

\[ 0 = I_2 - W I_1 \]

We will solve for \( x \) by decomposing \( A \) into the form \( LDL^T \). This decomposition is performed by passing each row of the matrix through the network, using the following steps:

Step 1. Pass row 1 of the matrix through the array to calculate the first row of weights.

\[
\begin{align*}
    w_{11} &= a_{12}/a_{11} = 4/2 = 2 \\
    w_{12} &= a_{13}/a_{11} = 6/2 = 2 \\
    d_{11} &= a_{11} = 2
\end{align*}
\]

Step 2. Pass row 2 of matrix through the array to calculate second row of weights.

\[
\begin{align*}
    w_{22} &= I_2^T/I_1 = \frac{20}{4} = 2 \\
    12 - 2(4) &= 4 \\
    20 - 3(4) &= 8
\end{align*}
\]

\[ d_{22} = 4 \]
Step 3. Pass row 3 of the matrix through the array to calculate $d_{33}$.

\[
\begin{array}{ccc}
6 & 20 & 36 \\
\downarrow & \downarrow & \downarrow \\
2 & 3 & \\
\end{array}
\]

\[
20 - 2(6) = 8 \quad \Rightarrow \quad 36 - 3(6) = 18
\]

\[
18 - 2(8) = 2
\]

$d_{33} = 2$

We now know that the L and D matrices are as follows:

\[
L = \begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 0 \\
3 & 2 & 1 \\
\end{bmatrix} \quad D = \begin{bmatrix}
2 & 0 & 0 \\
0 & 4 & 0 \\
0 & 0 & 2 \\
\end{bmatrix}
\]

To solve the system of equations, we have to perform the following steps as well, because $LDL^T x = b \rightarrow x = [L^{-1} D^{-1} (L^{-1} b)]$:

Step 4. $y = L^{-1} b$

Step 5. $z = D^{-1} y$

Step 6. $x = L^{-1} z$
Step 4. Pass $b$ through the array to perform $L^{-1}b$

\[ y_1 = 4, \quad y_2 = 8, \quad y_3 = 2 \]

Step 5. $z = D^{-1}y$; and since $D$ is diagonal, it is easy to invert.

\[
D^{-1} = \begin{bmatrix}
1/2 & 0 & 0 \\
0 & 1/4 & 0 \\
0 & 0 & 1/2
\end{bmatrix}
\quad y = \begin{bmatrix} 4 \\ 8 \\ 2 \end{bmatrix}
\quad z = D^{-1}y = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}
\]

Step 6. $x = L^{-1}z$, which will be done by the reverse flow method. We pass $z$ through the network in opposite direction from normal.

The vector $x$ has now been determined.

149
We now present a solution to the problem being solved which differs from the previous solution because it eliminates steps 4 and 5 by using an augmented form of the matrix (to be explained below).

Solve \( Ax = b \)

\[
A = \begin{bmatrix}
2 & 4 & 6 \\
4 & 12 & 20 \\
6 & 20 & 36
\end{bmatrix}
\]

\[
X = \begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
\]

\[
b = \begin{bmatrix}
4 \\
16 \\
30
\end{bmatrix}
\]

\[
A \text{ augmented} = \begin{bmatrix}
2 & 4 & 6 & 4 \\
4 & 12 & 20 & 16 \\
6 & 20 & 36 & 30
\end{bmatrix}
\]

\( A \text{ augmented} \) is simply the \( A \) matrix with \( b \) added as the last column. The network also has an added column as shown below.

The steps of the augmented form of solution are as follows:
Step 1. Pass row 1 of matrix through array to calculate first row of weights.

\[ W_{11} = \frac{a_{12}}{a_{11}} = \frac{4}{2} = 2 \]
\[ W_{12} = \frac{a_{13}}{a_{11}} = \frac{6}{2} = 3 \]
\[ W_{13} = \frac{b_1}{a_{11}} = \frac{4}{2} = 2 \]

Step 2. Pass row 2 of matrix through array to calculate second row of weights.

\[ W_{22} = \frac{8}{4} = 2 \]
\[ W_{23} = \frac{8}{4} = 2 \]

Step 3. Pass row 3 of matrix through array to calculate third row of weights.
Note that

\[ W_{13} = 2 \]
\[ W_{23} = 2 \]
\[ W_{33} = 1 \]

These values are the same as \( z \) in step 5 of the previous solution. We can proceed with the reverse flow or unit vector method to determine \( x \).

Step 4. We now show one step of the unit vector method as discussed in Section 2.6.1. The unit vector method passes a unit vector through the array and performs the dot product of the array output with the vector \( z \) to determine an element of \( x \).

A column of the array performs a calculation very similar to dot product and can be used in that manner, as shown below:

\[
\begin{align*}
0_1 &= I_1 - W_1 I_2 \\
0_2 &= I_1 - W_2 I_3 \\
0_3 &= I_1 - W_3 I_4 \\
0_3 &= I_1 - W_1 I_2 - W_2 I_3 - W_3 I_4 \\
&= I_1 - \sum_{i=1}^{3} W_i I_{i+1}
\end{align*}
\]

By setting \( I_1 = 0 \), the column output is the negative of the dot product of the column weights and the input vector. We now perform the unit vector method to calculate \( x_1 \).
The array output is 1; therefore, \( x_1 = -1 \) because the last column output is the negative of the dot product. \( x_1 = -1 \) is the same answer obtained in the previous solution.

**Multiple b's (Multiple Steering Vectors)**

This modification consists of adding enough extra columns to handle the additional b vectors. The illustration shows the configuration for 4b vectors, and a matrix of order 3.
Let $N$ be the matrix dimension. Then $N$ additional processors are needed for each $b$ vector. For $Kb$ vectors, $KN$ additional processors are needed.

