TOEPLITZ MATRIX PROBLEMS, FINITE RINGS AND VLSI ARCHITECTURES

by

Christopher J. Zarowski

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Doctor of Philosphy
in
Electrical Engineering

Winnipeg, Manitoba

ISBN 0-315-48106-4

TOEPLITZ MATRIX PROBLEMS, FINITE RINGS AND VLSI ARCHITECTURES

BY

CHRISTOPHER J. ZAROWSKI

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

This thesis is concerned with the solution of certain problems involving Toeplitz matrices. Specifically, the inversion and/or *LDU* factorization, reflection coefficient computation, and the solution of Toeplitz systems of equations are all of interest. These Toeplitz matrix problems may be solved in many ways, but of particular interest are the Schur and split Schur algorithms. A parallel-pipelined processor architecture for the implementation of the Schur algorithm is due to Kung and Hu, and we similarly develop an architecture for the split Schur algorithms. In both cases the parallel-pipelined processor system is a linear array of $O(n)$ processors (Toeplitz matrix is order n). The resulting machines have time complexities of $O(n)$. A Schur algorithm for the Hermitian Toeplitz matrices of any rank profile is developed as well, and a parallel processor implementation of it is considered. This latter algorithm is based upon the Levinson-Durbin algorithm for such matrices developed by Delsarte, Genin and Kamp. The behaviour of the Schur and split Schur algorithms under fixed-point arithmetic implementation conditions is considered. It is found that they are numerically stable, but that one must beware of ill-conditioned input data. To handle the ill-conditioned data cases, quantization error-free computation implementations of the Schur algorithm are considered. It is shown that quantization error-free computation should take place in finite rings and fields. Hensel codes and rational arithmetic are shown to be unsatisfactory quantization error-free computation methods. Quantization error-free computation in the finite ring of integers modulo p^r, denoted Z_{p^r}, is considered for the special cases of $p = 2^n \pm 1$. Hardware structures for modulo p^r arithmetic (addition, subtraction, and multiplication) are described, and a technique for mapping integer data into Z_{p^r} without the need for integer division is presented. Computation in the ring Z_{p^r} is advocated because it is easy to achieve a ring of large size simply by increasing r while holding p fixed. Large rings are needed because the quantization error-free solution of Toeplitz matrix problems produces numbers of large size in general. Furthermore, large quadratic residue number systems can be constructed from Z_{p^r} when p is a Gaussian prime of the form $2^n + 1$. This is useful in the complex-valued data case.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. H. C. Card for his patient encouragement and supervision throughout the course of this work.

The financial assistance of the Natural Sciences and Engineering Research Council (NSERC) in the form of Postgraduate Scholarships is gratefully acknowledged.

TABLE OF CONTENTS

Chapter

Chapter

Chapter

Appendix

Chapter I

INTRODUCTION

1. The Motivation

This thesis is concerned with a particular class of matrix algebra problems. The matrices of interest here are finite-dimensional, nonsingular Toeplitz matrices, and these may be real-valued or complex-valued. We are specifically interested in the problems of Toeplitz matrix inversion and/or *LDU* factorization, Toeplitz system solution, and reflection coefficient computation. We refer to these problems as "Toeplitz matrix problems" in the remainder of this thesis. Note that these problems are more precisely defined in Chapter II.

As will be demonstrated in Chapter II, these Toeplitz matrix problems arise in diverse ways and have enormous applications in all areas of science and engineering, and for this reason they have been widely studied by many researchers for many years. Thus, the study of Toeplitz matrix problems at a theoretical (i.e., mathematical) level is certainly well motivated.

In the engineering literature especially, most of the theoretical studies of Toeplitz matrix problems revolve around the issue of how to solve them in a computationally efficient manner. The search for computationally efficient algorithms is often driven by the need to solve Toeplitz matrix problems at a speed that is sufficient to meet the needs of the application at hand. Studies in this area have yielded many fast algorithms for the solution of Toeplitz matrix problems, and in the process have yielded much insight into the properties of Toeplitz matrices. Note that classically it is intended that such efficient algorithms shall run on conventional sequential processing systems.

More recently, due to the need for real-time solutions to Toeplitz matrix problems, there has been a search for algorithms that are amenable to parallel processing system implementation. In this regard it has been discovered that some of the classical algorithms for the solution of Toeplitz matrix problems are suitable for such an implementation while others are not as suitable (see Chapter IV).

Since some Toeplitz matrix problems are ill-conditioned (see Chapter V), conventional finite precision arithmetic implementations (i.e., fixed-point or floating-point arithmetic) of these algorithms will yield poor results (Chapter V). Thus, we are motivated to investigate the quantization error-free implementation of solutions to the Toeplitz matrix problems (see Chapters VI and VII).

Because of the high complexity of quantization error-free parallel processor implementations of the algorithms for solving Toeplitz matrix problems, very large scale integration (VLSI) technology, or even wafer scale integration (WSI) technology will be needed to successfully implement them. Since quantization error-free computation should take place in a finite ring or field (see Chapter VI), we are motivated to examine VLSI/WSI implementable architectures for arithmetic in finite rings (see Chapter VIII).

2. Objectives

Given the motivations of the preceding section, this thesis seeks to study certain algorithms for the solution of Toeplitz matrix problems, their finite precision arithmetic properties, and the quantization error-free parallel processor implementation of them using arithmetic in finite rings. Note that the finite precision parallel processor implementations are very similar to the quantization error-free parallel processor implementations.

3. Contributions of this Thesis

To the knowledge of the author, this thesis represents the first attempt at applying quantization error-free computation techniques to the solution of Toeplitz matrix problems. Arising from a study of this problem, the contributions of this thesis are:

(i) A new derivation of the split Schur algorithms of Delsarte and Genin based upon the Kung-Hu form of the Schur algorithm, rather than upon the Le Roux-Gueguen form of the Schur algorithm (see Chapter II, section 2.5).

(ii) A demonstration of the fact that the Le Roux-Gueguen and Kung-Hu Schur algorithms are actually the same algorithm (see Chapter III, section 1).

(iii) An inverse mapping from the split Schur variables to the Schur variables is developed (see Chapter III, section 2).

(iv) A Schur algorithm for Hermitian Toeplitz matrices of any rank profile is presented (see Chapter III, section 3).

(v) Parallel-pipelined processor arrays to implement the split Schur algorithms of Delsarte and Genin are presented (see Chapter IV, section 3).

(vi) A parallel-pipelined processor implementation of the Schur algorithm for Hermitian Toeplitz matrices of any rank profile is discussed (see Chapter IV, section 4).

(vii) The behaviour of the Schur and split Schur algorithms when implemented with fixed-point arithmetic is studied using the method of Alexander and Rhee. It is shown that the Schur and split Schur algorithms are numerically stable, although one must beware of ill-conditioned Toeplitz matrices (see Chapter V).

(viii) It is shown that quantization error-free computation should only take place in a finite ring or field, and that Hensel codes and rational arithmetic are ineffective quantization error-free computation methods (see Chapter VI, sections 1 and 2).

(ix) The quadratic residue number system is extended to include complex-valued data with rational-valued real and imaginary parts (see Chapter VI, sections 4.1 and 4.2).

(x) A conjecture concerning the structure of a certain subset of the finite ring Z_{p^r} is stated, and certain evidence is presented that supports the conjecture (see Chapter VI, section 4.3).

(xi) Quantization error-free forms of the Schur algorithm are developed (see Chapter VII).

(xii) Serial and parallel VLSI/WSI implementable hardware structures for addition, subtraction (via negation), and multiplication modulo p^r when $p = 2^n \pm 1$ are presented (see Chapter VIII, section 1).

(xiii) A VLSI/WSI implementable architecture for mapping from the integers to the finite ring Z_{p^r} is presented, and the problem of mapping data in Z_{p^r} back to the integers is also considered (see Chapter VIII, section 2).

4. Thesis Organization

The subject matter of this thesis is organized into nine chapters. Chapter II largely presents background material that is either essential to understanding the remainder of this thesis, or else helps the reader to understand how this thesis relates to the work on Toeplitz matrix problems that has already been done by others. In this chapter all of the classical methods for solving Toeplitz matrix problems are presented, and a listing of various areas where Toeplitz matrix problems arise is presented. Chapter III presents some new results concerning Schur and split Schur algorithms. Chapter IV describes various parallel-pipelined processor arrays for the implementation of the Schur and split Schur algorithms. Chapter V considers the behaviour of the Schur and split Schur algorithms under fixed-point arithmetic implementation conditions, and shows that the Schur and split Schur algorithms are numerically stable. Chapter VI describes the options available for implementing quantization error-free computation, and rejects two of them leaving only computation in finite rings and fields. Chapter VII presents quantization error-free forms of the Schur algorithm. These forms are suitable for implementation with arithmetic in finite rings and fields. Chapter VIII presents hardware architectures for arithmetic in the finite ring Z_{p^r}, and a means of mapping data from the integers into this ring (and vice versa). Chapter IX considers the prospects for successfully implementing fault-tolerant designs of the systems that compute in Z_{p^r}, since these systems will be quite large in general (as is shown in Chapter VII). Their large size will cause them to suffer from serious reliability problems unless fault-tolerant design methods are employed to mitigate them. Finally, Chapter X presents the conclusions and several directions that future work may take.

Chapter II

BACKGROUND: ALGORITHMS FOR TOEPLITZ MATRIX PROBLEMS

We now present, as background material, a discussion of the origin of Toeplitz matrix problems, and classical algorithms for their solution, as well as a few more modern approaches to the solution of these problems. Later chapters of this thesis will often draw upon the material of this chapter. In particular, we shall see that some of the algorithms to be presented are more amenable to parallel processor implementation than others.

1. Toeplitz Matrix Problems Defined

In this thesis we are interested in the solution of three problems:

(i) Toeplitz matrix inversion and/or LDU factorization.

(ii) The solution of Toeplitz systems of equations.

(iii) The computation of reflection coefficients.

The need to solve these problems arises in many different circumstances, as we'll soon see. A Toeplitz matrix T is simply

$$T = [t_{ij}]_{(n+1)\times(n+1)} = [t_{j-i}]_{(n+1)\times(n+1)} \quad , 0 \le i,j \le n \quad ,$$

where T is $(n+1) \times (n+1)$, and it may be real or complex valued, and t_{ij} is the element in the ith row and jth column of T. Since $t_{ij} = t_{j-i}$, all of the elements along a given diagonal of T are equal to each other: this is the meaning of Toeplitzness. The LDU factorization of T is

$$T = LDU \quad ,$$

where L is a lower triangular matrix, D is a diagonal matrix, and U is an upper triangular matrix. L and U consist entirely of ones on the main diagonal ($U = [u_{ij}]$, $L = [l_{ij}]$, $u_{ii} = l_{ii} = 1$ for all i). Reflection coefficients will be defined later on. We note that reflection coefficients are often alternatively called partial correlation, or PARCOR, coefficients in the literature.

2. Algorithms for the Solution of Toeplitz Matrix Problems

Toeplitz matrix problems have been, and continue to be, the subject of intense study. Some of the reasons for this are considered in section 3. New reasons to be interested in Toeplitz matrix problems emerge periodically. As a result, many algorithms for the solution of Toeplitz matrix problems have appeared over the years, and we will consider some of these algorithms here.

2.1 The Levinson-Durbin Class of Algorithms

Consider the linear system of equations

$$T_n a_n = e_n \quad , \tag{1}$$

where T_n is a complex Toeplitz matrix of dimension $(n+1) \times (n+1)$, and we have the complex vectors

$$\begin{aligned} a_n &= [a_{n,0}\ a_{n,1}\ \cdots\ a_{n,n}]^T \quad , \\ e_n &= [\sigma_n\ 0\ \cdots\ 0]^T \quad , \end{aligned} \tag{2}$$

where a_n is the $(n+1) \times 1$ solution vector. Note that T_n is not necessarily symmetric, or even Hermitian. The Levinson-Durbin algorithm will be used to obtain a_n and σ_n given T_n. We will let $T_n = [t_{j-i}]_{(n+1)\times(n+1)}$ where i is the row index and j is the column index. Note that historically Levinson [1] only considered the case where T_n was real and symmetric, and $a_{n,0} = 1$ (this is merely a normalization convention). Certain extensions were made by Durbin [2] to this original problem. The problem in (1) is a well-known generalization of the early work in [1,2], and is in fact based upon an exercise from the textbook [3] by Roberts and Mullis (see Problem 11.13 on page 551 of [3]).

Define the $(n+1) \times (n+1)$ *exchange matrix* J_n as the matrix consisting entirely of ones along the main antidiagonal but with zeros elsewhere. For example,

$$J_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} .$$

It is easy to see that $J_n^2 = I_n$ ($(n+1)$th order identity matrix). It may also be readily verified that

$$J_n T_n J_n = T_n^T . \tag{3}$$

Property (3) is called *persymmetry* in Blahut [4] and in Datta and Morgera [5] (Note: On page 362 of [4] it is said that $J_n T_n J_n = T_n$, but this is false !). If T_n is symmetric then clearly $J_n T_n J_n = T_n$; this is called *centrosymmetry [5]*.

Define the column vectors

$$t_c = [t_{k+1} \ \cdots \ t_1]^T , \tag{4}$$

$$t_r = [t_{-(k+1)} \ \cdots \ t_{-1}]^T ,$$

and so

$$T_{k+1} = \begin{bmatrix} T_k & t_c \\ t_r^T & t_0 \end{bmatrix} = \begin{bmatrix} t_0 & \hat{t}_c^T \\ \hat{t}_r & T_k \end{bmatrix} , \tag{5}$$

where $\hat{t}_c = J_k t_c$, $\hat{t}_r = J_k t_r$, so that the hat denotes a vector with its elements written in reverse order. We may augment (1) as

$$T_k \ [a_k \ b_k] = [e_k \ \hat{e}_k] . \tag{6}$$

We will let $a_{k,0} = b_{k,0} = 1$ (normalization). If we expand (6) we get

$$T_k \begin{bmatrix} 1 & b_{k,k} \\ a_{k,1} & b_{k,k-1} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ a_{k,k-1} & b_{k,1} \\ a_{k,k} & 1 \end{bmatrix} = \begin{bmatrix} \sigma_k & 0 \\ 0 & 0 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 0 & 0 \\ 0 & \sigma_k \end{bmatrix} . \tag{7}$$

It is clear that

$$T_{k+1} \begin{bmatrix} a_k & 0 \\ 0 & b_k \end{bmatrix} = \begin{bmatrix} e_k & \eta_k \\ \gamma_k & \hat{e}_k \end{bmatrix} , \tag{8}$$

where

$$\eta_k = \hat{t}_c^T b_k = \sum_{i=0}^{k} t_{(k+1)-i} b_{k,i} \ , \ \ \gamma_k = t_r^T a_k = \sum_{i=0}^{k} t_{i-(k+1)} a_{k,i} \ , \tag{9}$$

via (5) and (6). Expanding (8) gives

$$T_{k+1} \begin{bmatrix} 1 & 0 \\ a_{k,1} & b_{k,k} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ a_{k,k} & b_{k,1} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \sigma_k & \eta_k \\ 0 & 0 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 0 & 0 \\ \gamma_k & \sigma_k \end{bmatrix} . \tag{10}$$

Postmultiplying (10) by

$$\begin{bmatrix} 1 & K_{k+1}^r \\ K_{k+1}^f & 1 \end{bmatrix} , \tag{11}$$

where the *reflection coefficients* are

$$K_{k+1}^f = -\frac{\gamma_k}{\sigma_k} \ , \ \ K_{k+1}^r = -\frac{\eta_k}{\sigma_k} \ , \tag{12}$$

yields

$$T_{k+1} \, [a_{k+1} \ b_{k+1}] = [e_{k+1} \ \hat{e}_{k+1}]^T , \tag{13}$$

where

$$\sigma_{k+1} = \sigma_k + \eta_k K_{k+1}^f = \sigma_k + \gamma_k K_{k+1}^r = \sigma_k (1 - K_{k+1}^r K_{k+1}^f) , \tag{14}$$

and

$$\begin{aligned} a_{k+1,i} &= a_{k,i} + K_{k+1}^f b_{k,k+1-i} \ , \ (0 \le i \le k+1) \\ b_{k+1,i} &= b_{k,i} + K_{k+1}^r a_{k,k+1-i} \ , \ (0 \le i \le k+1) \end{aligned} \tag{15}$$

where $a_{k,k+1} = b_{k,k+1} = 0$, $a_{k+1,0} = b_{k+1,0} = 1$. Thus, we have derived the Levinson-Durbin algorithm for solving (1). This derivation is essentially the same as that in Kung and Hu [6].

We may summarize the previous algorithm in the form of pseudocode:

```
$\sigma_0 := t_0$;
$a_{0,0} := 1$;
$b_{0,0} := 1$;
For $k := 0$ to $n-1$ do begin
    $\eta_k := \sum_{i=0}^{k} t_{(k+1)-i} b_{k,i}$;
    $\gamma_k := \sum_{i=0}^{k} t_{i-(k+1)} a_{k,i}$;
    $K_{k+1}^r := -\frac{\eta_k}{\sigma_k}$;
    $K_{k+1}^f := -\frac{\gamma_k}{\sigma_k}$;
    $\sigma_{k+1} := \sigma_k(1 - K_{k+1}^f K_{k+1}^r)$;
    For $i := 0$ to $k+1$ do begin
        $a_{k+1,i} := a_{k,i} + K_{k+1}^f b_{k,k+1-i}$;
        $b_{k+1,i} := b_{k,i} + K_{k+1}^r a_{k,k+1-i}$;
        end;
    end;
```

The input to this algorithm is T_n from (1). Clearly, the algorithm has a time complexity of $O(n^2)$ on a sequential processor. This is a more efficient means of solving (1) than Gaussian elimination which requires $O(n^3)$ operations on a sequential processor.

The Levinson-Durbin algorithm yields a *UDL* decomposition of T_n^{-1}. This may be demonstrated as follows. Via (10) we may write

$$T_k U_k^B = \Sigma_k^L , \quad L_k^A T_k = \Sigma_k^U , \tag{16}$$

where.

$$U_k^B = \begin{bmatrix} 1 & b_{1,1} & \cdot\cdot & b_{k-1,k-1} & b_{k,k} \\ 0 & 1 & \cdot\cdot & b_{k-1,k-2} & b_{k,k-1} \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & \cdot\cdot & 1 & b_{k,1} \\ 0 & 0 & \cdot\cdot & 0 & 1 \end{bmatrix}, \; \Sigma_k^L = \begin{bmatrix} \sigma_0 & 0 & \cdot\cdot & 0 & 0 \\ \times & \sigma_1 & \cdot\cdot & 0 & 0 \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ \times & \times & \cdot\cdot & \sigma_{k-1} & 0 \\ \times & \times & \cdot\cdot & \times & \sigma_k \end{bmatrix}, \tag{17a}$$

$$L_k^A = \begin{bmatrix} 1 & 0 & \cdot\cdot & 0 & 0 \\ a_{1,1} & 1 & \cdot\cdot & 0 & 0 \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ a_{k-1,k-1} & a_{k-1,k-2} & \cdot\cdot & 1 & 0 \\ a_{k,k} & a_{k,k-1} & \cdot\cdot & a_{k,1} & 1 \end{bmatrix}, \; \Sigma_k^U = \begin{bmatrix} \sigma_0 & \times & \cdot\cdot & \times & \times \\ 0 & \sigma_1 & \cdot\cdot & \times & \times \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & \cdot\cdot & \sigma_{k-1} & \times \\ 0 & 0 & \cdot\cdot & 0 & \sigma_k \end{bmatrix}, \tag{17b}$$

where the entries marked '×' denote 'don't care' entries. The second equation in (16) follows from the fact that $J_k T_k J_k J_k a_k = \hat{e}_k$ which implies that $T_k^T \hat{a}_k = \hat{e}_k$ so $\hat{a}_k^T T_k = \hat{e}_k^T$, or

$$[a_{k,k} \; a_{k,k-1} \; \cdots \; a_{k,1} \; 1] \, T_k = [0 \; 0 \; \cdots \; 0 \; \sigma_k] \; . \tag{18}$$

From (16)

$$L_k^A T_k U_k^B = L_k^A \Sigma_k^L = \Sigma_k^U U_k^B = D_k = diag \left\{ \sigma_0 \; \sigma_1 \; \cdots \; \sigma_{k-1} \; \sigma_k \right\} , \tag{19}$$

since $L_k^A \Sigma_k^L$ is lower triangular and $\Sigma_k^U U_k^B$ is upper triangular. Thus,

$$T_k = (L_k^A)^{-1} D_k (U_k^B)^{-1} , \tag{20}$$

or

$$T_k^{-1} = U_k^B D_k^{-1} L_k^A \; . \tag{21}$$

Equation (21) is equation (2.8) in Kung and Hu [6]. Hence, given T_k and y_k in $T_k x_k = y_k$ we may solve for x_k via (21). This will take $O(k^2)$ operations.

From (20), the determinant of T_k is given by

$$det(T_k) = \prod_{i=0}^{k} \sigma_i , \tag{22}$$

and from (22)

$$\sigma_k = \frac{det(T_k)}{det(T_{k-1})} \quad . \tag{23}$$

Thus, it is clear that the Levinson-Durbin algorithm will compute T_n^{-1} if and only if $det(T_k) \neq 0$ for $k = 0,...,n$. This implies that the Levinson-Durbin algorithm must be terminated if and when $\sigma_k = 0$. If T_n has singular leading principal submatrices, and yet T_n^{-1} exists, then the algorithm of Rissanen [7] may be used to compute T_n^{-1} in $O(n^2)$ time on a sequential processor. If T_n is Hermitian, so that $T_n = T_n^H$ (H denotes the Hermitian (complex-conjugate) transpose) then Delsarte, Genin and Kamp [8] have shown how to modify the Levinson-Durbin algorithm to accommodate the singular submatrix case.

It is worth considering the special case of Hermitian Toeplitz matrices separately, since some simplifications in the previously presented Levinson-Durbin algorithm arise, and this special case is particularly useful in practice. If T_n is Hermitian then

$$J_n T_n J_n = \overline{T}_n \quad , \tag{24}$$

where the bar denotes conjugation, and it is clearly true that $t_{-i} = \overline{t}_i$. This is the *centro-Hermitian symmetry* property [5]. Because T_n is Hermitian and persymmetric it is called *Hermitian persymmetric* [5].

As before, $T_k a_k = e_k = [\sigma_k \ 0 \ \cdots \ 0]^T$ so that

$$T_{k+1} \begin{bmatrix} a_k \\ 0 \end{bmatrix} = \begin{bmatrix} e_k \\ \gamma_k \end{bmatrix} \quad , \tag{25}$$

where

$$\gamma_k = \sum_{i=0}^{k} \overline{t}_{k+1-i}\, a_{k,i} \quad . \tag{26}$$

Since

$$J_{k+1} T_{k+1} \begin{bmatrix} a_k \\ 0 \end{bmatrix} = J_{k+1} \begin{bmatrix} e_k \\ \gamma_k \end{bmatrix} = \begin{bmatrix} \gamma_k \\ \hat{e}_k \end{bmatrix}$$

we can use (24) to write

$$\overline{T}_{k+1}J_{k+1}\begin{bmatrix} a_k \\ 0 \end{bmatrix} = \overline{T}_{k+1}\begin{bmatrix} 0 \\ \hat{a}_k \end{bmatrix} = \begin{bmatrix} \gamma_k \\ \hat{e}_k \end{bmatrix}$$

which may be conjugated to yield

$$T_{k+1}\begin{bmatrix} 0 \\ \bar{\hat{a}}_k \end{bmatrix} = \begin{bmatrix} \overline{\gamma}_k \\ \bar{\hat{e}}_k \end{bmatrix} . \qquad (27)$$

Combining (27) with (25) yields

$$T_{k+1}a_{k+1} = T_{k+1}\left\{\begin{bmatrix} a_k \\ 0 \end{bmatrix} + K_{k+1}\begin{bmatrix} 0 \\ \bar{\hat{a}}_k \end{bmatrix}\right\} = \begin{bmatrix} e_k \\ \gamma_k \end{bmatrix} + K_{k+1}\begin{bmatrix} \overline{\gamma}_k \\ \bar{\hat{e}}_k \end{bmatrix} = e_{k+1} , \qquad (28)$$

where

$$K_{k+1} = -\frac{\gamma_k}{\overline{\sigma}_k} ,$$

and so

$$\sigma_{k+1} = \sigma_k + K_{k+1}\overline{\gamma}_k = \sigma_k(1 - |\, K_{k+1} \,|^2) . \qquad (29)$$

Clearly, σ_k is real for all k, since $\sigma_0 = t_0$ is real and $|\, K_{k+1} \,|$ is the magnitude of K_{k+1}. Thus, the $(k+1)$th reflection coefficient is

$$K_{k+1} = -\frac{\gamma_k}{\sigma_k} . \qquad (30)$$

From (28),

$$a_{k+1,i} = a_{k,i} + K_{k+1}\overline{a}_{k,k+1-i} , \quad (0 \le i \le k+1) \qquad (31)$$

where $a_{k,0} = 1$, $a_{k,k+1} = 0$ for all $k = 0,1,...,n-1$.

Thus we may summarize the Levinson-Durbin algorithm for Hermitian Toeplitz matrices as follows:

```
$\sigma_0 := t_0$;
$a_{0,0} := 1$;
For $k := 0$ to $n-1$ do begin
    $\gamma_k := \sum_{i=0}^{k} \overline{t}_{(k+1)-i}\, a_{k,i}$;
```

```
$K_{k+1} := -\frac{\gamma_k}{\sigma_k}$;
$\sigma_{k+1} := \sigma_k(1 - |K_{k+1}|^2)$;
For $i := 0$ to $k+1$ do begin
        $a_{k+1,i} := a_{k,i} + K_{k+1}\overline{a}_{k,k+1-i}$;
        end;
end;
```

The case where T_n is real and symmetric follows trivially. This case is also discussed in Roberts and Mullis [3] (see pp. 520-523).

Since $T_k a_k = e_k$, $\overline{T}_k \hat{a}_k = \hat{e}_k$ (via (24)), and so $\hat{a}_k^T \overline{T}_k^T = \hat{e}_k^T$ which is $\hat{a}_k^T T_k = \hat{e}_k^T$ and this yields

$$L_k^A T_k = \Sigma_k^U \ , \tag{32}$$

where L_k^A and Σ_k^U have the same form as in (17b). As well, $T_k \overline{\hat{a}}_k = \overline{\hat{e}}_k$ yields

$$T_k (L_k^A)^H = \Sigma_k^L \ , \tag{33}$$

where Σ_k^L has the same form as in (17a). Combining (32) and (33) gives

$$L_k^A T_k (L_k^A)^H = \Sigma_k^U (L_k^A)^H = L_k^A \Sigma_k^L = D_k = diag\left\{\sigma_0 \ \cdots \ \sigma_k\right\} . \tag{34}$$

Thus,

$$T_k^{-1} = (L_k^A)^H D_k^{-1} L_k^A \ . \tag{35}$$

The relations in (22) and (23) continue to hold.

The Levinson-Durbin algorithm has been derived and extended in various alternative ways not covered here. Other derivations and extensions may be found in Blahut [4], Hönig and Messerschmitt [9], Friedlander [10], Carayannis [11], Robinson and Treitel [12], Bruckstein and Kailath [13], and in Markel and Gray [14].

2.2 The Trench Algorithm and the Gohberg-Semencul Formula

Trench's algorithm [15] is another popular "classical" approach to the solution of Toeplitz systems of equations and Toeplitz matrix inversion. Zohar [16] argues that

Trench's original derivation is unnecessarily complex and so Zohar presents a simplified derivation. Various extensions to the original Trench algorithm are to be found in Akaike [17] and Zohar [18]. The derivation that we present below is essentially that of Zohar [16]. Note that the Trench algorithm also presumes that the leading principal submatrices of T_n are nonsingular. This is called *strong nonsingularity* in Zohar [16].

Define the column vectors

$$t_{k+1}^c = [t_{k+1} \quad \cdots \quad t_1]^T \ , \tag{36}$$

$$t_{k+1}^r = [t_{-(k+1)} \quad \cdots \quad t_{-1}]^T \ ,$$

so that

$$T_{k+1} = \begin{bmatrix} T_k & t_{k+1}^c \\ (t_{k+1}^r)^T & t_0 \end{bmatrix} . \tag{37}$$

Define

$$B_{k+1} = T_{k+1}^{-1} = \begin{bmatrix} M_k & b_{k+1}^c \\ (b_{k+1}^r)^T & b_{k+1}^0 \end{bmatrix} . \tag{38}$$

Thus,

$$T_{k+1}B_{k+1} = \begin{bmatrix} T_kM_k+t_{k+1}^c(b_{k+1}^r)^T & T_kb_{k+1}^c+b_{k+1}^0t_{k+1}^c \\ (t_{k+1}^r)^TM_k+t_0(b_{k+1}^r)^T & (t_{k+1}^r)^Tb_{k+1}^c+b_{k+1}^0t_0 \end{bmatrix} = \begin{bmatrix} I_k & 0 \\ 0 & 1 \end{bmatrix} = I_{k+1} \ , \tag{39}$$

which implies that

$$T_kM_k + t_{k+1}^c(b_{k+1}^r)^T = I_k \ , \tag{40a}$$

$$T_kb_{k+1}^c + b_{k+1}^0t_{k+1}^c = 0 \ , \tag{40b}$$

$$(t_{k+1}^r)^TM_k + t_0(b_{k+1}^r)^T = 0 \ , \tag{40c}$$

$$(t_{k+1}^r)^Tb_{k+1}^c + b_{k+1}^0t_0 = 1 \ . \tag{40d}$$

Premultiplying (40a,b) by B_k gives

$$M_k = B_k - B_kt_{k+1}^c(b_{k+1}^r)^T \ ,$$

$$b_{k+1}^{c} = -b_{k+1}^{0} B_k t_{k+1}^{c}$$

which combine to yield

$$M_k = B_k + \frac{b_{k+1}^{c} (b_{k+1}^{r})^T}{b_{k+1}^{0}} \quad . \tag{41}$$

Substituting (41) into (38) gives us

$$B_{k+1} = \begin{bmatrix} B_k + \dfrac{b_{k+1}^{c} (b_{k+1}^{r})^T}{b_{k+1}^{0}} & b_{k+1}^{c} \\ (b_{k+1}^{r})^T & b_{k+1}^{0} \end{bmatrix} \quad . \tag{42}$$

We know that $J_k T_k J_k = T_k^T$ and so $J_k T_k^{-1} J_k = T_k^{-T}$ which implies that $J_k B_k J_k = B_k^T$, i.e., B_k is persymmetric as is T_k. This fact, combined with (42), enables us to write

$$B_{k+1} = \begin{bmatrix} b_{k+1}^{0} & (\hat{b}_{k+1}^{c})^T \\ \hat{b}_{k+1}^{r} & B_k + \dfrac{\hat{b}_{k+1}^{r} (\hat{b}_{k+1}^{c})^T}{b_{k+1}^{0}} \end{bmatrix} \quad . \tag{43}$$

If we let $B_k = [b_{ij,k}]_{(k+1)\times(k+1)}$ then via (42)

$$b_{ij,k+1} = b_{ij,k} + \frac{1}{b_{k+1}^{0}} [b_{k+1}^{c} (b_{k+1}^{r})^T]_{ij} \ , \quad (0 \le i,j \le k) \tag{44}$$

where $[X]_{ij}$ means 'component ij of the matrix X'. From (43),

$$b_{i+1,j+1,k+1} = b_{ij,k} + \frac{1}{b_{k+1}^{0}} [\hat{b}_{k+1}^{r} (\hat{b}_{k+1}^{c})^T]_{ij} \ , \quad (0 \le i,j \le k) \ . \tag{45}$$

Subtracting (44) from (45) produces

$$b_{i+1,j+1,k+1} = b_{ij,k+1} + \frac{1}{b_{k+1}^{0}} [\hat{b}_{k+1}^{r} (\hat{b}_{k+1}^{c})^T - b_{k+1}^{c} (b_{k+1}^{r})^T]_{ij} \quad , \tag{46}$$

and using (44) and (45) again we get

$$b_{i-1,j-1,k+1} = b_{ij,k+1} + \frac{1}{b_{k+1}^{0}} [b_{k+1}^{c} (b_{k+1}^{r})^T - \hat{b}_{k+1}^{r} (\hat{b}_{k+1}^{c})^T]_{i-1,j-1} \quad , \tag{47}$$

where in this instance $1 \le i,j \le k+1$.

From (40b)

$$J_k T_k J_k J_k b^c_{k+1} + b^0_{k+1} J_k t^c_{k+1} = 0$$

or

$$T^T_k \hat{b}^c_{k+1} + b^0_{k+1} \hat{t}^c_{k+1} = 0 \, ,$$

and so we have

$$\hat{b}^c_{k+1} = -b^0_{k+1} B^T_k \hat{t}^c_{k+1} \tag{48}$$

which may be expanded as

$$\hat{b}^c_{k+1} = -b^0_{k+1} \begin{bmatrix} B^T_{k-1} + \dfrac{b^r_k (b^c_k)^T}{b^0_k} & b^r_k \\ (b^c_k)^T & b^0_k \end{bmatrix} \begin{bmatrix} \hat{t}^c_k \\ t_{k+1} \end{bmatrix}$$

$$= -b^0_{k+1} \begin{bmatrix} -\dfrac{\hat{b}^c_k}{b^0_k} + \dfrac{b^r_k (b^c_k)^T}{b^0_k} \hat{t}^c_k + b^r_k t_{k+1} \\ (b^c_k)^T \hat{t}^c_k + b^0_k t_{k+1} \end{bmatrix} .$$

This can be rewritten as

$$\frac{\hat{b}^c_{k+1}}{b^0_{k+1}} = \begin{bmatrix} \dfrac{\hat{b}^c_k}{b^0_k} \\ 0 \end{bmatrix} - \begin{bmatrix} \dfrac{b^r_k (b^c_k)^T}{b^0_k} & b^r_k \\ (b^c_k)^T & b^0_k \end{bmatrix} \begin{bmatrix} \hat{t}^c_k \\ t_{k+1} \end{bmatrix} . \tag{49}$$

If we compute the product $B_{k+1} T_{k+1}$ using (37) and (38) then we get

$$(b^r_{k+1})^T T_k + b^0_{k+1} (t^r_{k+1})^T = 0 \; . \tag{50}$$

From (50)

$$J_k T^T_k J_k J_k b^r_{k+1} + b^0_{k+1} J_k t^r_{k+1} = 0$$

or

$$T_k \hat{b}^r_{k+1} + b^0_{k+1} \hat{t}^r_{k+1} = 0$$

which implies that

$$\hat{b}^r_{k+1} = -b^0_{k+1} B_k \hat{t}^r_{k+1} \quad . \tag{51}$$

We may expand (51) to get

$$\hat{b}^r_{k+1} = -b^0_{k+1} \begin{bmatrix} B_{k-1} + \dfrac{b^c_k (b^r_k)^T}{b^0_k} & b^c_k \\ (b^r_k)^T & b^0_k \end{bmatrix} \begin{bmatrix} \hat{t}^r_k \\ t_{-(k+1)} \end{bmatrix}$$

$$= -b^0_{k+1} \begin{bmatrix} -\dfrac{\hat{b}^r_k}{b^0_k} + \dfrac{b^c_k (b^r_k)^T}{b^0_k} \hat{t}^r_k + b^c_k t_{-(k+1)} \\ (b^r_k)^T \hat{t}^r_k + b^0_k t_{-(k+1)} \end{bmatrix} .$$

This may be rewritten as

$$\frac{\hat{b}^r_{k+1}}{b^0_{k+1}} = \begin{bmatrix} \dfrac{\hat{b}^r_k}{b^0_k} \\ 0 \end{bmatrix} - \begin{bmatrix} \dfrac{b^c_k (b^r_k)^T}{b^0_k} & b^c_k \\ (b^r_k)^T & b^0_k \end{bmatrix} \begin{bmatrix} \hat{t}^r_k \\ t_{-(k+1)} \end{bmatrix} . \tag{52}$$

Let us define certain normalized variables namely

$$c^c_k = \frac{b^c_k}{b^0_k} \; , \; c^r_k = \frac{b^r_k}{b^0_k} \; , \; \lambda_k = \frac{1}{b^0_k} \quad . \tag{53}$$

We can use these definitions along with (49) and (52) to write

$$\hat{c}^c_{k+1} = \begin{bmatrix} \hat{c}^c_k \\ 0 \end{bmatrix} - \frac{(c^c_k)^T \hat{t}^c_k + t_{k+1}}{\lambda_k} \begin{bmatrix} c^r_k \\ 1 \end{bmatrix} , \tag{54a}$$

$$\hat{c}^r_{k+1} = \begin{bmatrix} \hat{c}^r_k \\ 0 \end{bmatrix} - \frac{(c^r_k)^T \hat{t}^r_k + t_{-(k+1)}}{\lambda_k} \begin{bmatrix} c^c_k \\ 1 \end{bmatrix} . \tag{54b}$$

Now, all we need is a recursion for λ_k.

From (42) and (43)

$$b^0_{k+1} = b_{00,k} + \frac{1}{b^0_{k+1}} [b^c_{k+1} (b^r_{k+1})^T]_{00} \, , \tag{55}$$

but $b_{00,k} = b^0_k$. From (49) and (52)

$$b_{k+1,0}^{c} = -b_{k+1}^{0}\ [(b_k^c)^T \hat{t}_k^c + b_k^0 t_{k+1}]\ , \tag{56}$$

$$b_{k+1,0}^{r} = -b_{k+1}^{0}\ [(b_k^r)^T \hat{t}_k^r + b_k^0 t_{-(k+1)}]\ ,$$

which are the first elements of the vectors b_{k+1}^c and b_{k+1}^r, respectively. Thus, substituting (56) into (55) yields

$$\begin{aligned} b_{k+1}^0 &= b_k^0 + b_{k+1}^0 [(b_k^c)^T \hat{t}_k^c + b_k^0 t_{k+1}]\ [(b_k^r)^T \hat{t}_k^r + b_k^0 t_{-(k+1)}] \\ &= b_k^0 + b_{k+1}^0 (b_k^0)^2 [(c_k^c)^T \hat{t}_k^c + t_{k+1}]\ [(c_k^r)^T \hat{t}_k^r + t_{-(k+1)}] \end{aligned}$$

which reduces to

$$\lambda_{k+1} = \lambda_k - \frac{[(c_k^r)^T \hat{t}_k^r + t_{-(k+1)}]\ [(c_k^c)^T \hat{t}_k^c + t_{k+1}]}{\lambda_k}\ , \tag{57}$$

and this is the desired recursion for λ_k. Equations (46) (or (47)), (54a,b), (57) and the definitions in (53) constitute the Trench algorithm. Simply iterate using (54a,b) and (57) until $k = n-1$ and then use (53) and (46) (or (47)). The time complexity of the Trench algorithm is $O(n^2)$ on a sequential processor.