To determine $x$ for $Ax=b$, either reverse flow or unit vector methods can be used. If the unit vector method is used, the total time is independent of the number of $b$ vectors. The total time would be derived as follows:

- $N =$ dimension of matrix
- $t =$ time per array stage

There are $N$ unit vectors and $N$ stages; thus, total time is

$$T_{UV} = Nt + (N-1)t = (2N-1)t$$

For the reverse flow method, the time is a function of the number of $b$ vectors ($K$). Using the same definitions as above, the total time

$$T_{RF} = (N-1)t + (K-1)t = (N+K-2)t$$

The reverse flow method is faster for $K \leq N+1$ but also requires more complicated processors.
Appendix D

RECI PROCAL AND SQUARE-ROOT CALCULATION

Depending on which Gram-Schmidt method is used (LDL* or LL*), either the reciprocal or the reciprocal of the square root or a real number must be calculated at each level. These calculations are performed using Newton-Raphson iteration. Newton-Raphson has the advantage of quadratic conversion (the number of accurate bits doubles with each iteration step) and the processing can be performed without divisions. The general recursive relationship is $x_{i+1} = x_i - F(x_i)/F'(x_i)$. For reciprocal of A the function $F(x) = (1/x - A)$. Applying the reciprocal function to the general function we have the following:

$$x_{i+1} = x_i - \frac{F(x)}{F'(x)}$$

$$x_{i+1} = x_i - \frac{1}{x_i} - A$$

$$x_{i+1} = x_i - \frac{x_i - A}{x_i^2}$$

$$x_{i+1} = x_i + x_i - Ax_i^2$$

$$x_{i+1} = (2-Ax_i)x_i$$

The seed value for $x_0$ is determined by table lookup and is an approximation of $1/A$. Each iteration step doubles the accurate bits.

The square root operation is performed almost identically. The function $F = (1/x^2 - A)$, so the recursive formula becomes

$$x_{i+1} = (3 - x_i^2 A)x_i/2$$
The divide by 2 is easy to implement, using a shift operation. The reciprocal of the square-root operation is slower than the reciprocal operation due to an extra multiplication and shift.
Appendix E

TIMING EQUATIONS FOR BLOCK AVERAGE AND O(n) PROCESSORS

The processor computation abilities are those listed in Section 3.4. The O(n) processors are connected and perform the processing as shown in Figure E-1. The reason for the processor on the first input is to compute the first denominator and its reciprocal.

The timing equations developed in Section 2.6 for an O(n^2) array apply to the O(n) array when one equation is to be solved. When more equations are to be solved (in a pipeline by the O(n^2) array) the differences in time between the O(n) and O(n^2) processor implementations become apparent. The three stages of calculation using the O(n) array are:

1. Calculate all internal weights.
2. Pass S through the array.
3. Solve for the weight W.

We build up the timing equation in the above order for three implementations:

- Sequential
- Full pipeline
- Optimal

E.1 CALCULATE ALL INTERNAL WEIGHTS

For each implementation method, we have the following steps:

A. Compute numerator and denominator.
B. Compute reciprocal.
C. Multiply by reciprocal.
D. Pass data through array, applying internal weights.

The timings for these basic steps are:
a. Processor configuration.

b. Processing configuration.

Figure E-1. Model used for \( O(n) \) timing measurements.
### Timing Equations

We must calculate internal weights at N-1 levels; so the general timing equation is

\[ T = (N-1)A + (N-1)B + (N-1)C + (N-2)D \]

The specific timing equations are:

- **Arbitrary N,S**: \( S = 2N \)
- **Sequential**: \( (21NS+9N-32S-9)t \)
- **Full Pipeline**: \( (25N+28N-3S-38)t \)
- **Optimal**: \( (2SN+16N-3S-20)t \)

The only difference between the times for \( O(n) \) processors and \( O(n^2) \) processors is that with \( O(n^2) \) processors, there is overlap between applying the internal weights calculating the numerator. This overlap can be obtained with \( O(n) \) processors if the processing power of each processor is doubled. Another approach is to use twice the number of processors as shown in Figure E-2. This is still an \( O(n) \) processor configuration and the processors can be identical.

### E.2 PASS S THROUGH THE ARRAY

With only \( O(n) \) processors, no pipelining can be performed. By designing the processors with recirculating pipelines, we can handle \( L \) steering vectors at a time, where \( L \) is the length of the pipeline. For \( K \) steering vectors, \( \lceil K/L \rceil \) passes are required. Another method is to pass all \( K \) vectors...
Figure E-2. $O(n)$ processor configuration with same speed to calculate internal weights as $O(n^2)$ processors using block averaging.
through sequentially. In this case, a memory (first-in, first-out (FIFO) buffer) of length \(K-L\) is needed for recirculating (Figure E-3). If \(K-L \geq L\), then additional processing may be desired on the feedback path instead of simply a FIFO-buffer; this is the model we will use. This model applies only to full pipeline and optimal implementations. The sequential implementation does not offer any pipeline possibilities. The timing formulas are:

\[
\begin{align*}
\text{Arbitrary } N,K & \quad K = 1 \\
\text{Sequential} & \quad (N-1)l_{11}Kt \\
\text{Full Pipeline} & \quad (N-1)\max(K,11)t + (K-1)t \\
\text{Optimal} & \quad (N-1)\max(K,5)t + (K-1)t
\end{align*}
\]

The maximum function is due to the recirculating buffer. If \(K\) is less than the length of the pipeline, then we use the pipeline length, or else we must include the recirculating path.

**E.3 SOLVE FOR WEIGHTS**

We solve for both unit vector and reverse flow methods.

**Unit Vector Timings**

For the unit vector method, we must pass through \(K+NK\) vectors. We can use the timing formulas just developed for passing \(K\) steering vectors by substituting \(K+NK\) for \(K\). These formulas are:

\[
\begin{align*}
\text{Arbitrary } N, K+NK & \quad N \geq 10 \\
\text{Sequential} & \quad (N^2-1)l_{11}Kt \\
\text{Full Pipeline} & \quad (N-1)\max(K+NK,11)t + (K+NK-1)t \\
\text{Optimal} & \quad (N-1)\max(K+NK,5)t + (K+NK-1)t
\end{align*}
\]

Since we are interested in large cases, the overall timing formulas now given are with \(N \geq 10\); so a recirculating pipeline is used.
For \( K \) Input Vectors

\[ L \] Forward Pipeline Length

Require

\( K - L \) Feedback Stages

Effective Pipeline Length = \( \text{Max}(K,L) \)

Figure E-3. Recirculating pipelines.
Timing Formulas for Complete Process

Arbitrary $S, K$; $N > 10$

- Sequential: $(21SN + 11N^2K + 9N - 11K - 32S - 9)t$
  
  $(53N^2 - 55N - 20)t$

- Full Pipeline: $(2SN + N^2K + NK + 28N - 3S - 39)t$
  
  $(5N^2 + 23N - 39)t$

- Optimal: $(2SN + N^2K + NK + 16N - 3S - 21)t$
  
  $(8N^2 + 11N - 21)t$

For the special case ($N > 10$, $S = 2N, K = 1$) the above are approximately two times slower than the $O(n^2)$ implementations. As $K$ increases, this difference becomes larger due to this $N^2K$ term in the above formulas.

Reverse Flow Timings

The reverse flow timings, in general, are not simply twice the forward flow for $K$ steering vectors as is the case with $O(n^2)$ processors. The exception is when diagonal collapsing is used because then the array is symmetric from both the input and output ports. The reverse flow timing for horizontal comparison is the same as forward flow from $K$ steering vectors for vertical comparison. For vertical comparison just interchange horizontal and vertical in the above sentence. This is summarized in Table E-1. We will only derive the timing for vertical collapsing.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Pass K Steering Vectors Through Array</th>
<th>Reverse Flow K Steering Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical Collapsing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Horizontal Collapsing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagonal Collapsing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Times at end of double header arrows are the same.
The last processor must operate on \((N-1)\) streams before the other processors can compute their tasks. After the last finished there are \((N-2)\) processors waiting for the output, which must be processed in order. A question which arises is whether to process one steering vector at a time or the \(i^{th}\) element from all steering vectors before the \(i-1^{st}\) element. Both of these methods have the same throughput rate.

The timing equations for reverse flow only are:

- **Arbitrary \(N, K\)**
  \[ (KN-K+N-2)t \]
- **Sequential**
  \[ (KN-K+1N-12)t \]
- **Full Pipeline**
  \[ (KN-K+5N-6)t \]
- **Optimal**
  \[ (KN-K+5N-6)t \]

The computer timing equations for all three steps are:

- **Arbitrary \(N, K, S\)** when \(S=2N, K=1\)
  \[ (21SN+22NK+20N-22K-32S-31)t \]
- **Sequential**
  \[ (21SN+22NK+20N-22K-32S-31)t \]
  \[ (42N^2-22N-53)t \]
- **Full Pipeline**
  \[ [25N+NK+39N-3S-51+(N-1)\max(K,11)]t \]
  \[ (4N^2+45N-62)t \]
- **Optimal**
  \[ [25N+NK+21N-3S-27+(N-1)\max(K,5)]t \]
  \[ (4N^2+21N-32)t \]

Because of the symmetries between horizontal and vertical collapsing, the above formulas apply to both.

**Comparison of Unit Vector and Reverse Flow Times**

The reverse flow method is faster if the following conditions hold:

- **Sequential** All cases
  \[ N > 10 \]
- **Full Pipeline** \(N^2K+12>11N+(N-1)\max(K,11)\)
- **Optimal** All cases
If we add the restriction that \( N \) must be greater than 22, reverse flow is always superior. This restriction is not unreasonable, since our effort is directed at large adaptive arrays.

Special Case

As already mentioned an \( O(n) \) system can determine the internal weights as quickly as an \( O(n^2) \) system if the processors are twice as powerful or doubled up. The process of applying the transformation to the steering vector and then using the unit vector or reverse flow method can also be performed on an \( O(n) \) system at the same rate as on an \( O(n^2) \) system with the same processor speeds. The method depends upon the diagonal collapsing (Figure E-4). In this processor, organization of each row has independent processors as does each column. Therefore, if only one input vector exists (one steering vector, \( K=1 \)) then at each stage no waiting is encountered for busy processors. The following table specifies the conditions in which this method is equal in time to an \( O(n^2) \) array.

<table>
<thead>
<tr>
<th>Unit Vector</th>
<th>Reverse Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Never</td>
</tr>
<tr>
<td>Full Pipeline</td>
<td>( K+NK &lt; 11 )</td>
</tr>
<tr>
<td>Optimal</td>
<td>( K+NK &lt; 5 )</td>
</tr>
</tbody>
</table>

The above table shows that the unit vector method is never equal to \( O(N^2) \) array times for \( N \) of interest \((N > 10)\). The reverse flow method rating is independent of \( N \), which is desirable. We also know that with diagonal collapsing, the reverse flow method can be used with forward flow.
Figure E-4. Diagonal collapsing.
Appendix F

PROCESSING TIMING FOR BLOCK AVERAGE $O(n^2)$ SYSTEM

This appendix determines system throughputs as a function of the degrees of freedom, $N$, the number of samples, $S$, and the number of steering vectors, $K$. This process can be divided into three subfunctions:

1. Calculate all internal weights.
2. Pass $S$ through array.
3. Solve for weights, using unit vector or reverse flow method.