We may initialize the Trench algorithm in the following manner. Consider the case $n = 1$, then

$$T_1 = \begin{bmatrix} t_0 & t_1 \\ t_{-1} & t_0 \end{bmatrix},\ B_1 = \frac{1}{t_0^2 - t_{-1}t_1} \begin{bmatrix} t_0 & -t_1 \\ -t_{-1} & t_0 \end{bmatrix}.$$

Thus,

$$\lambda_1 = t_0 - \frac{t_{-1}t_1}{t_0}\ ,\ c_1^r = -\frac{t_{-1}}{t_0}\ ,\ c_1^c = -\frac{t_1}{t_0}\ , \tag{58a}$$

or, alternatively,

$$b_1^r = -\frac{t_{-1}}{t_0^2 - t_{-1}t_1}\ ,\ b_1^c = -\frac{t_1}{t_0^2 - t_{-1}t_1}\ . \tag{58b}$$

We now wish to obtain the *Gohberg-Semencul formula* [19], which is an alternative means of writing T_n^{-1}. The Gohberg-Semencul formula expresses T_n^{-1} as a sum of products of upper and lower triangular Toeplitz matrices. This fact proves useful in certain applications to be discussed later on in this chapter.

We begin by noting that

$$B_k = U_k^B D_k^{-1} L_k^A = \begin{bmatrix} \times & \times & \cdot\ \cdot & \times & \sigma_k^{-1} b_{k,k} \\ \times & \times & \cdot\ \cdot & \times & \sigma_k^{-1} b_{k,k-1} \\ \cdot & \cdot & & \cdot & \vdots \\ \cdot & \cdot & & \cdot & \cdot \\ \times & \times & \cdot\ \cdot & \times & \sigma_k^{-1} b_{k,1} \\ \sigma_k^{-1} a_{k,k} & \sigma_k^{-1} a_{k,k-1} & \cdot\ \cdot & \sigma_k^{-1} a_{k,1} & \sigma_k^{-1} \end{bmatrix} , \tag{59}$$

where we have used (17a,b) and (21). Comparing (59) with (42) reveals that

$$b_k^0 = \sigma_k^{-1} \ , \tag{60a}$$

$$b_k^c = \sigma_k^{-1} \, [b_{k,k} \ b_{k,k-1} \ \cdots \ b_{k,2} \ b_{k,1}]^T \ , \tag{60b}$$

$$b_k^r = \sigma_k^{-1} \, [a_{k,k} \ a_{k,k-1} \ \cdots \ a_{k,2} \ a_{k,1}]^T \ . \tag{60c}$$

As well, $\lambda_k = \sigma_k$ via (53). Therefore, the Trench algorithm provides us with the same information about T_n as the Levinson-Durbin algorithm, but in a different way. This turns out to provide useful insight into the structure of T_n^{-1} as we shall now see.

From (46) and (60a,b,c)

$$b_{i+1,j+1,k} = b_{ij,k} + \sigma_k^{-1} \, [a_{k,i+1} b_{k,j+1} - b_{k,k-i} a_{k,k-j}] \ , \tag{61}$$

where $b_{i0,k} = \sigma_k^{-1} a_{k,i}$, $b_{0j,k} = \sigma_k^{-1} b_{k,j}$ $(a_{k,0} = b_{k,0} = 1)$ from (43), and $0 \le i,j \le k-1$. From (61) we may write

$$\sigma_k T_k^{-1} = \begin{bmatrix} 1 & 0 & . & 0 & 0 \\ a_{k,1} & 1 & . & 0 & 0 \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ a_{k,k-1} & a_{k,k-2} & . & 1 & 0 \\ a_{k,k} & a_{k,k-1} & . & a_{k,1} & 1 \end{bmatrix} \begin{bmatrix} 1 & b_{k,1} & . & b_{k,k-1} & b_{k,k} \\ 0 & 1 & . & b_{k,k-2} & b_{k,k-1} \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & . & 1 & b_{k,1} \\ 0 & 0 & . & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 & . & 0 & 0 \\ b_{k,k} & 0 & . & 0 & 0 \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ b_{k,2} & b_{k,3} & . & 0 & 0 \\ b_{k,1} & b_{k,2} & . & b_{k,k} & 0 \end{bmatrix} \begin{bmatrix} 0 & a_{k,k} & . & a_{k,2} & a_{k,1} \\ 0 & 0 & . & a_{k,3} & a_{k,2} \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & . & 0 & a_{k,k} \\ 0 & 0 & . & 0 & 0 \end{bmatrix} . \tag{62}$$

This is one form of the Gohberg-Semencul formula. Other forms may be found in Kailath, Vieira and Morf [20]. We may write (62) compactly as

$$T_k^{-1} = \sigma_k^{-1} \, (L_a^f U_b^f - L_b^r U_a^r) \ , \tag{63}$$

where L_a^f and L_b^r are lower triangular Toeplitz matrices with first columns

$$[1 \ a_{k,1} \ \cdots \ a_{k,k-1} \ a_{k,k}]^T \text{ , and } [0 \ b_{k,k} \ \cdots \ b_{k,2} \ b_{k,1}]^T \ ,$$

respectively, and U_b^f and U_a^r are upper triangular Toeplitz matrices with first rows

$$[1 \; b_{k,1} \; \cdots \; b_{k,k-1} \; b_{k,k}] \text{ , and } [0 \; a_{k,k} \; \cdots \; a_{k,2} \; a_{k,1}] \quad ,$$

respectively.

2.3 The Schur Algorithm

We will now derive the Schur algorithm for T_n in (1) via the method presented in Kung and Hu [6]. We will illustrate the method for the special case of $n = 3$ as was done in [6]. The extension to arbitrary n is straightforward.

Consider the augmented form of T_3, where we will let $T = T_3$,

$$\tilde{T} = \left[\begin{array}{ccc|cccc} t_{-3} & t_{-2} & t_{-1} & t_0 & t_1 & t_2 & t_3 \\ 0 & t_{-3} & t_{-2} & t_{-1} & t_0 & t_1 & t_2 \\ 0 & 0 & t_{-3} & t_{-2} & t_{-1} & t_0 & t_1 \\ 0 & 0 & 0 & t_{-3} & t_{-2} & t_{-1} & t_0 \end{array}\right] , \tag{64}$$

and so T is the rightmost four columns of $\tilde{T}$. We wish to compute $(L^*)^{-1}$ and U in $(L^*)^{-1}\tilde{T} = [X \mid U]$, where this may be expanded as follows

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{10} & 1 & 0 & 0 \\ l_{20} & l_{21} & 1 & 0 \\ l_{30} & l_{31} & l_{32} & 1 \end{bmatrix} \tilde{T} = \left[\begin{array}{ccc|cccc} \times & \times & \times & u_{00} & u_{01} & u_{02} & u_{03} \\ \times & \times & \times & 0 & u_{11} & u_{12} & u_{13} \\ \times & \times & \times & 0 & 0 & u_{22} & u_{23} \\ \times & \times & \times & 0 & 0 & 0 & u_{33} \end{array}\right] . \tag{65}$$

We may write

$$\begin{bmatrix} 1 & K_1^r & 0 & 0 \\ K_1^f & 1 & 0 & 0 \end{bmatrix} \tilde{T} = \left[\begin{array}{ccc|cccc} v_3^{(2)} & v_2^{(2)} & v_1^{(2)} & v_0^{(2)} & 0 & v_{-2}^{(2)} & v_{-3}^{(2)} \\ u_3^{(2)} & u_2^{(2)} & u_1^{(2)} & 0 & u_{-1}^{(2)} & u_{-2}^{(2)} & u_{-3}^{(2)} \end{array}\right] , \tag{66a}$$

where

$$K_1^f = -\frac{t_{-1}}{t_0} \, , \quad K_1^r = -\frac{t_1}{t_0} \, . \tag{66b}$$

The first rows of $(L^*)^{-1}$ and U are obviously

$$[1 \; 0 \; 0 \; 0] \text{ , and } [t_0 \; t_1 \; t_2 \; t_3] \quad ,$$

respectively. From (66a) the second rows of $(L^*)^{-1}$ and U are, respectively,

$$[K_1^f \; 1 \; 0 \; 0] \text{ , and } [0 \; u_{-1}^{(2)} \; u_{-2}^{(2)} \; u_{-3}^{(2)}] \quad .$$

Because T is Toeplitz (and so is $\tilde{T}$), we can shift the second row of (66a) to get

$$\begin{bmatrix} 1 & K_1^r & 0 & 0 \\ 0 & K_1^\ell & 1 & 0 \end{bmatrix} \tilde{T} = \left[\begin{array}{ccc|cccc} v_3^{(2)} & v_2^{(2)} & v_1^{(2)} & v_0^{(2)} & 0 & v_{-2}^{(2)} & v_{-3}^{(2)} \\ 0 & u_3^{(2)} & u_2^{(2)} & u_1^{(2)} & 0 & u_{-1}^{(2)} & u_{-2}^{(2)} \end{array}\right] . \tag{67}$$

Thus,

$$\begin{bmatrix} 1 & K_2^r \\ K_2^\ell & 1 \end{bmatrix} \begin{bmatrix} 1 & K_1^r & 0 & 0 \\ 0 & K_1^\ell & 1 & 0 \end{bmatrix} \tilde{T} = \left[\begin{array}{ccc|cccc} v_3^{(3)} & v_2^{(3)} & v_1^{(3)} & v_0^{(3)} & 0 & 0 & v_{-3}^{(3)} \\ u_3^{(3)} & u_2^{(3)} & u_1^{(3)} & 0 & 0 & u_{-2}^{(3)} & u_{-3}^{(3)} \end{array}\right] , \tag{68a}$$

where

$$K_2^\ell = -\frac{u_1^{(2)}}{v_0^{(2)}} , \quad K_2^r = -\frac{v_{-2}^{(2)}}{u_{-1}^{(2)}} . \tag{68b}$$

We have

$$\begin{bmatrix} 1 & K_2^r \\ K_2^\ell & 1 \end{bmatrix} \begin{bmatrix} 1 & K_1^r & 0 & 0 \\ 0 & K_1^\ell & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & K_1^r + K_1^\ell K_2^r & K_2^r & 0 \\ K_2^\ell & K_1^\ell + K_1^r K_2^\ell & 1 & 0 \end{bmatrix} ,$$

and so the third rows of $(L^*)^{-1}$ and U are, respectively,

$$[K_2^\ell \;\; K_1^\ell + K_1^r K_2^\ell \;\; 1 \;\; 0] \text{ , and } [0 \;\; 0 \;\; u_{-2}^{(3)} \;\; u_{-3}^{(3)}] \; .$$

We may repeat the preceding process one final time producing

$$\begin{bmatrix} 1 & K_3^r \\ K_3^\ell & 1 \end{bmatrix} \begin{bmatrix} 1 & K_1^r + K_1^\ell K_2^r & K_2^r & 0 \\ 0 & K_2^\ell & K_1^\ell + K_1^r K_2^\ell & 1 \end{bmatrix} \tilde{T} = \left[\begin{array}{ccc|cccc} v_3^{(4)} & v_2^{(4)} & v_1^{(4)} & v_0^{(4)} & 0 & 0 & 0 \\ u_3^{(4)} & u_2^{(4)} & u_1^{(4)} & 0 & 0 & 0 & u_{-3}^{(4)} \end{array}\right] , \tag{69a}$$

where

$$K_3^\ell = -\frac{u_1^{(3)}}{v_0^{(3)}} , \quad K_3^r = -\frac{v_{-3}^{(3)}}{u_{-2}^{(3)}} , \tag{69b}$$

and so the final (fourth) rows of $(L^*)^{-1}$ and U are, respectively,

$$[K_3^\ell \;\; K_2^\ell + K_3^\ell (K_1^r + K_1^\ell K_2^r) \;\; K_1^\ell + K_1^r K_2^\ell + K_2^r K_3^\ell \;\; 1] \text{ , and } [0 \;\; 0 \;\; 0 \;\; u_{-3}^{(4)}] \; .$$

Note that we have again employed shifting to obtain (69a).

From the preceding, we may write the Schur algorithm in the form of pseudocode:

```
For $i := -n$ to $n$ do begin
    $v_i^{(1)} := t_{-i}$ ;  $u_i^{(1)} := t_{-i}$;
    end;
```

```
For $k$ := 1 to $n$ do begin
    $K_k^f := -\frac{u_1^{(k)}}{v_0^{(k)}}$;
    $K_k^r := -\frac{v_{-k}^{(k)}}{u_{-k+1}^{(k)}}$;
    For $i$ := $-n$ to $n$ do begin
        $u_i^{(k+1)} := K_k^f v_i^{(k)} + u_{i+1}^{(k)}$;
        $v_i^{(k+1)} := v_i^{(k)} + K_k^r u_{i+1}^{(k)}$;
    end;
end;
```

Since we have $(L^*)^{-1}T = U$, it is possible to write

$$(L_n^*)^{-1}T_n = \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & u_{-2}^{(1)} & \cdots & u_{-(n-1)}^{(1)} & u_{-n}^{(1)} \\ 0 & u_{-1}^{(2)} & u_{-2}^{(2)} & \cdots & u_{-(n-1)}^{(2)} & u_{-n}^{(2)} \\ 0 & 0 & u_{-2}^{(3)} & \cdots & u_{-(n-1)}^{(3)} & u_{-n}^{(3)} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & u_{-n}^{(n+1)} \end{bmatrix} . \tag{70}$$

But what is L_n^* ?

Suppose $T_n = L_n D_n^{-1} U_n$, where $L_n = L_n^* D_n$, and $U_n = D_n U_n^*$. Thus the proper LDU factorization of T_n is $T_n = L_n^* D_n U_n^*$. Note that the asterisk denotes a triangular matrix with the main diagonal consisting entirely of ones. Define $\dot{T}_n = T_n^T$, and so $\dot{T}_n = \dot{L}_n \dot{D}_n^{-1} \dot{U}_n$, and since $T_n^T = U_n^T D_n^{-1} L_n^T$, we conclude that

$$\dot{L}_n = U_n^T \, , \ \dot{D}_n = D_n \, , \ \dot{U}_n = L_n^T \, . \tag{71}$$

Running the Schur algorithm with $\dot{T}_n$ as the input produces variables $\dot{K}_k^f$, $\dot{K}_k^r$, $\dot{v}_i^{(k)}$, and $\dot{u}_i^{(k)}$ that may be related to the variables K_k^f , K_k^r , $v_i^{(k)}$, and $u_i^{(k)}$ produced by the Schur algorithm when T_n is the input. We will now find this relationship, and in so doing we will answer the question of the previous paragraph.

It is clear that $\dot{t}_i = t_{-i}$ where $\dot{t}_i$ is an element of $\dot{T}_n$. For $k = 1$, from the Schur algorithm we have

$$\dot{K}_1^f = -\frac{\dot{u}_1^{(1)}}{\dot{v}_0^{(1)}} = -\frac{t_1}{t_0} = K_1^r \quad ,$$

$$\dot{K}_1^r = -\frac{\dot{v}_{-1}^{(1)}}{\dot{u}_0^{(1)}} = -\frac{t_{-1}}{t_0} = K_1^f \quad .$$

Thus we surmise that $\dot{K}_k^f = K_k^r$ and $\dot{K}_k^r = K_k^f$. In fact, we also have

$$\dot{v}_i^{(k)} = u_{-i-k+1}^{(k)} \ , \ \dot{u}_i^{(k)} = v_{-i-k+1}^{(k)} \quad , \tag{72}$$

and this may be proven inductively as follows. Suppose that (72) holds for k, so then

$$\dot{K}_k^f = -\frac{\dot{u}_1^{(k)}}{\dot{v}_0^{(k)}} = -\frac{v_{-k}^{(k)}}{u_{-k+1}^{(k)}} = K_k^r \quad ,$$

$$\dot{K}_k^r = -\frac{\dot{v}_{-k}^{(k)}}{\dot{u}_{-k+1}^{(k)}} = -\frac{u_1^{(k)}}{v_0^{(k)}} = K_k^f \quad .$$

As well,

$$\begin{aligned} \dot{u}_i^{(k+1)} &= \dot{K}_k^f \dot{v}_i^{(k)} + \dot{u}_{i+1}^{(k)} \\ &= K_k^r u_{-i-k+1}^{(k)} + v_{-i-k}^{(k)} = v_{-i-(k+1)+1}^{(k+1)} \quad , \\ \dot{v}_i^{(k+1)} &= \dot{v}_i^{(k)} + \dot{K}_k^r \dot{u}_{i+1}^{(k)} \\ &= u_{-i-k+1}^{(k)} + K_k^f v_{-i-k}^{(k)} = u_{-i-(k+1)+1}^{(k+1)} \quad , \end{aligned}$$

and so (72) is true for $k+1$. Therefore, (72) is valid by induction.

Because of (72), row j $(0 \le j \le n)$ of $\dot{U}_n$ is

$$[0 \ \cdots \ 0 \ \dot{u}_{-j}^{(j+1)} \ \cdots \ \dot{u}_{-n}^{(j+1)}] = [0 \ \cdots \ 0 \ v_0^{(j+1)} \ \cdots \ v_{n-j}^{(j+1)}] \quad .$$

But $L_n = \dot{U}_n^T$ from (71) and so

$$L_n = \begin{bmatrix} v_0^{(1)} & 0 & \cdots & 0 \\ v_1^{(1)} & v_0^{(2)} & \cdots & 0 \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ v_n^{(1)} & v_{n-1}^{(2)} & \cdots & v_0^{(n+1)} \end{bmatrix} \quad . \tag{73}$$

Since $T_n = L_n D_n^{-1} U_n$ we must have

$$D_n = diag \left\{ v_0^{(1)} \; v_0^{(2)} \; \cdots \; v_0^{(n+1)} \right\} \tag{74}$$

$$= diag \left\{ u_0^{(1)} \; u_{-1}^{(2)} \; \cdots \; u_{-n}^{(n+1)} \right\} ,$$

that is,

$$v_0^{(k)} = u_{-k+1}^{(k)} \quad . \tag{75}$$

Finally, from (73) and (70),

$$T_n = \begin{bmatrix} v_0^{(1)} & 0 & \cdot & 0 \\ v_1^{(1)} & v_0^{(2)} & \cdot & 0 \\ \vdots & \vdots & & \vdots \\ v_n^{(1)} & v_{n-1}^{(2)} & \cdot & v_0^{(n+1)} \end{bmatrix} \begin{bmatrix} (v_0^{(1)})^{-1} & 0 & \cdot\cdot & 0 \\ 0 & (v_0^{(2)})^{-1} & \cdot\cdot & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdot\cdot & (v_0^{(n+1)})^{-1} \end{bmatrix} \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & \cdot & u_{-n}^{(1)} \\ 0 & u_{-1}^{(2)} & \cdot & u_{-n}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdot & u_{-n}^{(n+1)} \end{bmatrix} . \tag{76}$$

This is equation (3.13a,b) in [6]. The proof of (76) that we have just presented is essentially contained in Appendix A of [6].

If we compare (20) with (76) we see that

$$\sigma_k = v_0^{(k+1)} \tag{77}$$

for $k = 0,1,...,n$. Hence the Schur algorithm will work only when the leading principal submatrices of T_n are nonsingular. From (70) $(L_k^*)^{-1}T_k = U_k$ and from (20) $L_k^A T_k = D_k (U_k^B)^{-1}$ and so $L_k^A = (L_k^*)^{-1}$ (note that we are assuming T_k is strongly nonsingular again). If we compare the elements in the lower left corners of L_k^A (see (17b)) and $(L_k^*)^{-1}$ (generalize derivation in equations (64) - (69a,b) to arbitrary n) we get

$$K_k^f = a_{k,k} \quad .$$

From (15) $K_k^f = a_{k,k}$, and so the reflection coefficients K_k^f produced by the Schur algorithm are identical to the reflection coefficients K_k^f produced by the Levinson-Durbin algorithm. It should be evident that K_k^r in the Schur algorithm equals K_k^r in the Levinson-Durbin algorithm for all k as well. This fact follows by using an argument similar to the one used in proving that $\dot{K}_k^f = K_k^r$ and $\dot{K}_k^r = K_k^f$.

If T_n is Hermitian then it can be shown that the Schur algorithm becomes:

```
For $i$ := 0 to $n$ do begin
```

```
        $u_{-i}^{(1)} := t_i$ ; $u_i^{(1)} := \overline{t}_i$;
        end;
    For $k := 1$ to $n$ do begin
        $K_k := -\dfrac{u_1^{(k)}}{u_{-k+1}^{(k)}}$;
        For $i := 0$ to $n-k$ do begin
            $u_{-(i+k)}^{(k+1)} := u_{-k-i+1}^{(k)} + K_k \overline{u}_{i+1}^{(k)}$;
            $u_i^{(k+1)} := K_k \overline{u}_{-k-i+1}^{(k)} + u_{i+1}^{(k)}$;
            end;
        end;
```

We have used the fact that $L_n^* = (U_n^*)^H$, and $v_0^{(k)} = u_{-k+1}^{(k)}$ so that

$$v_i^{(k)} = \overline{u}_{-k-i+1}^{(k)} \tag{78}$$

holds.

2.4 The Bareiss Algorithm

Bareiss [21] considers the solution of the Toeplitz system of equations $Tx = b$ where T is $(n+1) \times (n+1)$ and is not necessarily symmetric (i.e., T has the form of T_n in (1)), and $b = [b_0 \cdots b_n]^T$ is an arbitrary column vector. T may of course be complex-valued. Brent and Luk [22] compactly write the Bareïss algorithm as follows:

```
    $T^{(0)} := T$ ; $b^{(0)} := b$;
    For $k := 1$ to $n$ do begin
        $m_{-k} := \dfrac{t_{k,0}^{(1-k)}}{t_0}$;
        $T^{(-k)} := T^{(-k+1)} - m_{-k} Z_{-k} T^{(k-1)}$;
        $b^{(-k)} := b^{(-k+1)} - m_{-k} Z_{-k} b^{(k-1)}$;
        $m_k := \dfrac{t_{0,k}^{(k-1)}}{t_{n,n}^{(-k)}}$;
        $T^{(k)} := T^{(k-1)} - m_k Z_k T^{(-k)}$;
        $b^{(k)} := b^{(k-1)} - m_k Z_k b^{(-k)}$;
```

end;

Z_{-k} and Z_k are shift matrices defined by

$$Z_{-k} = [z_{ji}^{(-k)}] = [\delta_{j-i-k}] \ , \ \ Z_k = [z_{ji}^{(k)}] = [\delta_{j-i+k}] \ , \tag{79}$$

where δ is the Kronecker delta. If Z_{-k} is applied to a matrix as indicated in the pseudocode above, that matrix will be shifted down by k rows and the top k rows will be filled in with zeros. Similarly, Z_k applied to a matrix as indicated above will shift that matrix up by k rows and will fill the bottom k rows with zeros. Thus Z_{-k} is a *downshift matrix,* and Z_k is an *upshift matrix.*

It is evident that the Bareiss algorithm produces a sequence of systems of linear equations, namely

$$T^{(-1)}x = b^{(-1)} \, , T^{(1)}x = b^{(1)} \, , \ . \ . \ . \ , \ T^{(-n)}x = b^{(-n)} \, , T^{(n)}x = b^{(n)} \, , \tag{80}$$

where $T^{(-n)}$ is upper triangular and $T^{(n)}$ is lower triangular. The algorithm has a time complexity of $O(n^2)$ (including the back-substitution step).

The operation of this algorithm becomes clearer if we consider an example. Let us consider the special case of $n = 3$ as we did in section 2.3, and so T is as in (64). The Bareiss algorithm will then produce the following sequence of parameters $m_{\pm k}$ and matrices $T^{(\pm k)}$:

$$m_{-1} = \frac{t_{1,0}^{(0)}}{t_0} = \frac{t_{-1}}{t_0} \ , \tag{81a}$$

$$T^{(-1)} = T^{(0)} - m_{-1}Z_{-1}T^{(0)} \tag{81b}$$

$$= \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_{-1} & t_0 & t_1 & t_2 \\ t_{-2} & t_{-1} & t_0 & t_1 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix} - m_{-1} \begin{bmatrix} 0 & 0 & 0 & 0 \\ t_0 & t_1 & t_2 & t_3 \\ t_{-1} & t_0 & t_1 & t_2 \\ t_{-2} & t_{-1} & t_0 & t_1 \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ 0 & t_0^{(-1)} & t_1^{(-1)} & t_2^{(-1)} \\ t_{-2}^{(-1)} & 0 & t_0^{(-1)} & t_1^{(-1)} \\ t_{-2}^{(-1)} & t_{-2}^{(-1)} & 0 & t_0^{(-1)} \end{bmatrix} ,$$

$$m_1 = \frac{t_{0,1}^{(0)}}{t_{3,3}^{(-1)}} = \frac{t_1}{t_0^{(-1)}} \ , \tag{81c}$$

$$T^{(1)} = T^{(0)} - m_1Z_1T^{(-1)} \tag{81d}$$

$$= \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_{-1} & t_0 & t_1 & t_2 \\ t_{-2} & t_{-1} & t_0 & t_1 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix} - m_1 \begin{bmatrix} 0 & t_0^{(-1)} & t_1^{(-1)} & t_2^{(-1)} \\ t_{-2}^{(-1)} & 0 & t_0^{(-1)} & t_1^{(-1)} \\ t_{-3}^{(-1)} & t_{-2}^{(-1)} & 0 & t_0^{(-1)} \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} t_0 & 0 & t_2^{(1)} & t_3^{(1)} \\ t_{-1}^{(1)} & t_0 & 0 & t_2^{(1)} \\ t_{-2}^{(1)} & t_{-1}^{(1)} & t_0 & 0 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix},$$

$$m_{-2} = \frac{t_{2,0}^{(-1)}}{t_0} = \frac{t_{-2}^{(-1)}}{t_0}, \tag{82a}$$

$$T^{(-2)} = T^{(-1)} - m_{-2} Z_{-2} T^{(1)} \tag{82b}$$

$$= \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ 0 & t_0^{(-1)} & t_1^{(-1)} & t_2^{(-1)} \\ t_{-2}^{(-1)} & 0 & t_0^{(-1)} & t_1^{(-1)} \\ t_{-3}^{(-1)} & t_{-2}^{(-1)} & 0 & t_0^{(-1)} \end{bmatrix} - m_{-2} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ t_0 & 0 & t_2^{(1)} & t_3^{(1)} \\ t_{-1}^{(1)} & t_0 & 0 & t_2^{(1)} \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ 0 & t_0^{(-1)} & t_1^{(-1)} & t_2^{(-1)} \\ 0 & 0 & t_0^{(-2)} & t_1^{(-2)} \\ t_{-3}^{(-2)} & 0 & 0 & t_0^{(-2)} \end{bmatrix},$$

$$m_2 = \frac{t_{0,2}^{(1)}}{t_{3,3}^{(-2)}} = \frac{t_2^{(1)}}{t_0^{(-2)}}, \tag{82c}$$

$$T^{(2)} = T^{(1)} - m_2 Z_2 T^{(-2)} \tag{82d}$$

$$= \begin{bmatrix} t_0 & 0 & t_2^{(1)} & t_3^{(1)} \\ t_{-1}^{(1)} & t_0 & 0 & t_2^{(1)} \\ t_{-2}^{(1)} & t_{-1}^{(1)} & t_0 & 0 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix} - m_2 \begin{bmatrix} 0 & 0 & t_0^{(-2)} & t_1^{(-2)} \\ t_{-3}^{(-2)} & 0 & 0 & t_0^{(-2)} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} t_0 & 0 & 0 & t_3^{(2)} \\ t_{-1}^{(2)} & t_0 & 0 & 0 \\ t_{-2}^{(1)} & t_{-1}^{(1)} & t_0 & 0 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix},$$

$$m_{-3} = \frac{t_{3,0}^{(-2)}}{t_0} = \frac{t_{-3}^{(-2)}}{t_0}, \tag{83a}$$

$$T^{(-3)} = T^{(-2)} - m_{-3} Z_{-3} T^{(2)} \tag{83b}$$

$$= \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ 0 & t_0^{(-1)} & t_1^{(-1)} & t_2^{(-1)} \\ 0 & 0 & t_0^{(-2)} & t_1^{(-2)} \\ t_{-3}^{(-2)} & 0 & 0 & t_0^{(-2)} \end{bmatrix} - m_{-3} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ t_0 & 0 & 0 & t_3^{(2)} \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ 0 & t_0^{(-1)} & t_1^{(-1)} & t_2^{(-1)} \\ 0 & 0 & t_0^{(-2)} & t_1^{(-2)} \\ 0 & 0 & 0 & t_0^{(-3)} \end{bmatrix},$$

$$m_3 = \frac{t_{0,3}^{(2)}}{t_{3,3}^{(-3)}} = \frac{t_3^{(2)}}{t_0^{(-3)}}, \tag{83c}$$

$$T^{(3)} = T^{(2)} - m_3 Z_3 T^{(-3)} \tag{83d}$$

$$= \begin{bmatrix} t_0 & 0 & 0 & t_3^{(2)} \\ t_{-1}^{(2)} & t_0 & 0 & 0 \\ t_{-2}^{(1)} & t_{-1}^{(1)} & t_0 & 0 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix} - m_3 \begin{bmatrix} 0 & 0 & 0 & t_0^{(-3)} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} t_0 & 0 & 0 & 0 \\ t_{-1}^{(2)} & t_0 & 0 & 0 \\ t_{-2}^{(1)} & t_{-1}^{(1)} & t_0 & 0 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix} .$$

The Bareiss algorithm works as long as the leading principal submatrices of T are nonsingular. An LU factorization of T is

$$T = LU \quad , \tag{84}$$

where $L = \frac{1}{t_0}(T^{(n)})^{T2}$, and $U = T^{(-n)}$, and where 'T2' denotes matrix transposition about the main antidiagonal (see [22]).

2.5 The Split Schur Algorithms of Delsarte and Genin

Delsarte and Genin [23] have shown that the Levinson-Durbin algorithm, for the case where T_n is real and symmetric, is redundant in complexity by a constant factor. Specifically, if the Levinson-Durbin algorithm of section 2.1 (real-symmetric T_n) requires $c_1 n^2$ multiplications (large n assumed), and if the split (symmetric or antisymmetric form) Levinson-Durbin algorithm requires $c_2 n^2$ multiplications, then $c_2 < c_1$. The method in [23] has been extended by Krishna and Morgera [24] to accommodate the Hermitian Toeplitz case. As well, Delsarte and Genin [25] have presented symmetric and antisymmetric split Schur algorithms for the case where T_n is real and symmetric. The split Schur algorithms of [25] are faster than the Schur algorithm of section 2.3 by a constant factor, at least insofar as the computation of reflection coefficients is concerned. We will only derive the symmetric split Schur algorithm here, and we will simply state the antisymmetric split Schur algorithm, as its derivation is so close to that of the symmetric case.

From the Schur algorithm for Hermitian T_n we have

$$u_{\pm i}^{(1)} = t_i \ , \ \ (i = 0,1,\ldots,n) \tag{85a}$$

$$K_k = -\frac{u_1^{(k)}}{u_{-k+1}^{(k)}} \ , \tag{85b}$$

$$u_{-(i+k)}^{(k+1)} = u_{-(i+k)+1}^{(k)} + K_k u_{i+1}^{(k)} \ , \tag{85c}$$

$$u_i^{(k+1)} = K_k u_{-(i+k)+1}^{(k)} + u_{i+1}^{(k)} \ , \tag{85d}$$

where the bars have been eliminated since we are now assuming that T_n is real.

Define the variables

$$v_{k+1,i} = u_{i+1}^{(k)} + u_{-(i+k)+1}^{(k)} \ , \tag{86a}$$

$$v_{k+1,i}^* = u_{i+1}^{(k)} - u_{-(i+k)+1}^{(k)} \ . \tag{86b}$$

Variables $v_{k,i}$ are used in the construction of the *symmetric split Schur algorithm*, and variables $v_{k,i}^*$ are used in the construction of the *antisymmetric split Schur algorithm*. We call $v_{k,i}$ the *symmetric Schur variables*, $v_{k,i}^*$ the *antisymmetric Schur variables*, and $u_i^{(k)}$ the *Schur variables* (as in [25]).

Using (85c,d) and the definition (86a) gives

$$(1 + K_k)v_{k+1,i} = u_i^{(k+1)} + u_{-(k+i)}^{(k+1)} \ . \tag{87}$$

From (85d) and (86a)

$$v_{k+1,i} = u_i^{(k+1)} + (1 - K_k)u_{-(i+k)+1}^{(k)} \ . \tag{88}$$

From (87)

$$(1 + K_{k-1})v_{k,i} = u_i^{(k)} + u_{-(i+k)+1}^{(k)} \ . \tag{89}$$

If we substitute (89) into (88) we get

$$v_{k+1,i} = (1 - K_k)(1 + K_{k-1})v_{k,i} - u_i^{(k)} + u_i^{(k+1)} + K_k u_i^{(k)} \ . \tag{90}$$

But from (85c) $K_k u_i^{(k)} = u_{-(i+k)+1}^{(k+1)} - u_{-(k+i)+2}^{(k)}$ and this may be substituted into (90) giving

$$v_{k+1,i} = (1 - K_k)(1 + K_{k-1})v_{k,i} - u_i^{(k)} + u_i^{(k+1)} + u_{-(i+k)+1}^{(k+1)} - u_{-(i+k)+2}^{(k)} \ . \tag{91}$$

From (86a)

$$v_{k+1,i-1} = u_i^{(k)} + u_{-(k+i)+2}^{(k)} \ ,$$

$$v_{k+2,i-1} = u_i^{(k+1)} + u_{-(k+i)+1}^{(k+1)} \ ,$$

and so (91) becomes

$$v_{k+1,i} = (1 - K_k)(1 + K_{k-1})v_{k,i} - v_{k+1,i-1} + v_{k+2,i-1}$$

or

$$v_{k+2,i} = v_{k+1,i} + v_{k+1,i+1} - \alpha_k v_{k,i+1} \ , \tag{92}$$

where

$$\alpha_k = (1 - K_k)(1 + K_{k-1}) \ . \tag{93}$$

We call α_k the kth *symmetric split reflection coefficient.* Straightforwardly,

$$K_k = 1 - \frac{\alpha_k}{1 + K_{k-1}} \ . \tag{94}$$

Since $k = 1, \ldots, n$, we will let $K_0 = 0$ as a convention (as is done in [25]).

We need initial conditions, and we need an expression for the symmetric split reflection coefficient in terms of the symmetric Schur variables. Clearly, from (86a) and (85b)

$$v_{k+1,0} = u_1^{(k)} + u_{-k+1}^{(k)} = (1 - K_k)u_{-k+1}^{(k)} = (1 - K_k)\sigma_{k-1} \ , \tag{95}$$

where we have also used (75) and (77). From the Levinson-Durbin algorithm $\sigma_k = (1 - K_k^2)\sigma_{k-1} = (1 - K_k)(1 + K_k)\sigma_{k-1}$. Hence, $\sigma_k = v_{k+1,0}(1 + K_k)$ or $\sigma_{k-1} = v_{k,0}(1 + K_{k-1})$. As a result, from (93),

$$\alpha_k = (1 - K_k)(1 + K_{k-1}) = \frac{v_{k+1,0}}{v_{k,0}} \ . \tag{96}$$

From (86a)

$$v_{2,i} = u_{i+1}^{(1)} + u_{-i}^{(1)} = t_i + t_{i+1} \ , \ (0 \le i \le n-1) \ , \tag{97}$$

and via (85b) $K_1 = -u_1^{(1)}/u_0^{(1)} = -t_1/t_0$. Since $K_1 = 1 - \alpha_1$ and $\alpha_1 = \dfrac{v_{2,0}}{v_{1,0}} = (t_0+t_1)/t_0$ we must have $v_{1,0} = t_0$. From (89)

$$(1 + K_0)v_{1,i} = u_i^{(1)} + u_{-i}^{(1)} = 2t_i \ ,$$

or $v_{1,i} = 2t_i \ \ (1 \le i \le n)$.

Thus, the *symmetric split Schur algorithm* is:

```
$K_0 := 0;$
$v_{1,0} := t_0$ ; $v_{1,i} := 2t_i \quad (1 \le i \le n);$
$v_{2,i} := t_i + t_{i+1} \quad (0 \le i \le n-1);$
For $k := 1$ to $n$ do begin
    $\alpha_k := \frac{v_{k+1,0}}{v_{k,0}};$
    $K_k := 1 - \frac{\alpha_k}{1 + K_{k-1}};$
    For $i := 0$ to $n-k$ do begin
        $v_{k+2,i} := v_{k+1,i} + v_{k+1,i+1} - \alpha_k v_{k,i+1};$
    end;
end;
```

We may reindex the variables of this pseudocode program to get:

```
$K_0 := 0;$
$v_{0,0} := t_0$ ; $v_{0,i} := 2t_i \quad (1 \le i \le n);$
$v_{1,i} := t_i + t_{i+1} \quad (0 \le i \le n-1);$
For $k := 1$ to $n$ do begin
    $\alpha_k := \frac{v_{k,0}}{v_{k-1,0}};$
    $K_k := 1 - \frac{\alpha_k}{1 + K_{k-1}};$
    For $i := 0$ to $n-k$ do begin
        $v_{k+1,i} := v_{k,i} + v_{k,i+1} - \alpha_k v_{k-1,i+1};$
    end;
end;
```

This is essentially the symmetric split Schur algorithm of Table VI in [25]. The only difference is that the innermost For-do loop of the above program continues until $i = n-k$ whereas the corresponding loop in the Table VI program of [25] only continues until $i = n-k-1$. This difference is insignificant, and is due to the use of a different form of the Schur algorithm as a starting point for the derivation of the split Schur algorithms in [25]. We note that the Schur algorithm in [25] is essentially the same as that in [6]. The Schur algorithm in [25] is that of Le Roux and Gueguen [26], and is derived directly from the Levinson-Durbin algorithm via the derivation of Markel and Gray [14], which is different from the Levinson-Durbin algorithm derivation of section 2.1.

We may state the *antisymmetric split Schur algorithm* in the notation of Delsarte and Genin [25] as follows:

```
$K_0 := 0;$
$v^*_{0,0} := t_0;\ v^*_{0,i} := 0\ (1 \le i \le n);$
$v^*_{1,i} := t_i - t_{i+1}\ \ (0 \le i \le n-1);$
For $k := 1$ to $n$ do begin
    $\alpha^*_k := \frac{v^*_{k,0}}{v^*_{k-1,0}};$
    $K_k := -1 + \frac{\alpha^*_k}{1 - K_{k-1}};$
    For $i := 0$ to $n-k$ do begin
        $v^*_{k+1,i} := v^*_{k,i} + v^*_{k,i+1} - \alpha^*_k v^*_{k-1,i+1};$
        end;
    end;
```

We call α^*_k the kth *antisymmetric split reflection coefficient.*

A comparison of the split Schur algorithms with the Schur algorithm reveals that the split Schur algorithms use about half of the multiplications that the Schur algorithm does in order to compute the reflection coefficients K_k. It is still true though that both types of algorithms have time complexities of $O(n^2)$.