We build up the timing equation in the above order for three implementations:

Sequential (worst case)

Full pipeline

Optimal (best case)

We also assume separate processors for denominator calculations.

F.1 CALCULATE ALL INTERNAL WEIGHTS

For each implementation method we have the following steps:

A. Compute numerator and denominator.
B. Compute reciprocal.
C. Multiply by reciprocal.
D. Pass data through, applying internal weight.

The last step can be combined with the first step, so that as the internal weights are applied the outputs are used to calculate the numerator and denominator of the next row. The timings for these basic steps are:
For a system of degree \( N \) there are \( N-1 \) internal weights; so the general timing formula in terms of stages is:

\[
T = A + (N-1)B + (N-1)C + (N-2)\text{[DA]},
\]

where \( \text{[DA]} \) represents the time to apply the weights and calculate the numerator in the pipeline. The resultant specific formulas are:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>10St</td>
<td>7t</td>
<td>2t</td>
<td>11St</td>
</tr>
<tr>
<td>Full pipeline</td>
<td>10t + (S-1)t</td>
<td>7t</td>
<td>2t</td>
<td>11t + (S-1)t</td>
</tr>
<tr>
<td>Optimal</td>
<td>4t + (S-1)t</td>
<td>7t</td>
<td>2t</td>
<td>5t + (S-1)t</td>
</tr>
</tbody>
</table>

**F.2 PASS S THROUGH ARRAY**

This is the same calculation as applying the internal weights to \( K \) input vectors (remember \( K \) is the number of steering vectors). The time formulas are as follows:

<table>
<thead>
<tr>
<th></th>
<th>( N,K )</th>
<th>( K=1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>((11N + 11K - 22)t)</td>
<td>((11N - 11)t)</td>
</tr>
<tr>
<td>Full pipeline</td>
<td>((11N + K - 12)t)</td>
<td>((11N - 11)t)</td>
</tr>
<tr>
<td>Optimal</td>
<td>((5N + K - 6)t)</td>
<td>((5N - 5)t)</td>
</tr>
</tbody>
</table>
After passing the steering vectors through the array, the resultant vector must be multiplied by all the reciprocals of the denominators. Since these values have already been calculated, only a real-times-complex multiply must be performed, as follows:

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Full pipeline</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>Multiply</td>
<td>Multiply</td>
</tr>
<tr>
<td>Time</td>
<td>$2t$</td>
<td>$2t$</td>
</tr>
<tr>
<td>Total Time</td>
<td>$(11N + 11K - 20)t$</td>
<td>$(11N + K - 10)t$</td>
</tr>
</tbody>
</table>

F.3 SOLVE FOR WEIGHTS

We first explore the unit vector (UV) method. This method, as explained in Section 2.6.1, consists of passing $N$ (or $N-1$) unit vectors through the arrays and performing a dot product on the output. This process is similar to passing $S$ data vectors through the arrays, so the same formula holds. Since the steering vector (or vectors) will pass through the arrays first, we are passing a total of $K+NK$ vectors ($N$ unit vectors for each steering vector). The processing time for dot product is entirely overlapped by the pipeline nature of the array except for one complex multiply and add at the end. The delay for this last multiply/add is equivalent to applying the weight for one input ($11t$, $11t$, and $5t$ for the three methods discussed).
Unit Vector Times

Sequential \[ 11NK + 22N + 11K - 22 \]
Full pipeline \[ NK + 17N + K - 12 \]
Optimal \[ KN + 5N + K - 6 \]

Time for the entire processing time can now be determined by adding these times to the times for passing through the raw data vectors. The resultant timing formulas are:

\[
\text{Arbitrary } N, S, K \quad \text{K=1, S=2N} \\
\begin{align*}
\text{Sequential} & \quad 11SN + 11NK + 33N - 12S + 11K - 23 \quad 22N^2 + 20N - 12 \\
\text{Full pipeline} & \quad SN + NK + 41N - S + K - 31 \quad 2N^2 + 40N - 30 \\
\text{Optimal} & \quad SN + KN + 22N - S + K - 19 \quad 2N^2 + 21N - 18 \\
\end{align*}
\]

Reverse Flow Times

The reverse flow method is exactly the opposite of passing the steering vectors through the array, so the basic timing formula is:

\[
\text{PASS RAW DATA} + 2 \times \text{PASS STEERING VECTOR} + \text{COMPLEX MULTIPLY}
\]

The extra complex multiply is to perform the final calculation of passing the steering vectors. The timing formulas are:

\[
\text{Arbitrary } N, S, K \quad \text{K=1, S=2N} \\
\begin{align*}
\text{Sequential} & \quad 11SN + 44N - 12S + 22K - 51 \quad 22N^2 + 20N - 29 \\
\text{Full pipeline} & \quad SN + 52N - S + 2K - 42 \quad 2N^2 + 50N - 40 \\
\text{Optimal} & \quad SN + 27N - S + 2K - 24 \quad 2N^2 + 25N - 22 \\
\end{align*}
\]
Comparison of Reverse Flow and Unit Vector Times

We can compare the timing equations for reverse flow and unit vector methods to determine which is faster. The results of this comparison are shown below:

<table>
<thead>
<tr>
<th>Method</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>All cases</td>
</tr>
<tr>
<td>Full pipeline</td>
<td>( NK + 11 &gt; 11N + K )</td>
</tr>
<tr>
<td>Optimal</td>
<td>( NK + 6 &gt; 5N + K )</td>
</tr>
</tbody>
</table>

As \( K \) approaches \( N \) the reverse flow method is better. Which method is actually implemented is determined not only by \( K \) and \( N \) but also by the timing requirements of the system to determine the required level of complexity in the processors.
Appendix G

PROOF OF RESULTS ON GROWTH OF INTERMEDIATE RESULTS

We first consider the case of the algorithm without square roots, where the actual elements of the covariance matrix are passed through the array. Since the array does nothing more than the Cholesky LDL* decomposition of $M$, we will first change notation to show how the $Z_i^j(k)$ correspond to intermediate values in the Cholesky decomposition, and then with this simpler notation prove the stated results.

Our claim is that

$$Z_i^j(k) = m_{kk} - \sum_{r=1}^{i-1} \bar{q}_{kr} \bar{x}_{kr} dr \quad i < k \quad (G-1a)$$

$$Z_i^j(k) = m_{kk} - \sum_{r=1}^{k-1} \bar{q}_{kr} \bar{x}_{kr} dr = d_k \quad (G-1b)$$

$$Z_j^i(k) = m_{jk} - \sum_{r=1}^{i-1} \bar{q}_{kr} \bar{x}_{jr} dr \quad i < k \quad (G-1c)$$

$$w_{kj} = Z_j^i(k)/Z_k^k(k) = \bar{q}_{jk} \quad j > k \quad (G-1d)$$

where $L = \{Z_{jk}\}$ (unit lower triangular), $D = \text{diag}\{d_j\}$, $M = \{m_{jk}\}$, and $M = LDL^*$. The proof is by noting that the array and the Cholesky decomposition perform the same column operations on $M$, and that the array recomputes some intermediate values (because no time is lost due to parallelism, and interprocessor communication is simplified).

We may now show

$$\lambda_{\min} \leq Z_k^k(k) = m_{kk} - \sum_{r=1}^{k-1} \bar{q}_{kr} \bar{x}_{kr} dr \leq m_{\max} \quad (G-2a)$$

$$\lambda_{\min} \leq Z_i^j(k) = m_{kk} - \sum_{r=1}^{i-1} \bar{q}_{kr} \bar{x}_{kr} dr \leq m_{\max} \quad (i < k) \quad (G-2b)$$

where $\lambda_{\min}$ = minimum eigenvalue of $M$ and $m_{\max} = \max_{ij} |m_{ij}|$. 

172
We need several well-known facts. \( m_{\text{max}} = \max_{i} |m_{ii}| \) since our matrix is positive definite Hermitian [Isaacson and Keller 1966]. The separation theorem [Franklin 1968]* states that if \( M \) is an \( n \times n \) positive definite Hermitian matrix, and if \( M_m \) is the \( m \times m \) matrix consisting of the first \( m \) rows and \( n \) columns of \( M \) with eigenvalues \( \lambda^m_1 \leq \lambda^m_2 \leq \ldots \leq \lambda^m_m \), then the \( \lambda^i_j \) satisfy

\[
\lambda^m_1 \leq \lambda^{m-1}_2 \leq \lambda^m_2 \leq \ldots \leq \lambda^{m-1}_m \leq \lambda^m_m.
\]

In other words, the \( m-1 \) eigenvalues lie in between, or separate the \( m \) eigenvalues of the larger matrix. Also, if \( M=LDL^* \), \( D=\text{diag}(d) \), then \( \det M_m = \prod_{j=1}^{m} \lambda^m_j = \prod_{j=1}^{m} d_j \), the equality of which follows from the fact that \( \det M = \det D = \prod d_j \) and the Cholesky decomposition of \( M_m \) produces the same \( \lambda^i_j \) and \( d_i \) values for \( i \leq m \). Finally, to prove Eq. (G-2a), we write

\[
d_m = \prod_{j=1}^{m} d_j / \prod_{j=1}^{m-1} d_j = \frac{\det M_m}{\det M_m} = \prod_{j=1}^{m} \lambda^m_j / \prod_{j=1}^{m-1} \lambda^m_j
\]

so since

\[
\lambda^{m-1}_j \leq \lambda^m_{j+1},
\]

\[
d_m = \prod_{j=1}^{m} \lambda^m_j / \prod_{j=1}^{m-1} \lambda^m_{j+1} = \lambda^m_1 \geq \lambda^m_m = \lambda_{\text{min}}.
\]

The other half of the inequality follows by our noticing that the defining equation for \( d_k \) (Eq. (G-1b)) starts with \( m_{kk} \leq m_{\text{max}} \) and subtracts positive numbers \( |z_{kr}|^2 d_r \) (\( d_r \geq \lambda_{\text{min}} > 0 \) since \( M \) is positive definite). Eq. (G-2b) follows from (G-2a) since \( \lambda_{\text{min}} \leq z_k^k(k) \leq z_k^k(k) \leq m_{kk} \leq m_{\text{max}} \), and the sum for \( z_k^k(k) \) subtracts more positive terms than the sum for \( z_k^k(k) \). That these bounds are sharp is obvious by considering any diagonal matrix.