2.6 Positive Definite Toeplitz Matrices

If T_n is an autocorrelation matrix, then T_n is positive semidefinite [3]. This is often symbolized by writing $T_n \ge 0$. For complex data, T_n will be Hermitian if it is an autocorrelation matrix. From [3], a necessary and sufficient condition for the positive definiteness of T_n (symbolized by $T_n > 0$) is that

$$det(T_k) > 0 \text{ for } k = 0,1,...,n \quad .$$

If we recall equation (23) of section 2.1, the Levinson-Durbin algorithm has a built-in test for positive definiteness. Since $\sigma_{k+1} = \sigma_k(1 - | K_{k+1} |^2)$ (equation (29)), then T_n is positive definite if and only if $| K_k | < 1$, for $k = 1,2,...,n$. Clearly, if $T_n > 0$ then $0 < \sigma_{k+1} \leq \sigma_k$, $k \geq 0$ as well.

It can be shown that the Schur variables, in the Hermitian Toeplitz autocorrelation matrix case, satisfy $| u_i^{(k)} | \leq t_0$ (see [25], or [26]). Similarly, the symmetric and antisymmetric Schur variables satisfy $| v_{k,j} | \leq 2t_0$, $| v_{k,j}^* | \leq 2t_0$, and the split reflection coefficients satisfy $0 < \alpha_k$, $\alpha_k^* < 4$, (see [25]). The bounds satisfied by the Schur and split Schur variables make the Schur and split Schur algorithms suitable for fixed-point arithmetic implementation.

2.7 Other Algorithms

The list of algorithms for the solution of Toeplitz matrix problems that we have so far presented is not complete. We can name others. All of the preceding algorithms have time complexities of $O(n^2)$, but there exist algorithms with time complexities of $O(n \log^2 n)$. These algorithms are not practical unless n is quite large, however. The reader should see Kumar [27], Brent, Gustavson, and Yun [28], and Bitmead and Anderson [29]. All three algorithms use so-called "doubling strategies". Doubling strategies are introduced in Blahut [4]. Connections between Euclid's algorithm, Padé approximation and Toeplitz problems are discussed in [28]. A summary of some of the results in [28] may be found in Gustavson and Yun [30]. Bitmead and Anderson [29] employ the Gohberg-Semencul formula in their algorithm.

Some Toeplitz matrices are banded. This means that there are integers p and q such that $1 \leq p,q < n$ and $t_p \neq 0$, $t_{-q} \neq 0$ but $t_i = 0$ for $i > p$ and $i < -q$. If p is of the same order as q, then Jain [31] presents an algorithm for solving $T_n x = y$, based upon the use of fast Fourier transforms (FFT), and the Trench algorithm, and it has a time complexity of $O(n \log n + p^2)$, on a sequential processor. Dickinson [32] presents an algorithm that uses results from Trench [15] and Zohar [18]. It has a time complexity of $O(pn + qn + (p+q)^2)$ which is much faster than that of Jain if p and q are fixed while n grows.

Many concepts that have been used to solve Toeplitz matrix problems can be extended to solve certain non-Toeplitz matrix problems. For example, the Levinson-Durbin algorithm can be extended to solve Toeplitz-plus-Hankel matrices as is shown in Merchant and Parks [33]. If T is Toeplitz and H is Hankel, then $T + H$ is a Toeplitz-plus-Hankel matrix. H is Hankel if $H = [h_{ij}] = [h_{i+j}]$, that is, H is "Toeplitz in its antidiagonals". Note that if H is Hankel, then JH is Toeplitz and so the algorithms of the previous sections can potentially be used to solve $Hx = y$. Levinson-Durbin-like algorithms can be found for the solution of systems of equations involving so-called *diagonal innovations matrices* (DIM), and *peripheral innovations matrices* (PIM). This is shown in Carayannis, Kalouptsidis, and Manolakis [34]. Toeplitz matrices belong to the class of PIM matrices. The *displacement rank theory* of Kailath, Kung and Morf [35,36] allows the extension of the Levinson-Durbin class of algorithms to *near-to-Toeplitz matrices*. A matrix is near-to-Toeplitz if its displacement rank is fixed compared with n, the size of the matrix. The reader should consult [35,36] for an explanation of this idea. Applications of displacement rank theory can be found in Friedlander, Morf, Kailath, and Ljung [37]. Improvements to the work in [37] are suggested in Kalouptsidis, Manolakis, and Carayannis [38]. An extension of the Levinson-Durbin algorithm to the *covariance method of linear prediction* can be found in Morf, Dickinson, Kailath, and Vieira [39].

3. On The Origin of Toeplitz Matrix Problems

We have seen that many algorithms exist for the solution of Toeplitz matrix problems. Now we shall consider some examples of where Toeplitz matrix problems arise.

3.1 A Miscellany

We begin by presenting a list of applications with short explanations. Succeeding sections will consider other examples in greater detail.

(i) Complex-valued, block Toeplitz matrices arise in the analysis of diffraction gratings (see Jull, Hui, Facq [40], and Facq [41]).

(ii) The discrete form of the Gel'fand-Levitan integral equation, which arises in geophysical applications, has a Toeplitz-plus-Hankel structure (see Merchant and Parks [33], and Aki and Richards [42]).

(iii) The numerical solution of boundary value problems in ordinary differential equations leads to the need to invert banded Toeplitz matrices (see Usmani [43]).

(iv) Linear predictive deconvolution to remove reverberations in marine seismograms, and Wiener filtering can involve the need to invert Toeplitz matrices (see Wood and Treitel [44], Wiggins and Robinson [45]).

(v) The linear predictive analysis of speech signals can involve the need to invert Toeplitz matrices (see Schafer and Rabiner [46], and Markel and Gray [14]).

(vi) Reed-Solomon error control codes can be decoded via Toeplitz matrix inversion (see Blahut [4,47,48]).

(vii) Finding the poles of an autoregressive (AR) system function can involve the need to compute reflection coefficients (see Jones and Steinhardt [49]).

(viii) Parametric bispectrum analysis can involve the need to invert a nonsymmetric Toeplitz matrix (see Raghuveer and Nikias [50]).

(ix) Linear interpolation can require the Gohberg-Semencul formula (see Kay [51]).

(x) Moving average (MA) system identification problems can be solved by an iterative algorithm that uses Trench's algorithm, and the resulting method is applicable to the modeling of so-called 2-phase flows in fluid mechanics (see Ohsmann [52]).

(xi) The identification of nonlinear systems can involve the need to solve Toeplitz systems of equations, and this has applications in the modeling of nonlinear physiological systems such as neurons (see Korenberg [53]).

(xii) Capon's [54] maximum likelihood method (MLM) is used for wavenumber spectral analysis for wave propagation with spatially distributed sensor arrays, and the

Gohberg-Semencul formula can be used in the high-speed computation of Capon's MLM (see Musicus [55]).

3.2 Padé Approximants

From Gragg [56], the Padé table of a power series

$$C(z) = \sum_{m=0}^{\infty} c_m z^m \tag{98}$$

is a doubly infinite array of rational functions

$$r_{mn}(z) = \frac{p_{mn}(z)}{q_{mn}(z)} = \frac{a_0 + a_1 z + \cdots + a_m z^m}{b_0 + b_1 z + \cdots + b_n z^n} \tag{99}$$

determined in such a manner that the Maclaurin expansion of r_{mn} agrees with $C(z)$ as far as possible, where $m,n \geq 0$. It is possible that $deg(p_{mn}) < m$, and $deg(q_{mn}) < n$. $C(z)$ is *normal* if, for each pair (m,n), this agreement is exact through to the power z^{m+n}. Note that z, c_m, a_i and b_i may be complex-valued in general. The convergence of (98) is not essential. $r_{mn}(z)$ is a *Padé form* of type (m,n) for $C(z)$ if $q_{mn}(z) \neq 0$ and

$$C(z)q_{mn}(z) - p_{mn}(z) = O(z^{m+n+1}) \ . \tag{100}$$

$O(z^{m+n+1})$ means that the right side is a power series beginning exactly with a power $z^{m+n+k+1}$, where $0 \leq k \leq \infty$; $k = \infty$ implies that $Cq_{mn} - p_{mn} = 0$. The Padé form approximates the power series $C(z)$, and so is called a *Padé approximant.* Equation (100) is equivalent to a linear system of $m + n + 1$ equations in $m + n + 2$ unknowns $a_0, \ldots, a_m$,and $b_0, \ldots, b_n$:

$$\sum_{j=0}^{n} c_{i-j}\, b_j = \begin{cases} a_i \ , & i = 0, \ldots, m \ , \\ 0 \ , & i = m+1, \ldots, m+n \ , \end{cases} \tag{101}$$

and this is a Toeplitz system of equations. Various results on the existence and uniqueness of solutions, and how to find them, are found in [56].

It is beyond the scope of this thesis to consider the details of the theory of Padé approximants and their relationship to Toeplitz matrix problems, Euclid's algorithm, AR signal modeling, etc. The interested reader is refered to such publications as Brent, Gustavson and Yun [28], Gustavson and Yun [30], Cybenko [57], Weiss and McDonough [58], and McEliece and Shearer [59]. The Padé approximant problem is related to the *minimal partial realization problem* (see Imamura [60] or Kailath [61] (pp. 322-326 or pp. 491-492)). The Padé approximant problem has applications in system identification since, for example, $C(z)$ might represent the impulse response of some linear time-invariant system.

3.3 Pisarenko's Harmonic Decomposition

Pisarenko's harmonic decomposition (PHD) [62] is a spectral analysis technique that involves the computation of eigenvalues and eigenvectors of autocorrelation matrices. A summary of this method and a comparison of it with other spectral analysis methods may be found in Kay and Marple [63], and the method is also discussed in Roberts and Mullis [3] (see pp. 535-538), albeit rather briefly. However, we shall summarize the main concepts here.

The PHD method assumes that the signal, for which a spectral estimate is desired, is composed of a finite sum of sinusoids plus white noise. Thus, the PHD approach will yield a line spectral estimate of the signal in question. Specifically, given n real sinusoids in additive white noise, and the autocorrelation matrix (R) of order $2n + 1$ for this signal, then the minimum eigenvalue of R is the variance of the noise (see [3],[62-63]). The eigenvector corresponding to the minimum eigenvalue of R may be used to construct a polynomial, the zeros of which have unity magnitude and the arguments specify the frequencies of the sinusoids (see [62-63]). Once the noise variance and frequencies of the sinusoids are known, it is possible to find the amplitudes of the sinusoids (see [62-63]).

Iterative approaches to the computation of the minimum eigenvalue and corresponding eigenvector of R may be found in Hu and Kung [64], and in Hayes and Clements [65]. The method proposed in [64] involves Toeplitz system solution and so can use the parallel-pipelined architecture of Kung and Hu [6]. As a result, the time complexity of the Toeplitz eigensystem solver in [64] is $O(kn)$, where k is the number of iterations. In the case of Hayes and Clements [65], the computation of reflection coefficients is required, and the parallel-pipelined Schur algorithm machine in [6] can be used here as well. In this instance, the time complexity of the Hayes-Clements algorithm will also be $O(kn)$, where k is again the number of iterations.

3.4 Gaussian Signal Detection

The problem of Gaussian signal detection is a fundamental one in pattern recognition theory (see Tou and Gonzalez [66]), and in communications theory (see Van Trees [67]). It turns out that the Gohberg-Semencul formula is applicable to the design of linear time-invariant, and fast Gaussian signal detectors (see Kailath, Levy, Ljung, and Morf [68]). We shall explain how this is so in the context of pattern recognition.

Suppose that there exists a set of M pattern classes $\omega_1, \ldots, \omega_M$, and that we have a measured signal vector $x = [x_0 \cdots x_n]^T$ that originates from one of these classes. We would naturally like to know from which class this vector originated. We will assume, for simplicity, that the a priori probabilities for the occurrence of each class are equal, and that $p(x \mid \omega_i)$ is the probability density function of x given that x came from ω_i. If we assume that $p(x \mid \omega_i)$ is Gaussian, then

$$p(x \mid \omega_i) = \frac{1}{(2\pi)^{(n+1)/2} \, (det(R_n^i))^{\frac{1}{2}}} \exp[-\frac{1}{2}(x - m_i)^T (R_n^i)^{-1}(x - m_i)] \; , \qquad (102)$$

where R_n^i is the covariance matrix of class ω_i, and m_i the corresponding mean vector. R_n^i is of order $n+1$. We have $x \in \omega_i$ if and only if

$$\frac{p(x \mid \omega_i)}{p(x \mid \omega_j)} > 1 \quad \text{for } \textit{every } j \neq i \; . \qquad (103)$$

From (102),

$$\frac{p(x \mid \omega_i)}{p(x \mid \omega_j)} = \left[\frac{det(R_n^j)}{det(R_n^i)}\right]^{\frac{1}{2}} \exp\left\{-\frac{1}{2}[(x-m_i)^T (R_n^i)^{-1}(x-m_i) - (x-m_j)^T (R_n^j)^{-1}(x-m_j)]\right\} . \qquad (104)$$

Taking the logarithm of (104), $x \in \omega_i$ if and only if

$$\Lambda(x) = (x-m_i)^T(R_n^i)^{-1}(x-m_i) - (x-m_j)^T(R_n^j)^{-1}(x-m_j) < \ln\frac{det(R_n^j)}{det(R_n^i)} \quad , \qquad (105)$$

for every $j \neq i$. *Likelihood function* $\Lambda(x)$ must be computed and compared to a threshold in order to ascertain from which class x is most likely to have come. From (105) it is clear that the principal operation involved in computing the likelihood function is the evaluation of expressions of the form

$$y^T T_n^{-1} y \quad , \qquad (106)$$

where T_n is Toeplitz. As is noted in [68], the operation in (106) is in the form of a linear time-variant filtering operation. However, it is also noted in [68] that the Gohberg-Semencul formula can be used to rewrite (106) in the form of a linear time-invariant filtering operation.

For real-valued data, T_n will be real and symmetric. Thus, (63) will have the form

$$T_n^{-1} = \sigma_n^{-1}\,(L_a^f(L_a^f)^T - (U_a^r)^T U_a^r) \, , \qquad (107)$$

where L_a^f is lower triangular Toeplitz, and U_a^r is upper triangular Toeplitz. Hence, (106) becomes

$$y^T T_n^{-1} y = \sigma_n^{-1}\,[((L_a^f)^T y)^T (L_a^f)^T y - (U_a^r y)^T U_a^r y] \ . \qquad (108)$$

$(L_a^f)^T y$ and $U_a^r y$ may be rapidly computed through the use of fast Fourier transforms or fast convolution algorithms (see Blahut [4]). It is clear that these are linear time-invariant filtering operations.

3.5 The Layered Earth Model - A Geophysical Application

We will consider a model of the stratified (layered) earth as in Figure 1(a). This model, and the attendant notation, is from Robinson and Treitel [12]. From this example it will be seen how the reflection coefficients K_k of section 2 got their name.

In the model of Figure 1(a) each layer is assumed to have a thickness such that the travel time through it is one-half time unit (i.e., the two-way travel time is one unit). In other words, all layers have the same travel time. Interface 0 is the earth-air, or perhaps water-air, interface. The present model supports travelling-wave motion from bottom to top *(upgoing waves),* and from top to bottom *(downgoing waves).* Partial reflection and transmission of the waves occurs at the interface between the layers. Let c_n denote the *reflection coefficient* of interface n, and τ_n the *transmission coefficient.* By convention, all wave motion is measured in physical units proportional to the square root of energy (i.e., the square of the amplitude of a wave is in terms of energy). A *pulse* is a narrow spike-like waveform associated with a particular discrete-time instant. Thus, the present model is actually a discrete-time model, and each pulse has an energy proportional to the square of its amplitude.

As in Figure 1(b) (left), a unit energy downgoing pulse is partially reflected and partially transmitted. The reflected pulse has amplitude c_n, and the transmitted pulse has amplitude τ_n. By the law of conservation of energy, $c_n^2 + \tau_n^2 = 1$ or $\tau_n = \sqrt{1 - c_n^2}$ (positive square root chosen by convention). The case of the upgoing pulse incident upon the bottom of interface n is shown in the right of Figure 1(b).

Let z denote the unit time delay operator (electrical engineers would normally choose z^{-1} but we shall adopt the geophysicists' conventions). Thus, a half unit delay is $z^{\frac{1}{2}}$, and a half unit advance is $z^{-\frac{1}{2}}$.

From Figure 1(c), let $d_n(t)$ and $u_n(t)$ be the downgoing and upgoing waves, respectively, at the top of layer n, and let $d'_n(t)$ and $u'_n(t)$ be the downgoing and upgoing waves, respectively, at the bottom of layer n (all at time t). Since waves propagate through a layer unchanged, it must be true that

$$\begin{bmatrix} d'_n(t) \\ u'_n(t) \end{bmatrix} = \begin{bmatrix} z^{\frac{1}{2}} & 0 \\ 0 & z^{-\frac{1}{2}} \end{bmatrix} \begin{bmatrix} d_n(t) \\ u_n(t) \end{bmatrix} . \qquad (109)$$

It may be readily shown that, at interface n,

$$u'_n(t) = c_n d'_n(t) + \tau_n u_{n+1}(t) , \qquad (110a)$$

$$d_{n+1}(t) = -c_n u_{n+1}(t) + \tau_n d'_n(t) . \qquad (110b)$$

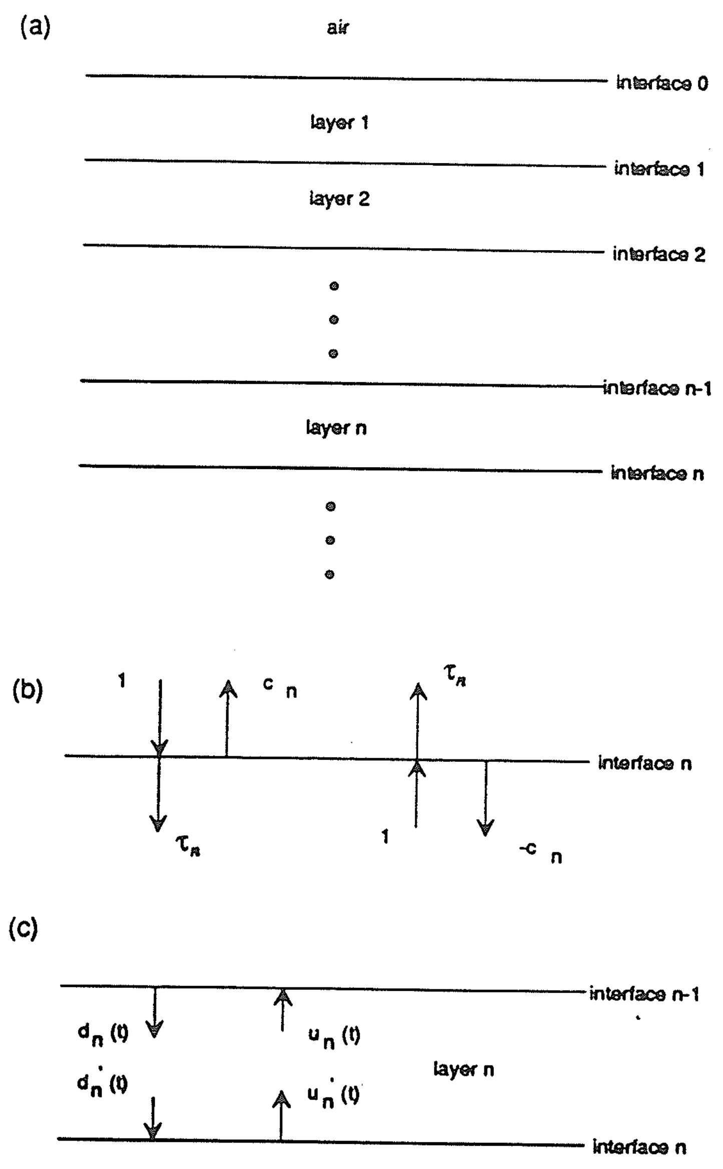


Figure 1: (a) Layered earth model; (b) Downgoing unit pulse (left), and upgoing unit pulse (right) incident upon interface n (c_n = square root energy reflection coefficient, τ_n = square root energy transmission coefficient); (c) Upgoing and downgoing waves at the top, and bottom of layer n.

Equations (110a,b) solve to yield

$$\begin{bmatrix} d_{n+1}(t) \\ u_{n+1}(t) \end{bmatrix} = \frac{1}{\tau_n} \begin{bmatrix} 1 & -c_n \\ -c_n & 1 \end{bmatrix} \begin{bmatrix} d'_n(t) \\ u'_n(t) \end{bmatrix} , \qquad (111)$$

and this combined with (109) gives us

$$\begin{bmatrix} d_{n+1}(t) \\ u_{n+1}(t) \end{bmatrix} = \frac{z^{-\frac{1}{2}}}{\tau_n} \begin{bmatrix} z & -c_n \\ -c_n z & 1 \end{bmatrix} \begin{bmatrix} d_n(t) \\ u_n(t) \end{bmatrix} . \tag{112}$$

We can relate the waves of layer $n+1$ to those of layer 1 using (112) as follows:

$$\begin{bmatrix} d_{n+1}(t) \\ u_{n+1}(t) \end{bmatrix} = \frac{z^{-\frac{n}{2}}}{T_n} \begin{bmatrix} P_n^* & Q_n^* \\ Q_n & P_n \end{bmatrix} \begin{bmatrix} d_1(t) \\ u_1(t) \end{bmatrix} , \tag{113}$$

where $T_n = \prod_{i=1}^{n} \tau_i$, and

$$\begin{bmatrix} P_n^* & Q_n^* \\ Q_n & P_n \end{bmatrix} = \begin{bmatrix} z & -c_n \\ -c_n z & 1 \end{bmatrix} \cdots \begin{bmatrix} z & -c_1 \\ -c_1 z & 1 \end{bmatrix} . \tag{114}$$

It may be shown that $P_n^*(z) = z^n P_n(z^{-1})$, $Q_n^*(z) = z^n Q_n(z^{-1})$ (see Robinson and Treitel [69]).

Let us now consider the marine seismogram (as in section V of [12]). We will excite the layered system with a surface source. Assume that it is a downgoing unit pulse δ_t set off at $t = 0$ just below interface 0 (surface). We have $\delta_t = 1$ if $t = 0$, and $\delta_t = 0$ if $t \neq 0$ (Kronecker delta). We assume that the surface is a perfect reflector (a water-air interface is a good approximation of this). As a result, $| c_0 | = 1$ (so $c_0 = \pm 1$). Therefore, upgoing wave $u_1(t)$ (at the top of layer 1) is reflected at the surface to yield downgoing wave $-c_0 u_1(t)$. Thus,

$$d_1(t) = \delta_t - c_0 u_1(t) . \tag{115}$$

It is clear that the first nonzero value of $u_1(t)$ occurs at $t = 1$, and so $u_1(t)$ is the time series $u_1(1)$, $u_1(2)$, $u_1(3)$, $\cdots$, which has its first *break* (or *arrival time*) at $t = 1$. The downgoing wave $d_1(t)$ is the time series 1 , $-c_0 u_1(1)$, $-c_0 u_1(2)$, $\cdots$, which has its first break at $t = 0$ (see (115)). In general, it is easy to see that the first break of upgoing wave $u_{n+1}(t)$ at the top of layer $n+1$ occurs one time unit after the first break of downgoing wave $d_{n+1}(t)$ at the top of the same layer. Furthermore, the first break of $d_{n+1}(t)$ is produced by the unit source pulse travelling down through the first n layers directly from the surface and through interface n. The amplitude of this pulse is then $T_n = \prod_{i=1}^{n} \tau_n$, and its travel time is $\frac{n}{2}$ units. Hence, the downgoing wave $d_{n+1}(t)$ is the time series $d_{n+1}(\frac{n}{2})$, $d_{n+1}(\frac{n}{2} + 1)$, $d_{n+1}(\frac{n}{2} + 2)$, $\cdots$. Recall that $u_{n+1}(t)$ arrives one time unit after $d_{n+1}(t)$. Thus, $u_{n+1}(t)$ is of the form $u_{n+1}(\frac{n}{2} + 1)$, $u_{n+1}(\frac{n}{2} + 2)$, $\cdots$. Pulse $u_{n+1}(\frac{n}{2} + 1)$ is the reflection of direct pulse $d_{n+1}(\frac{n}{2})$ from interface $n+1$, and so $u_{n+1}(\frac{n}{2} + 1)$ has amplitude $T_n c_{n+1}$.

Given $G(z)$, let us denote $G(z^{-1})$ by $\overline{G(z)}$, or simply $\overline{G}$. The z-transform of (113) is

$$\begin{bmatrix} D_{n+1} \\ U_{n+1} \end{bmatrix} = \frac{z^{-\frac{n}{2}}}{T_n} \begin{bmatrix} P_n^* & Q_n^* \\ Q_n & P_n \end{bmatrix} \begin{bmatrix} D_1 \\ U_1 \end{bmatrix} . \qquad (116)$$

Replacing z by z^{-1} in (116) yields

$$\begin{bmatrix} \overline{D}_{n+1} \\ \overline{U}_{n+1} \end{bmatrix} = \frac{z^{-\frac{n}{2}}}{T_n} \begin{bmatrix} P_n & Q_n \\ Q_n^* & P_n^* \end{bmatrix} \begin{bmatrix} \overline{D}_1 \\ \overline{U}_1 \end{bmatrix} . \qquad (117)$$

Combining (116) and (117) gives

$$\begin{bmatrix} D_{n+1} & \overline{U}_{n+1} \\ U_{n+1} & \overline{D}_{n+1} \end{bmatrix} = \frac{z^{-\frac{n}{2}}}{T_n} \begin{bmatrix} P_n^* & Q_n^* \\ Q_n & P_n \end{bmatrix} \begin{bmatrix} D_1 & \overline{U}_1 \\ U_1 & \overline{D}_1 \end{bmatrix} . \qquad (118)$$

The determinant of (118) is

$$D_{n+1}\overline{D}_{n+1} - U_{n+1}\overline{U}_{n+1} = \frac{z^{-n}}{T_n^2} (P_n^* P_n - Q_n^* Q_n)(D_1\overline{D}_1 - U_1\overline{U}_1) , \qquad (119)$$

where we have used the fact that $det(kC) = k^m det(C)$ (k is a scalar, and C is an order m matrix). The determinant of (114) is

$$P_n P_n^* - Q_n Q_n^* = z^n \prod_{i=1}^{n} (1 - c_i^2) = z^n T_n^2 , \qquad (120)$$

and so (119) becomes

$$D_{n+1}\overline{D}_{n+1} - U_{n+1}\overline{U}_{n+1} = D_1\overline{D}_1 - U_1\overline{U}_1 . \qquad (121)$$

This may be interpreted to mean that the net downgoing energy in layer $n+1$ equals the net downgoing energy in layer 1 (see Robinson and Treitel [69-70]). At infinite depth $U_\infty = 0$, and the net downgoing energy is then $D_\infty \overline{D}_\infty$ which is in the form of a *spectral function* (z-transform of an autocorrelation function - see [69-70]). That is, $D_\infty \overline{D}_\infty$ is the z-transform of the autocorrelation of $d_\infty(t)$. Let this autocorrelation function be r_t. Thus,

$$R(z) = \sum_{t=-\infty}^{\infty} r_t\, z^t = D_\infty \overline{D}_\infty \; .$$

But by (121),

$$R = D_{n+1}\overline{D}_{n+1} - U_{n+1}\overline{U}_{n+1} = D_1\overline{D}_1 - U_1\overline{U}_1 \; . \tag{122}$$

From (115) $D_1 = 1 - c_0 U_1$ and so

$$D_1\overline{D}_1 - U_1\overline{U}_1 = (1 - c_0U_1)(1 - c_0\overline{U}_1) - U_1\overline{U}_1 = 1 - c_0U_1 - c_0\overline{U}_1 \; , \tag{123}$$

as $c_0^2 = 1$. Thus,

$$R = (1 - c_0U_1) - c_0\overline{U}_1 = D_1 - c_0\overline{U}_1 \tag{124a}$$

$$R = (1 - c_0\overline{U}_1) - c_0U_1 = \overline{D}_1 - c_0U_1 \; . \tag{124b}$$

From (124a,b) we have $R = \overline{R}$ (i.e., autocorrelation is symmetric about $t = 0$ as we would expect).

Adding $-c_0$ times (116) to (117) (suitably rearranged) gives

$$\begin{bmatrix} -c_0D_{n+1} + \overline{U}_{n+1} \\ -c_0U_{n+1} + \overline{D}_{n+1} \end{bmatrix} = \frac{z^{-\frac{n}{2}}}{T_n} \begin{bmatrix} P_n^* & Q_n^* \\ Q_n & P_n \end{bmatrix} \begin{bmatrix} -c_0D_1 + \overline{U}_1 \\ -c_0U_1 + \overline{D}_1 \end{bmatrix} . \tag{125}$$

From (125)

$$-c_0U_{n+1} + \overline{D}_{n+1} = \frac{z^{-\frac{n}{2}}}{T_n} \left[Q_n(-c_0D_1 + \overline{U}_1) + P_n(-c_0U_1 + \overline{D}_1)\right] . \tag{126}$$

From (124a)

$$-c_0D_1 + \overline{U}_1 = -c_0(D_1 - c_0\overline{U}_1) = -c_0R \; ,$$

where we have used $c_0^2 = 1$, and from (124b), $R = -c_0 U_1 + \overline{D}_1$, and so (126) becomes

$$-c_0 U_{n+1} + \overline{D}_{n+1} = \frac{z^{-\frac{n}{2}}}{T_n} (P_n - c_0 Q_n) R \ . \tag{127}$$

Let $A_n = P_n - c_0 Q_n$, and so (127) becomes

$$A_n R = T_n z^{\frac{n}{2}} (-c_0 U_{n+1} + \overline{D}_{n+1}). \tag{128}$$

If we expand the right-hand side of (128), then

$$T_n z^{\frac{n}{2}}(-c_0 U_{n+1} + \overline{D}_{n+1}) = .. + T_n d_{n+1}(\frac{n}{2}+1)z^{-1} + T_n^2 + .. - T_n^2 c_0 c_{n+1} z^{n+1} - T_n c_0 u_{n+1}(\frac{n}{2}+2)z^{n+2} + .. \ . \tag{129}$$

Note that the time-domain equivalent of (129) is zero for $t = 1, \ldots, n$. Hence, operator A_n acts upon R to annihilate $r_1, \ldots, r_n$. We can relate the present discussion to the Levinson-Durbin algorithm in the following revealing way.

Consider the autocorrelation sequence r_t, with $r_t = r_{-t}$. An operator defined by the parameters $a_{k,0}, a_{k,1}, \cdots, a_{k,k}$ $(a_{k,0} = 1)$ acts upon r_t to produce g_t as follows:

$$g_t = \sum_{s=0}^{k} a_{k,s} \, r_{t-s} \ . \tag{130}$$

We want $g_t = 0$ for $t = 1,...,k$, and $g_0 = \sigma_k$. Clearly then, (130) is like (1) in section 2.1 in this event $(t_i = r_i)$. We recall from section 2.1 that the Levinson-Durbin recursion computes a_{k+1} given a_k (and a few other parameters) such that $g_{k+1} = 0$. From (130) then

$$g_{k+1} = \sum_{s=0}^{k} r_{(k+1)-s} \, a_{k,s} \tag{131}$$

which is really γ_k in the pseudocode for the Levinson-Durbin algorithm in the Hermitian Toeplitz case. Thus, $K_{k+1} = -\frac{g_{k+1}}{\sigma_k}$ or $g_{k+1} = -\sigma_k K_{k+1} = -g_0 K_{k+1}$. The z-domain form of g_t is

$$\cdots + g_{-1}z^{-1} + g_0 + g_{n+1}z^{n+1} + g_{n+2}z^{n+2} + \cdots \quad , \tag{132}$$

and this may be compared to (129) giving

$$g_0 = T_n^2 \ ,$$

$$g_{n+1} = -T_n^2 c_0 c_{n+1} = -g_0 c_0 c_{n+1} \ ,$$

or $-g_0 K_{n+1} = -g_0 c_0 c_{n+1}$ implying that

$$K_{n+1} = c_0 c_{n+1} \ . \tag{133}$$

This kind of comparison is reasonable since $A_n R$ from (128) is essentially the right-hand side of (130). Evidently, the reflection coefficients produced by the Levinson-Durbin algorithm are essentially the same as the "physical" reflection coefficients c_n of the layered earth model.

3.6 Lattice Filters

We now demonstrate a connection between Toeplitz matrix problems, and so-called *orthogonal polynomials*. The central result will be that the solution of a Toeplitz matrix problem yields a class of digital filters called *lattice filters*. The results to follow are taken from Markel and Gray [14], and from Gray and Markel [71].

We begin by presenting an alternative derivation of the Levinson-Durbin algorithm for Hermitian Toeplitz matrices. What follows is based primarily upon results in [14], but we note that [14] does not actually consider the Hermitian Toeplitz case. Markel and Gray [14] only consider the case of a real and symmetric Toeplitz matrix, and so the derivation to follow is somewhat more general than that in [14].

Suppose we are given a discrete-time sequence of complex-valued numbers, and that these numbers are used to produce the complex-valued autocorrelation sequence c_i , $i = 0,1,...,n$. Let us assume the following model for this signal. Consider a digital filter with system function

$$H_n(z) = \frac{\sigma}{x_n(z)} \ , \tag{134a}$$

where

$$x_n(z) = \sum_{i=0}^{n} x_{n,i} z^i \quad , x_{n,0} = 1 \quad . \tag{134b}$$

Note that, except for $x_{n,0}$, $x_{n,i}$ are complex-valued in general. It is clearly true that the impulse response sequence of $H_n(z)$, which we shall designate as h_k, satisfies

$$h_k = \sigma\delta_k - \sum_{l=1}^{n} x_{n,l} h_{k-l} \ , \tag{135}$$

where δ_k is the Kronecker delta. If the filter is stable, then the autocorrelation sequence r_k of output h_k can be defined as

$$\overline{r}_{l-k} = \sum_{i=-\infty}^{\infty} h_{i-l} \overline{h}_{i-k} = \sum_{i=-\infty}^{\infty} h_i \overline{h}_{i+l-k} \ , \tag{136}$$

where the bar denotes complex-conjugation. We have $r_k = \overline{r}_{-k}$. We may rewrite (135) as

$$\sigma\delta_k = \sum_{i=0}^{n} x_{n,i} h_{k-i} \quad . \tag{137}$$

Multiplying (137) by $\overline{h}_{k-\nu}$ and summing over all k yields

$$\sigma\overline{h}_{-\nu} = \sum_{i=0}^{n} x_{n,i} \overline{r}_{i-\nu} \quad . \tag{138}$$

From (135), $h_0 = \sigma$, where we are assuming that the filter is causal so that $h_k = 0$ for $k < 0$. Thus, (138) becomes

$$\sum_{i=0}^{n} x_{n,i} \overline{r}_i = | \sigma |^2 \ , \tag{139a}$$

$$\sum_{i=0}^{n} x_{n,i} \overline{r}_{i-k} = 0 \ , \ (k = 1,\ldots,n) \ . \tag{139b}$$

If we set the measured values c_i equal to the model values r_i, then

$$\sum_{i=0}^{n} x_{n,i} \overline{c}_i = \sigma_n \ , \tag{140a}$$

$$\sum_{i=0}^{n} x_{n,i} \overline{c}_{i-k} = 0 \ , \ (k = 1,\ldots,n) \ , \tag{140b}$$

where $\sigma_n = |\sigma|^2$. Clearly, (140a,b) is a Toeplitz system of equations of the same form as (1) in section 2.1. Thus, our measured autocorrelation sequence is assumed to have originated from an all-pole filter $H_n(z)$ that was "struck" with an impulse. A system of equations like (140a,b) can also be obtained on the assumption that $H_n(z)$ is being driven by white noise (see [9]).

If $F(z) = \sum_{k=-\infty}^{\infty} f_k z^k$, then let $F^\dagger(z) = \sum_{k=-\infty}^{\infty} \bar{f}_k z^{-k}$. Let $C(z) = \sum_{k=-\infty}^{\infty} c_k z^k$; this is the z-transform of the measured autocorrelation sequence, and it is a spectral function. From *Laurent's theorem* (see Kreyszig [72], page 711)

$$c_k = \bar{c}_{-k} = \frac{1}{2\pi\sqrt{-1}} \int_\Gamma C(z)\, z^{-k-1}\, dz \tag{141}$$

for $k = 0,1,2,\ldots$. Γ is a simple closed contour enclosing the origin of the complex z-plane. This is essentially (12a) in [14]. From [14] we have the inner product

$$<F(z),G(z)> = \frac{1}{2\pi\sqrt{-1}} \int_\Gamma C(z)\, F^\dagger(z) G(z) z^{-1}\, dz \ , \tag{142}$$

where

$$<cF(z),bG(z) + aH(z)> = b\bar{c}<F(z),G(z)> + a\bar{c}<F(z),H(z)> \ , \tag{143a}$$

$$<z^n F(z),z^n G(z)> = <F(z),G(z)> \ , \tag{143b}$$

$$<F(z),G(z)> = <1,F^\dagger(z)G(z)> \ . \tag{143c}$$

We have

$$<z^k,z^i> = \frac{1}{2\pi\sqrt{-1}} \int_\Gamma C(z)\, z^{-(k-i)-1}\, dz = c_{k-i} = \bar{c}_{i-k} \ . \tag{144}$$

Thus, (140b) becomes

$$\sum_{i=0}^{n} x_{n,i}<z^k,z^i> = <z^k, \sum_{i=0}^{n} x_{n,i} z^i> = <z^k,x_n(z)> = 0 \ , \tag{145}$$

for $k = 1,\ldots,n$. In other words, $x_n(z)$ is orthogonal to z^k for $k = 1,\ldots,n$, and so is called an orthogonal polynomial.

We may write

$$<z^{\nu},x_n(z)> = <x_n^{\dagger}(z),z^{-\nu}> = <z^{n+1}x_n^{\dagger}(z),z^{-\nu+(n+1)}> = 0\ , \tag{146}$$

where $\nu = 1,...,n$. Define

$$y_n(z) = z^{n+1}x_n^{\dagger}(z) = \sum_{i=0}^{n} \bar{x}_{n,i}\, z^{n+1-i} = \sum_{j=1}^{n+1} \bar{x}_{n,(n+1)-j}\, z^j = \sum_{j=1}^{n+1} y_{n,j} z^j\ , \tag{147}$$

so that $y_{n,j} = \bar{x}_{n,(n+1)-j}$ and since $x_{n,0} = 1$, we have $y_{n,n+1} = 1$. Thus, (146) becomes

$$<y_n(z),z^{\nu}> = 0\ , \tag{148}$$

for $\nu = 1,...,n$. $y_n(z)$ is another orthogonal polynomial, and its coefficients are the same as those of $x_n(z)$ except that they are in reverse order and are complex-conjugates. As well, the degree of $y_n(z)$ is $n+1$, whereas the degree of $x_n(z)$ is n.