Next, we show

\[ |z_j^k(k)| \leq m_{\text{max}} \quad (G-2c) \]

\[ |z_j^i(k)| \leq 2m_{\text{max}} \quad (i \neq k) \quad (G-2d) \]

Let \( \tilde{\kappa}_{kr} = \kappa_{kr} \sqrt{d_r} \) so that \( d_k \tilde{\kappa}_{jk} = z_j^{k}(k) = m_{jk} - \sum_{r=1}^{k-1} \tilde{\kappa}_{kr} \tilde{\kappa}_{jr} \) (k \neq j) so

\[ z_j^{k}(k) = \sqrt{d_k} \left( \sqrt{d_k} \tilde{\kappa}_{jk} \right) = \tilde{\kappa}_{kk} \tilde{\kappa}_{jk}. \]

Since \( m_{kk} = \sum_{r=1}^{k} |\tilde{\kappa}_{kr}|^{2} \) and

\[ m_{jj} = \sum_{r=1}^{j} |\tilde{\kappa}_{jr}|^{2} \]

we have \( \tilde{\kappa}_{kk} < \sqrt{m_{\text{max}}} \) and \( \tilde{\kappa}_{jk} < \sqrt{m_{\text{max}}} \) so \( |z_j^{k}(k)| \leq m_{\text{max}} \), proving Eq. (G-2c). Also \( |z_j^{i}(k)| = |m_{jk} - \sum_{r=1}^{k-1} \tilde{\kappa}_{kr} \tilde{\kappa}_{jr}| = |m_{jk} - (k^{\text{th}} \text{ row of } L \text{ up to } i-1, j^{\text{th}} \text{ row of } L \text{ up to } i-1)| \leq |m_{jk}| + |(k^{\text{th}} \text{ row of } L \text{ up to } i-1, j^{\text{th}} \text{ row of } L \text{ up to } i-1)| \)

(\text{where } (\cdot, \cdot) \text{ is a complex dot product})

\[ \leq m_{\text{max}} + |k^{\text{th}} \text{ row of } L \text{ up to } i-1|||j^{\text{th}} \text{ row of } L \text{ up to } i-1|| \]

(by Cauchy Schwartz, where \( || \cdot || \) = vector norm)

\[ \leq m_{\text{max}} + |k^{\text{th}} \text{ row of } L|||j^{\text{th}} \text{ row of } L|| \]

\[ = m_{\text{max}} + \sqrt{m_{ik} m_{jj}} \leq m_{\text{max}} + m_{\text{max}} = 2m_{\text{max}} \]

proving Eq. (G-2d). To see that Eq. (G-2c) is sharp and (G-2d) is sharp to within a factor of 2, consider the matrix.
\[
\begin{bmatrix}
a+b & a-b \\
a-b & a+b
\end{bmatrix}
\text{ for } 0 < b << a.
\]

Finally, we prove
\[
|w_{jk}| = \frac{|z_j^k(k)| |z_k^k(k)|}{\sqrt{\frac{m_{\max}}{\lambda_{\min}}}}.
\]

(G-2e)

Since \( m_{kk} = \sum_{r=1}^{k} |\tilde{z}_{kr}|^2 \) we have \( |\tilde{z}_{kr}| \leq \sqrt{m_{\max}} \).

Also, \( |w_{kj}| = \frac{|\tilde{z}_{jk}| \sqrt{d_k}}{\sqrt{d_j}} \leq \frac{\sqrt{m_{\max}}}{\sqrt{d_k}} \leq \frac{\sqrt{m_{\max}}}{\sqrt{\lambda_{\min}}} \) since \( d_k \geq \lambda_{\min} \). To see that this bound is sharp to within a factor of 2, consider
\[
\begin{bmatrix}
\frac{1}{a^2} & \frac{1}{a} \\
\frac{1}{a} & 2
\end{bmatrix} = \begin{bmatrix}
1 & 0 \\
\frac{1}{a^2} & 0
\end{bmatrix} \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix} 
\]

for large \( a \);

so \( \lambda_{\min} \approx \frac{1}{2a^2} \), \( m_{\max} = 2, \sqrt{\frac{m_{\max}}{\lambda_{\min}}} \approx 2a \).

Next we consider the version of the algorithm without square roots when sample vectors are used to compute the \( w_{ij} \)'s. We denote the intermediate results by \( Z_{ij}^1(k) \) to distinguish them from the \( Z_{ij}^1(k) \) above, when the actual matrix was used. We interpret the inputs \( Z_{ij}^1(k) \) as Gaussian random variables with zero means and covariance matrix \( M \), and compute the distributions of the \( Z_{ij}^1(k) \) which are functions of the \( Z_{ij}^1(k) \). The actual distributions of the \( Z_{ij}^1(k) \) are too complicated to compute exactly, but we
make the approximation of replacing the \( w_{ij} \)'s by their means as soon as they are computed. When the number of samples is large, this is an excellent approximation. (For other properties of these distributions, see [Rao 1965].) It is in fact possible to write down the exact distribution of the \( w_{ij} \)'s, see Rao, p. 508. With this approximation we note that \( Z_j^i(k) \) is a linear combination of the input random variables, and hence is Gaussian with zero mean. If we compute its variance, then we will know the distribution of its possible values and hence how many bits are required to represent it.

We now prove

\[
\operatorname{var}[Z_j^i(k)] = Z_j^i(j) \quad (G-3a)
\]

where \( Z_j^i(j) \) is computed as though the actual matrix were being passed through. Thus \( \lambda_{\min} \leq \operatorname{var}[Z_j^i(k)] \leq \lambda_{\max} \), using the results of the last analysis.

In other words, after the random variables have passed through \( i \) rows of the array, their variances equal the \( i \)th partial sums of the expressions for \( d_j \). Let \( V \) be the column vector of random variables, so that \( \mathbb{E}(VV^*) = M \) and \( \mathbb{E}[(L^{-1}V)(L^{-1}V)^*] = L^{-1}\mathbb{E}(VV^*)L^{-1} = D \). Let \( L_m \) represent the transformation performed by the first \( m \) rows. Then

\[ \begin{bmatrix}
1 \\
\vdots \\
1 \\
-w_{mn+1} \\
\vdots \\
-w_{mn}
\end{bmatrix}
\begin{bmatrix}
1 \\
-w_{12} \\
\vdots \\
-w_{13} \\
\vdots \\
-w_{1n}
\end{bmatrix}
\begin{bmatrix}
1 \\
w_{m+1m+2} \\
\vdots \\
w_{m+1n} \\
w_{n-1n} \\
-w_{n-1n}
\end{bmatrix}
\begin{bmatrix}
w_{m+1m+2} \\
\vdots \\
1 \\
-w_{m+1n} \\
-w_{m+1n}
\end{bmatrix}
\cdot L^{-m}
\]

\[ \begin{bmatrix}
1 \\
\vdots \\
1 \\
w_{m+1m+2} \\
\vdots \\
w_{m+1n} \\
w_{n-1n} \\
1
\end{bmatrix}
\cdot L^{-1}
\]
Partitioning $L$ as

$$L = \begin{bmatrix}
1 & & & & & 0 \\
& w_{12} & 1 & & & \\
& & & \ddots & & \\
& & & & \ddots & \ddots \\
& w_{1m} & & & \cdots & 1 \\
w_{1m+1} & \cdots & w_{nm+1} & 1 & & \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
w_{1n} & w_{mn} & w_{m+1n} & & & 1
\end{bmatrix} = \begin{bmatrix}
L_1 & 0 \\
0 & A & L_2
\end{bmatrix}$$

and $D$ as

$$D = \begin{bmatrix}
d_1 & & & & 0 \\
& & \ddots & & \\
& d_m & & \ddots & \ddots \\
& & & \ddots & \ddots & \ddots \\
& & & & d_{m+1} & \ddots \\
0 & & & \ddots & \ddots & d_n
\end{bmatrix} = \begin{bmatrix}
D_1 & 0 \\
0 & D_2
\end{bmatrix}$$

we see we can write

$$L^{-m} = \begin{bmatrix}
1 & & \ & 0 \\
& & \ddots & \ddots & \ddots \\
& & & 0 & \ddots & \ddots \\
& & & & & 0 \\
& & & & & L_2
\end{bmatrix}L^{-1}.$$
so $M_m = E[(L_m V)(L_m V)^*] = L_m M L_m^*$

$$
= \begin{bmatrix}
I & 0 \\
0 & L_2
\end{bmatrix} L^{-1} LDL^* L^{-1}^* \begin{bmatrix}
I & 0 \\
0 & L_2^*
\end{bmatrix}
$$

$$
= \begin{bmatrix}
D_1 & 0 \\
0 & L_2 D_2 L_2^*
\end{bmatrix}
$$

It is the diagonal elements of $M_m$ that are our answers. The first $m$ diagonal elements are obviously equal to their final values. Now $M = LDL^*$

$$
\begin{bmatrix}
L_1 & 0 \\
A & L_2
\end{bmatrix} \begin{bmatrix}
D_1 & 0 \\
0 & D_2
\end{bmatrix} \begin{bmatrix}
L_1^* & A^* \\
0 & L_2^*
\end{bmatrix} = \begin{bmatrix}
L_1 D_1 L_1^* & L_1 D_1 A^* \\
AD_1 L_1^* & AD_1 A^* + L_2 D_2 L_2^*
\end{bmatrix}
$$

Hence

$$(*) \ (L_2 D_2 L_2^*)_{ij} = (AD_1 A^* + L_2 D_2 L_2^*)_{ij} - (AD_1 A^*)_{ij}
$$

$$
= m_{ij} - \sum_{r=1}^{m} \lambda_{ir} \bar{x}_{jr} \ dr
$$

Set $i=j$ and we get the desired result.

Next we prove that $\tilde{w}_{ij}$ is approximately Gaussian with mean

$$E(\tilde{w}_{ij}) = Z_j^*(i); \text{ so } |E(\tilde{w}_{ij})| \leq m_{\max} \text{ and variance}$$
\[
\text{var}(\tilde{w}_{ij}) = \frac{1}{S} \left( z_i^1(i) z_j^1(j) + |z_i^1(j)|^2 \right)
\]
\[
= \frac{1}{S} \left[ d_i \cdot \left( m_{ij} - \sum_{r=1}^{i-1} |x_{jr}|^2 \right) + |m_{ij} - \sum_{r=1}^{i-1} x_{jr} |^2 \right]
\]
\[
\leq 2 \frac{m_{\text{max}}^2}{S} 
\]

If \( Q_1 \) and \( Q_2 \) are normal with covariance matrix \( \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \), it is easy to see \( \text{E}(Q_1 Q_2) = m_{12} \) and \( \text{var}(Q_1 Q_2) = m_{11} m_{22} + m_{12} m_{21} \). Since linear combinations of multivariate Gaussian random variables are multivariate Gaussian, we need only find out what the variances and covariances are of the variables used to form \( \tilde{w}_{ij} \). We approximate \( \tilde{w}_{ij} \) as Gaussian by the central limit theorem. They are given by (*) with \( m = i \), and this is our result. The \( 1/S \) factor is used to cancel one of the \( S \) factors "concealed" in \( m_{\text{max}}^2 \). Since \( \tilde{w}_{ij} \approx N(\mu, \sigma^2) \) with \( |\mu| \leq m_{\text{max}} \) and \( \sigma^2 \leq 2m_{\text{max}}^2/S \) we have

\[
P(|\tilde{w}_{ij}| \geq 2^t \frac{m_{\text{max}}}{S}) = P(\tilde{w}_{ij} \leq 2^t \frac{m_{\text{max}}}{S}) + P(\tilde{w}_{ij} \leq -2^t \frac{m_{\text{max}}}{S})
\]