We want $x_{n+1}(z)$ such that $x_{n+1,0} = 1$ and $<z^{\nu},x_{n+1}(z)> = 0$ for $\nu = 1,...,n+1$. We can construct such a function via

$$x_{n+1}(z) = x_n(z) + k_n y_n(z) \tag{149}$$

since $\deg\{y_n\} = n+1$, $y_{n,0} = 0$ and $x_{n,0} = 1$. Thus, we must find k_n. This can be accomplished as follows:

$$\begin{aligned} 0 = <z^{n+1},x_{n+1}(z)> &= <z^{n+1},x_n(z) + k_n y_n(z)> \\ &= <z^{n+1},x_n(z)> + k_n <z^{n+1},y_n(z)> \\ &= \beta_n + k_n \alpha_n\ . \end{aligned} \tag{150}$$

Hence,

$$k_n = -\frac{\beta_n}{\alpha_n}\ . \tag{151}$$

For $n = 0$, $\alpha_0 = <z,y_0(z)> = <z,z> = c_0$, $\beta_0 = <z,x_0(z)> = <z,1> = c_1$, and we have $x_{0,0} = 1$. These are initial conditions.

We may write

$$\begin{aligned}
\beta_n &= \langle z^{n+1}, x_n(z)\rangle = \langle z^{n+1}, \sum_{i=0}^{n} x_{n,i} z^i \rangle \\
&= \langle 1, \sum_{i=0}^{n} x_{n,i} z^{-(n+1)+i} \rangle \\
&= \langle 1, \sum_{j=1}^{n+1} x_{n,(n+1)-j} z^{-j} \rangle \\
&= \sum_{j=1}^{n+1} x_{n,(n+1)-j} \langle 1, z^{-j} \rangle \\
&= \sum_{j=1}^{n+1} x_{n,(n+1)-j} c_j \\
&= \sum_{i=0}^{n} c_{(n+1)-i} x_{n,i} \quad ,
\end{aligned} \tag{152}$$

and this is equivalent to γ_n in (26) of section 2.1. Similarly,

$$\begin{aligned}
\alpha_n &= \langle z^{n+1}, y_n(z)\rangle = \langle z^{n+1}, z^{n+1} x_n^{\dagger}(z)\rangle \\
&= \langle 1, x_n^{\dagger}(z)\rangle \\
&= \langle x_n(z), 1\rangle \\
&= \langle 1, \sum_{i=0}^{n} \overline{x}_{n,i} z^{-i} \rangle \\
&= \sum_{i=0}^{n} \overline{x}_{n,i} \langle 1, z^{-i} \rangle \\
&= \sum_{i=0}^{n} \overline{x}_{n,i} c_i = \overline{\sigma}_n \quad ,
\end{aligned} \tag{153}$$

where the last equality is due to (140a). Note that

$$\begin{aligned}
\langle x_n(z), x_n(z)\rangle &= \langle \sum_{i=0}^{n} x_{n,i} z^i, \sum_{j=0}^{n} x_{n,j} z^j \rangle \\
&= \sum_{j=0}^{n} x_{n,j} \langle x_n(z), z^j \rangle
\end{aligned}$$

$$= \sum_{j=0}^{n} x_{n,j} \left[\sum_{k=0}^{n} \overline{x}_{n,k} c_{k-j} \right]$$

$$= \overline{\sigma}_n \quad ,$$

where we have used (140a,b) again. Thus,

$$\alpha_n = \langle x_n(z), x_n(z) \rangle = \overline{\sigma}_n \quad . \tag{154}$$

From (153) $\alpha_n = \langle x_n(z), 1 \rangle$ and so

$$\begin{aligned}
\alpha_{n+1} - \alpha_n &= \langle x_{n+1}(z), 1 \rangle - \langle x_n(z), 1 \rangle \\
&= \langle x_n(z) + k_n y_n(z), 1 \rangle - \langle x_n(z), 1 \rangle \\
&= \langle 1, x_n^{\dagger}(z) + \overline{k}_n y_n^{\dagger}(z) \rangle - \langle x_n(z), 1 \rangle \\
&= \overline{k}_n \langle 1, y_n^{\dagger}(z) \rangle \\
&= \overline{k}_n \langle z^{n+1} x_n^{\dagger}(z), 1 \rangle \\
&= \overline{k}_n \langle z^{n+1}, x_n(z) \rangle \\
&= \overline{k}_n \beta_n \\
&= \overline{k}_n (-\alpha_n k_n) \quad ,
\end{aligned}$$

and so

$$\alpha_{n+1} = \alpha_n (1 - | \, k_n \, |^2) \quad , \tag{155}$$

or $\overline{\sigma}_{n+1} = \overline{\sigma}_n (1 - | \, k_n \, |^2)$. Since $\sigma_0 = c_0$ is real, σ_n is real for all n and $\alpha_n = \sigma_n$ is real for all n. If we note that (149) is essentially the same as (31), since $x_k = a_k$, then we have in fact derived the Levinson-Durbin algorithm for Hermitian Toeplitz matrices by a different means. However, the present orthogonal polynomial interpretation leads to interesting (and useful) digital filter structures.

We have $x_{n+1}(z) = x_n(z) + k_n y_n(z)$ (from (149)), so

$$y_{n+1}(z) = z^{n+2} x_{n+1}^{\dagger}(z)$$

$$
\begin{aligned}
&= z^{n+2}[x_n^\dagger(z) + \bar{k}_n y_n^\dagger(z)] \\
&= z^{n+2}[z^{-(n+1)} y_n(z) + \bar{k}_n z^{-(n+1)} x_n(z)] \\
&= z y_n(z) + \bar{k}_n z x_n(z) \ .
\end{aligned}
\tag{156}
$$

We may combine (149) with (156) to get

$$
\begin{bmatrix} x_{n+1}(z) \\ y_{n+1}(z) \end{bmatrix} = \begin{bmatrix} 1 & k_n \\ \bar{k}_n z & z \end{bmatrix} \begin{bmatrix} x_n(z) \\ y_n(z) \end{bmatrix} , \tag{157}
$$

where $x_0(z) = 1$, and $y_0(z) = z$. Notice the similarity between (157) and (112) of section 3.5. Given the results of this and the previous section, we should not be surprised. Interpreting z as a unit time delay operator, we may use (157) to construct the digital filter of Figure 2(a,b). This is essentially the *lattice nth order predictor* of Figure 4.9 in Hönig and Messerschmitt [9] (see p. 102). It is a structure due to Gray and Markel [71]. We shall not investigate the origin of the term "predictor" here, but shall instead refer the reader to [9] for an explanation. It is evident that the reflection coefficients k_n parametrize the lattice filter.

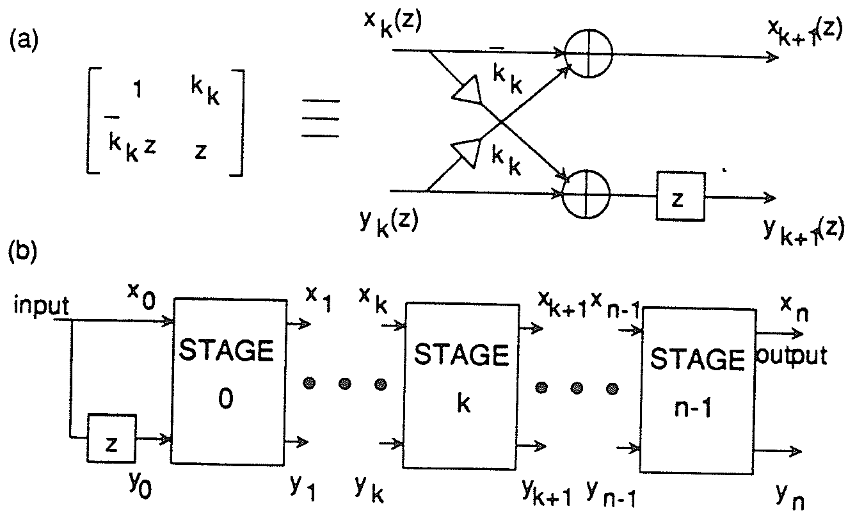


Figure 2: (a) Correspondence between the 2×2 matrix operator of equation (157) and a filter section; (b) Gray-Markel lattice nth order predictor.

The filter of Figure 2(b) implements the denominator polynomial $x_n(z)$ of $H_n(z)$ in (134a). From the standpoint of a VLSI implementation it is evident that this filter has desirable features. The filter is composed of a single basic building block (as shown in Figure 2(a)), and it can be readily pipelined for high throughput applications. The filter also has a regular linear layout.

It is possible to construct a lattice filter to realize $H_n(z)$, the all-pole filter. However, we shall not show how to do this here. Instead, we refer the reader to Gray and Markel [71], or to [9]. We note as well that lattice pole-zero filters can be constructed (see [9] or [71]). These filters are known to be superior to direct form digital filters under finite precision arithmetic implementation conditions (see Markel and Gray [73], or Gray and Markel [74]).

The Gray-Markel orthogonal filter structures that we have discussed here have been generalized in many ways. Generalizations may be found in Delsarte, Genin and Kamp [75], Lev-Ari and Kailath [76], Lev-Ari, Kailath and Cioffi [77], and in Hönig and Messerschmitt [9], Friedlander [10], and Delsarte and Genin [25]. We emphasize that this list of references is far from complete.

REFERENCES

[1] N. Levinson, "The Wiener RMS Error Criterion in Filter Design and Prediction," J. Math. Phys., vol. 25, Jan. 1946, pp. 261-278.

[2] J. Durbin, "The Fitting of Time Series Models," Rev. Int. Stat. Inst., vol. 28, 1960, pp. 233-244.

[3] R. A. Roberts, C. T. Mullis, *Digital Signal Processing*. Reading, Massachusetts: Addison-Wesley, 1987.

[4] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, Massachusetts: Addison-Wesley, 1985.

[5] L. Datta, S. D. Morgera, "Some Results on Matrix Symmetries and a Pattern Recognition Application," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-34, Aug. 1986, pp. 992-994.

[6] S.-Y. Kung, Y. H. Hu, "A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-31, Feb. 1983, pp. 66-75.

[7] J. Rissanen, "Solution of Linear Equations with Hankel and Toeplitz Matrices," Numerische Mathematik, vol. 22, 1974, pp. 361-366.

[8] P. Delsarte, Y. V. Genin, Y. G. Kamp, "A Generalization of the Levinson Algorithm for Hermitian Toeplitz Matrices with Any Rank Profile," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Aug. 1985, pp. 964-971.

[9] M . L. Hönig, D. G. Messerschmitt, *Adaptive Filters Structures Algorithms, and Applications*. Hingham, Massachusetts: Kluwer Academic Publ., 1984.

[10] B. Friedlander, "Lattice Filters for Adaptive Processing," Proc. IEEE, vol. 70, Aug. 1982, pp. 829-867.

[11] G. Carayannis, "An Alternative Formulation for the Recursive Solution of the Covariance and Autocorrelation Equations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-25, Dec. 1977, pp. 574-577.

[12] E. A. Robinson, S. Treitel, "Maximum Entropy and the Relationship of the Partial Autocorrelation to the Reflection Coefficients of a Layered System," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-28, April 1980, pp. 224-235.

[13] A. Bruckstein, T. Kailath, "An Inverse Scattering Framework for Several Problems in Signal Processing," IEEE ASSP Magazine, vol. 4, Jan. 1987, pp. 6-20.

[14] J. D. Markel, A. H. Gray, Jr., "On Autocorrelation Equations as Applied to Speech Analysis," IEEE Trans. on Audio and Electroacoustics, vol. AU-21, April 1973, pp. 69-79.

[15] W. F. Trench, "An Algorithm for the Inversion of Finite Toeplitz Matrices," J. SIAM, vol. 12, Sept. 1964, pp. 515-522.

[16] S. Zohar, "Toeplitz Matrix Inversion: The Algorithm of W. F. Trench," J. ACM, vol. 16, Oct. 1969, pp. 592-601.

[17] H. Akaike, "Block Toeplitz Matrix Inversion," SIAM J. Appl. Math., vol. 24, March 1973, pp. 234-241.

[18] S. Zohar, "The Solution of a Toeplitz Set of Linear Equations," J. ACM, vol. 21, April 1974, pp. 272-276.

[19] I. C. Gohberg, A. A. Semencul, "On the Inversion of Finite Toeplitz Matrices and Their Continuous Analogs," Mat. Issled. (in Russian), vol. 2, 1972, pp. 201-233.

[20] T. Kailath, A. Vieira, M. Morf, "Inverses of Toepitz Operators, Innovations, and Orthogonal Polynomials," SIAM Review, vol. 20, Jan. 1978, pp. 106-119.

[21] E. H. Bareiss, "Numerical Solution of Linear Equations with Toeplitz and Vector Toeplitz Matrices," Numerische Mathematik, vol. 13, 1969, pp. 404-424.

[22] R. P. Brent, F. T. Luk, "A Systolic Array for the Linear-Time Solution of Toeplitz Systems of Equations," J. of VLSI and Comp. Sys., vol. 1, 1983, pp. 1-22.

[23] P. Delsarte, Y. V. Genin, "The Split Levinson Algorithm," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-34, June 1986, pp. 470-478.

[24] H. Krishna, S. D. Morgera, "The Levinson Recurrence and Fast Algorithms for Solving Toeplitz Systems of Linear Equations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, June 1987, pp. 839-848.

[25] P. Delsarte, Y. V. Genin, "On the Splitting of Classical Algorithms in Linear Prediction Theory," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, May 1987, pp. 645-653.

[26] J. Le Roux, C. Gueguen, "A Fixed Point Computation of Partial Correlation Coefficients," IEEE Trans, on Acoust., Speech, and Signal Proc., vol. ASSP-25, June 1977, pp. 257-259.

[27] R. Kumar, "A Fast Algorithm for Solving a Toeplitz System of Equations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Feb. 1985, pp. 254-267.

[28] R. P. Brent, F. G. Gustavson, D. Y. Y. Yun, "Fast Solution of Toeplitz Systems of Equations and Computation of Padé Approximants," J. Algorithms, vol. 1, 1980, pp. 259-295.

[29] R. R. Bitmead, B. D. O. Anderson, "Asymptotically Fast Solution of Toeplitz and Related Systems of Linear Equations," Lin. Alg. and its Appl., vol. 34, 1980, pp. 103-116.

[30] F. G. Gustavson, D. Y. Y. Yun, "Fast Algorithms for Rational Hermite Approximation and Solution of Toeplitz Systems," IEEE Trans. on Circ. and Syst., vol. CAS-26, Sept. 1979, pp. 750-755.

[31] A. K. Jain, "Fast Inversion of Banded Toeplitz Matrices by Circular Decompositions," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-26, April 1978, pp. 121-126.

[32] B. W. Dickinson, "Efficient Solution of Linear Equations with Banded Toeplitz Matrices," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-27, Aug. 1979, pp. 421-423.

[33] G. A. Merchant, T. W. Parks, "Efficient Solution of a Toeplitz-Plus-Hankel Coefficient Matrix System of Equations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-30, Feb. 1982, pp. 40-44.

[34] G. Carayannis, N. Kalouptsidis, D. G. Manolakis, "Fast Recursive Algorithms for a Class of Linear Equations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-30, April 1982, pp. 227-239.

[35] T. Kailath, S.-Y. Kung, M. Morf, "Displacement Ranks of a Matrix," Bull. Amer. Math. Soc., vol. 1, Sept, 1979, pp. 769-773.

[36] T. Kailath, S.-Y. Kung, M. Morf, "Displacement Ranks of Matrices and Linear Equations," J. Math. Anal. Appl., vol. 68, 1979, pp. 395-407.

[37] B. Friedlander, M. Morf, T. Kailath, L. Ljung, "New Inversion Formulas for Matrices Classified in Terms of Their Distance from Toeplitz Matrices," Lin. Alg. and is Appl., vol. 27, 1979, pp. 31-60.

[38] N. Kalouptsidis, D. Manolakis, G. Carayannis, "Efficient Triangularization, Inversion and System Solution for Near-to-Toeplitz Matrices," Proc. 1983 IEEE Int. Conf. on Acoust., Speech, and Signal Proc., pp. 24-27.

[39] M. Morf, B. Dickinson, T. Kailath, A. Vieira, "Efficient Solution of Covariance Equations for Linear Prediction," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-25, Oct. 1977, pp. 429-433.

[40] E. V. Jull, D. C. W. Hui, P. Facq, "Scattering by Dual-Blazed Corrugated Conducting Strips and Small Reflection Gratings," J. Opt. Soc. Amer. A, vol. 2, July 1985, pp. 1049-1056.

[41] P. Facq, "Diffraction par des Structures Cylindriques Périodiques Limitées," Ann. Telecommun., vol. 31, 1976, pp. 99-107.

[42] K. Aki, P. G. Richards, *Quantitative Seismology Theory and Methods*. San Francisco, California: Freeman, 1980.

[43] R. A. Usmani, *Applied Linear Algebra*. Course notes to Engineering Analysis III, course no. 24.810, Dept. of Appl. Math., University of Manitoba, Winnipeg, Manitoba, Canada, 1984.

[44] L. C. Wood, S. Treitel, "Seismic Signal Processing," Proc. IEEE, vol. 63, April 1975, pp. 649-661.

[45] R. A. Wiggins, E. A. Robinson, "Recursive Solution to the Multichannel Filtering Problem," J. Geophys. Res., vol. 70, April 1965, pp. 1885-1890.

[46] R. W. Schafer, L. R. Rabiner, "Digital Representations of Speech Signals," Proc. IEEE, vol. 63, April 1975, pp. 662-677.

[47] R. E. Blahut, "Algebraic Fields, Signal Processing, and Error Control," Proc. IEEE, vol. 73, May 1985, pp. 874-893.

[48] R. E. Blahut, *Theory and Practice of Error Control Codes*. Reading, Massachusetts: Addison-Wesley, 1983.

[49] W. B. Jones, A. O. Steinhardt, "Finding the Poles of the Lattice Filter," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Oct. 1985, pp. 1328-1331.

[50] M. R. Raghuveer, C. L. Nikias, "Bispectrum Estimation: A Parametric Approach," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Oct. 1985, pp. 1213-1230.

[51] S. Kay, "Some Results in Linear Interpolation Theory," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-31, June 1983, pp. 746-749.

[52] M. Ohsmann, "An Iterative Method for the Identification of MA Systems," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-36, Jan. 1988, pp. 106-109.

[53] M. Korenberg, "Functional Expansions, Parallel Cascades and Nonlinear Difference Equations," in *Advanced Methods of Physiological System Modeling*, vol. I (V. Z. Marmarelis, editor). University of Southern California, Los Angeles, California: Biomedical Simulations Resource, 1987, pp. 221-240.

[54] J. Capon, "High Resolution Frequency-Wavenumber Spectrum Analysis," Proc. IEEE, vol. 57, Aug. 1969, pp. 1408-1418.

[55] B. R. Musicus, "Fast MLM Power Spectrum Estimation from Uniformly Spaced Correlations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Oct. 1985, pp. 1333-1335.

[56] W. B. Gragg, "The Padé Table and Its Relation to Certain Algorithms of Numerical Analysis," SIAM Review, vol. 14, Jan. 1972, pp. 1-62.

[57] G. Cybenko, "Restrictions of Normal Operators, Padé Approximation and Autoregressive Time Series," SIAM J. Math. Anal., vol. 15, July 1984, pp. 753-767.

[58] L. Weiss, R. N. McDonough, "Prony's Method, z-Transforms, and Padé Approximation," SIAM Review, vol. 5, April 1963, pp. 145-149.

[59] R. J. McEliece, J. B. Shearer, "A Property of Euclid's Algorithm and an Application to Padé Approximation," SIAM J. Appl. Math., vol. 34, June 1978, pp. 611-615.

[60] K. Imamura, "Two Recursive Algorithms for Solving the Scalar Partial Realization Problem," Proc. 1985 IEEE Int. Symp. on Circ. and Syst., Kyoto, Japan, June 5-7, 1985, pp. 823-824.

[61] T. Kailath, *Linear Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

[62] V. F. Pisarenko, "The Retrieval of Harmonics from a Covariance Function," Geophys. J. Roy. Astron. Soc., vol. 33, 1973, pp. 347-366.

[63] S. M. Kay, S. L. Marple, "Spectrum Analysis - A Modern Perspective," Proc. IEEE, vol. 69, Nov. 1981, pp. 1380-1419.

[64] Y. H. Hu, S.-Y. Kung, "Toeplitz Eigensystem Solver," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Oct. 1985, pp. 1264-1271.

[65] M. H. Hayes, M. A. Clements, "An Efficient Algorithm for Computing Pisarenko's Harmonic Decomposition Using Levinson's Recursion," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-34, June 1986, pp. 485-491.

[66] J. T. Tou, R. C. Gonzalez, *Pattern Recognition Principles*. Reading, Massachusetts: Addison-Wesley, 1974.

[67] H. L. Van Trees, *Detection, Estimation and Modulation Theory,* Part I. New York, New York: John Wiley & Sons, 1968.

[68] T. Kailath, B. C. Levy, L. Ljung, M. Morf, "Fast Time-Invariant Implementations of Gaussian Signal Detectors," IEEE Trans. on Info. Theo., vol. IT-24, July 1978, pp. 469-477.

[69] E. A. Robinson, S. Treitel, "The Spectral Function of a Layered System and the Determination of Waveforms at Depth," Geophys. Prosp., vol. 25, 1977, pp. 434-459.

[70] E. A. Robinson, S. Treitel, "The Fine Structure of the Normal Incidence Synthetic Seismogram," Geophys. J. Roy. Astron. Soc., vol. 53, 1978, pp. 289-310.

[71] A. H. Gray, Jr., J. D. Markel, "Digital Lattice and Ladder Filter Synthesis," IEEE Trans. on Audio and Electroacoustics, vol. AU-21, Dec. 1973, pp. 491-500.

[72] E. Kreyszig, *Advanced Engineering Mathematics,* 4th edition. New York, New York: John Wiley & Sons, 1979.

[73] J. D. Markel, A. H. Gray, Jr., "Roundoff Noise Characteristics of a Class of Orthogonal Polynomial Structures," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-23, Oct. 1975, pp. 473-486.

[74] A. H. Gray, Jr., J. D. Markel, "A Normalized Digital Filter Structure," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-23, June 1975, pp. 268-

276.

[75] P. Delsarte, Y. Genin, Y. Kamp, "Orthogonal Polynomial Matrices on the Unit Circle," IEEE Trans. on Circ. and Syst., vol. CAS-25, Mar. 1978, pp. 149-160.

[76] H. Lev-Ari, T. Kailath, "Lattice Filter Parametrization and Modeling of Nonstationary Processes," IEEE Trans. on Info. Theo., vol. IT-30, Jan. 1984, pp. 2-16.

[77] H. Lev-Ari, T. Kailath, J. Cioffi, "Least-Squares Adaptive Lattice and Transversal Filters: A Unified Geometric Approach," IEEE Trans. on Info. Theo., vol. IT-30, Mar. 1984, pp. 222-236.

Chapter III

SOME FURTHER RESULTS ON SCHUR AND SPLIT SCHUR ALGORITHMS

In Zarowski and Card [1] it is shown that the Schur algorithm of Le Roux and Gueguen [2] is equivalent to the Schur algorithm of Kung and Hu [3]. To our knowledge, this has not been noted previously. This equivalence, which we have stated in Chapter II, shall be demonstrated in this chapter. The symmetric split Schur algorithm of Delsarte and Genin [4] was rederived in [1] using the Schur algorithm in Kung and Hu [3], rather than the algorithm of Le Roux and Gueguen [2]. The rederivation was presented in Chapter II of this thesis (see section 2.5). In [1] an inverse mapping from the split Schur variables to the Schur variables is obtained, as no such mapping was derived by Delsarte and Genin [4]. This inverse mapping is important in the context of a parallel-pipelined processor implementation of the split Schur algorithms (see Chapter IV), and so we shall include it here. Finally, we shall present the derivation of a Schur algorithm for Hermitian Toeplitz matrices of any rank profile. This algorithm is due to Zarowski and Card [5], and it is derived using the Levinson-Durbin algorithm of Delsarte, Genin and Kamp [6] for such matrices, and the Kung-Hu Schur algorithm [3] for strongly nonsingular Hermitian Toeplitz matrices (see Chapter II, section 2.3).

1. The Le Roux-Gueguen and Kung-Hu Schur Algorithms are Equivalent

Let us assume that T_n is a real and symmetric Toeplitz matrix (so $T_n = [t_{|j-i|}]_{(n+1)\times(n+1)}$). The Le Roux-Gueguen [2] Schur algorithm is summarized in Table II of [4], and we repeat it here:

```
For $i$ := 0 to $n$ do begin
    $e_{0,i} := t_i$ ; $e_{0,-i} := t_i$ ;
```

```
        end;
    For $k$ := 1 to $n$ do begin
        $K_k := -\frac{e_{k-1,k}}{e_{k-1,0}}$;
        For $i$ := 0 to $n-k-1$ do begin
            $e_{k,-i} := e_{k-1,-i} + K_k e_{k-1,k+i}$;
            $e_{k,k+i+1} := K_k e_{k-1,-(i+1)} + e_{k-1,k+i+1}$;
            end;
        end;
```

From [2], Schur variables $e_{k,i}$ are defined as

$$e_{k,i} = <z^i, a_k(z)> \quad , \tag{1}$$

where this expression is taken from Chapter II, section 3.6 (with trivial notational changes). Recall that the material of section 3.6 in Chapter II is adapted from Markel and Gray [7]. Equation (1) then expands as

$$e_{k,i} = \sum_{j=0}^{k} t_{|i-j|} a_{k,j} \quad . \tag{2}$$

Although the similarity of the above algorithm to the Kung-Hu algorithm of Chapter II, section 2.3 is evident, because of the manner in which the variables $e_{k,i}$ are obtained in [2], it is not clear how they relate to the elements $u_i^{(k)}$ of the LDU decomposition of T_n. However, from Chapter II, section 2.3 we have

$$L_k^A T_k = U_k \quad , \tag{3}$$

where L_k^A is defined in (17b) (Chapter II, section 2.1), and U_k is the right hand side of (70) (Chapter II, section 2.3). Equation (3) expands to become

$$u_{-i}^{(k)} = \sum_{j=0}^{k-1} t_{|j+i-k+1|} a_{k-1,j} \quad . \tag{4}$$

Thus, we can compare (2) with (4), and this yields

$$e_{k,i} = u_{i-k}^{(k+1)} \quad . \tag{5}$$

We may therefore conclude that the Schur algorithm of Le Roux and Gueguen [2], and of Kung and Hu [3] are equivalent.

2. An Inverse Mapping

Given $v_{k,i}$ (symmetric Schur variables of Chapter II), we might want $u_i^{(k)}$, the elements of U_n. Thus we need an inverse mapping from values $v_{k,i}$ to values $u_i^{(k)}$. From (86a) (Chapter II, section 2.5) $v_{k+2,i-1} = u_i^{(k+1)} + u_{-(k+i)+1}^{(k+1)}$, and this may be substituted in to (87) (Chapter II, section 2.5) giving

$$u_{-(i+k)}^{(k+1)} = u_{-(i+k)+1}^{(k+1)} + (1 + K_k)v_{k+1,i} - v_{k+2,i-1} \quad , \tag{6}$$

where $0 \le k \le n$, $1 \le i \le n-k$ since we want the elements of U_n in (70) (Chapter II, section 2.3). Equation (6) is recursive, and for a given k we need $u_{-(i+k)+1}^{(k+1)} \,|_{i=1} = u_{-k}^{(k+1)} = \sigma_k$. This is an initial condition. Recall from Chapter II, section 2.5 that $\sigma_k = v_{k+1,0}(1 + K_k)$.

For the purpose of computing reflection coefficients, the split Schur algorithms require approximately half of the number of multiplications that the Schur algorithm does. However, if we are to use the split Schur algorithms to *LDU* factorize T_n, then the number of multiplications is about the same in both cases. It is the presence of a multiplication operation in (6) that makes the number of multiplications comparable. There appears to be no way of eliminating this extra multiplication. Thus, there is no apparent advantage in using the split Schur algorithms to *LDU* factorize Toeplitz matrices. However, in Chapter IV we shall see that this conclusion is false in the context of a parallel-pipelined processor implementation of the split Schur algorithms. That is, the extra multiplications due to (6) causes no trouble in this context. Only in the case of a sequential processor implementation of the split Schur algorithms is the presence of these extra multiplications a problem.

3. A Schur Algorithm for Hermitian Toeplitz Matrices of Any Rank Profile

Delsarte, Genin and Kamp [6] have derived an extended form of the Levinson-Durbin algorithm for Hermitian Toeplitz matrices. Their algorithm is able to cope with the case when one or more of the leading principal submatrices of the matrix is singular. Recall from Chapter II, section 2.1 that the classical form of the Levinson-Durbin algorithm described therein must terminate when a singular submatrix is encountered. In this section we shall use the algorithm of [6] and the Kung-Hu Schur algorithm of [3] to derive a Schur algorithm for the singular submatrix case. The result will be a Schur algorithm for Hermitian Toeplitz matrices of any rank profile. In the derivation to follow we adopt the notation of [6], which differs slightly from the notation of Chapter II. However, this will facilitate any comparisons of the results in this section to those in [6] that the reader might wish to make.

3.1 A Summary of the Delsarte, Genin and Kamp Algorithm

Let there be an $m \times m$ Hermitian Toeplitz matrix

$$C_m = \begin{bmatrix} c_0 & \overline{c}_1 & \cdots & \overline{c}_{m-1} \\ c_1 & c_0 & \cdots & \overline{c}_{m-2} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ c_{m-1} & c_{m-2} & \cdots & c_0 \end{bmatrix} , \tag{7}$$

where c_0 is real, c_i $(1 \le i \le m-1)$ are complex, and $c_{-i} = \overline{c}_i$. Let $C_k = [c_{i-j}]_{k \times k}$ ($i =$ row index, $j =$ column index) denote the kth order submatrix of C_m, so $1 \le k \le m$ and $0 \le i,j \le k-1$. Let $f_k = det(C_k)$, with $f_0 = 1$ by convention. Let $a_{k-1} = [a_{k-1,0}\ a_{k-1,1}\ \cdots\ a_{k-1,k-1}]^T$ satisfy

$$C_k a_{k-1} = [1\ 0\ \cdots\ 0]^T , \tag{8a}$$

and let $x_{k-1} = [x_{k-1,0}\ x_{k-1,1}\ \cdots\ x_{k-1,k-1}]^T$ satisfy

$$C_k x_{k-1} = [\sigma_k\ 0\ \cdots\ 0]^T , \tag{8b}$$

where $a_{k-1,0} = \dfrac{f_{k-1}}{f_k}$, $x_{k,0} = 1$ and $\sigma_k = \dfrac{f_k}{f_{k-1}}$. We may construct the *Levinson* and *predictor polynomials*, respectively,

$$a_k(z) = \sum_{i=0}^{k} a_{k,i} z^i , \quad x_k(z) = \sum_{i=0}^{k} x_{k,i} z^i , \tag{9}$$

and $a_{k-1} = \dfrac{x_{k-1}}{\sigma_k}$ (assuming $\sigma_k \neq 0$). The *reciprocal* of $a_k(z)$ is $\hat{a}_k(z) = z^k \overline{a}_k(\overline{z}^{-1})$.

Index r is called a *right singular point* as $f_r \neq 0$, but $f_{r+1} = 0$. Similarly, index n is called a *left singular point* since $f_n \neq 0$ but $f_{n-1} = 0$. It is known that

$$n - r = 2l \quad , \tag{10}$$

where l is called the *Iohvidov index* (see [6],[8]).

The algorithm of Delsarte, Genin and Kamp [6] may be summarized in pseudocode form as:

```
$k := 0$ ; $\sigma_{k+1} := c_0$; $x_k(z) := 1$; $a_{k-1}(z) := 0$;
While $k \leq m-2$ do begin
    If $\sigma_{k+1} \neq 0$ then begin
        $K_{k+1} := -\frac{1}{\sigma_{k+1}} \sum_{i=0}^{k} c_{(k+1)-i} x_{k,i}$;
        $\sigma_{k+2} := \sigma_{k+1}(1 - | K_{k+1} |^2)$;
        $a_k(z) := \frac{x_k(z)}{\sigma_{k+1}}$;
        $x_{k+1}(z) := x_k(z) + K_{k+1} z \hat{x}_k(z)$;
        $k := k + 1$;
        end
    else begin {$r = k$}
        Find smallest $l$ satisfying $\overline{c}_l x_{r,0} + \overline{c}_{l+1} x_{r,1} + .. + \overline{c}_{l+r} x_{r,r} \neq 0$;
        If $r = 0$ then begin
            $\psi_i := -\delta_{i,l}$ $(0 \leq i \leq l)$;
            $\alpha_i := c_{l+i}$ $(0 \leq i \leq l)$;
            end
        else begin
            $\psi_i := \sum_{j=0}^{r-1} c_{r+l-i-j} a_{r-1,j}$ $(0 \leq i \leq l)$;
            $\alpha_i := \sum_{j=0}^{r} c_{r+l+i-j} x_{r,j}$ $(0 \leq i \leq l)$;
            end;
        Solve $\begin{bmatrix} \alpha_0 & 0 & . & 0 \\ \alpha_1 & \alpha_0 & . & 0 \\ . & . & & . \\ . & . & & . \\ \alpha_l & \alpha_{l-1} & . & \alpha_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ . \\ . \\ p_l \end{bmatrix} := - \begin{bmatrix} \psi_l \\ \psi_{l-1} \\ . \\ . \\ \psi_0 \end{bmatrix}$;
        $p(z) := \sum_{i=0}^{2l} p_i z^i \quad (p_{l+i} = -\bar{p}_{l-i})$;
        $\beta_0 := \frac{1}{\bar{p}_0}$; $n := k + 2l$;
        $a_{n-1}(z) := \beta_0^{-1} z^l x_k(z)$;
        $x_n(z) := \bar{\beta}_0 [z^l a_{k-1}(z) + p(z) x_k(z)]$;
        $\sigma_{n+1} := |\beta_0|^2 [p_l + \bar{p}_l + a_{k-1}(0)]$;
        $k := n$;
        end;
    end;
```

Note that this program assumes C_m is nonsingular, otherwise if at any stage no l satisfying

$$\sum_{i=0}^{r} \bar{c}_{l+i} x_{r,i} \neq 0 \tag{11}$$

can be found, then the program must be modified to terminate.

3.2 The Kung and Hu Schur Algorithm

We may restate, for convenience, the Schur algorithm of Chapter II using the notation of section 3.1 (this chapter) above:

```
For $i := 0$ to $m-1$ do begin
    $u_{-i}^{(1)} := \bar{c}_i$ ; $u_i^{(1)} := c_i$;
For $k := 1$ to $m-1$ do begin
    $K_k := -\frac{u_1^{(k)}}{u_{-k+1}^{(k)}}$;
    For $i := 0$ to $m-k-1$ do begin
        $u_{-(i+k)}^{(k+1)} := u_{-k-i+1}^{(k)} + K_k \bar{u}_{i+1}^{(k)}$;
        $u_i^{(k+1)} := K_k \bar{u}_{-k-i+1}^{(k)} + u_{i+1}^{(k)}$;
        end;
    end;
```

It is clear from the results of Chapter II that we have

$$C_k = U_k^H D_k^{-1} U_k \quad , \tag{12a}$$

$$C_k^{-1} = L_k^H D_k^{-1} L_k \quad , \tag{12b}$$

where

$$D_k = diag \left\{ \sigma_1 \; \sigma_2 \; \cdots \; \sigma_k \right\} \; , \tag{13a}$$

$$U_k = \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & \cdots & u_{-k+1}^{(1)} \\ 0 & u_{-1}^{(2)} & \cdots & u_{-k+1}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & u_{-k+1}^{(k)} \end{bmatrix} , \; L_k = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ x_{1,1} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ x_{k-2,k-2} & x_{k-2,k-3} & \cdots & 1 & 0 \\ x_{k-1,k-1} & x_{k-1,k-2} & \cdots & x_{k-1,1} & 1 \end{bmatrix} , \tag{13b}$$

and where $\sigma_k = u_{-k+1}^{(k)}$. As well,

$$L_k = D_k U_k^{-H} \quad , \tag{14a}$$

$$L_k C_k = U_k \quad , \tag{14b}$$

and the latter equation expands to give us

$$\begin{aligned} u_{-i}^{(k)} &= \sum_{j=0}^{k-1} \overline{c}_{j+i-k+1} x_{k-1,j} \\ &= \sum_{j=0}^{k-1} c_{k-i-j-1} x_{k-1,j} \quad . \end{aligned} \tag{15}$$

3.3 The Desired Result

We must relate the variables in the Delsarte, Genin and Kamp algorithm summarized in section 3.1 above, to the Schur variables in order to find the required Schur algorithm.

Using (15) it is straightforward to verify that

$$u_{-(r+l)}^{(r+1)} = \sum_{i=0}^{r} c_{-(l+i)} x_{r,i} = \sum_{i=0}^{r} \overline{c}_{l+i} x_{r,i} \quad , \tag{16a}$$

$$u_{-(i-l-1)}^{(r)} = \sigma_r \sum_{j=0}^{r-1} c_{r+l-i-j} a_{r-1,j} = \sigma_r \psi_i \quad , \tag{16b}$$

$$u_{l+i}^{(r+1)} = \sum_{j=0}^{r} c_{r+l+i-j} x_{r,j} = \alpha_i \quad . \tag{16c}$$

Thus, (16a) may be used to find l, the Iohvidov index, and (16b,c) may be used to find ψ_i and α_i, respectively.

We want to compute the order n and $n+1$ Schur variables $u_i^{(n)}$ and $u_i^{(n+1)}$, respectively, if possible. Thus, from the algorithm of section 3.1,

$$a_{n-1}(z) = \beta_0^{-1} z^l x_k(z) \quad , \tag{17a}$$

$$x_n(z) = \overline{\beta}_0 [z^l a_{k-1}(z) + p(z)\, x_k(z)] \quad , \tag{17b}$$

where $k = r$. We have

$$b_{n-1} = C_n a_{n-1} = [1 \; 0 \; \cdots \; 0]^T \quad , \tag{18a}$$

$$y_n = C_{n+1} x_n = [\sigma_{n+1} \; 0 \; \cdots \; 0]^T \quad . \tag{18b}$$

Hence we may substitute (17a) into (18a) and get

$$b_{n-1,i} = \beta_0^{-1} \sum_{j=0}^{r} \overline{c}_{l+j-i} x_{r,j} \; , \; (0 \le i \le n-1) \quad . \tag{19a}$$

Similarly, substituting (17b) into (18b) produces

$$y_{n,i} = \overline{\beta}_0 \sum_{j=0}^{r-1} \overline{c}_{l+j-i} a_{r-1,j} + \overline{\beta}_0 \sum_{\nu=0}^{2l} p_\nu \sum_{j=0}^{r} \overline{c}_{\nu+j-i} x_{r,j} \; , \; (0 \le i \le n) \quad . \tag{19b}$$

We can use (15) again to express vectors b_i and y_i in terms of Schur variables. Therefore,

$$b_{n-1,i} = \beta_0^{-1} u_{i-(r+l)}^{(r+1)} \quad , \tag{20a}$$

$$y_{n,i} = \overline{\beta}_0 \left[\frac{1}{\sigma_r} u_{i-(r+l)+1}^{(r)} + \sum_{\nu=0}^{2l} p_\nu u_{i-(r+\nu)}^{(r+1)} \right] \quad . \tag{20b}$$

We have $u_i^{(0)} = 0$ (all i) since $a_{-1}(z) = 0$. We also have, from (18a) and (18b), respectively,

$$b_{n-1,i} = \sum_{j=0}^{n-1} \bar{c}_{j-i} a_{n-1,j} \ , \ y_{n,i} = \sum_{j=0}^{n} \bar{c}_{j-i} x_{n,j} \ . \tag{21}$$

The equations in (21) can be expressed in terms of Schur variables as

$$b_{n-1,i} = \frac{1}{\sigma_n} u_{i-n+1}^{(n)} \ , \ y_{n,i} = u_{i-n}^{(n+1)} \ . \tag{22}$$

There is clearly a technical problem involving the expression for $b_{n-1,i}$ in (22) however. Since $\sigma_n = \dfrac{f_n}{f_{n-1}}$ and $f_{n-1} = 0$, we have $\sigma_n = \pm\infty$. Hence, we cannot write $\dfrac{1}{\sigma_n} u_{i-n+1}^{(n)} = \beta_0^{-1} u_{i-(r+l)}^{(r+1)}$ (obtained by combining (20a) with the first equation in (22)). Thus, $u_i^{(n)}$ is unobtainable, although we do have

$$u_{i-n}^{(n+1)} = \bar{\beta}_0 \left[\frac{1}{\sigma_r} u_{i-(r+l)+1}^{(r)} + \sum_{v=0}^{2l} p_v u_{i-(r+v)}^{(r+1)} \right] , \tag{23}$$

which is obtained by combining the second equation in (22) with (20b). Equation (23) is to be evaluated for $i = 0,-1,\ldots,n-m+1$ and $i = n,\ldots,m-1$ in general.

Although we cannot get $u_i^{(n)}$, $b_{n-1,i}$ exists and is finite (see (21)). This fact is useful since if $\sigma_{n+1} = 0$, then the else case of the "If $\sigma_k \neq 0$ then" statement is executed again, where now $k = n + 1$ and $r = n$. The problem is that

$$\psi_i = \frac{1}{\sigma_n} u_{-(i-l-1)}^{(n)} \quad (0 \leq i \leq l) \tag{24}$$

in this case. But $b_{n-1,i} = \dfrac{1}{\sigma_n} u_{i-n+1}^{(n)}$, and so

$$\psi_{n+l-i} = b_{n-1,i} \quad , or \quad \psi_i = b_{n-1,n+l-i} \ . \tag{25}$$

From (20a)

$$\psi_{n+l-i} = \beta_0^{-1} u_{i-(r+l)}^{(r+1)} \quad , or \ \psi_i = \beta_0^{-1} u_{2l-i}^{(r+1)} \ . \tag{26}$$

We want $\psi_0 \cdots \psi_l$ and so $i = n, \cdots ,n+l$ in (26). Thus,

$$\psi_0 = \beta_0^{-1} u_{2l}^{(r+1)} , \ \cdots \ , \psi_l = \beta_0^{-1} u_l^{(r+1)} \ . \tag{27}$$

Since $u_i^{(r+1)}$ exists, we can obtain ψ_i as in (27). Note that it is genuinely possible that $\sigma_{n+1} = 0$. An example with this property is presented in [6]. We emphasize that the variables l and r in (26) and (27) have the values that they had just prior to the computation of $\sigma_{n+1}(=0)$. Similarly, β_0 in (26) and (27) is that value used to obtain σ_{n+1}.

Equations (16a,b,c) and (23) represent the Schur algorithm for the case where $\sigma_{k+1} = 0$ in the algorithm of section 3.1. The algorithm does not allow us to determine $u_i^{(r+2)} \cdots u_i^{(n)}$, and so some of the rows of U_m in (13b) are missing as a result. The size of the resulting gap, which we shall call a *singular gap,* is $n - (r+2) + 1 = 2l - 1$ $(l \geq 1)$ (using (10)). It is possible to use the known Schur variables, and (14a) (nonsingular case) to find the polynomials $a_{n-1}(z)$ and $x_n(z)$ however.

We may finally summarize the Schur algorithm for Hermitian Toeplitz matrices of any rank profile using pseudocode:

```
$k := 1$ ; $\sigma_k := c_0$ ; $u_{-i}^{(1)} := \bar{c}_i$ , $u_i^{(1)} := c_i$ $(0 \leq i \leq m-1)$ ;
While $k \leq m-1$ do begin
    If $\sigma_k \neq 0$ then begin
        $K_k := -\dfrac{u_1^{(k)}}{u_{-k+1}^{(k)}}$ ;
        For $i := 0$ to $m-k-1$ do begin
            $u_{-(i+k)}^{(k+1)} := u_{-k-i+1}^{(k)} + K_k \bar{u}_{i+1}^{(k)}$ ;
            $u_i^{(k+1)} := K_k \bar{u}_{-k-i+1}^{(k)} + u_{i+1}^{(k)}$ ;
            end;
        $\sigma_{k+1} := u_{-k}^{(k+1)}$ ;
        $k := k + 1$ ;
        end
    else begin $\{r = k - 1\}$
        Find smallest $l$ such that $u_{-(r+l)}^{(r+1)} \neq 0$ ;
        If $r = 0$ then begin
```

```
                $\psi_i := -\delta_{i,l}\ (0 \le i \le l)$ ;
                $\alpha_i := c_{l+i}\ (0 \le i \le l)$ ;
                end
            else begin
                $\psi_i := \frac{1}{\sigma_r} u^{(r)}_{-(i-l-1)}\ (0 \le i \le l)$ ;
                $\alpha_i := u^{(r+1)}_{l+i}\ (0 \le i \le l)$ ;
                end;
            Solve $\begin{bmatrix} \alpha_0 & 0 & . & 0 \\ \alpha_1 & \alpha_0 & . & 0 \\ . & . & & . \\ . & . & & . \\ \alpha_l & \alpha_{l-1} & . & \alpha_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ . \\ . \\ p_l \end{bmatrix} := - \begin{bmatrix} \psi_l \\ \psi_{l-1} \\ . \\ . \\ \psi_0 \end{bmatrix}$ ;
            $p_{l+i} := -p_{l-i}\ (1 \le i \le l)$ ;
            $\beta_0 := \frac{1}{\bar{p}_0}$ ; $n := 2l + k - 1$ ;
            $u^{(n+1)}_{i-n} := \bar{\beta}_0 \left[ \frac{1}{\sigma_r} u^{(r)}_{i-(r+l)+1} + \sum_{v=0}^{2l} p_v u^{(r+1)}_{i-(r+v)} \right]$
                $(0 \ge i \ge n-m+1\ ,\ n \le i \le m-1)$ ;
            $\sigma_{n+1} := |\ \beta_0\ |^2\ [p_l + \bar{p}_l + \frac{1}{\sigma_r}]$ ;
                {replace $\frac{1}{\sigma_r}$ by 0 if $r = 0$}
            $k := n + 1$ ;
            end;
    end;
```

The pseudocode omits to account for the case where $\sigma_{n+1} = 0$. This is done to simplify the presentation of the algorithm. As a final remark, it may be readily shown that the above algorithm and that of Delsarte, Genin and Kamp [6], both have time complexities of $O(m^2)$ (on a sequential processor).

3.4 Numerical Examples

We may clarify the operation of the algorithm of the section 3.3 with the aid of two numerical examples.

Example 1

Consider the order $m = 5$ matrix

$$C_5 = \begin{bmatrix} 2 & -j & 1 & -1 & 1 \\ j & 2 & -j & 1 & -1 \\ 1 & j & 2 & -j & 1 \\ -1 & 1 & j & 2 & -j \\ 1 & -1 & 1 & j & 2 \end{bmatrix} ,$$

and so $c_0 = 2$, $c_1 = j$, $c_2 = 1$, $c_3 = -1$, $c_4 = 1$. If we execute the algorithm of section 3.1 we get:

$$\sigma_1 = 2 ,\ x_0(z) = 1 ,\ a_{-1}(z) = 0 ,$$

$$K_1 = -\frac{1}{2}j ,\ \sigma_2 = \frac{3}{2} ,\ a_0(z) = \frac{x_0(z)}{\sigma_1} = \frac{1}{2} ,\ x_1(z) = x_0(z) + K_1 z\hat{x}_0(z) = 1 - \frac{1}{2}jz ,$$

$$K_2 = -1 ,\ \sigma_3 = 0 ,\ a_1(z) = \frac{x_1(z)}{\sigma_2} = \frac{2}{3} - \frac{1}{3}jz ,\ x_2(z) = x_1(z) + K_2 z\hat{x}_1(z) = 1 - jz - z^2 .$$

Since $\sigma_3 = 0$ we have $r = 2$, and the smallest value of l for which $\sum_{i=0}^{r} \bar{c}_{l+i} x_{r,i} \neq 0$ is $l = 1$. It may be readily shown that

$$\psi_0 = -\frac{2}{3} - \frac{1}{3}j ,\ \psi_1 = 1 ,$$

$$\alpha_0 = -1 - 2j ,\ \alpha_1 = j .$$

Therefore, $\beta_0 = \dfrac{1}{\bar{p}_0} = 1 - 2j$, since

$$p_0 = \frac{1}{5} - \frac{2}{5}j ,\ p_1 = -\frac{8}{75} + \frac{2}{25}j ,\ p_2 = -\frac{1}{5} - \frac{2}{5}j .$$

At this point in the algorithm of section 3.1, $k = 2$ and so $n = k + 2l = 4$. It is also straightforward to show that

$$a_3(z) = \beta_0^{-1} z x_2(z) = (\frac{1}{5}+\frac{2}{5}j)z + (\frac{2}{5}-\frac{1}{5}j)z^2 + (-\frac{1}{5}-\frac{2}{5}j)z^3 ,$$

$$x_4(z) = \overline{\beta}_0[za_1(z) + p(z)x_2(z)]$$

$$= 1 + (\frac{2}{5}+\frac{1}{5}j)z + (\frac{2}{15}-\frac{13}{15}j)z^2 + (-\frac{8}{15}-\frac{7}{15}j)z^3 + (-\frac{3}{5}+\frac{4}{5}j)z^4 ,$$

$$\sigma_5 = |\beta_0|^2[p_1 + \overline{p}_1 + a_1(0)] = \frac{34}{15} .$$

If we now use the Schur algorithm of section 3.2, then

$$u_{-4}^{(1)} = 1 , u_{-3}^{(1)} = -1 , u_{-2}^{(1)} = 1 , u_{-1}^{(1)} = -j , u_0^{(1)} = 2 \ (= \sigma_1) ,$$

$$u_1^{(1)} = j , u_2^{(1)} = 1 , u_3^{(1)} = -1 , u_4^{(1)} = 1 ,$$

$$K_1 = -\frac{1}{2}j , u_{-4}^{(2)} = -1 - \frac{1}{2}j , u_{-3}^{(2)} = 1 + \frac{1}{2}j , u_{-2}^{(2)} = -\frac{3}{2}j , u_{-1}^{(2)} = \frac{3}{2} \ (= \sigma_2) ,$$

$$u_0^{(2)} = 0 , u_1^{(2)} = \frac{3}{2} , u_2^{(2)} = -1 - \frac{1}{2}j , u_3^{(2)} = 1 + \frac{1}{2}j ,$$

$$K_2 = -1 , u_{-4}^{(3)} = j , u_{-3}^{(3)} = 1 - 2j , u_{-2}^{(3)} = 0 \ (= \sigma_3) ,$$

$$u_0^{(3)} = 0 , u_1^{(3)} = -1 - 2j , u_2^{(3)} = j .$$

Since $\sigma_3 = 0$, $r = 2$ as before. From (16a) $u_{-(l+2)}^{(3)} = 0$ if $l = 0$, but $u_{-3}^{(3)} \neq 0$ and so $l = 1$. From (16b,c) the values of ψ_i and α_i are correctly produced. From (23)

$$u_{i-4}^{(5)} = (1 + 2j) \ [\frac{2}{3}u_{i-2}^{(2)} + \sum_{\nu=0}^{2} p_\nu u_{i-(\nu+2)}^{(3)}] ,$$

which yields $u_{-4}^{(5)} = \frac{34}{15}$ $(= \sigma_5)$. We can now write

$$U_5 = \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & u_{-2}^{(1)} & u_{-3}^{(1)} & u_{-4}^{(1)} \\ 0 & u_{-1}^{(2)} & u_{-2}^{(2)} & u_{-3}^{(2)} & u_{-4}^{(2)} \\ 0 & 0 & u_{-2}^{(3)} & u_{-3}^{(3)} & u_{-4}^{(3)} \\ 0 & 0 & 0 & u_{-3}^{(4)} & u_{-4}^{(4)} \\ 0 & 0 & 0 & 0 & u_{-4}^{(5)} \end{bmatrix} = \begin{bmatrix} 2 & -j & 1 & -1 & 1 \\ 0 & \frac{3}{2} & -\frac{3}{2}j & 1+\frac{1}{2}j & -1-\frac{1}{2}j \\ 0 & 0 & 0 & 1-2j & j \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \frac{34}{15} \end{bmatrix} ,$$

where the $\times$ denotes an unknown value. In summary, the Schur algorithm of section 3.3 produces results in agreement with those produced by the algorithm of section 3.1, as we expect.

Example 2

Now consider the order $m = 4$ matrix

$$C_4 = \begin{bmatrix} 0 & -1 & 1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} ,$$

where $c_0 = 0$, $c_1 = -1$, $c_2 = 1$, $c_3 = 0$. Let us again begin by executing the algorithm of section 3.1. Clearly, $\sigma_1 = c_0 = 0$ and so $r = 0$. The smallest value of l such that $\sum_{i=0}^{0} \overline{c}_{l+i} x_{0,i} \neq 0$ is $l = 1$. Thus, $\psi_i = -\delta_{i,1}$, and so $\psi_0 = 0$, $\psi_1 = -1$. As well, $\alpha_i = c_{i+1}$, giving $\alpha_0 = -1$, $\alpha_1 = 1$. Therefore, $p_0 = -1$, $p_1 = -1$, $p_2 = 1$. Hence, $\beta_0 = -1$, and

$$a_1(z) = \beta_0^{-1} z x_0(z) = -z \quad ,$$

$$x_2(z) = \overline{\beta}_0[z a_{-1}(z) + p(z) x_0(z)] = 1 + z - z^2 \quad ,$$

$$\sigma_3 = | \ \beta_0 \ |^2 [p_1 + \overline{p}_1 + a_{-1}(0)] = -2 \quad .$$

We have $n = k + 2l = 0 + 2 \cdot 1 = 2$. At this point k becomes n, so $k = 2$. Since $\sigma_3 \neq 0$,

$$K_3 = -\frac{1}{\sigma_3} \sum_{i=0}^{2} c_{3-i} x_{2,i} = 1 \quad ,$$

$$\sigma_4 = \sigma_3(1 - | \ K_3 \ |^2) = 0 \quad ,$$

$$a_2(z) = \frac{x_2(z)}{\sigma_3} = -\frac{1}{2} - \frac{1}{2} z + \frac{1}{2} z^2 \quad ,$$

$$x_3(z) = x_2(z) + K_3 z \hat{x}_2(z) = 1 + z^3 \quad .$$

Now let us use the Schur algorithm for Hermitian Toeplitz matrices of any rank profile. We have

$$u_{-3}^{(1)} = 0 \ , \ u_{-2}^{(1)} = 1 \ , \ u_{-1}^{(1)} = -1 \ , \ u_0^{(1)} = 0 \ (= \sigma_1) \quad ,$$

$$u_1^{(1)} = -1 \ , \ u_2^{(1)} = 1 \ , \ u_3^{(1)} = 0 \ ,$$

and so $\sigma_1 = 0$ implying that $r = 0$. Quantities ψ_i , α_i , l , n , $p(z)$ and β_0 are therefore going to be computed correctly and will take on the values of the previous paragraph. We don't know $u_i^{(2)}$, but via (23),

$$u_{i-2}^{(3)} = -\sum_{\nu=0}^{2} p_\nu u_{i-\nu}^{(1)} = u_i^{(1)} + u_{i-1}^{(1)} - u_{i-2}^{(1)} \ ,$$

and this gives us

$$u_{-2}^{(3)} = -2 \ , \ u_{-3}^{(3)} = 0 \ ,$$

$$u_0^{(3)} = 0 \ , \ u_1^{(3)} = 2 \ .$$

Now $k = 3$, and we may use the Schur algorithm of section 3.2. Hence,

$$K_3 = -\frac{u_1^{(3)}}{u_{-2}^{(3)}} = 1 \ ,$$

$$u_{-(i+3)}^{(4)} = u_{-(i+2)}^{(3)} + K_3 \bar{u}_{i+1}^{(3)} \ ,$$

$$u_i^{(4)} = K_3 \bar{u}_{-(i+2)}^{(3)} + u_{i+1}^{(3)} \ .$$

For $i = 0$, the latter two equations give

$$u_{-3}^{(4)} \ (= \sigma_4) = 0 \ , \ u_0^{(4)} = 0 \ .$$

Finally,

$$U_4 = \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & u_{-2}^{(1)} & u_{-3}^{(1)} \\ 0 & u_{-1}^{(2)} & u_{-2}^{(2)} & u_{-3}^{(2)} \\ 0 & 0 & u_{-2}^{(3)} & u_{-3}^{(3)} \\ 0 & 0 & 0 & u_{-3}^{(4)} \end{bmatrix} = \begin{bmatrix} 0 & -1 & 1 & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \ .$$

REFERENCES

[1] C. J. Zarowski, H. C. Card, "Relations Between the Schur and Split Schur Algorithms," submitted to the 14th Biennial Symposium on Communications, Queen's University, Kingston, Ontario, Canada, 1988.

[2] J. Le Roux, C. Gueguen, "A Fixed Point Computation of Partial Correlation Coefficients," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-25, June 1977, pp. 257-259.

[3] S.-Y. Kung, Y. H. Hu, "A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-31, Feb. 1983, pp. 66-75.

[4] P. Delsarte, Y. V. Genin, "On the Splitting of Classical Algorithms in Linear Prediction Theory," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, May 1987, pp. 645-653.

[5] C. J. Zarowski, H. C. Card, "A Schur Algorithm for Hermitian Toeplitz Matrices of Any Rank Profile," submitted to the IEEE Trans. on Acoust., Speech, and Signal Proc.

[6] P. Delsarte, Y. V. Genin, Y. Kamp, "A Generalization of the Levinson Algorithm for Hermitian Toeplitz Matrices of Any Rank Profile," IEEE Trans., on Acoust., Speech, and Signal Proc., vol. ASSP-33, Aug. 1985, pp. 964-971.

[7] J. D. Markel, A. H. Gray, Jr., "On Autocorrelation Equations as Applied to Speech Analysis," IEEE Trans. on Aud. and Electroac., vol. AU-21, April 1973, pp. 69-79.

[8] I. S. Iohvidov, *Hankel and Toeplitz Matrices and Forms*. Boston, Massachusetts: Birkhaüser, 1982.

Chapter IV

PARALLEL-PIPELINED PROCESSOR ARCHITECTURES FOR THE SCHUR AND SPLIT SCHUR ALGORITHMS

In this chapter we present parallel-pipelined processor implementations of the Schur and split Schur algorithms described in the previous two chapters. As we have seen, on a sequential processor, these algorithms have a time complexity of $O(n^2)$ (nth order matrix). The Schur and split Schur algorithms can be implemented, in parallel form, on a linear array of $O(n)$ processors. In this situation these algorithms will run in $O(n)$ time. As will be seen, in many cases it is possible to arrange the computations so that processors in the array need only communicate with their nearest neighbors. This is done through the use of pipelining. Thus, the need for global communications can often be eliminated. The parallel-pipelined processor implementation of the Schur algorithm is due mainly to Kung and Hu [1], but partly to Brent and Luk [2]. The parallel-pipelined processor implementation of the split Schur algorithms is due to Zarowski and Card [3], as is the implementation of the Schur algorithm for Hermitian Toeplitz matrices of any rank profile.

1. Primary Issues in the VLSI Implementation of Parallel-Pipelined Processing Systems

We shall begin by discussing the formulation of parallel algorithms to solve problems. In this regard we shall first note that algorithms which are efficient on sequential machines are not necessarily efficient on parallel machines. For example, we have seen in Chapter II that the Levinson-Durbin and Schur algorithms both have a time complexity of $O(n^2)$ on a sequential processor. However, it is noted in Kung and Hu [1] that the Levinson-Durbin algorithm, when implemented on a linear array of processors, will have a time complexity of $O(n \log n)$. On the other hand, the Schur algorithm will have a time complexity of $O(n)$. This is due to fundamental differences in the structure of the two algorithms. In particular, the Levinson-Durbin class of algorithms all possess inner product operations, i.e., operations of the form

$$\sum_{i=1}^{n} x_i y_i \ .$$

On a linear array of n processors this operation can be performed in log n time at best. Clearly, the products $x_i y_i$ can be computed in $O(1)$ time. The bottleneck is due to the need to sum the products, and this takes logarithmic time on a linear array of n processors. The Schur-type algorithms possess no inner product operations and so are completely unconstrained by this operation. As a result, in formulating algorithms to solve arbitrary problems in parallel, very close attention must be paid to algorithm structure. It is reasonable to state that the theory of algorithms is more important in a parallel-processing context than it is in a sequential processing context.

In a sequential processing environment, computational complexity is traditionally measured in terms of the number of arithmetic operations required by the algorithm. However, we have seen that some algorithms are more amenable to parallel implementation than others, even when they all possess the same time complexity (operations count) on a sequential machine. Thus, operations count is not a valid criterion for judging which algorithm is most amenable to parallel implementation. Kung and Hu [1] suggest that throughput rate replace operations count as the performance criterion of a parallel solution to a problem. We shall adopt this criterion here. We may therefore state that in formulating a parallel algorithm, **structure the algorithm to achieve the maximum parallelism (concurrency) and, therefore, the maximum throughput rate.** According to this criterion, the Schur-type algorithms are superior to the Levinson-Durbin-type algorithms at solving Toeplitz problems (defined in Chapter II, section 1).

In Chapter II other algorithms, besides the Levinson-Durbin and Schur algorithms, were described. Specifically, the Trench and Bareiss algorithms were also derived. The presence of inner product operations in the Trench algorithm leads us to conclude that it is a poor candidate for parallel processor implementation. However, the Bareiss algorithm is shown by Brent and Luk [2] to be a good candidate for such an implementation. In fact, they show that in some respects the Bareiss algorithm is superior to the Schur algorithm of [1] in this context. For example, the Schur algorithm can *LDU* factorize and compute the reflection coefficients of a nonsymmetric Toeplitz matrix. But the parallel processor implementation of it requires that the reflection coefficients of this form of the Schur algorithm be globally broadcasted to all other processors in the array. This is undesirable, especially in the context of a VLSI (very large scale integration) implementation. The parallel implementation of the Bareiss algorithm proposed by Brent and Luk [2] does not have this problem, i.e., communications between processors can be strictly local. Note however, that the Bareiss algorithm is no better than the Schur algorithm when the Toeplitz matrix is either symmetric, or Hermitian. It is only these latter two cases that are of interest to us in this chapter. It appears that these are the most useful cases in practical applications.

It is well known that contemporary VLSI technology is limited by communications constraints (see Mead and Conway [4], or Kung, Whitehouse, and Kailath [5]). A parallel algorithm can be readily implemented with VLSI technology only if communications is localized (i.e., between neighboring processors only). Architectures such as the systolic array ([4],[5]), and the wavefront array processor ([5],[6]) employ localized interprocessor communications, and so are viable candidates for VLSI implementation. Thus, if possible, we must **arrange a computation so that communications constraints are satisfied, and processing throughput rate is simultaneously maximized.** The parallel-pipelined processor implementations of the Schur algorithm in [1] and the split Schur algorithms in [3] satisfy this requirement quite well. The communications constraint is satisfied by pipelining. Notice that in general the requirement of satisfying communications constraints conflicts with the requirement of maximizing throughput. Thus, a compromise may be necessary. If such a compromise proves unacceptable, then it will be necessary to find a better algorithm. Unfortunately, it may well be that no such algorithm exists and so compromises may be unavoidable afterall.

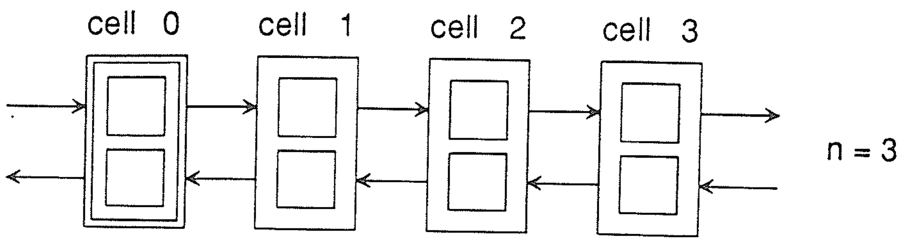
Note that the criterion of the preceding paragraph neglects the problem of data input and output to the VLSI chip. This is a significant consideration in practice since all VLSI chips have a finite number of I/O pins. It turns out however, that the problems of interest to us in this thesis have pin I/O requirements which are fixed and independent of n, the size of the problem. This fact shall become obvious to the reader in the remainder of this chapter.

2. Parallel-Pipelined Architectures for the Schur Algorithm

In this section we describe the parallel-pipelined processor implementation of the Schur algorithm due to Kung and Hu [1], along with a certain improvement suggested by Brent and Luk [2]. Although the material of this section is found in [1,2], the presentation of this section is more complete and detailed, and we believe that the present exposition is easier to understand than that in [1,2]. As in Kung and Hu [1], we shall assume that T_n is an $(n+1) \times (n+1)$ symmetric Toeplitz matrix. The extension to the Hermitian Toeplitz case is straightforward (simply take conjugates at the appropriate places). Furthermore, we shall consider the special case of $n = 3$, as generalization to arbitrary n is straightforward, and special cases are easier to visualize than the general case.

Figure 1(a) shows the linear array of processors suggested by Kung and Hu [1] for the parallel implementation of their form of the Schur algorithm. The processors are represented as larger boxes containing two smaller boxes. The smaller boxes denote storage locations for the numbers $u_{-(i+k)+1}^{(k)}$ and $u_{i+1}^{(k)}$. Number $u_{-(i+k)+1}^{(k)}$ appears in cell i, while $u_{i+1}^{(k)}$ appears in cell $i+1$. Cell 0 is represented by a double box. This is to signify the fact that this is the only cell that must be capable of performing division. Recall that division is used to produce the reflection coefficients K_k, but division will also be used in certain back-substitution operations to be described later in this section.

(a)



(b)

time	cell			
	0	1	2	3
0	$u_0^{(1)}$ X	$u_{-1}^{(1)}$ $u_1^{(1)}$	$u_{-2}^{(1)}$ $u_2^{(1)}$	X $u_3^{(1)}$
1	$u_{-1}^{(2)}$ $u_0^{(2)}$	$u_{-2}^{(2)}$ $u_1^{(2)}$	$u_{-3}^{(2)}$ $u_2^{(2)}$	X X
2	$u_{-2}^{(3)}$ $u_0^{(3)}$	$u_{-3}^{(3)}$ $u_1^{(3)}$	X X	X X
3	$u_{-3}^{(4)}$ $u_0^{(4)}$	X X	X X	X X

X = don't care

(c)

time	cell			
	0	1	2	3
0	$u_0^{(1)}$ X	$u_{-1}^{(1)}$ $u_1^{(1)}$	$u_{-2}^{(1)}$ $u_2^{(1)}$	X $u_3^{(1)}$
1	$\boxed{u_{-1}^{(2)}\ u_0^{(2)}}$	$u_{-1}^{(1)}$ $u_1^{(1)}$	$u_{-2}^{(1)}$ $u_2^{(1)}$	X $u_3^{(1)}$
2	$u_{-1}^{(2)}$ $u_0^{(2)}$	$\boxed{u_{-2}^{(2)}\ u_1^{(2)}}$	$u_{-2}^{(1)}$ $u_2^{(1)}$	X $u_3^{(1)}$
3	$\boxed{u_{-2}^{(3)}\ u_0^{(3)}}$	$u_{-2}^{(2)}$ $u_1^{(2)}$	$\boxed{u_{-3}^{(2)}\ u_2^{(2)}}$	X $u_3^{(1)}$
4	$u_{-2}^{(3)}$ $u_0^{(3)}$	$\boxed{u_{-3}^{(3)}\ u_1^{(3)}}$	$u_{-3}^{(2)}$ $u_2^{(2)}$	X $u_3^{(1)}$
5	$\boxed{u_{-3}^{(4)}\ u_0^{(4)}}$	$u_{-3}^{(3)}$ $u_1^{(3)}$	$u_{-3}^{(2)}$ $u_2^{(2)}$	X $u_3^{(1)}$

Figure 1: (a) Linear array of processors used to execute the Schur algorithm of Kung and Hu; (b) Computation of the Schur variables without pipelining; (c) Computation of the Schur variables with pipelining.

It is important to note that we can interpret indices k and i in $u_i^{(k)}$ as time and space (or cell) indices, respectively. With this interpretation in mind, Figure 1(b) shows the flow of the Schur variables through the machine of Figure 1(a). At time $t = 0$ the initial state of the machine is shown. It is clear that the machine can be initialized in $O(n)$ time. The reader must visualize the computation, and broadcasting to all cells (processors), of the reflection coefficient K_{k+1} between $t = k$ and $t = k+1$ $(k \geq 0)$. Since

$$K_1 = \frac{u_1^{(1)}}{u_0^{(1)}} , \quad K_2 = \frac{u_1^{(2)}}{u_{-1}^{(2)}} , \quad K_3 = \frac{u_1^{(3)}}{u_{-2}^{(3)}} ,$$

K_k is computed in cell 0, hence the need for this cell to be capable of performing division. Since to compute $u_i^{(k+1)}$ requires K_k and $u_i^{(k)}$, it is necessary that K_k be available to all processors at any given t in Figure 1(b), hence the apparent need to globally broadcast K_k. It is important to note that the Schur variables in a given cell, say cell i, are functionally dependent upon the Schur variables of cells i and $i+1$ exclusively (from one t to the next) via

$$\begin{bmatrix} u_{-(i+k)}^{(k+1)} \\ u_i^{(k+1)} \end{bmatrix} = \begin{bmatrix} 1 & K_k \\ K_k & 1 \end{bmatrix} \begin{bmatrix} u_{-(i+k)+1}^{(k)} \\ u_{i+1}^{(k)} \end{bmatrix} , \qquad (1)$$

which is taken from the innermost For-do loop of the Schur algorithm (see Chapter II, section 2.3). Thus, aside from the global broadcasting of reflection coefficients, interprocessor communications is localized. Clearly, a Toeplitz matrix will be factorized in $O(n)$ time on the machine of Figure 1(a).

The global broadcasting of reflection coefficients can be eliminated entirely by the use of pipelining, as is demonstrated in [1]. We thus satisfy our communications constraints (i.e., local communications requirements) without reducing processing throughput rate. This is illustrated in Figure 1(c). Here K_1 is computed between $t = 0$ and $t = 1$, K_2 between $t = 2$ and $t = 3$,and K_3 between $t = 4$ and $t = 5$. During $t = 1$, K_1 is used to obtain $u_{-1}^{(2)}$ and $u_0^{(2)}$, and is simultaneously broadcasted to cell 1. During $t = 2$, K_1 is used to compute $u_{-2}^{(2)}$ and $u_1^{(2)}$ (K_1 is in cell 1 now), and K_1 is simultaneously sent to cell 2. During $t = 3$, K_2 is used to compute $u_{-2}^{(3)}$ and $u_0^{(3)}$, while K_1 is used to compute $u_{-3}^{(2)}$ and $u_2^{(2)}$ since K_2 is in cell 0, and K_1 is in cell 2. Simultaneously, K_1 is sent to cell 3, and K_2 is sent to cell 1. Similar operations are applied to obtain the remaining Schur variables. Notice that in Figure 1(b), the Schur variables $u_i^{(2)}$ are all computed during $t = 1$, the Schur variables $u_i^{(3)}$ are computed during $t = 2$, and so on. In Figure 1(c) on the other hand, the Schur variables $u_i^{(2)}$ are produced staggered in time, as are $u_i^{(3)}$ and $u_i^{(4)}$. Despite this staggering of the outputs, the matrix T_n will be LDU factorized in $O(n)$ time. In addition, about half of the cells in Figure 1(c) are inactive for a given t. As we shall soon see, it is possible to use these inert processors to perform useful work.

Often we wish to solve

$$T_n x_n = y_n \tag{2}$$

for a given T_n and y_n. Since the Schur algorithm LDU factorizes T_n, we can compute x_n via two successive back-substitutions. From Chapter II

$$T_n = U_n^T D_n^{-1} U_n \quad , \tag{3}$$

where U_n is in (70) (Chapter II), and D_n is in (74) (Chapter II). Thus, (2) may be written as

$$U_n^T D_n^{-1} U_n x_n = y_n \quad , \tag{4}$$

and if we define $b_n = D_n^{-1} U_n x_n$, then

$$U_n^T b_n = y_n \quad , \tag{5a}$$

$$D_n^{-1} U_n x_n = b_n \quad . \tag{5b}$$

If we also define $\tilde{b}_n = D_n b_n$ then (5b) can be replaced by

$$U_n x_n = \tilde{b}_n \quad . \tag{6}$$

Equations (5a,b) show that x_n can be computed in two successive back-substitution operations, given U_n (D_n is the main diagonal of U_n), and y_n.

We will begin by considering the *first back-substitution* represented by (5a). This one is of the form

$$L_n x_n = y_n \quad , \tag{7}$$

where $L_n = [l_{ij}]_{(n+1)\times(n+1)}$ ($i =$ row index, $j =$ column index) is lower triangular, $x_n = [x_{n,0} \cdots x_{n,n}]^T$, $y_n = [y_{n,0} \cdots y_{n,n}]^T$. The following algorithm readily solves the first back-substitution problem in (7) (hence in (5a)):

First Back-Substitution Algorithm:

```
        $y_n^{(0)} := y_n$ ;
        For $k$ := 0 to $n$ do begin
        $x_{n,k} := \frac{y_{n,k}^{(k)}}{l_{k,k}}$;
        For $i := k+1$ to $n$ do begin
            $y_{n,i}^{(k+1)} := y_{n,i}^{(k)} - x_{n,k} l_{i,k}$ ;
        end;
    end;
```

On a sequential processor this algorithm has a time complexity of $O(n^2)$. There is a linear systolic array parallel processor solution to this problem in Mead and Conway [4] (see pp. 285-288), with a time complexity of $O(n)$, and Figure 2 illustrates its development for $n = 3$. Figure 2(a) assumes that it is possible to allow global broadcasting of the numbers $x_{n,k}$ to all cells. In this case

$$x_{3,0} = \frac{y_{3,0}^{(0)}}{l_{0,0}} , x_{3,1} = \frac{y_{3,1}^{(1)}}{l_{1,1}} , x_{3,2} = \frac{y_{3,2}^{(2)}}{l_{2,2}} , x_{3,3} = \frac{y_{3,3}^{(3)}}{l_{3,3}} ,$$

and so cell 0 is the only cell that must be capable of performing division. In Figure 2(a), $x_{n,k}$ is computed and broadcasted to all other cells between $t = k$ and $t = k+1$. Elements in cell i are updated using the contents of cell $i+1$ at any given t. Thus, aside from the global broadcasting of $x_{n,k}$, communications is strictly localized. As in the case of the Schur algorithm, pipelining may be used to eliminate the need for globally broadcasting $x_{n,k}$. This is illustrated in Figure 2(b), and the result is essentially the linear back-substitution array in [4].

If $L_n = U_n^T$, then

$$l_{ij} = u_{-i}^{(j+1)} , \tag{8}$$

and the First Back-Substitution Algorithm may be rewritten using (8), and by replacing x_n with b_n as per equation (5a). Thus,

$$\tilde{b}_{n,k} = y_{n,k}^{(k)} , \quad b_{n,k} = \frac{y_{n,k}^{(k)}}{u_{-k}^{(k+1)}} , \tag{9}$$

where we have used the First Back-Substitution Algorithm with the suggested variable name changes. Figure 2(b) then becomes Figure 3, which shows that the first back-

(a)

time	cell			
	0	1	2	3
0	l_{00} $y_{30}^{(0)}$	l_{10} $y_{31}^{(0)}$	l_{20} $y_{32}^{(0)}$	l_{30} $y_{33}^{(0)}$
1	l_{11} $y_{31}^{(1)}$	l_{21} $y_{32}^{(1)}$	l_{31} $y_{33}^{(1)}$	× ×
2	l_{22} $y_{32}^{(2)}$	l_{32} $y_{33}^{(2)}$	× ×	× ×
3	l_{33} $y_{33}^{(3)}$	× ×	× ×	× ×

$$L_3 = \begin{bmatrix} l_{00} & 0 & 0 & 0 \\ l_{10} & l_{11} & 0 & 0 \\ l_{20} & l_{21} & l_{22} & 0 \\ l_{30} & l_{31} & l_{32} & l_{33} \end{bmatrix}$$

(b)

time	cell			
	0	1	2	3
0	l_{00} $y_{30}^{(0)}$			
1		l_{10} $y_{31}^{(0)}$		
2	l_{11} $y_{31}^{(1)}$		l_{20} $y_{32}^{(0)}$	
3		l_{21} $y_{32}^{(1)}$		l_{30} $y_{33}^{(0)}$
4	l_{22} $y_{32}^{(2)}$		l_{31} $y_{33}^{(1)}$	
5		l_{32} $y_{33}^{(2)}$		
6	l_{33} $y_{33}^{(3)}$			

Figure 2: (a) Back-substitution using global communications; (b) Back-substitution with pipelining to eliminate global communications.

substitution of (5a) can be carried out concurrently with the computation of U_n. Alternatively, variables $y_{n,i}^{(k)}$ can be computed when a cell is otherwise inactive.