\[
= P \left( \frac{\tilde{w}_{ij} - \mu}{\frac{\sigma}{\sqrt{S} \frac{m_{\text{max}}}{S}}} \geq \frac{2^t \frac{m_{\text{max}}}{S} - \mu}{\frac{\sigma}{\sqrt{S} \frac{m_{\text{max}}}{S}}} \right) + P \left( \frac{\tilde{w}_{ij} - \mu}{\frac{\sigma}{\sqrt{S} \frac{m_{\text{max}}}{S}}} \leq \frac{-2^t \frac{m_{\text{max}}}{S} + \mu}{\frac{\sigma}{\sqrt{S} \frac{m_{\text{max}}}{S}}} \right)
\]

\[
\leq P \left[ N(0,1) \geq \frac{2^t \frac{m_{\text{max}}}{S} - \mu}{\frac{\sigma}{\sqrt{S} \frac{m_{\text{max}}}{S}}} \right] + P \left[ N(0,1) \leq \frac{-2^t \frac{m_{\text{max}}}{S} + \mu}{\frac{\sigma}{\sqrt{S} \frac{m_{\text{max}}}{S}}} \right]
\]

\[
= P \left[ N(0,1) \geq \sqrt{\frac{S}{2}} (2^t - 1) \right] + P \left[ N(0,1) \leq -\sqrt{\frac{S}{2}} (2^t - 1) \right]
\]

180
\[ P \left( \tilde{w}_{ij} > \frac{\sqrt{\frac{5}{2}}}{2} (2^t - 1) \right) = P \left| N(0,1) > \sqrt{\frac{5}{2}} (2^t - 1) \right|. \]

\( \tilde{w}_{ij} \) is approximately \( \chi^2 \) with \( S \) degrees of freedom, mean

\[ E(\tilde{w}_{ij}) = Z_i(i), \ |E(\tilde{w}_{ij})| \leq m_{\text{max}} \] and variance

\[ \text{var}(\tilde{w}_{ij}) \leq \frac{2 m_{\text{max}}^2}{S}. \] (G-3c)

Since \( E(\tilde{w}_{ij}) = \text{var}[Z_i^*(i)] \), the proof of the first part is easy. Recalling that the expectation of the fourth power of a standard normal variable is 3, we obtain \( \text{var}(\tilde{w}_{ij}) = 2 \text{var}[Z_i^*(i)]/S \leq 2 m_{\text{max}}^2/S. \) Thus

\[ P(\tilde{w}_{ij} > 2 m_{\text{max}}) = P \left( \frac{\tilde{w}_{ij}}{m_{\text{max}}} \frac{S}{2} > 2 t \right) = P(\chi_3^2 < 2 t). \]
Appendix H

SAMPLE VOLTAGE VECTOR MODEL

To determine if the sample covariance matrix approach is feasible when \( N \), the number of weights, is at least 200, we had to use a model of the sample voltage vectors to write a computer simulation to test the algorithm and its implementation.

The radar test problem is arranged so that the interferers can be specified by their eigenvalues. To do this simply, a linear antenna array with uniform spacing and weighting was chosen. With this configuration it is easy to form multiple beams and place an interferer in each beam so that each beam output contains only power from that interferer. Since each beam output is then independent, the covariance matrix of the beam output is diagonal and the interferer powers are the eigenvalues. It is easy to transform the problem to element space, using a unitary transformation.

To implement this approach, the interferers are placed so that all except one are at nulls of the beam pattern for each beam. The voltage output of a beam formed by summing all element outputs is given by

\[
X_b = \left[ 1 + e^{i\psi} + e^{i(N-1)\psi} \right] = \frac{\sin(N\psi/2)}{\sin\psi/2}
\]

where

\[
\psi = 2\pi D \sin \theta,
\]

\( \theta \) = an angle measured from the boresight of the antenna to a point source, and

\( D \) = the antenna element separation in wavelengths.

If interferer positions are chosen so that \( \sin \left[ N(\psi_n - \psi_m)/2 \right] = 0 \) for \( n \neq m \), they satisfy the condition for independence. This will be true when
\(N/2(\psi_n - \psi_m) = k\pi\), where \(k\) is an integer. The desired interferer positions are, therefore,

\[\psi_k = \frac{2\pi k}{N}, \quad k = 1, \ldots, N-1\]  \hspace{1cm} (H-1)

In element space the voltage at each of \(N\) elements is given by

\[x_n = \sum_{k=1}^{K} R_k V_k e^{i \frac{2\pi k (N-1)}{N}N} + V_n \cdot R_n\]  \hspace{1cm} (H-2)

for the \(K = N-1\) interferers. The \(\lambda_k\) are the powers in each interferer and the \(R\)'s are independent zero-mean random variables with \(|R_k|^2 = 1\). The \(R\) are included, when needed, to simulate the interferer variations between samples, and receiver noise whose power is \(Q_n\).

Since the interferer positions are known, the true covariance matrix can be computed as

\[M_{m,n} = \frac{X^*_m X_n}{1/K} = \sum_{k=1}^{K} \lambda_k e^{-i\frac{2\pi k}{N(m-n)}}\]  \hspace{1cm} (H-3)

A stochastic sample covariance matrix using \(S\) samples can be obtained by forming

\[\hat{M}_{m,n} = \frac{1}{S} \sum_{s=1}^{S} s^* X^*_m s X_n\]  \hspace{1cm} (H-4)

where the voltages are obtained from Eq. (H-2).

With this model the eigenvalues \(\lambda_k\) can be chosen to span any desired range of values, and the number of interferers can be varied up to \(N-1\) while one specifies the eigenvalues.
MISSION

of

Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.