Now let us consider the *second and final back-substitution* operation represented by (6), which came from (5b). Equation (6) has the form

$$U_n x_n = y_n \quad , \tag{10}$$

where $U_n = [u_{ij}]_{(n+1)\times(n+1)}$ (i = row index, j = column index) is upper triangular, with x_n and y_n as in (7). The following algorithm readily solves the second back-substitution problem of (10) (hence of (6)):

Second Back-Substitution Algorithm:

```
$y_n^{(0)} := y_n$ ;
For $k$ := 0 to $n$ do begin
    $x_{n,n-k} := \dfrac{y_{n,n-k}^{(k)}}{u_{n-k,n-k}}$ ;
```

time	cell			
	0	1	2	3
0	$u_0^{(1)}$ $y_{30}^{(0)}$	$u_{-1}^{(1)}$ $y_{31}^{(0)}$	$u_{-2}^{(1)}$ $y_{32}^{(0)}$	$u_{-3}^{(1)}$ $y_{33}^{(0)}$
1	$u_{-1}^{(2)}$ $y_{31}^{(1)}$	$u_{-1}^{(1)}$ $y_{31}^{(0)}$	$u_{-2}^{(1)}$ $y_{32}^{(0)}$	$u_{-3}^{(1)}$ $y_{33}^{(0)}$
2	$u_{-1}^{(2)}$ $y_{31}^{(1)}$	$u_{-2}^{(2)}$ $y_{32}^{(1)}$	$u_{-2}^{(1)}$ $y_{32}^{(0)}$	$u_{-3}^{(1)}$ $y_{33}^{(0)}$
3	$u_{-2}^{(3)}$ $y_{32}^{(2)}$	$u_{-2}^{(2)}$ $y_{32}^{(1)}$	$u_{-3}^{(2)}$ $y_{33}^{(1)}$	$u_{-3}^{(1)}$ $y_{33}^{(0)}$
4	$u_{-2}^{(3)}$ $y_{32}^{(2)}$	$u_{-3}^{(3)}$ $y_{33}^{(2)}$	$u_{-3}^{(2)}$ $y_{33}^{(1)}$	$u_{-3}^{(1)}$ $y_{33}^{(0)}$
5	$u_{-3}^{(4)}$ $y_{33}^{(3)}$	$u_{-3}^{(3)}$ $y_{33}^{(2)}$	$u_{-3}^{(2)}$ $y_{33}^{(1)}$	$u_{-3}^{(1)}$ $y_{33}^{(0)}$

$\rightarrow \tilde{b}_{30}, b_{30}$

$\rightarrow \tilde{b}_{31}, b_{31}$

$\rightarrow \tilde{b}_{32}, b_{32}$

$\rightarrow \tilde{b}_{33}, b_{33}$

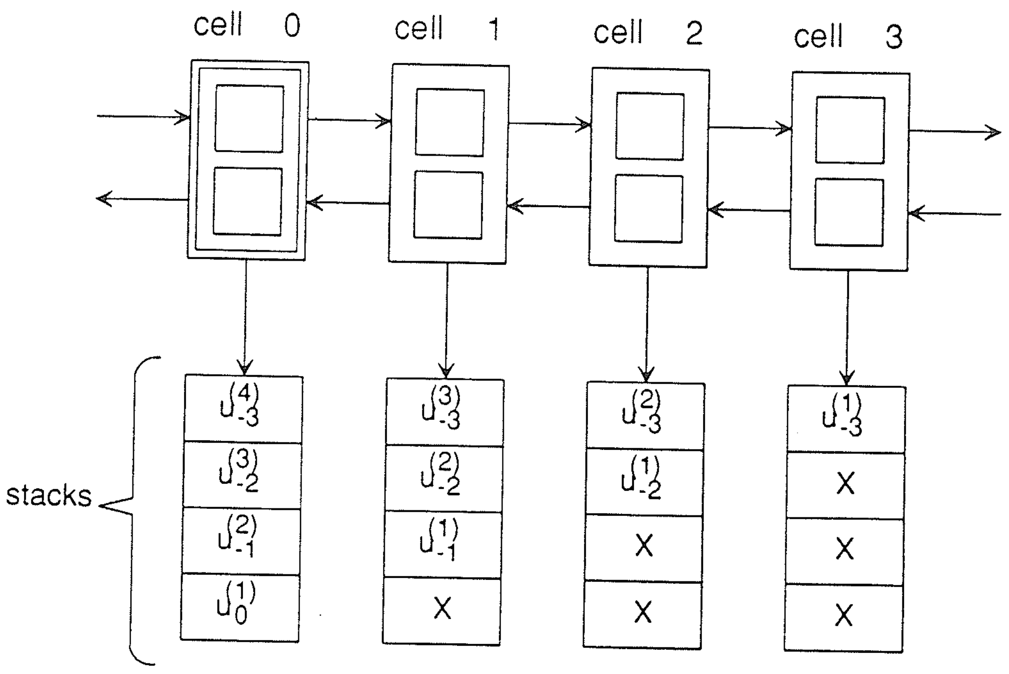
Figure 3: The first back-substitution may be carried out concurrently with the generation of U_n.

```
        For $i$ := 0 to $n-k-1$ do begin
            $y_{n,i}^{(k+1)} := y_{n,i}^{(k)} - x_{n,n-k}u_{i,n-k}$ ;
        end;
    end;
```

As with the First Back-Substitution Algorithm, a sequential processor implementation will have a time complexity of $O(n^2)$, and the linear systolic array solution with have a time complexity of $O(n)$.

There are two ways of incorporating the second back-substitution represented by (6) into the array of Figure 1(a). One of these methods is suggested by Kung and Hu [1], and the other is suggested by Brent and Luk [2]. We shall first describe the Kung-Hu solution.

Kung and Hu [1] suggest the stacking of the elements of U_n (from $T_n = U_n^T D_n^{-1} U_n$) as indicated in Figure 4. The element $\tilde{b}_{n,k}$ can be stored in cell $n-k$ during the first back-substitution phase. As a result, all of the necessary parameters will be in place for the final back-substitution phase. The second back-substitution is shown in Figure 5, assuming the global broadcasting of $x_{n,k}$, and where $u_{ij} = u_{-j}^{(i+1)}$ in the Second Back-Substitution Algorithm. Comparing Figure 5 with Figure 4 reveals that the elements of U_n, when stacked as indicated in Figure 4, will indeed be available at the proper place and time for the second back-substitution.



$$U_3 x_3 = \bar{b}_3 \quad \rightarrow \quad \begin{bmatrix} u_0^{(1)} & u_{-1}^{(1)} & u_{-2}^{(1)} & u_{-3}^{(1)} \\ 0 & u_{-1}^{(2)} & u_{-2}^{(2)} & u_{-3}^{(2)} \\ 0 & 0 & u_{-2}^{(3)} & u_{-3}^{(3)} \\ 0 & 0 & 0 & u_{-3}^{(4)} \end{bmatrix} \begin{bmatrix} x_{30} \\ x_{31} \\ x_{32} \\ x_{33} \end{bmatrix} = \begin{bmatrix} \bar{b}_{30} \\ \bar{b}_{31} \\ \bar{b}_{32} \\ \bar{b}_{33} \end{bmatrix}$$

Figure 4: Storage of U_n on stacks to facilitate the second and final back-substitution. Illustrated is the state of the stacks after the first back-substitution, and before the second back-substitution.

The Kung-Hu solution has the advantage of simplicity, but it has the obvious disadvantage of requiring $O(n^2)$ storage. If n is large then this could be a serious drawback. It has been shown by Brent and Luk [2] that the $O(n^2)$ storage problem can be avoided by regenerating the elements of U_n as they are needed during the second back-substitution. This can be accomplished with only $O(n)$ storage and some extra computations. One way to do this is to store

time	cell			
	0	1	2	3
0	$u_{-3}^{(4)}$ $y_{33}^{(0)}$	$u_{-3}^{(3)}$ $y_{32}^{(0)}$	$u_{-3}^{(2)}$ $y_{31}^{(0)}$	$u_{-3}^{(1)}$ $y_{30}^{(0)}$
1	$u_{-2}^{(3)}$ $y_{32}^{(1)}$	$u_{-2}^{(2)}$ $y_{31}^{(1)}$	$u_{-2}^{(1)}$ $y_{30}^{(1)}$	× ×
2	$u_{-1}^{(2)}$ $y_{31}^{(2)}$	$u_{-1}^{(1)}$ $y_{30}^{(2)}$	× ×	× ×
3	$u_{0}^{(1)}$ $y_{30}^{(3)}$	× ×	× ×	× ×

$$\rightarrow x_{33} = \frac{y_{33}^{(0)}}{u_{-3}^{(4)}}$$

$$\rightarrow x_{32} = \frac{y_{32}^{(1)}}{u_{-2}^{(3)}}$$

$$\rightarrow x_{31} = \frac{y_{31}^{(2)}}{u_{-1}^{(2)}}$$

$$\rightarrow x_{30} = \frac{y_{30}^{(3)}}{u_{0}^{(1)}}$$

Figure 5: Second back-substitution assuming global broadcasting of $x_{n,k}$ (kth element of the solution vector x_n).

$$u_0^{(i)}\ ,\ u_{-n}^{(i)}\ (1 \le i \le n+1)\ ,\ K_i\ \ (1 \le i \le n)$$

in cell $n-i+1$ during the LDU factorization (Schur algorithm) phase. It is then possible to use, for $1 \le k \le n$, $0 \le i \le n-k$,

$$\begin{bmatrix} u_{-(i+k)+1}^{(k)} \\ u_{i+1}^{(k)} \end{bmatrix} = \frac{1}{1-K_k^2} \begin{bmatrix} 1 & -K_k \\ -K_k & 1 \end{bmatrix} \begin{bmatrix} u_{-(i+k)}^{(k+1)} \\ u_i^{(k+1)} \end{bmatrix} , \tag{11}$$

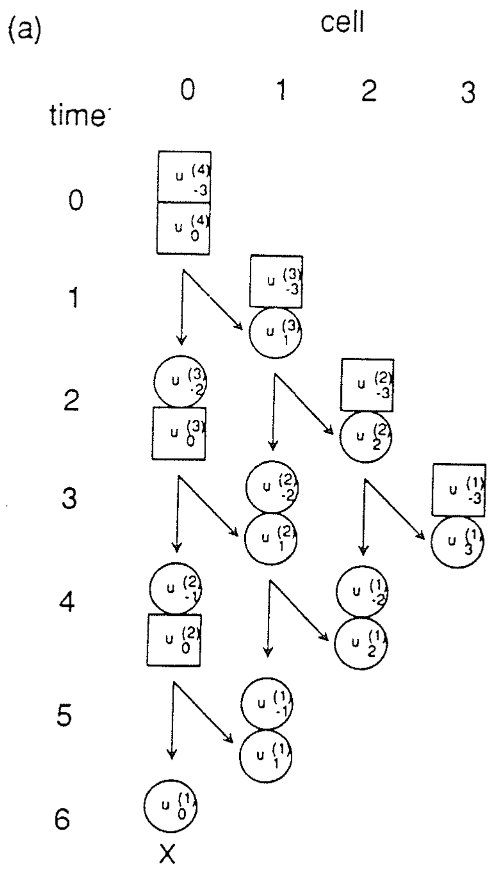
which is derived from (1), to regenerate the Schur variables $u_{-i}^{(k)}, (i \ge 0)$ as they are needed during the final back-substitution phase.

Figure 6(a) shows the sequence in which the Schur variables are computed using (11). Note that the prestored variables are contained in boxes and the computable variables are contained in circles. Figure 6(b) shows the desired locations, for a given t, of the prestored Schur variables $u_0^{(i)}$, and the reflection coefficients K_i, during the course of the regeneration operation illustrated in Figure 6(a).

3. Parallel-Pipelined Architectures for the Split Schur Algorithms

Now we present the parallel-pipelined processor architecture which may be used to implement either of the split Schur algorithms in $O(n)$ time. We will only consider the case of the symmetric split Schur algorithm; the antisymmetric case follows with

(a)



(b)

time	cell 0	1	2	3
0	X $u_0^{(4)}$	K_3 $u_0^{(3)}$	K_2 $u_0^{(2)}$	K_1 $u_0^{(1)}$
1	$u_0^{(4)}$	K_3	$u_0^{(2)}$	
2	K_3	$u_0^{(3)}$	K_2	$u_0^{(1)}$
3	$u_0^{(3)}$	K_2	$u_0^{(2)}$	K_1
4	K_2	$u_0^{(2)}$	K_1	
5	$u_0^{(2)}$	K_1		
6	K_1			

□ = stored values (during LDU factorization)

○ = computable from stored and previously computed values

X = don't care

Figure 6: (a) Elements of U_n can be regenerated by storing a certain set (see text) of $O(n)$ values during the execution of the Schur algorithm. This figure illustrates the relevant dependencies; (b) Position of K_i and $u_0^{(i)}$ at any time during the regeneration of U_n.

trivial modifications. In this section we will use the symmetric split Schur algorithm of Delsarte and Genin [7], namely:

```
$v_{0,0} := r_0$; $K_0 := 0$;
For $j := 1$ to $n$ do begin
    $v_{0,j} := 2r_j$;
    end;
For $j := 0$ to $n-1$ do begin
    $v_{1,j} := r_j + r_{j+1}$;
    end;
For $k := 1$ to $n$ do begin
```

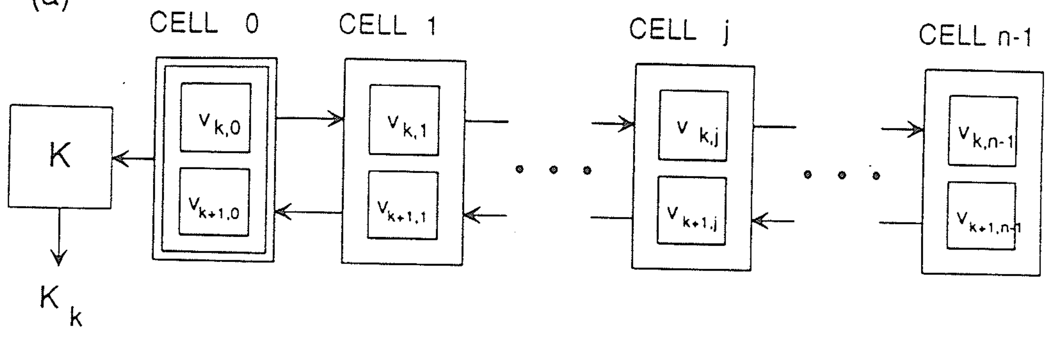
```
        $\alpha_k := v_{k,0}/v_{k-1,0}$; $K_k := 1-\alpha_k/(1+K_{k-1})$;
        For $j$ := 0 to $n-k-1$ do begin
              $v_{k+1,j} := v_{k,j} + v_{k,j+1} - \alpha_k v_{k-1,j+1}$
              end;
        end;
```

Recall that the only difference between the above algorithm and that of Chapter II is in the duration of the innermost For-do loop. The symmetric split Schur algorithm of Chapter II has an innermost For-do loop that terminates at $n-k$.

Figure 7(a) depicts the parallel-pipelined processor which is used to implement the symmetric or antisymmetric split Schur algorithms. The machine consists of n processors in a linear array (labeled *CELL* 0 , . . . , *CELL* j , . . . , *CELL* $n-1$) which are used to compute α_k and $v_{k,j}$, plus one additional processor (box labeled K) that computes K_k. All processors must be capable of performing addition/subtraction, and other operations to be specified presently. The processor *CELL* 0 is in bold lines. We distinguish *CELL* 0 in this way in order to symbolize the fact that *CELL* 0 must be capable of performing division so that it may compute α_k. Processor K must also perform division, but *CELL* 1 to *CELL* $n-1$ need only perform multiplication. *CELL* j $(0 \le j \le n-1)$ contains two boxes, one of which symbolizes a storage location for $v_{k,j}$ and the other symbolizes a storage location for $v_{k+1,j}$. A storage location must also be provided for α_k in each cell, but this is not shown.

For the special case of $n = 4$, Figure 7(b) depicts the flow of data through the machine of Figure 7(a). Note that j is interpreted as a space (cell) index, and k is interpreted as a time index. The basic unit of time is the duration of a multiply-add or a divide-add step. For $k = 0$ in Figure 7(b), the initial state of the processor array is depicted. In between the time instants for each value of k depicted two things must happen. First of all, α_k must be computed by *CELL* 0 and broadcast to all other cells (including K). This is assumed to take one time unit. Next, each processor computes $v_{k+1,j}$ for $0 \le j \le n-k-1$ according to the innermost For-do loop of the split Schur algorithm. This will take one time unit. Hence, each value of k shown in the figure

(a)



(b)

time (k)	cell (j)			
	0	1	2	3
0	$v_{0,0}$ $v_{1,0}$	$v_{0,1}$ $v_{1,1}$	$v_{0,2}$ $v_{1,2}$	$v_{0,3}$ $v_{1,3}$
1	$v_{1,0}$ $v_{2,0}$	$v_{1,1}$ $v_{2,1}$	$v_{1,2}$ $v_{2,2}$	$v_{1,3}$ ×
2	$v_{2,0}$ $v_{3,0}$	$v_{2,1}$ $v_{3,1}$	$v_{2,2}$ ×	× ×
3	$v_{3,0}$ $v_{4,0}$	$v_{3,1}$ ×	× ×	× ×

Figure 7: (a) Parallel-pipelined processor array to implement the split Schur algorithms; (b) Flow of data through the machine assuming global broadcasting of the split reflection coefficients (not shown).

represents two basic time units. During the first time unit quantity $1 + K_{k-1}$ may be computed by processor K. During the second time unit K may compute $K_k = 1 - \alpha_k/(1 + K_{k-1})$.

Aside from the apparent need to globally broadcast split reflection coefficients, communications between the processors of Figure 7(a) is strictly local, i.e., between neighboring processors only. Through pipelining, it is in fact possible to eliminate global data transfers altogether. For $n = 4$, Figure 8 shows the flow of data through the machine of Figure 7(a), assuming the use of pipelining, and assuming that the machine is initialized as indicated (*time* $= 0$ entry of Figure 8). Figure 8 depicts the flow of the split reflection coefficients explicitly. The unit of time in Figure 8 is the basic time unit previously stated. Note that the machine may be initialized in $O(n)$ time quite readily.

time	cell			
	0	1	2	3
0	$v_{0,0}$ $v_{1,0}$ $\times$	$v_{0,1}$ $v_{1,1}$ $\times$	$v_{0,2}$ $v_{1,2}$ $\times$	$v_{0,3}$ $v_{1,3}$ $\times$
1	$\boxed{\begin{matrix}v_{0,0}\\ v_{1,0}\\ \alpha_1\end{matrix}}$	$v_{0,1}$ $v_{1,1}$ $\times$	$v_{0,2}$ $v_{1,2}$ $\times$	$v_{0,3}$ $v_{1,3}$ $\times$
2	$v_{1,0}$ $v_{2,0}$ α_1	$\boxed{\begin{matrix}v_{0,1}\\ v_{1,1}\\ \alpha_1\end{matrix}}$	$v_{0,2}$ $v_{1,2}$ $\times$	$v_{0,3}$ $v_{1,3}$ $\times$
3	$\boxed{\begin{matrix}v_{1,0}\\ v_{2,0}\\ \alpha_2\end{matrix}}$	$v_{1,1}$ $v_{2,1}$ α_1	$\boxed{\begin{matrix}v_{0,2}\\ v_{1,2}\\ \alpha_1\end{matrix}}$	$v_{0,3}$ $v_{1,3}$ $\times$
4	$v_{2,0}$ $v_{3,0}$ α_2	$\boxed{\begin{matrix}v_{1,1}\\ v_{2,1}\\ \alpha_2\end{matrix}}$	$v_{1,2}$ $v_{2,2}$ α_1	$\boxed{\begin{matrix}v_{0,3}\\ v_{1,3}\\ \alpha_1\end{matrix}}$
5	$\boxed{\begin{matrix}v_{2,0}\\ v_{3,0}\\ \alpha_3\end{matrix}}$	$v_{2,1}$ $v_{3,1}$ α_2	$\boxed{\begin{matrix}v_{1,2}\\ v_{2,2}\\ \alpha_2\end{matrix}}$	$v_{1,3}$ $\times$ α_1
6	$v_{3,0}$ $v_{4,0}$ α_3	$\boxed{\begin{matrix}v_{2,1}\\ v_{3,1}\\ \alpha_3\end{matrix}}$	$v_{2,2}$ $\times$ α_2	$v_{1,3}$ $\times$ α_2
7	$\boxed{\begin{matrix}v_{3,0}\\ v_{4,0}\\ \alpha_4\end{matrix}}$	$v_{3,1}$ $\times$ α_3	$v_{2,2}$ $\times$ α_3	$\times$ $\times$ α_2

$\boxed{}$ = inactive

Figure 8: Pipelined flow of data through the machine of Figure 7(a) eliminating all global data transfers.

It is not necessary to fully initialize the machine before computation can begin. For $n = 4$, Figure 9 depicts the flow of data without initialization. Figure 9 depicts two successive sets of inputs. The first set is $\{v_{0,j}, v_{1,j}\}$ and the second set

$\{\overline{v}_{0,j}\ , \overline{v}_{1,j}\}$ is denoted by barred variables to distinguish it from the first set. The outputs due to the second set of inputs are similarly barred. It is necessary, in this scenario, for all cells to be able to perform division. It is assumed, for simplicity, that α_k is immediately output to processor K upon its creation, but this may be avoided by further pipelining. It is clear that the overall throughput of the machine is now determined by the length of our basic time unit. As a result, a high throughput is possible with the arrangement of Figure 9.

From the inverse mapping of equation (6) in Chapter III, it is possible to obtain the Schur variables $u_{-i}^{(k)}$ from the split Schur variables $v_{k,i}$, and so to obtain the elements of matrix U_n in $T_n = U_n^T D_n^{-1} U_n$. An inspection of (6) (Chapter III) reveals that the Schur variables may be computed when the processors of Figure 7(a) are inactive during the times indicated in Figure 8. A processor is inactive if it is not computing a split Schur variable.

From the standpoint of the computation of reflection coefficients and the computation of the elements of U_n, the split Schur algorithms are more efficient than the Schur algorithm, in the context of a parallel-pipelined processor implementation. Although the inverse mapping from the split Schur variables to the Schur variables requires extra multiplications, these multiplications may be performed when the processors in the array of Figure 7(a) are otherwise inactive. Recall that these extra multiplications rendered the split Schur algorithms no more efficient at computing the elements of U_n than the Schur algorithm, in the context of a sequential processor implementation (see Chapter III, section 2). However, in the context of a parallel processor implementation, this limitation is lifted.

4. Parallel-Pipelined Architectures for the Schur Algorithm for Hermitian Toeplitz Matrices of Any Rank Profile

We will now consider the parallel-pipelined processor implementation of the Schur algorithm for Hermitian Toeplitz matrices of any rank profile derived in section 3 of Chapter III. The relevant algorithm is summarized in the form of pseudocode in

time	output	cell				input data stream			
		0	1	2	3				
0		× × ×	× × ×	× × ×	× × ×	$v_{0,0}$ $v_{1,0}$ ×	$v_{0,1}$ $v_{1,1}$ ×	$v_{0,2}$ $v_{1,2}$ ×	$v_{0,3}$ $v_{1,3}$ ×
1		× × ×	× × ×	× × ×	$v_{0,0}$ $v_{1,0}$ ×	$v_{0,1}$ $v_{1,1}$ ×	$v_{0,2}$ $v_{1,2}$ ×	$v_{0,3}$ $v_{1,3}$ ×	$\bar{v}_{0,0}$ $\bar{v}_{1,0}$ ×
2	α_1	× × ×	× × ×	$v_{0,0}$ $v_{1,0}$ α_1	$v_{0,1}$ $v_{1,1}$ ×	$v_{0,2}$ $v_{1,2}$ ×	$v_{0,3}$ $v_{1,3}$ ×	$\bar{v}_{0,0}$ $\bar{v}_{1,0}$ ×	$\bar{v}_{0,1}$ $\bar{v}_{1,1}$ ×
3		× × ×	$v_{1,0}$ $v_{2,0}$ α_1	$v_{0,1}$ $v_{1,1}$ α_1	$v_{0,2}$ $v_{1,2}$ ×	$v_{0,3}$ $v_{1,3}$ ×	$\bar{v}_{0,0}$ $\bar{v}_{1,0}$ ×	$\bar{v}_{0,1}$ $\bar{v}_{1,1}$ ×	$\bar{v}_{0,2}$ $\bar{v}_{1,2}$ ×
4	α_2	$v_{1,0}$ $v_{2,0}$ α_2	$v_{1,1}$ $v_{2,1}$ α_1	$v_{0,2}$ $v_{1,2}$ α_1	$v_{0,3}$ $v_{1,3}$ ×	$\bar{v}_{0,0}$ $\bar{v}_{1,0}$ ×	$\bar{v}_{0,1}$ $\bar{v}_{1,1}$ ×	$\bar{v}_{0,2}$ $\bar{v}_{1,2}$ ×	$\bar{v}_{0,3}$ $\bar{v}_{1,3}$ ×
5		$v_{2,0}$ $v_{3,0}$ α_2	$v_{1,1}$ $v_{2,1}$ α_2	$v_{1,2}$ $v_{2,2}$ α_1	$\bar{v}_{0,0}$ $\bar{v}_{1,0}$ ×	$\bar{v}_{0,1}$ $\bar{v}_{1,1}$ ×	$\bar{v}_{0,2}$ $\bar{v}_{1,2}$ ×	$\bar{v}_{0,3}$ $\bar{v}_{1,3}$ ×	× × ×
6	α_3 $\bar{\alpha}_1$	$v_{2,0}$ $v_{3,0}$ α_3	$v_{2,1}$ $v_{3,1}$ α_2	$\bar{v}_{0,0}$ $\bar{v}_{1,0}$ $\bar{\alpha}_1$	$\bar{v}_{0,1}$ $\bar{v}_{1,1}$ ×	$\bar{v}_{0,2}$ $\bar{v}_{1,2}$ ×	$\bar{v}_{0,3}$ $\bar{v}_{1,3}$ ×	× × ×	× × ×
7		$v_{3,0}$ $v_{4,0}$ α_3	$\bar{v}_{1,0}$ $\bar{v}_{2,0}$ $\bar{\alpha}_1$	$\bar{v}_{0,1}$ $\bar{v}_{1,1}$ $\bar{\alpha}_1$	$\bar{v}_{0,2}$ $\bar{v}_{1,2}$ ×	$\bar{v}_{0,3}$ $\bar{v}_{1,3}$ ×	× × ×	× × ×	× × ×
8	α_4 $\bar{\alpha}_2$	$\bar{v}_{1,0}$ $\bar{v}_{2,0}$ $\bar{\alpha}_2$	$\bar{v}_{1,1}$ $\bar{v}_{2,1}$ $\bar{\alpha}_1$	$\bar{v}_{0,2}$ $\bar{v}_{1,2}$ $\bar{\alpha}_1$	$\bar{v}_{0,3}$ $\bar{v}_{1,3}$ ×	× × ×	× × ×	× × ×	× × ×
9		$\bar{v}_{2,0}$ $\bar{v}_{3,0}$ $\bar{\alpha}_2$	$\bar{v}_{1,1}$ $\bar{v}_{2,1}$ $\bar{\alpha}_2$	$\bar{v}_{1,2}$ $\bar{v}_{2,2}$ $\bar{\alpha}_1$	× × ×	× × ×	× × ×	× × ×	× × ×
10	$\bar{\alpha}_3$	$\bar{v}_{2,0}$ $\bar{v}_{3,0}$ $\bar{\alpha}_3$	$\bar{v}_{2,1}$ $\bar{v}_{3,1}$ $\bar{\alpha}_2$	× × ×	× × ×	× × ×	× × ×	× × ×	× × ×
11		$\bar{v}_{3,0}$ $\bar{v}_{4,0}$ $\bar{\alpha}_3$	× × ×	× × ×	× × ×	× × ×	× × ×	× × ×	× × ×
12	$\bar{\alpha}_4$	× × ×	× × ×	× × ×	× × ×	× × ×	× × ×	× × ×	× × ×

Figure 9: Flow of data through the machine of Figure 7(a) without the use of a separate initialization phase.

section 3.3 of Chapter III. For the sake of brevity, we shall refer to this algorithm as the Schur algorithm in the remainder of this section. It is apparent that the Schur algorithm may be most plausibly implemented on a linear array of processors like that in

Figure 1(a).

In the Schur algorithm, the processor array of Figure 1(a) must cope with two cases: $\sigma_k \neq 0$, which gives rise to what we usually mean by the Schur algorithm (i.e., Chapter II Schur algorithm), and $\sigma_k = 0$ which gives rise to modifications that facilitate "jumping over the singular gap". Since $\sigma_k = u_{-k+1}^{(k)}$ is computed in cell 0, it is cell 0 that must determine which case applies. Furthermore, it must communicate whether or not $\sigma_k = 0$ to the other processors in the array in order that they may take the proper course of action. As long as $\sigma_k \neq 0$, the array may run in the usual way (e.g., as depicted in Figure 1(c)). If $\sigma_k = 0$, the array must compute the Schur variables $u_i^{(1)} \cdots u_i^{(r+1)}$ ($k = r + 1$, r is the right singular point), and it is necessary to save the variables $u_i^{(r)}$ and $u_i^{(r+1)}$, in the cells in which they were created, since they are used to jump over the singular gap (see the Schur algorithm for the case where $\sigma_k = 0$). Thus, when cell 0 detects that $\sigma_k = 0$, it must notify the remaining processors that they are only going to compute the Schur variables of up to order $r+1$, that is, $u_i^{(r+1)}$. It is then necessary to execute the "If $\sigma_k \neq 0$ then" statement of the Schur algorithm for the case where $\sigma_k = 0$. This will involve computing l (Iohvidov index), α_i , ψ_i , p_i , β_0 , σ_{n+1} and the order $n+1$ Schur variables, $u_i^{(n+1)}$. We will discuss, in greater detail, the computation of these parameters below.

In what follows we shall assume that the processor (cell) which computes r , l , σ_r , or β_0 is allowed to globally broadcast the result to all other processors. It turns out that cell 0 computes r , σ_r , and β_0, while cell l computes l. In practice, such limited global communications simplifies the parallel implementation of the Schur algorithm.

It will be useful to consider Toeplitz matrices C_m with many singular gaps. For example, consider a C_m with leading principal submatrices $C_1, \ldots, C_m$, where, recalling that $f_k = det(C_k)$ (Chapter III),

$$f_1 \neq 0 , f_2 = 0 , f_3 \neq 0 , \cdots , f_{m-1} = 0 , f_m \neq 0 . \qquad (12a)$$

For this to make sense m must be odd and $m > 1$. Thus we have the sequence of right singular points

$$r_1 = 1\ ,\ r_2 = 3\ ,\ \cdots\ ,\ r_{\frac{m-1}{2}} = m-2\ , \tag{12b}$$

(so $r_i = 2i - 1$), and the sequence of Iohvidov indices

$$l_1 = 1\ ,\ l_2 = 1\ ,\ \cdots\ ,\ l_{\frac{m-1}{2}} = 1\ . \tag{12c}$$

A sequence of left singular points (n_i) is also generated ($n_i = r_i + 2l_i$ via (10) in Chapter III). Notice that

$$\sum_{i=1}^{\frac{m-1}{2}} l_i \equiv O(m)\ . \tag{13}$$

Equation (13) simply says that the size of all singular gaps combined is of the order of m (the size of the Toeplitz matrix). Note that we are assuming that a matrix C_m with the properties of (12a,b,c) exists (for all odd m , $m > 1$). This might not be so. More generally, C_m has many singular gaps if it possesses an Iohvidov index sequence satisfying

$$\sum_{i \in S} l_i \equiv O(m)\ ,\ \ S^\circ \equiv O(m)\ , \tag{14}$$

where $S^\circ =$ the number of elements in the index set S. The example of C_m satisfying (12a,b,c) is intended merely to indicate how such a matrix might arise. It is clear though that the existence problem remains since we do not know if there exists a C_m satisfying (14) for all m. We will simply conjecture that such matrices exist.

It is important to note that Toeplitz matrices satisfying (14) represent the worst-case scenario. If C_m satisfies (14), then the "else" case of the "If $\sigma_k \neq 0$ then" statement in the Schur algorithm will be executed often enough to dominate the complexity of the overall algorithm. Hence, a complexity analysis of the Schur algorithm (or that in [6]) must consider this case, and this case alone.

If $\sigma_k = 0$ then it is first necessary to compute l, before we can find p_i , $u_i^{(n+1)}$, etc. Recall that l is the smallest positive integer such that $u_{-(r+l)}^{(r+1)} \neq 0$. The order $r+1$ Schur variable $u_{-(r+l)}^{(r+1)}$ resides in cell l. One way to find l is as follows. Since cell 0 has computed r, and broadcasted this knowledge to all other processors, all of the processors know that they are to do nothing until l is found (other than to continue computing the order $r+1$ Schur variables). Cell 0 has $u_{-r}^{(r+1)} = 0$, and it can notify cell 1 of this fact. If $u_{-(r+1)}^{(r+1)} = 0$ (this is in cell 1) then cell 1 notifies cell 2 of this fact, and so on, until cell l is reached where $u_{-(r+l)}^{(r+1)} \neq 0$ holds. At this point cell l knows what l is, and it can globally broadcast l to all of the remaining processors. Since in the worst possible case (14) holds, a total of no more than $O(m)$ clock cycles are used by the processor array in computing and broadcasting each value l_i. Thus, the computation and communication of Iohvidov indices does not lead to a parallel processing bottleneck.

Having computed l, it is now necessary to compute p_i for $i = 0,1,...,2l$. This requires the parameters α_i and ψ_i which are Schur variables. Recall that

$$\alpha_i = u_{l+i}^{(r+1)} \, , \quad \psi_i = \frac{1}{\sigma_r} u_{-(i-l-1)}^{(r)} \tag{15a}$$

if $r \neq 0$, and that

$$\alpha_i = c_{l+i} \, , \quad \psi_i = -\delta_{i,l} \tag{15b}$$

if $r = 0$, for $i = 0,1,...,l$. The location of these parameters in the cells of Figure 1(a) is depicted in Figure 10. To compute p_i requires solving

$$\begin{bmatrix} \alpha_0 & 0 & .. & 0 \\ \alpha_1 & \alpha_0 & .. & 0 \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \alpha_l & \alpha_{l-1} & .. & \alpha_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \cdot \\ \cdot \\ p_l \end{bmatrix} = - \begin{bmatrix} \psi_l \\ \psi_{l-1} \\ \cdot \\ \cdot \\ \psi_0 \end{bmatrix} , \tag{16}$$

and we have $p_{l+i} = -\overline{p}_{l-i}$ $(i = 1,...,l)$. It is clear that (16) can be solved via the First Back-Substitution algorithm (see section 2 of this Chapter). From Figure 10, the parameters ψ_i and α_i must be relocated in order to facilitate the execution of this algorithm. Specifically, ψ_i should be located in cell $l-i$, and α_i should be located in cell i. However, from the figure we initially have ψ_i in cell $l-i+1$, and α_i is in cell $l+i$. Clearly, all that we need to do is to shift the elements α_i in cells l to $2l$ leftwards by l cell positions. This will take l clock cycles. Similarly, the elements ψ_i in cells 1 to $l+1$ need only be shifted leftwards by one cell position. This will only take one clock cycle. Since cell 0 is capable of performing division, by comparison with Figure 2, the First Back-Substitution algorithm may proceed in the usual manner. Parameters $p_0, \ldots, p_l$ (in this order) will then emerge from cell 0. It is suggested that p_i be stored in RAM (random access memory) in cell 0. The reasons for this will become apparent later on in this section. It is clear that p_i (all i) will be computed in $O(l)$ time. This includes both the shifting into their proper cell positions of the parameters α_i and ψ_i, and the back-substitution operation itself. Once again, since (14) holds in the worst case, this will not result in a parallel processing bottleneck.

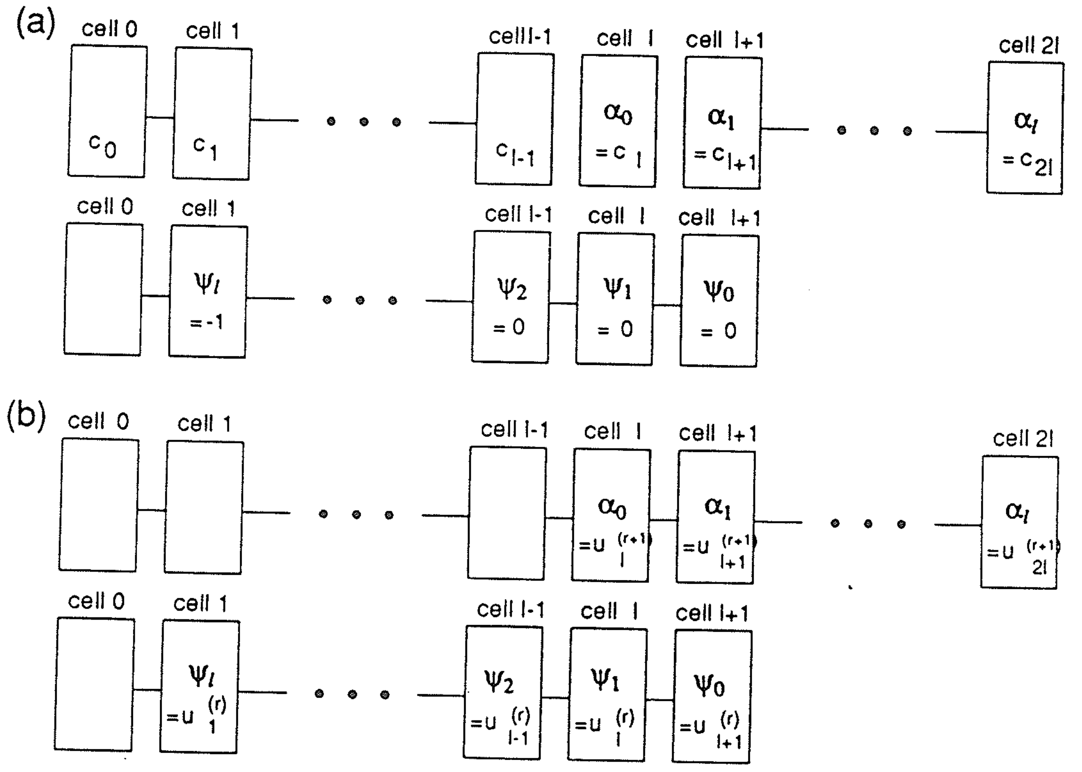


Figure 10: Positions of the parameters ψ_i and α_i in the cells of Figure 1(a) before shifting them into their proper cell positions. (a) case $r = 0$; (b) case $r \neq 0$.

The order $n+1$ Schur variables $u_i^{(n+1)}$ are computed via

$$u_{i-n}^{(n+1)} = \bar{\beta}_0 \left[\frac{1}{\sigma_r} u_{i-(r+l)+1}^{(r)} + \sum_{v=0}^{2l} p_v u_{i-(r+v)}^{(r+1)} \right] , \tag{17}$$

for $0 \geq i \geq n-m+1$, $n \leq i \leq m-1$. The summation term of (17) is a convolutional summation. As such, it can be put into "matrix form", where the matrix will have a

Toeplitz (or possibly Hankel) structure. Thus, for $0 \geq i \geq n-m+1$, (17) becomes

$$\begin{bmatrix} u_{-n}^{(n+1)} \\ \cdot \\ \cdot \\ u_{-(m-1)}^{(n+1)} \end{bmatrix} = \frac{\beta_0}{\sigma_r} \begin{bmatrix} u_{-(r+l)+1}^{(r)} \\ \cdot \\ \cdot \\ u_{l-m+2}^{(r)} \end{bmatrix} + \bar{\beta}_0 \begin{bmatrix} u_{-r}^{(r+1)} & \cdot\cdot & u_{-n}^{(r+1)} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ u_{2l-m+1}^{(r+1)} & \cdot\cdot & u_{-m+1}^{(r+1)} \end{bmatrix} \begin{bmatrix} p_0 \\ \cdot \\ \cdot \\ p_{2l} \end{bmatrix} , \tag{18a}$$

and for $n \leq i \leq m-1$, (17) becomes

$$\begin{bmatrix} u_0^{(n+1)} \\ \cdot \\ \cdot \\ u_{m-n-1}^{(n+1)} \end{bmatrix} = \frac{\beta_0}{\sigma_r} \begin{bmatrix} u_{l+1}^{(r)} \\ \cdot \\ \cdot \\ u_{m-r-l} \end{bmatrix} + \bar{\beta}_0 \begin{bmatrix} u_{2l}^{(r+1)} & \cdot\cdot & u_0^{(r+1)} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ u_{m-r-1}^{(r+1)} & \cdot\cdot & u_{m-n-1}^{(r+1)} \end{bmatrix} \begin{bmatrix} p_0 \\ \cdot \\ \cdot \\ p_{2l} \end{bmatrix} . \tag{18b}$$

From (18a,b) we see that to compute all order $n+1$ Schur variables requires all of the order $r+1$ Schur variables and certain order r Schur variables. The order $r+1$ Schur variables are present in cells 0 through to $m-r-1$. The order r Schur variables $u_{-(r+l)+1}^{(r)} \cdots u_{l-m+2}^{(r)}$ are in cells l to $l+m-n-1$, respectively. The order r Schur variables $u_{l+1}^{(r)} \cdots u_{m-r-1}^{(r)}$ are contained in cells $l+1$ to $l+m-n$, respectively. We want the order $n+1$ Schur variables $u_{-n}^{(n+1)} \cdots u_{-(m-1)}^{(n+1)}$, and $u_0^{(n+1)} \cdots u_{m-n-1}^{(n+1)}$ in cells 0 to $m-n-1$, respectively. Thus, the order r Schur variables in (18a,b) must be shifted into these cell positions, since they are used to determine the order $n+1$ Schur variables. This shifting operation will take no more than $O(l)$ time.

Let $p = [p_0 \cdots p_{2l}]^T$, let the matrix in (18a) be H, and let the matrix in (18b) be T. Both H and T are $(m-n) \times (2l+1)$ matrices. Inspections of H and T reveal that they are Hankel and Toeplitz, respectively. Define,

$$u^+ = Tp \ , \ u^- = Hp \ . \tag{19}$$

The vectors u^+ and u^- are the results of matrix-vector products. Matrix-vector products may be implemented on linear systolic arrays in $O(w)$ time, where w is the matrix bandwidth (see [4], pp. 274-276). The bandwidth of a matrix is here defined as the sum of the number of rows and columns minus unity. We will briefly digress to explain the linear systolic implementation of a matrix-vector product.

Often (as in (19)) we wish to perform matrix-vector multiplication:

$$y = Ax \quad . \tag{20}$$

Here $y = [y_0 \ \cdots \ y_{n-1}]^T$, $x = [x_0 \ \cdots \ x_{m-1}]^T$, and $A = [a_{ij}]_{n \times m}$ ($i = 0,1,...,n-1$, $j = 0,1,...,m-1$). Figure 11 depicts systolic matrix-vector multiplication for the special case where $n = m = 3$. The systolic array is, as we have noted, a linear array of $n + m - 1$ processors (cells). Vector x is fed into the array at its left-hand end (cell 0), and vector $y = 0$ is fed into the array at its right-hand end (cell $n+m-2 = 4$). The solution y appears at the left-hand end beginning with the component y_0 at $n + m - 1$ clock cycles after we began feeding data into the array. The elements of matrix A are fed into the "tops" of the cells. For example, a_{02} feeds into cell 0, a_{12} and a_{01} feed into cell 1, a_{22} , a_{11} and a_{00} feed into cell 2, etc. Thus, A is fed into the array "diagonal-wise". The flow of data through the machine is shown in Figure 11. To understand this flow, consider the progress of the element y_0 through the array. At time = 0, y_0 enters cell 4 (with an initial value of zero), and after every clock cycle it moves to the left by one cell position. When it reaches cell 2 at time = 2, cell 2 performs

$$y_0 = y_0 + a_{00}x_0 \quad ,$$

and passes this new value for y_0 on to cell 1, where cell 1 then performs

$$y_0 = y_0 + a_{01}x_1 \quad ,$$

and this value is passed on to cell 0, where cell 0 performs

$$y_0 = y_0 + a_{02}x_2 \quad .$$

At time = 5, y_0 emerges from cell 0 with the final value

$$y_0 = a_{00}x_0 + a_{01}x_1 + a_{02}x_2$$

which is the correct value for y_0 in the vector $y = Ax$. Similar reasoning applies to the vector components y_1 and y_2.

We may apply the systolic array of Figure 11 to the problem of computing u^+ and u^- in (19). Since the elements of A feed into the array diagonal-wise, it is desirable for A to be Toeplitz, since then all of the elements of A on a given diagonal are

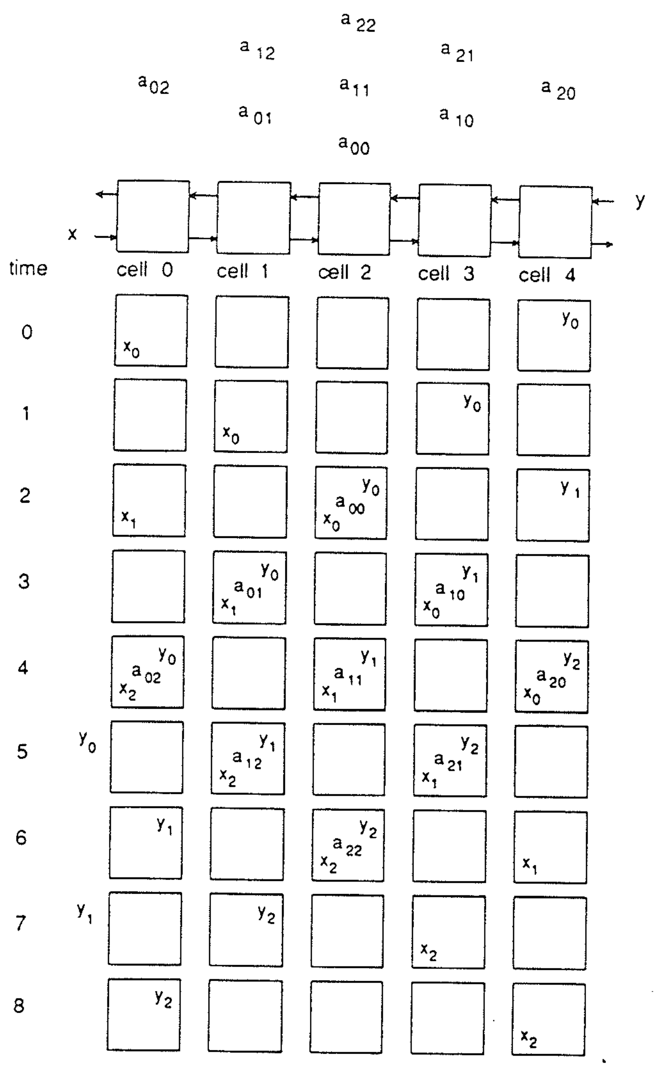


Figure 11: Linear systolic array for matrix-vector multiplication.

the same. Clearly, T in (19) is Toeplitz, but H is not. However, we can write

$$u^- = (HJ)(Jp) \ , \tag{21}$$

where HJ is Toeplitz, and $Jp = [p_{2l}\ p_{2l-1}\ \cdots\ p_0]^T$ (p with the elements in reverse order). Recall that J is our symbol for the exchange matrix (see Chapter II, section 2.1).

As an example, let us consider H and T for $m = 7$, $r = 2$, $l = 1$, and so $n = 4$. Thus, from (18a,b)

$$H = \begin{bmatrix} u_{-2}^{(3)} & u_{-3}^{(3)} & u_{-4}^{(3)} \\ u_{-3}^{(3)} & u_{-4}^{(3)} & u_{-5}^{(3)} \\ u_{-4}^{(3)} & u_{-5}^{(3)} & u_{-6}^{(3)} \end{bmatrix}, \quad T = \begin{bmatrix} u_{2}^{(3)} & u_{1}^{(3)} & u_{0}^{(3)} \\ u_{3}^{(3)} & u_{2}^{(3)} & u_{1}^{(3)} \\ u_{4}^{(3)} & u_{3}^{(3)} & u_{2}^{(3)} \end{bmatrix} . \tag{22}$$

Figure 12(a) shows the locations of the Schur variables, in the array of Figure 1(a), that make up T and H prior to the computation of the order $n+1$ Schur variables. Clearly,

$$HJ = \begin{bmatrix} u_{-4}^{(3)} & u_{-3}^{(3)} & u_{-2}^{(3)} \\ u_{-5}^{(3)} & u_{-4}^{(3)} & u_{-3}^{(3)} \\ u_{-6}^{(3)} & u_{-5}^{(3)} & u_{-4}^{(3)} \end{bmatrix} . \tag{23}$$

To compute u^- we use HJ, since the elements of it are located in the proper cell positions to facilitate the systolic multiplication of HJ by Jp. That is, p must be fed into cell 0 in reverse order (i.e., p_{2l} first, then p_{2l-1}, ..., and finally p_0). This is illustrated in the middle of Figure 12 (see Figure 12(b)). Since the elements of T are in their proper cells, Tp may be computed in the usual way (see the bottom of Figure 12(b)). The reader can now see the desirability of storing p_i in RAM in cell 0.

In Figure 11 notice that processors are inactive half of the time, and on alternating cycles. In other words, it is possible to interleave the computation of two matrix-vector products. This is advantageous in our present problem since we wish to compute the two products Hp and Tp. By interleaving the computation of Hp and Tp we can compute (18a) and (18b) simultaneously.

The order $n+1$ Schur variables of (18a) will be produced by cell 0 in the order $u_{-n}^{(n+1)} \cdots u_{-(m-1)}^{(n+1)}$. Similarly, the order $n+1$ Schur variables of (18b) will be produced in the order $u_0^{(n+1)} \cdots u_{m-n-1}^{(n+1)}$. Recall that we want $u_{-n}^{(n+1)} \cdots u_{-(m-1)}^{(n+1)}$ to be finally located in cells 0 to $m-n-1$, respectively, and similarly, we want $u_0^{(n+1)} \cdots u_{m-n-1}^{(n+1)}$ to be finally located in cells 0 to $m-n-1$, respectively. It shall take $O(m-n)$ clock cycles to ensure that this is so.

Since T and H are $(m-n) \times (2l+1)$, the order $n+1$ Schur variables can be computed on the linear systolic array in $O(m-n+2l) = O(m-r)$ time. Recall that the bandwidths of H and T determine the time complexity, and their bandwidths are

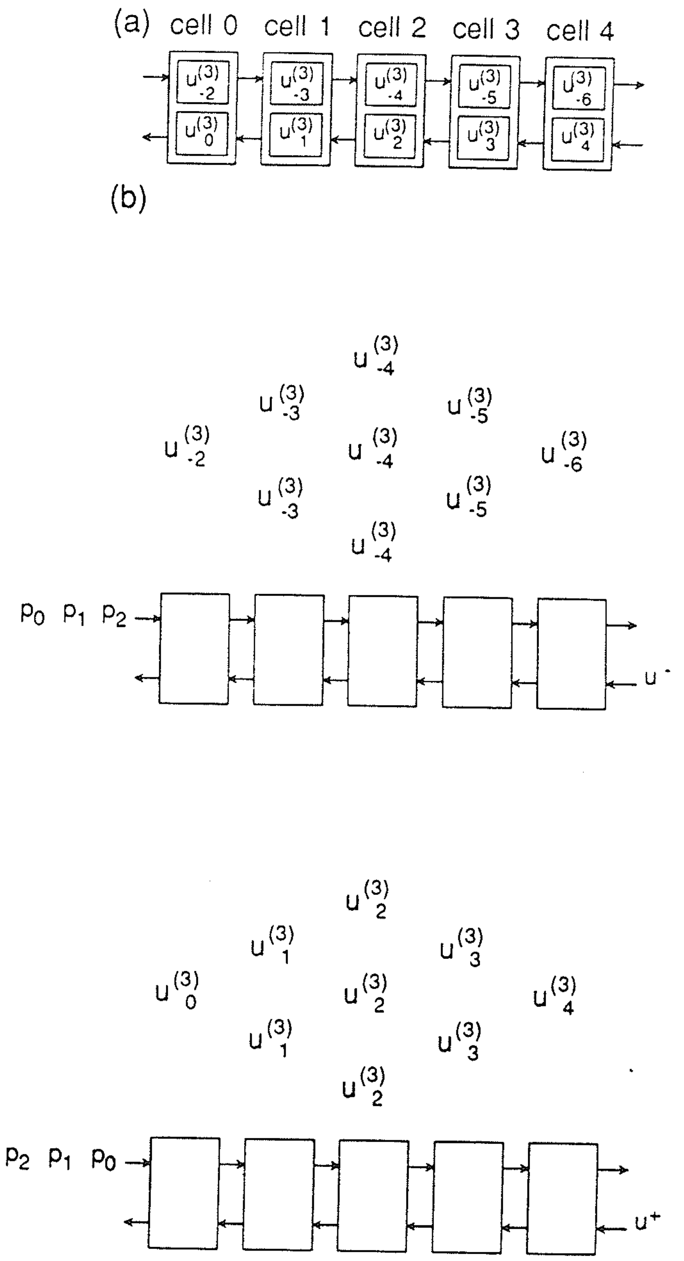


Figure 12: Example of the computation of $u_{i-n}^{(n+1)}$ for $m = 7$, $r = 2$, $l = 1$, $n = 4$. (a) Contents of the cells before the computation of $u_{i-n}^{(n+1)}$; (b) Systolic matrix-vector multiplication to compute u^+ and u^- which are used to determine $u_{i-n}^{(n+1)}$.

$m-n+2l$ (see [4], page 274). If C_m satisfies (14), then the amount of time that the array spends in computing the order $n+1$ Schur variables is

$$\sum_{i \in S} (m - r_i) = O(m^2) \quad . \tag{24}$$

Evidently this presents a serious problem since the machine of Figure 1(a) will then perform no better (asymptotically) than a sequential processing machine. In other words, it appears that the computation of order $n+1$ Schur variables is a parallel processing bottleneck.

Is it possible to overcome this bottleneck ? Unfortunately, we shall see that we cannot remove this bottleneck while using systolic matrix-vector multiplication. However, there remains the possibility that other matrix-vector multiplication schemes exist that do not give rise to the bottleneck to be described below.

In Figure 1(c) we note that it is possible to commence computing the order $k+1$ Schur variables only two clock cycles after commencing the computation of the order k Schur variables. The computation of Schur variables of all orders may then be said to be *overlapped.* It is because of overlapping that the machine of Figure 1(a), when operated as in Figure 1(c), continues to have a time complexity of $O(m)$, just as it did when it was operated as in Figure 1(b). If we could overlap the computation of the order $n+1$ Schur variables with the Schur variables of all other orders, much as in Figure 1(c), then the bottleneck could be removed. Note that it would be acceptable for the instant we begin to compute order k Schur variables to be separated from the instant we begin to compute the order $k+1$ Schur variables by $O(l)$ clock cycles. Such a separation would only occur when singular gaps are encountered however. This is vital since before we could begin to compute the order $n+1$ Schur variables it was necessary to spend $O(l)$ clock cycles computing the parameters p_i.

Recall that it takes at least $m+n-1$ clock cycles before the first component of the product vector appears, in a matrix-vector product, given that the matrix A is $n \times m$. This delay we shall call the *latency* of the systolic array. Thus, in our present problem, it will take at least $m-n+2l = m-r$ clock cycles to produce the first order $n+1$ Schur variable. Hence it will also be at least $m-r$ clock cycles before we can **begin** computing the order $n+2$ Schur variables (using the usual form of the Schur algorithm, which would assume that $\sigma_{n+1} \neq 0$). Because it takes so long to begin the computation of the order $n+2$ Schur variables, it will take a total of $O(m^2)$ clock cycles to compute the order $n_i + 1$ Schur variables associated with each singular gap i (as in (24)). It is now clear that to successfully eliminate the bottleneck requires a means of computing the matrix-vector products of (19) with a latency of not more than $O(l)$ clock cycles, rather than $O(m-r)$ clock cycles. Furthermore, the new method (assuming that it exists), must never require that all of the order $r+1$ Schur variables be available simultaneously, or else overlapping would once again be precluded. As well, the new method should run on a linear processor array (i.e., we do not want to change the array topology). It is not known whether such a method exists. Thus, for the present at least, the bottleneck is impossible to remove.

It is important to note that alternative matrix-vector multiplication schemes do exist (e.g., methods involving wavefront array processors [5,6]), and that it may be possible to adapt these methods to our present problem and so eliminate the unfortunate bottleneck. However, it is anticipated that significant extra work shall be required in this area, and so we shall leave this as an open problem. Other open problems exist in the area of implementing the Schur algorithm in a parallel processing environment. These are summarized in Chapter X.

Since we must conclude (tentatively) that the bottleneck is unremovable, we are in fact stating that the Schur algorithm is inherently sequential. But this of course assumes that C_m satisfies (14), and this is the worst-case scenario. Suppose that we now assume $S^\circ \equiv O(1)$. In other words, assume that the number of singular gaps in C_m is independent of m. If m is large, then C_m will have very few singular gaps. In this case the complexity of the Schur algorithm will be dominated by the complexity of the usual form of the Schur algorithm. Hence, no bottleneck will exist, and the parallel-pipelined processor implementation of the Schur algorithm will run in $O(m)$ time, despite our present inability to properly overlap the computation of the Schur variables. Furthermore, the pipelining of such parameters as r and l to all processors can be undertaken without any increase in the time complexity of the implementation. Thus, in this instance, all global communications can certainly be eliminated. It is now clear that the success of the Schur algorithm in its parallel form strongly depends upon the number of singular gaps (not their size) in C_m relative to the order of C_m.

REFERENCES

[1] S.-Y. Kung, Y. H. Hu, "A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-31, Feb. 1983, pp. 66-75.

[2] R. P. Brent, F. T. Luk, "A Systolic Array for the Linear-Time Solution of Toeplitz Systems of Equations," J. of VLSI and Comp. Syst., vol. 1, 1983, pp. 1-22.

[3] C. J. Zarowski, H. C. Card, "A Parallel-Pipelined Architecture for Implementing the Split Schur Algorithms for Reflection Coefficient Computation," to be submitted to CCVLSI'88.

[4] C. Mead, L. Conway, *Introduction to VLSI Systems*. Reading, Massachusetts: Addison-Wesley, 1980.

[5] S.-Y. Kung, H. J. Whitehouse, T. Kailath (eds.), *VLSI and Modern Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1985.

[6] S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, D. V. Bhaskar Rao, "Wavefront Array Processors: Language, Architecture, and Applications," IEEE Trans. on Comp., vol. C-31, Nov. 1982, pp. 1054-1066.

[7] P. Delsarte, Y. Genin, "On the Splitting of Classical Algorithms in Linear Prediction Theory," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, May 1987, pp. 645-653.

Chapter V

THE BEHAVIOUR OF THE SCHUR AND SPLIT SCHUR ALGORITHMS UNDER FIXED-POINT ARITHMETIC CONDITIONS

So far we have considered the Schur and split Schur algorithms (of Chapter II) without regard to the practical matter of how these algorithms behave when implemented using finite precision arithmetic. It is clear that the issue of finite precision arithmetic effects is important, as there is no point in implementing an algorithm that is numerically unstable. As well, it is also important to identify ill-conditioned problem instances. In the present chapter we shall therefore present a finite precision arithmetic analysis of the Schur and split Schur algorithms. Some simulation results will also be presented as a check of the analytical results. We shall confine our attention to Toeplitz matrices that are real-valued, positive definite autocorrelation matrices. Furthermore, we shall only consider fixed-point arithmetic. Recall from Chapter II (section 2.6) that the Schur and split Schur variables satisfy certain bounds that make the Schur and split Schur algorithms particularly desirable from the standpoint of a fixed-point arithmetic implementation. This is beneficial from an economic point of view as floating-point arithmetic hardware is generally more expensive than fixed-point arithmetic hardware. The fixed-point arithmetic analyses and simulations of the Schur and split Schur algorithms are due to Zarowski and Card [1,2].

1. Literature Review

Before we present the analyses of the behaviour of the Schur and split Schur algorithms under fixed-point arithmetic conditions, it is informative to consider the behaviour of the Levinson-Durbin algorithm under finite precision arithmetic conditions. The results on the Levinson-Durbin algorithm to follow are due to Cybenko

[3,4] and to Alexander and Rhee [5]. Since we may derive the Schur algorithm from the Levinson-Durbin algorithm (recall Le Roux and Gueguen [6]), we might reasonably expect that the numerical properties of the Schur and Levinson-Durbin algorithms will be qualitatively similar. Similarly, as the split Schur algorithms are derivable from the Schur algorithm, we would expect that the split Schur algorithms have numerical properties similar to those of the parent Schur algorithm. By the end of the chapter we shall see that this is so.

Cybenko [3,4] considers Toeplitz matrices $R_n = [r_{|i-j|}]_{n \times n}$ that are real, symmetric and positive definite, and they are normalized so that $r_0 = 1$ (hence $|r_i| < 1$, $i > 1$). Autocorrelation matrices for real-valued data are real and symmetric, and they are often positive definite as well. Recall that, in general, if R_n is an autocorrelation matrix it will at least be positive semidefinite (see Roberts and Mullis [7]). Under the assumption of positive definiteness, the reflection coefficients K_i associated with R_n will satisfy, for all i,

$$|K_i| < 1 \ . \tag{1}$$

The Durbin algorithm [3], or the Levinson-Durbin algorithm as it is called in [4], is used to solve

$$R_n a_n = -[r_1 \ \cdots \ r_n]^T = -r_n \ . \tag{2}$$

Equation (2) is somewhat different from equation (1) in Chapter II, but it turns out that there is no significant difference between the Levinson-Durbin algorithm of [4] and that in Chapter II (for the special case of T_n considered here). The solution vector a_n of (2) has the form $a_n = [a_{n,1} \, a_{n,2} \ \cdots \ a_{n,n}]^T$.

Cybenko [3,4] presents results on the *condition number* of R_n, and on the numerical stability of the Levinson-Durbin algorithm. We will first summarize Cybenko's results concerning the condition number of R_n. First of all, it is well known (see Golub and Van Loan [8]) that the condition number of a matrix A is given by $\kappa(A) = ||A|| \, ||A^{-1}||$, where $||A||$ is the norm of A (any suitable matrix norm will do). If $\kappa(A)$ is large, then no algorithm operating under finite precision arithmetic conditions can reasonably be expected to yield an accurate solution to $Ax = y$. A matrix with a large condition number is said to be *ill-conditioned.* In [4] Cybenko only considers 1-norms (he summarizes 2-norm results in [3]). If x is an n-vector then the 1-norm of x is $|| x || = \sum_{i=1}^{n} | x_i |$. If A is an $n \times n$ matrix, then the 1-norm of A is $|| A || = \sup_{|| x || = 1} || Ax ||$. From [4]

$$1 \le || R_n || \le n \ , \tag{3a}$$

and from Theorem 3.1 in [4]

$$\max \left\{ \frac{1}{\prod_{i=1}^{n-1}(1 - K_i^2)} , \frac{1}{\prod_{j=1}^{n-1}(1 - K_j)} \right\} \le || R_n^{-1} || \le \prod_{j=1}^{n-1} \frac{(1 + | K_j |)}{(1 - | K_j |)} \ . \tag{3b}$$

Equations (3a,b) and the definition of condition number readily yield bounds on $\kappa(R_n)$. From (3b) it is clear that $\kappa(R_n)$ is large if any reflection coefficient is "large" (i.e., close to unity in magnitude). Cybenko [4] observes that it is necessary for reflection coefficients to be quite large before the effects of ill-conditioning become very evident. However, this observation was made for rather small n (n on the order of 10).

We shall now summarize the results of Cybenko [4] concerning the numerical stability of the Levinson-Durbin algorithm. Both fixed-point and floating-point arithmetic results are to be found in [4], but we shall only examine the fixed-point case. The fixed-point and floating-point results are qualitatively the same. Let us begin by describing the quantization error model involved in obtaining the results in [4]. For fixed-point arithmetic, the rounding error model is (in the notation and language of [4])

$$fx(a + b) = a + b \ , \ fx(ab) = ab + \xi \ , \ fx(\frac{a}{b}) = \frac{a}{b} + \zeta \ , \tag{4}$$

where a and b are fixed-point numbers, and $fx(\ .\ )$ is the fixed-point representation of the argument. Quantities ξ and ζ are so-called *local rounding errors* satisfying $| \zeta | , | \xi | \le \Delta$, where Δ depends upon the wordlength and method of truncation. Suppose a_n is the true solution (infinite precision solution) to (2). The computed solution will be denoted by $\hat{a}_n = a_n + \alpha_n$, where α_n is the perturbation of the true solution due to the cummulative effects of all of the local rounding errors. Thus,

$$R_n \hat{a}_n = R_n\ (a_n + \alpha_n) = -r_n + \delta_n$$

so

$$R_n \alpha_n = \delta_n\ ,\ or\ \ \alpha_n = R_n^{-1} \delta_n\ . \tag{5}$$

Cybenko [3,4] calls δ_n the *residual vector*. A bound on $|| \ \delta_n \ ||$ (1-norm) may be obtained, and this bound determines the numerical stability of the Levinson-Durbin algorithm. Thus, from Theorem 4.1 in [4],

$$|| \ \delta_n \ || \leq \Delta \left[\prod_{j=1}^{n} (1 + | \ K_j \ |) \right] \left(\frac{n^3}{3} + \frac{3}{2} n^2 + n \right) + O(\Delta^2)\ , \tag{6}$$

where $O(\Delta^2)$ symbolizes the fact that only first order errors are considered in the derivation of (6). Cybenko [4] argues that higher order terms are insignificant, in the course of deriving (6). The bound in (6) is largest when K_i is large (as defined in the previous paragraph). However, despite this, the bound in (6) shows that the Levinson-Durbin algorithm is numerically stable. This is so because, as is argued in [4], the bound in (6) is comparable to that for the Cholesky algorithm (*LU* decomposition, see [8]). The Cholesky algorithm is known to be numerically stable. Thus, the Levinson-Durbin algorithm performs poorly only when it is exposed to ill-conditioned data. In this sense it performs no better or worse than any other method.

A more recent analysis of the Levinson-Durbin algorithm is due to Alexander and Rhee [5], However, their analysis technique is quite different from that of Cybenko [3,4]. Their method is an adaptation of the method commonly used to compute roundoff noise gains and variances in digital filter structures (see Roberts and Mullis [7], Chen [9], or Oppenheim and Schafer [10]). The analyses of the Schur and split Schur algorithms in Zarowski and Card [1,2] also make use of this technique. We shall now briefly summarize the results of Alexander and Rhee [5] concerning the Levinson-Durbin algorithm.

Alexander and Rhee [5] utilize a statistical model for roundoff errors (as is suggested for digital filters in [7,9,10]). If x is an infinite precision variable, and $Q[x]$ is the quantized (finite precision) value of x, then $\hat{x} = Q[x] = x + \eta_x$, where η_x is the error due to quantization. The error is modeled as a random variable (see, for example, Chen [9] (Chapter 11)). From Chapter II, section 3.6 we know that the reflection coefficients parametrize certain digital filter structures, such as the all-pole lattice filter. It is known (see [7]) that if $| K_i | < 1$ for all i then the all-pole, or pole-zero lattice filters will be stable (stability theory for digital filters is discussed in [7,9,10]). Instability results if $| K_i | \geq 1$ for any i. Alexander and Rhee [5] derive formulae that can be used to estimate the value of $Var[\Delta K_i]$ (subject to certain assumptions that will be specified later), the variance of the error ΔK_i in the ith reflection coefficient due to the cummulative effects of all rounding errors. Such results can be used in investigating the stability of lattice filters under finite precision arithmetic conditions, for example. Note that Alexander and Rhee [5] assume the use of fixed-point arithmetic only. The method of [5] takes into account the presence of quantization errors in the normalized autocorrelation sequence $r_1, \cdots, r_n$ (sequence is normalized if $r_0 = 1$). The error due to quantization in r_i is denoted by Δr_i. From [5]

$$Var[\Delta K_2] = \frac{Var[\Delta r_2] + 4r_1^2 Var[\Delta r_1] + \sigma_\eta^2}{(1 - r_1^2)^2}$$

$$= \frac{8r_1^4 + 4r_1^2 + 2r_2^2 + 2}{(1 - r_1^2)^2} \sigma_\eta^2 , \qquad (7)$$

where σ_η^2 is the variance of the quantization error of a single product or quotient. From (7) it is clear that the finite precision estimate of K_2 is likely to be poor if $r_1 \to \pm 1$. It is shown in [5] that this will happen when R_n is due to either a narrowband highpass or lowpass input signal. It is straightforward to show that $K_1 = r_1$ (by the conventions in [3,4,5]), so (7) is large if K_1 is large. In other words, it is ill-conditioned input data that lead to poor estimates of the reflection coefficients, and so the results in [5] actually agree with those in [3,4]. It is worth noting that ill-conditioned data can readily arise in practice. For example, Markel and Gray [11] sometimes observed the effects of ill-conditioning in their experiments with linear predictive (speech signal) vocoders that employ the Levinson-Durbin algorithm.

2. Analysis Assumptions

In this section we present the assumptions behind the analyses of the Schur and split Schur algorithms under fixed-point arithmetic conditions. These results appear in sections 3 and 4 of this chapter. The analysis assumptions are essentially those of Alexander and Rhee [5].

We shall assume (as in [5]) that R_n is an autocorrelation matrix, and that it is positive definite. Alexander and Rhee [5], as we have already noted, take into account quantization errors in the estimates of r_i (normalized autocorrelation coefficients). Thus, their results involve the analysis of a particular autocorrelation sequence estimator under finite precision arithmetic conditions. The estimator that they chose (and that we choose) was

$$R(i) = \frac{1}{L-i} \sum_{k=0}^{L-i-1} s(k)\, s(k+i) \tag{8}$$

for $i = 0,1,\ldots,n$, where $s(k)$ is a real-valued signal. In computing (8), the signal $s(k)$ is windowed, and so we may assume that $s(k)$ is zero for $k < 0$ and $k \geq L$. We will let

$$r_i = \frac{R(i)}{R(0)} \tag{9}$$

for $i = 0,1,\ldots,n$, and so $| r_i | \leq 1$ for all i. Equation (9) is the definition of the *lag i (or ith) normalized autocorrelation coefficient.* Other estimators (besides (8)) could have been chosen. However, the estimator in (8) is desirable in practice as it is unbiased and the variance of the estimates that it produces approach zero as $L \to \infty$, assuming that $s(k)$ is ergodic (see Chen [9], pp. 388-390). Hence, the estimator is statistically consistent (see [9]). Thus, we shall assume that $L \gg n$ to ensure good estimates.

Let x be any infinite precision (nominal) variable. Let $\hat{x}$ be the quantized form of x. Then, as before,

$$\hat{x} = Q[x] = x + \eta_x \ , \tag{10}$$

where η_x is the quantization error. We will assume $Q[\ ]$ to be a roundoff quantizer.

We will also assume that all quantization errors are zero mean, uncorrelated and uniformly distributed over the quantizer bin width. The quantizers themselves are uniform and produce a b-bit binary word for any variable x with the dynamic range $-x_{\max} \leq x \leq x_{\max}$. Note that this assumption will be slightly modified in section 4. Thus, the quantization error variance σ_x^2 is

$$\sigma_x^2 = E[\eta_x^2] = x_{\max}^2 \frac{2^{-2b}}{3} \quad , \tag{11}$$

where $E[\;]$ is the statistical expectation operator. For us, $x_{\max} = 1$ holds (e.g., $| u_k^{(i)} | \leq r_0 = 1$).

Multiplication and division produce roundoff errors that may be modeled as

$$Q[xy] = xy + \eta_p \quad , \tag{12a}$$

$$Q[\frac{x}{y}] = \frac{x}{y} + \eta_q \quad . \tag{12b}$$

We are assuming that the roundoff error is uniformly distributed over the interval $[-2^{-b}\ , 2^{-b}]$ which gives (11) ($x_{\max} = 1$). As in [5], η with a suitable subscript denotes the quantization error of a single product or quotient (local rounding error in the language of [3,4]), and Δx denotes the error in some variable x due to the cummulative effects of quantization error. Second order products such as $\Delta x\ \Delta y$ or $\Delta x\ \eta$ will be ignored as they arise in the computation of expressions for such parameters as ΔK_i since they are small relative to first order errors like Δx or η.

From Alexander and Rhee [5], $\Delta r_i = \hat{r}_i - r_i$ and

$$\Delta r_i = \frac{\eta_{R_i} - r_i \eta_{R_0}}{L\ R(0)} + \eta_{r_i} \tag{13}$$

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

$$E[\eta_{R_i}^2] = [(L - i)\ R(i)]^2 \sigma_\eta^2 \quad , \tag{14a}$$

$$E[\eta_{R_0}^2] = [L\ R(0)]^2 \sigma_\eta^2 \quad , \tag{14b}$$

$$E[\eta_{r_i}^2] = \sigma_\eta^2 \quad , \tag{14c}$$

and $\sigma_\eta^2 = \frac{2^{-2b}}{3}$. Equation (13) uses the assumption that $L \gg n$.

3. Fixed-Point Arithmetic Properties of the Schur Algorithm

In this section we use the model and assumptions of the previous section to analyse the Schur algorithm. The results to follow are from Zarowski and Card [1].

3.1 Analysis

We will begin by presenting some analytical results. For this we need to apply the model of section 2 to the infinite precision form of the Schur algorithm of Chapter II. Thus, according to this model, the Schur algorithm becomes:

```
$\hat{u}_{-k}^{(1)} := \hat{r}_k$ ; $\hat{u}_k^{(1)} := \hat{r}_k$ ;
For $i$ := 1 to $n$ do begin
    $\hat{K}_i := Q[-\hat{u}_1^{(i)} / \hat{u}_{-i+1}^{(i)}]$ ;
    For $k$ := 0 to $n - i$ do begin
        $\hat{u}_{-(k+i)}^{(i+1)} := \hat{u}_{-k-i+1}^{(i)} + Q[\hat{K}_i \hat{u}_{k+1}^{(i)}]$ ;
        $\hat{u}_k^{(i+1)} := Q[\hat{K}_i \hat{u}_{-k-i+1}^{(i)}] + \hat{u}_{k+1}^{(i)}$ ;
    end;
end;
```

This is the *finite precision Schur algorithm.*

It is now possible to obtain error expressions for the reflection coefficients K_i and the Schur variables $u_k^{(i)}$. Thus, from the finite precision Schur algorithm it is easy to see that

$$\hat{K}_i = -\frac{u_1^{(i)} + \Delta u_1^{(i)}}{u_{-i+1}^{(i)} + \Delta u_{-i+1}^{(i)}} + \eta_{K,i} , \tag{15}$$

and so

$$\Delta K_i = \hat{K}_i - K_i = -\frac{\Delta u_1^{(i)} + K_i \Delta u_{-i+1}^{(i)}}{u_{-i+1}^{(i)}} + \eta_{K,i} . \tag{16}$$

Similarly,

$$\hat{u}_{-(k+i)}^{(i+1)} = u_{-(k+i)}^{(i+1)} + \Delta u_{-k-i+1}^{(i)} + K_i \Delta u_{k+1}^{(i)} + \Delta K_i u_{k+1}^{(i)} + \eta_{u_{-(k+i)},i} , \qquad (17a)$$

$$\hat{u}_{k}^{(i+1)} = u_{k}^{(i+1)} + \Delta u_{k+1}^{(i)} + K_i \Delta u_{-k-i+1}^{(i)} + \Delta K_i u_{-k-i+1}^{(i)} + \eta_{u_k,i} . \qquad (17b)$$

From (17a,b) we get

$$\Delta u_{-(k+i)}^{(i+1)} = \Delta u_{-k-i+1}^{(i)} + K_i \Delta u_{k+1}^{(i)} + \Delta K_i u_{k+1}^{(i)} + \eta_{u_{-(k+i)},i} , \qquad (18a)$$

$$\Delta u_{k}^{(i+1)} = \Delta u_{k+1}^{(i)} + K_i \Delta u_{-k-i+1}^{(i)} + \Delta K_i u_{-k-i+1}^{(i)} + \eta_{u_k,i} . \qquad (18b)$$

We can use (16), and (18a,b) in combination with the infinite precision Schur algorithm of Chapter II (which gives us the nominal parameters K_i and $u_k^{(i)}$) to get closed-form expressions for ΔK_i. This is practical only for small i, since for large i the expressions are extremely unwieldy and hard to obtain because of the large effort involved.

Let us first consider the case of $i = 1$. From (16) it is clear that

$$\Delta K_1 = -\Delta r_1 + \eta_{K,1} . \qquad (19)$$

We can compute an estimate of the variance of this error, and the result is

$$Var[\Delta K_1] = Var[\Delta r_1] + \sigma_\eta^2 .$$

From [5], $Var[\Delta r_1] = [1 + 2r_1^2]\sigma_\eta^2$ (use (13) and (14a,b,c)), and so

$$Var[\Delta K_1] = 2[1 + r_1^2]\sigma_\eta^2 . \qquad (20)$$

From (19), if $\Delta r_1 = 0$, then $Var[\Delta K_1] = \sigma_\eta^2$. Thus, much of the error in our finite precision arithmetic estimate of K_1 is due to error in the estimate of r_1.

We can repeat the above analysis for the special case of $i = 2$. Once again, from (16), and with the aid of (18a,b),

$$\Delta K_2 = \frac{\Delta r_2}{r_1^2 - 1} - \frac{2r_1[r_2 - 1]\Delta r_1}{[r_1^2 - 1]^2} \qquad (21)$$
$$+ \frac{r_1[r_2 - 1]}{[r_1^2 - 1]^2}\eta_{K,1} + \frac{\eta_{u_1,1}}{r_1^2 - 1} + \frac{r_2 - r_1^2}{[r_1^2 - 1]^2}\eta_{u_{-1},1} + \eta_{K,2} .$$

The variance of ΔK_2 will consist of two parts denoted by P_1 and P_2 (so that

$Var[\Delta K_2] = P_1 + P_2$). The first part will involve the first two terms of (21), and will give the variance due to errors in the estimation of r_1 and r_2. The second part involves the last four terms of (21), and gives the error due to the quantization of products and quotients in the course of executing the finite precision Schur algorithm. From [5], $Var[\Delta r_2] = [1 + 2r_2^2]\sigma_\eta^2$, and so

$$P_1 = \frac{[1 + 2r_2^2][r_1^2 - 1]^2 + 4r_1^2[r_2 - 1]^2[1 + 2r_1^2]}{[r_1^2 - 1]^4}\sigma_\eta^2 , \qquad (22a)$$

$$P_2 = \frac{r_1^2[r_2 - 1]^2 + [r_2 - r_1^2]^2 + [r_1^2 - 1]^4 + [r_1^2 - 1]^2}{[r_1^2 - 1]^4}\sigma_\eta^2 . \qquad (22b)$$

For a second order AR (autoregressive) process with poles at $z = \rho e^{\pm j\theta}$, it is straightforward to show that

$$r_1 = \frac{2\rho\cos\theta}{1 + \rho^2} , \quad r_2 = \frac{\rho^2(\sin 3\theta - \rho^2\sin\theta)}{(1 + \rho^2)\sin\theta} . \qquad (23)$$

Such a process may be generated by passing white noise (zero-mean) through an all-pole filter with the said pole locations. Note that the filter must be stable, and so $0 \leq \rho < 1$. Equations (23) is obtained via the method described in Chen [9] (see Chapter 10, pp. 346-353). If $\theta \to 0$, and $\rho \to 1$, then $r_1 \to 1$, and $r_2 \to 1$. A signal such as this is a narrowband lowpass signal. Similarly, if $\theta \to \pi$, and $\rho \to 1$, then $r_1 \to -1$, and $r_2 \to 1$, and the result is a narrowband highpass signal. Either type of signal will lead to a large value for $Var[\Delta K_2]$, implying a poor estimate of K_i for $i \geq 2$. This may be readily seen by considering (22a,b).

3.2 Discussion and Simulation

We will now present some simulation results. These results confirm the validity of the previous analysis. We will also discuss how the present results concerning the Schur algorithm relate to the Levinson-Durbin results found in [3,4,5].

We will begin by describing the method of simulation. A series of 2nd order AR signals were constructed by passing zero-mean, white Gaussian noise through a 2nd order all-pole filter with poles at $z = \rho e^{\pm j\theta}$. In all of the experiments performed, ten

1000-point signals were constructed for various values of ρ and θ. Floating-point arithmetic was used to construct the test signals, and to compute the nominal values of the normalized autocorrelation coefficients. Floating-point arithmetic was then used to compute the nominal reflection coefficient values. The nominal values are taken to be the "infinite precision" values. Quantized normalized autocorrelation coefficients were produced by rounding the nominal normalized autocorrelation coefficients to b-bit 2s complement numbers. The resulting quantized coefficients were used to obtain the fixed-point reflection coefficients. Appendix A contains the C program used to perform the simulations (on the Data General Eclipse MV/8000 computer - all simulations in this thesis were performed on this machine), along with a page of typical program output. Note however that the reflection coefficients are indexed differently in the program output from the indexing that we have used so far. Specifically, $K(i)$ in the program output is K_{i-1} in the notation above. The experimentally derived variance estimates were obtained by squaring the difference between the nominal and fixed-point reflection coefficient estimates and averaging over the number of experiments (ten in this case). Appendix B contains tabular summaries of various experiments with lowpass signals (highpass results omitted).

Because of the manner in which quantized autocorrelation coefficients were obtained, equation (22a) must be modified. Specifically, $Var[\Delta r_1] = Var[\Delta r_2] = \sigma_\eta^2$ now. Thus,

$$Var[\Delta K_2] = P'_1 + P_2 \ , \tag{24}$$

where

$$P'_1 = \frac{[r_1^2 - 1]^2 + 4r_1^2[r_2 - 1]^2}{[r_1^2 - 1]^4} \sigma_\eta^2 \ . \tag{25}$$

Hence, equation (24) becomes the expression for the "theoretical variance" of the 2nd reflection coefficient. The simulation results are compared with the results provided by (24) in the tables of Appendix B. Note that the nominal normalized autocorrelation coefficients needed by (24) are obtained using (23).

Each of Tables I,II and III in Appendix B represents simulation results for a particular combination of ρ and θ. In general, there is good agreement between the theoretical predictions and the experimental (simulation) results in the sense that the discrepancies seem to decrease as b increases. This is reasonable since as b increases the assumption of uncorrelated quantization errors becomes more accurate. A comparison of the results in all of the tables confirms that the reflection coefficient error variance does indeed increase as $\rho \to 1$ and $\theta \to 0$.

If we compare the expressions for $Var[\Delta K_2]$ due to the Schur algorithm with the similar expression for the Levinson-Durbin algorithm (see (7)), we conclude that the two algorithms are qualitatively the same in terms of their finite precision arithmetic behaviour. In other words, the Schur algorithm is numerically stable. However, the expressions for $Var[\Delta K_2]$ in (24) and (22a,b), not surprisingly, indicate once again that narrowband lowpass and highpass signals yield ill-conditioned autocorrelation matrices. This of course agrees with Cybenko's results in [3,4]. As a result, the Schur algorithm performs no worse than any other algorithm for LDU factorization (such as Cholesky's algorithm) when exposed to ill-conditioned input data.

4. Fixed-Point Arithmetic Properties of the Split Schur Algorithms

As we did in section 3, in this section we shall use the model and assumptions of section 2 to analyse the split Schur algorithms (symmetric and antisymmetric forms) with real, symmetric, positive definite autocorrelation matrices as input, under fixed-point arithmetic conditions. The results to follow are from Zarowski and Card [2].

4.1 Analysis

We will again begin by presenting analytical results. As in section 3.1, application of the model of section 2 to the infinite precision symmetric split Schur algorithm yields the finite precision form:

$$\hat{v}_{0,0} := \hat{r}_0 ;$$

$$\hat{K}_0 := 0 ;$$

$$\hat{v}_{0,j} := 2\hat{r}_j \quad (1 \le j \le n) ;$$

```
$\hat{v}_{1,j} := \hat{r}_j + \hat{r}_{j+1}\ \ (0 \le j \le n-1)$ ;
For $k$ := 1 to $n$ do begin
    $\hat{\alpha}_k := Q[\hat{v}_{k,0}/\hat{v}_{k-1,0}]$ ;
    $\hat{K}_k := 1 - Q[\hat{\alpha}_k/(1+\hat{K}_{k-1})]$ ;
    For $j$ := 0 to $n-k-1$ do begin
        $\hat{v}_{k+1,j} := \hat{v}_{k,j} + \hat{v}_{k,j+1} - Q[\hat{\alpha}_k \hat{v}_{k-1,j+1}]$ ;
        end;
    end;
```

This is the *finite precision symmetric split Schur algorithm.* Similarly, the antisymmetric split Schur algorithm has the finite precision form:

```
$\hat{v}^*_{0,0} := \hat{r}_0$ ;
$\hat{K}_0 := 0$ ;
$\hat{v}^*_{0,j} := 0\ \ (1 \le j \le n)$ ;
$\hat{v}^*_{1,j} := \hat{r}_j - \hat{r}_{j+1}\ \ (0 \le j \le n-1)$ ;
For $k$ := 1 to $n$ do begin
    $\hat{\alpha}^*_k := Q[\hat{v}^*_{k,0}/\hat{v}^*_{k-1,0}]$ ;
    $\hat{K}_k := -1 + Q[\hat{\alpha}^*_k/(1-\hat{K}_{k-1})]$ ;
    For $j$ := 0 to $n-k-1$ do begin
        $\hat{v}^*_{k+1,j} := \hat{v}^*_{k,j} + \hat{v}^*_{k,j+1} - Q[\hat{\alpha}^*_k \hat{v}^*_{k-1,j+1}]$ ;
        end;
    end;
```

This is the *finite precision antisymmetric split Schur algorithm.*

We will now derive analytical results for the symmetric split Schur algorithm. Naturally, we need iterative expressions for the errors due to quantization of the variables α_k, $v_{k,j}$ and K_k. These errors are denoted by $\Delta\alpha_k$, ΔK_k and $\Delta v_{k,j}$, in conformity with the notational conventions of section 2. We shall determine closed-form expressions for $Var[\Delta\alpha_1]$ and $Var[\Delta K_1]$. Closed-form expressions for $\Delta\alpha_2$ and ΔK_2 shall also be found, from which expressions for the variances of these errors may be readily determined.

From the error model of section 2, and from the finite precision symmetric split Schur algorithm

$$\Delta\alpha_k = \hat{\alpha}_k - \alpha_k = \frac{v_{k,0} + \Delta v_{k,0}}{v_{k-1,0} + \Delta v_{k-1,0}} - \frac{v_{k,0}}{v_{k-1,0}} + \eta_{\alpha,k}$$

$$= \frac{\Delta v_{k,0} - \alpha_k \Delta v_{k-1,0}}{v_{k-1,0}} + \eta_{\alpha,k} \qquad (26)$$

for $k = 1,\ldots,n$, where we have assumed that $v_{k-1,0}$ is much larger than $\Delta v_{k-1,0}$. Similarly, it is easy to show that

$$\Delta K_k = \frac{(1-K_k)\Delta K_{k-1} - \Delta\alpha_k}{1 + K_{k-1}} + \eta_{K,k} \qquad (27)$$

for $k = 1,\ldots,n$. We have assumed that $1 + K_{k-1}$ is much larger than ΔK_{k-1} in the derivation of (27). Finally,

$$\Delta v_{k+1,j} = \Delta v_{k,j} + \Delta v_{k,j+1} - \alpha_k \Delta v_{k-1,j+1} - v_{k-1,j+1}\Delta\alpha_k + \eta_{v_{k+1},j} \ , \qquad (28)$$

where $k = 1,\ldots,n$ and $j = 0,1,\ldots,n-k-1$.

From the initiallization phase of the finite precision symmetric split Schur algorithm,

$$\Delta K_0 = 0 \ , \ \Delta v_{0,0} = 0 \ , \ \Delta v_{1,0} = \Delta r_1 \ .$$

As well, $K_0 = 0$ and $v_{0,0} = 1$. Thus, for $k = 1$ we may use (26) to obtain

$$\Delta\alpha_1 = \frac{\Delta v_{1,0} - \alpha_1 \Delta v_{0,0}}{v_{0,0}} + \eta_{\alpha,1} = \Delta r_1 + \eta_{\alpha,1} \ , \qquad (29)$$

where $\alpha_1 = 1 + r_1$. Similarly, via (27),

$$\Delta K_1 = -\Delta\alpha_1 + \eta_{K,1} = -\Delta r_1 - \eta_{\alpha,1} + \eta_{K,1} \ , \qquad (30)$$

where $K_1 = -r_1$. Using the fact that

$$Var[\Delta r_i] = [1 + 2{r_i}^2]\ \sigma_\eta^2 \ , \qquad (31)$$

(see [5]), we have

$$Var[\Delta\alpha_1] = Var[\Delta r_1] + \sigma_\eta^2 = 2(1 + r_1^2)\sigma_\eta^2 \ , \tag{32a}$$

$$Var[\Delta K_1] = Var[\Delta r_1] + 2\sigma_\eta^2 = (3 + 2r_1^2)\sigma_\eta^2 \ . \tag{32b}$$

Clearly, much of the error in the estimation of α_1 and K_1 is due to error in the estimate of the normalized autocorrelation coefficient r_1. This is especially true when r_1 is close to ± 1.

Now let us consider the case $k = 2$. Using (28)

$$\Delta v_{2,0} = -4r_1\Delta r_1 + \Delta r_2 - 2r_1\eta_{\alpha,1} + \eta_{v_{2,0}} \ , \tag{33}$$

where we have used $\Delta v_{1,1} = \Delta r_1 + \Delta r_2$ and $\Delta v_{0,1} = 2\Delta r_1$ ($v_{0,1} = 2r_1$) which follow from the initiallization parts of the finite precision symmetric split Schur algorithm. Therefore, substituting (33) into (26) and simplifying gives

$$D_1\Delta\alpha_2 = -[2r_1^2 + 4r_1 + r_2 + 1]\Delta r_1 + (1 + r_1)\Delta r_2 \tag{34}$$
$$- 2r_1(1 + r_1)\eta_{\alpha,1} + (1 + r_1)^2\eta_{\alpha,2} + (1 + r_1)\eta_{v_{2,0}} \ ,$$

where $D_1 = (1 + r_1)^2$. An expression for $Var[\Delta\alpha_2]$ may now be obtained, but for brevity we will not state it explicitly. It is enough to see from (34) that

$$Var[\Delta\alpha_2] \to \infty \text{ if } r_1 \to -1 \ .$$

For a second order AR process with poles at $z = \rho e^{\pm j\theta}$, and such that $\rho \to 1$ and $\theta \to \pi$, then $r_1 \to -1$ (as we know from section 3 - see (23)). Hence, if the input to the finite precision symmetric split Schur algorithm is from a narrowband highpass process, then large errors in the estimates of α_k ($k \geq 2$) can be expected.

From (27) with $k = 2$, substituting in the appropriate values, and simplifying gives

$$D_2\Delta K_2 = [(1 - r_1)(2r_1^2 - r_2 - 1)(1 + r_1)^2 + (1 - r_1^2)(2r_1^2 + 4r_1 + r_2 + 1)]\Delta r_1$$
$$- (1 + r_1)(1 - r_1^2)\Delta r_2$$
$$+ [(1 - r_1)(2r_1^2 - r_2 - 1)(1 + r_1)^2 + 2r_1(1 + r_1)(1 - r_1^2)]\eta_{\alpha,1}$$
$$- (1 + r_1)^2(1 - r_1^2)\eta_{\alpha,2} \tag{35}$$

$$+ (1 - r_1)(1 + r_2 - 2r_1^2)(1 + r_1)^2\eta_{K,1}$$

$$+ (1 - r_1)(1 - r_1^2)(1 + r_1)^2\eta_{K,2}$$

$$- (1 + r_1)(1 - r_1^2)\eta_{v_{2,0}} \ ,$$

where $D_2 = (1 - r_1)(1 - r_1^2)(1 + r_1)^2$. Again, it is easy (though tedious) to obtain an expression for $Var[\Delta K_2]$, but for brevity we shall once again decline to state it explicitly. From (35)

$$Var[\Delta K_2] \to \infty \text{ if } r_1 \to \pm 1 \ .$$

Thus, $Var[\Delta K_2] \to \infty$ if the input is from either a narrowband highpass or lowpass process. Obviously, we can expect large errors in the estimates of K_k for $k > 2$ for such signals as well. This is so since ΔK_k is a function of ΔK_{k-1}.

We will simply state the results for the antisymmetric split Schur algorithm as the process of obtaining them is identical to that used to obtain the above results for the symmetric split Schur algorithm. Thus, the expressions analogous to (26), (27) and (28) are

$$\Delta\alpha_k^* = \frac{\Delta v_{k,0}^* - \alpha_k^*\Delta v_{k-1,0}^*}{v_{k-1,0}^*} + \eta_{\alpha^*,k} \ , \tag{36a}$$

$$\Delta K_k = \frac{(1 + K_k)\Delta K_{k-1} + \Delta\alpha_k^*}{1 - K_{k-1}} + \eta_{K,k} \ , \tag{36b}$$

$$\Delta v_{k+1,j}^* = \Delta v_{k,j}^* + \Delta v_{k,j+1}^* - \alpha_k^*\Delta v_{k-1,j+1}^* - v_{k-1,j+1}^*\Delta\alpha_k^* + \eta_{v_{k+1,j}^*} \ , \tag{36c}$$

where $k = 1,...,n$ and $j = 0,1,...,n-k-1$. Using (36a,b,c) it can be shown that

$$\Delta\alpha_1^* = -\Delta r_1 + \eta_{\alpha^*,1} \ , \tag{37a}$$

$$\Delta K_1 = -\Delta r_1 + \eta_{\alpha^*,1} + \eta_{K,1} \ , \tag{37b}$$

$$D_3\Delta\alpha_2^* = (1 - r_2)\Delta r_1 - (1 - r_1)\Delta r_2 + (1 - r_1)^2\eta_{\alpha^*,2} + (1 - r_1)\eta_{v_{2,0}^*} \ , \tag{37c}$$

$$D_4\Delta K_2 = 2r_1(1 - r_1)(1 - r_2)\Delta r_1 - (1 - r_1)(1 - r_1^2)\Delta r_2$$

$$+ (1 - r_2)(1 - r_1)^2\eta_{\alpha^*,1} + (1 - r_1)^2(1 - r_1^2)\eta_{\alpha^*,2}$$

$$+ (1 - r_2)(1 - r_1)^2\eta_{K,1} + (1 - r_1^2)(1 - r_1)^2(1 + r_1)\eta_{K,2} \tag{37d}$$
$$+ (1 - r_1)(1 - r_1^2)\eta_{v_{2,0}^*} \quad ,$$

where $D_3 = (1 - r_1)^2$ and $D_4 = (1 - r_1^2)(1 - r_1)^2(1 + r_1)$.

From (37c) it is evident that

$$Var[\Delta\alpha_2^*] \rightarrow \infty \text{ if } r_1 \rightarrow 1 \quad ,$$

and this will happen if the input data are from a narrowband lowpass process. This is in contrast with the results for $Var[\Delta\alpha_2]$ since $Var[\Delta\alpha_2] \rightarrow \infty$ for a narrowband highpass process. As well,

$$Var[\Delta K_2] \rightarrow \infty \text{ if } r_1 \rightarrow \pm 1 \quad ,$$

and so errors in the estimation of K_2 (and of K_k, $k > 2$) will be large if the input data are from a narrowband highpass or lowpass process. This result is qualitatively the same as the result for $Var[\Delta K_2]$ in the case of the symmetric split Schur algorithm.

4.2 Discussion and Simulation

We will present simulation results for the finite precision symmetric split Schur algorithm. These results will confirm the validity of the analysis in section 4.1. We will also discuss how the present results relate to those for the Schur algorithm in section 3.

Because the variables α_k, and $v_{k,j}$ in general possess a nonzero integer part, the simulations which follow use twos complement binary words of the form

$$x_{-m} \cdots x_{-1}x_0x_1 \cdots x_{b-1} \quad , \tag{38}$$

where $x_i \in \{0,1\}$, and x_{-m} is the sign bit. Thus $Q[\;]$ quantizes nominal values down to $m+b$-bit numbers (in sections 2 and 3 we had $m = 0$). The integer parts of all products and quotients shall of course be retained. However, fractional parts will be quantized by rounding to $b-1$ bits. Hence, $\sigma_\eta^2 = \frac{2^{-2b}}{3}$, just as it was in sections 2 and 3. In all of the simulations that follow we pick $m = 2$. Thus, the total wordlength is $b+2$ bits with $b-1$ fractional bits.

In one set of simulations, a series of second order AR signals were constructed by passing zero-mean, white Gaussian noise through a second order all-pole filter with poles at $z = \rho e^{\pm j\theta}$. In all of the experiments, ten 1000-point signals were constructed for each of the various sets of ρ and θ. Floating-point arithmetic was used to construct the test signals and to compute the nominal normalized autocorrelation coefficients. Floating-point arithmetic was then used to compute the nominal reflection coefficient values, since we are interested specifically in comparing the experimentally measured values of $Var[\Delta K_2]$ with the theoretically predicted values. All nominal values derived via floating-point arithmetic were taken to be "infinite precision" values. Quantized normalized autocorrelation coefficients were produced by rounding the nominal normalized autocorrelation coefficients to b+2-bit twos complement numbers of the form in (38). Thus, $Var[\Delta r_k] = \sigma_\eta^2$ for $k \geq 1$. This condition also held for the second set of simulation experiments to be described below. The resulting quantized coefficients were used to obtain the fixed-point reflection coefficients. The C program used to produce the theoretical and experimental results is to be found in Appendix C. Typical program output is included. Appendix D contains Tables I to VI which summarize the results of certain experiments using this program. These results test the validity of the expression for ΔK_2 (which we know yields $Var[\Delta K_2]$) in (35). In general, the agreement between theory and experiment is good.

The reader may be disturbed by the fact that we have not presented closed-form expressions for ΔK_k when $k > 2$. This is simply due to the fact that such expressions are difficult to derive (note the complexity of (35)). Fortunately, it is not necessary to derive such expressions which would, if they were available, be useful as design equations (for the selection of a suitable b). It is possible to use *recursive programming* to "implement" the equations (26-28) in the form of software. Such a program has been written in PASCAL, and it may be found in Appendix E. Roughly speaking, a tree data structure is created (via dynamic allocation) such that each node in the tree (a PASCAL record type) contains the terms of the equations (26-28). Pointers from one node to the next symbolize the *individual* terms in (26-28). A suitable traversal of the tree may then be used to "collect like terms" in the variables Δr_k , $\eta_{\alpha,k}$, $\eta_{K,k}$ and

$\eta_{\nu_{k+1,j}}$. The program requires the nominal normalized autocorrelation coefficients as input. These may be produced by another PASCAL program in Appendix F (for fourth order AR models).

A second set of experiments was performed using fourth order AR signals constructed by passing zero-mean, white Gaussian noise through a 4th order all-pole filter with poles $\rho_1 e^{\pm j\theta_1}$ and $\rho_2 e^{\pm j\theta_2}$. The theoretical variance estimates ($Var[\Delta K_k]$) are obtained using the nominal normalized autocorrelation coefficients produced by the program in Appendix F as input to the program in Appendix E. The C program in Appendix G computes the experimental variance estimates. Typical output from this program is to be found in Appendix G. Results for various experiments with the programs of Appendix E and G are to be found in the form of Tables VII to IX in Appendix H. Once again, in all of the simulations, ten 1000-point signals were constructed for each set of ρ_1 , ρ_2 , θ_1 , θ_2 and b stated in order to produce the entries in the experimental columns of the tables. As before, the agreement between theory and experiment is reasonably good.

It is worth noting the possible sources of the discrepancies between the experimental and theoretical results. We have the following list of possibilities:

(i) The assumption of uncorrelated errors is clearly not completely valid, but it is necessary for reasons of analytic tractability.

(ii) Only ten 1000-point sequences were used to generate each experimental column entry in the tables.

(iii) The values of b used in the experiments may be too small for the assumption of uncorrelated errors to hold very accurately.

(iv) We have neglected any second order error terms that may have arisen in the course of deriving expressions for ΔK_k.

(v) The pseudorandom number generator is not perfectly random, and this could cause some deviation between the theoretical predictions and experimental results, since there may be hidden periodicities in the pseudorandom sequence.

(vi) The use of floating-point estimates to represent infinite precision values carries some risk (but is expedient).

Points (iv-vi) are likely to be rather insignificant. It is the first three points that no doubt matter the most. The above six possible discrepancy sources also apply to the results in section 3 of course.

It is clear from the results of this section that the symmetric split Schur algorithm is qualitatively similar to the Schur algorithm under fixed-point arithmetic implementation conditions. That is, the symmetric split Schur algorithm is numerically stable, and any poor estimates of K_k are due to the use of ill-conditioned input data. Similar conclusions can safely be reached concerning the antisymmetric split Schur algorithm.

We may finally conclude that the Levinson-Durbin, Schur, and split Schur algorithms are all numerically stable, and that poor reflection coefficient estimates are due to ill-conditioned inputs.

REFERENCES

[1] C. J. Zarowski, H. C. Card, "Finite Precision Arithmetic and the Schur Algorithm," submitted to the IEEE Trans. on Acoust., Speech, and Signal Proc.

[2] C. J. Zarowski, H. C. Card, "Finite Precision Arithmetic and the Split Schur Algorithms," submitted to the IEEE Trans. on Acoust., Speech, and Signal Proc.

[3] G. Cybenko, "Round-off Error Properties in Durbin's, Levinson's, and Trench's Algorithms," Proc. 1979 Int. Conf. on Acoust., Speech, and Signal Proc., Washington, D.C., April 2-4, 1979, pp. 498-501.

[4] G. Cybenko, "The Numerical Stability of the Levinson-Durbin Algorithm for Toeplitz Systems of Equations," SIAM J. Sci. Stat. Comp., vol. 1, Sept. 1980, pp. 303-319.

[5] S. T. Alexander, Z. M. Rhee, "Analytical Finite Precision Results for Burg's Algorithm and the Autocorrelation Method for Linear Prediction," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, May 1987, pp. 626-635.

[6] J. Le Roux, C. Gueguen, "A Fixed Point Computation of Partial Correlation Coefficients," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-25, Jan. 1977, pp. 257-259.

[7] R. A. Roberts, C. T. Mullis, *Digital Signal Processing*. Reading, Massachusetts: Addison-Wesley, 1987.

[8] G. H. Golub, C. F. Van Loan, *Matrix Computations*. Baltimore, Maryland: Johns Hopkins University Press, 1983.

[9] C.-T. Chen, *One-Dimensional Digital Signal Processing*. New York, New York: Marcel Dekker, 1979.

[10] A. V. Oppenheim, R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[11] J. D. Markel, A. H. Gray, Jr., "Fixed-Point Truncation Arithmetic Implementation of a Linear Prediction Autocorrelation Vocoder," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-22, Aug. 1974, pp. 273-282.

Chapter VI

THE QUADRATIC RESIDUE NUMBER SYSTEM, FAREY FRACTIONS, AND HENSEL CODES

As we know, some problem instances are ill-conditioned (e.g., narrowband high and lowpass signals give ill-conditioned autocorrelation matrices), and some algorithms are numerically unstable. It is desirable to consider the solution of such problems using arithmetic in finite number systems that enable the computation to proceed free of quantization errors. This is the meaning of *error-free computation.* While the Schur algorithm is numerically stable, it will give poor estimates of Schur variables and reflection coefficients when the input is ill-conditioned (see Chapter V). The Schur algorithm for Hermitian Toeplitz matrices of any rank profile (Chapter III) requires testing for zero, a risky operation to implement with any form of finite precision arithmetic (be it fixed-point or floating-point). Thus, such algorithms are candidates for error-free computation implementation. There are three principal means of performing error-free computations: (i) by rational arithmetic,. (ii) by Hensel code arithmetic, and (iii) by computing in finite rings and fields. In this chapter we shall show why we reject options (i) and (ii), thus leaving us with computation in finite rings and fields as our only option. Our rejection of option (ii) is based upon results from Zarowski and Card [1]. We shall review the concept of a quadratic residue number system (QRNS), a well-known means of performing error-free computation with complex-valued data in an efficient way. We shall straightforwardly extend the usual QRNS to accommodate fractional (meaning non-integer) data, as in Zarowski and Card [2].

1. Rational Arithmetic

In this form of error-free computation, all numbers are represented in the form $\frac{a}{b}$ ($a, b \in Z$, the set of integers). Thus, the addition and multiplication of such numbers proceeds as follows:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} \, , \quad \frac{a}{b} \, \frac{c}{d} = \frac{ac}{bd} \, .$$

Similar reasoning applies to subtraction and division. It is clear that rational arithmetic is very inefficient as, for example, addition requires three integer multiplications. Furthermore, as we do not quantize results, numbers can grow in size at a geometric rate. Note that this problem is common to all error-free computation methods. However, the growth in size of rational numbers is likely to be particularly great, since a result $\frac{a}{b}$ may be of the form

$$\frac{a}{b} = \frac{a_1 k}{b_1 k} = \frac{a_1}{b_1} \, .$$

That is, the numerator and denominator may contain common factors (i.e., $gcd(a,b) \neq 1$). This means that a and b may be unnecessarily large. Eliminating common factors is computationally laborious, as it involves employing Euclid's algorithm to find k. Thus, we can readily reject rational arithmetic as a viable means of performing error-free computation.

2. Hensel Codes

The Hensel codes are defined in Krishnamurthy, Rao and Subramanian [3], and in Gregory and Krishnamurthy [4]. As well, the basic operations of addition, subtraction (via negation), multiplication and division of Hensel codes is also defined. The Hensel codes of [3,4] are in one-to-one correspondence with a certain finite subset of the rational numbers Q, and it is intended that arithmetic with Hensel codes should correspond to arithmetic with the numbers in this subset. Let this subset of the rationals be denoted by $F_N \cup X$. F_N is the set of order-N Farey fractions, and X is the set of invalid order-N Farey fractions which is a finite subset of Q (see section 2.1 below). Let H denote the set of Hensel codes corresponding to the members of F_N, and let H_X denote the set of Hensel codes corresponding to the members of X. Then $H \cup H_X$ denotes the set of Hensel codes corresponding to the set $F_N \cup X$. Gorgui-Naguib and King [5] have shown that addition and multiplication of Hensel codes in $H \cup H_X$, using the arithmetic in [3,4], does not always correspond to addition and multiplication in $F_N \cup X$. Specifically, let $a, b \in F_N \cup X$, and let $H(p,r,a)$, $H(p,r,b) \in H \cup H_X$, then even if $a+b$, $ab \in F_N \cup X$, it will not necessarily be the case that $H(p,r,a)+H(p,r,b)$, $H(p,r,a)H(p,r,b) \in H \cup H_X$. We show that this difficulty never arises provided that the input data and final results of a computation with those inputs lies entirely within the set F_N.

Gorgui-Naguib and King [5] have also shown how to modify the operations of addition and multiplication on $H \cup H_X$ originally defined in [3,4] in order that the correct results are produced. However, their method involves mapping operands in $H \cup H_X$ back to the set $F_N \cup X$ when it is discovered that an incorrect Hensel code will be produced by a sum or product of those operands. The operands in $F_N \cup X$ are then mapped back to $H \cup H_X$, but this time r (the number of Hensel code digits) is larger than before. It is clear that this mapping back and forth precludes a special purpose hardware implementation of the methods in [5], although a software implementation may be more practical. It is worth noting that in many practical cases it is not a serious limitation if the inputs and final outputs are restricted to the set F_N (see [4]). This is particularly true if the only operations involved are addition, subtraction and multiplication. Furthermore, the hardware cost of such a restriction will often be minimal compared with the cost of implementing the schemes in [5].

We also demonstrate that the concept of Hensel codes becomes redundant when we restrict ourselves to using operands from the set F_N alone. This is because of the fact that there is a finite ring of Hensel codes containing H such that this ring is isomorphic to the ring $Z_{p^r} = \{0,1,\ldots,p^r-1\}$ under modulo p^r addition and multiplication.

Thus, we wish to find an alternative method of reintroducing the set X as operands (and final results), that precludes the need to map back and forth between $H \cup H_X$ and $F_N \cup X$ as is done in [5]. This is the final objective accomplished in this section. However, it turns out that in order to avoid mapping back and forth, very long Hensel code words will be required in general. As a result, our approach is probably no more practical than that of [5]. It does however provide an alternate perspective on the problem.

2.1 The order-N Farey Fractions and Ring Z_{p^r}

Here we define the order-N Farey fractions F_N, the invalid order-N Farey fractions X, and their relationship with the finite ring Z_{p^r} .

Let Z be the set of integers, and let p be any prime positive integer. Let Q be the set of rational numbers. Define the following subset of Q:

$$\hat{Q} = \left\{ \frac{a}{b} \mid gcd(b,p) = 1 \right\} .$$

It is easily demonstrated that $\hat{Q}$ is a commutative ring with identity under the usual definition of addition and multiplication that it inherits from the field of rationals Q (see [4], page 25). Set $Z_{p^r} = \{0,1,...,p^r-1\}$ forms a finite ring under modulo p^r addition and multiplication. If $x \in Z$, then let $| \, x \, |_{p^r}$ denote the modulo p^r reduction of x: $| \cdot |_{p^r} \mid Z \to Z_{p^r}$. We may extend the mapping $| \cdot |_{p^r}$ in the following manner. Let $\frac{a}{b} \in \hat{Q}$, then

$$\left| \frac{a}{b} \right|_{p^r} = | \, ab^{-1} \, |_{p^r} \ ,$$

where b^{-1} (modulo p^r) exists since b is mutually prime to p. It may be readily proven that $| \cdot |_{p^r} \mid \hat{Q} \to Z_{p^r}$ is a ring homomorphism (see [4], pp. 25-26). Let

$$Q_k = \left\{ \frac{a}{b} \in \hat{Q} \mid \left| \frac{a}{b} \right|_{p^r} = k \right\} ,$$

and so clearly $\hat{Q} = \bigcup_{k=0}^{p^r-1} Q_k$. Each Q_k is called a *generalized residue class*. Ideally, we would like to establish a one-to-one correspondence between a single representative from each Q_k and a member of Z_{p^r}. Unfortunately, we can identify such a unique element in only some of the generalized residue classes, but not all of them. These elements are specified by

Definition 1 ∘ The finite subset of $\hat{Q}$

$$F_N = \left\{ \frac{a}{b} \in \hat{Q} \mid gcd(a,b) = 1 \,,\, 0 \le | \, a \, | \le N \,,\, 0 < | \, b \, | \le N \right\}$$

where $N > 0$ is an integer, is called the set of *order-N Farey fractions*.

This is Definition 5.13 of [4] (see page 27). Also from [4] we have

Theorem 1 ∘ Let N be the largest integer satisfying the inequality

$$2N^2 + 1 \le p^r$$

and let the generalized residue class Q_k contain the order-N Farey fraction $x = a/b$. Then x is the only order-N Farey fraction in Q_k.

The proof is on page 27 of [4]. We shall always assume that N satisfies Theorem 1 from now on. Theorem 1 motivates the definition of set

$$\hat{Z}_{p^r} = \left\{ | \, \frac{a}{b} \, |_{p^r} \mid \frac{a}{b} \in F_N \right\} ,$$

and $\hat{Z}_{p^r} \subset Z_{p^r}$. From Theorem 5.17 of [4] (see p. 28), $| \cdot |_{p^r} \mid F_N \rightarrow \hat{Z}_{p^r}$ is one-to-one and onto. Thus, we can state the immediate

Theorem 2 ∘ Addition and multiplication in Z_{p^r} corresponds to addition and multiplication in F_N provided that the input data and final results of the arithmetic in F_N lie entirely within the set F_N.

Proof ∘ $| \cdot |_{p^r} \mid \hat{Q} \rightarrow Z_{p^r}$ is a ring homomorphism, and $F_N \subset \hat{Q}$, $\hat{Z}_{p^r} \subset Z_{p^r}$, and $| \cdot |_{p^r} \mid F_N \rightarrow \hat{Z}_{p^r}$ is one-to-one and onto.

It will prove useful later on to have

Definition 2 ∘ The finite subset of Q

$$X = \left\{ \frac{a}{b} \in Q \mid gcd(a,b) = 1 \ , b = cp \ (c \in Z), 0 \le | \ a \ | \le N \ , 0 < | \ b \ | \le N \right\}$$

where N is as in Definition 1, is called the set of *invalid order-N Farey fractions*. From Definition 2, $b = cp$ and so b^{-1} $(mod \ p^r)$ does not exist. Clearly,

$$F_N \cup X = \left\{ \frac{a}{b} \mid gcd(a,b) = 1 \ , 0 \le | \ a \ | \le N \ , 0 < | \ b \ | \le N \right\} .$$

As well, $X \cap F_N = \varnothing$ (empty set), and $X \cap \hat{Q} = \varnothing$.

2.2 The Field of p-adic Numbers Q_p

Here we will present a working definition of the p-adic numbers as well as the arithmetic operations upon this set that contribute to making it a true algebraic field. We note that it is the ring structure of the p-adic numbers that interests us most, and so we will largely ignore the operation of division. A good introductory treatment of the p-adic numbers is in Koblitz [6]. However, some of the material that follows is summarized from Gregory and Krishnamurthy [4] (see sections 1 - 3 of Chapter II).

The *field of p-adic numbers* Q_p is the completion of the rational numbers with respect to the *p-adic metric*, which is induced by the *p-adic norm*. If $\alpha = a/b \in Q$ is nonzero, and $gcd(a,b) = 1$, then it can be expressed uniquely as

$$\alpha = \frac{c}{d} p^n \tag{1}$$

where p is any prime number, $gcd(c,d) = gcd(c,p) = gcd(d,p) = 1$. The p-adic norm is defined as follows (see [4], Theorem 2.3, page 64).

Theorem 3 • The mapping $|| \ \cdot \ ||_p | \ Q \to R$ (field of real numbers) defined by

$$|| \ \alpha \ ||_p = \begin{cases} p^{-n}, & \alpha \ne 0 \\ 0, & \alpha = 0 \end{cases}$$

is a norm on Q.

The proof is in Koblitz [6] (see Proposition on page 2). The p-adic metric induced by the p-adic norm is simply $d(x,y) = \ || \ x - y \ ||_p$ $(x,y \in Q)$. The completion of Q

with respect to $|| \cdot ||_p$ is discussed in Koblitz [6] (Chapter I), and so will not be considered here. However, we are interested in one of its consequences, namely

Theorem 4 ∘ Any rational number $\alpha \in Q$ has the unique p-adic expansion

$$\alpha = \sum_{j=n}^{\infty} a_j \, p^j$$

where each coefficient (digit) a_j is an integer in $\{0,1,...,p-1\}$, and n is such that $|| \alpha ||_p = p^{-n}$.

In other words, the infinite series converges to $\alpha \in Q$ in the p-adic metric. Theorem 4 is a corollary to Theorem 2.15 in [4] (see page 67). As an example, $1 + p^2 + p^4 + \cdots$ converges to $(1 - p^2)^{-1}$ in the p-adic metric. However, in the more familiar absolute value norm ($| x | = x$ if $x \geq 0$, $| x | = -x$ if $x < 0$), it is obvious that the same series diverges.

We are interested primarily in p-adic expansions of the members of $\hat{Q}$, and hence of F_N. If $\alpha \in \hat{Q}$, then its p-adic expansion has the form

$$\alpha = \sum_{j=0}^{\infty} a_j p^j = a_0 + a_1 p + \cdots + a_{n-1} p^{n-1} + a_n p^n + \cdots \quad . \tag{2a}$$

We can dispense with the need to display the powers of p explicitly by employing a *p-adic point* (as is done in [3,4]):

$$\alpha = .a_0 a_1 \cdots a_{n-1} a_n \cdots \quad . \tag{2b}$$

If $\alpha \in \hat{Q}$, then α has the form in (1) with $n \geq 0$. Thus, $|| \alpha ||_p$ is as in Theorem 3 which implies that $n \geq 0$ in Theorem 4. As a result, $a_i = 0$ for $0 \leq i \leq n-1$ will hold in (2a,b).

It is possible to do arithmetic with the p-adic numbers since we have stated that they form a field. We will concentrate on negation, addition, subtraction (via negation), and multiplication.

First consider negation. Let

$$\alpha = a_n p^n + a_{n+1} p^{n+1} + a_{n+2} p^{n+2} + \cdots \quad ,$$

then

$$-\alpha = b_n p^n + b_{n+1} p^{n+1} + b_{n+2} p^{n+2} + \cdots ,$$

where $b_n = p - a_n$, $b_j = (p-1) - a_j$ for $j > n$. The proof is straightforward and is contained in [4] on page 69.

Now consider addition. Let $\alpha \in \hat{Q}$ be as in (2a), and let $\beta \in \hat{Q}$ be

$$\beta = \sum_{j=0}^{\infty} b_j p^j = b_0 + b_1 p + \cdots + b_{m-1} p^{m-1} + b_m p^m + \cdots ,$$

where $|| \beta ||_p = p^{-m}$ and $m \geq 0$. Thus,

$$\alpha + \beta = \sum_{j=0}^{\infty} a_j p^j + \sum_{j=0}^{\infty} b_j p^j$$

$$= (a_0 + b_0) + (a_1 + b_1) p + \cdots + (a_k + b_k) p^k + \cdots$$

$$= s_0 + s_1 p + \cdots + s_k p^k + \cdots ,$$

where

$$a_j + b_j + c_j = s_j + c_{j+1} p \quad (j \geq 0) ,$$

and $0 \leq s_j < p$, c_{j+1} is the carry out of position j, and $c_0 = 0$.. Via Theorem 4, the p-adic sum $\sum_{j=0}^{\infty} s_j p^j$ is the unique p-adic expansion of $\alpha + \beta$. Subtraction is accomplished by negating the subtrahend and adding it to the minuend: $\alpha - \beta = \alpha + (-\beta)$.

Lastly, we consider multiplication. Let α and β be defined as in the previous paragraph, then

$$\alpha\beta = \left[\sum_{j=0}^{\infty} a_j p^j \right] \left[\sum_{j=0}^{\infty} b_j p^j \right]$$

$$= \sum_{j=0}^{\infty} \left[\sum_{k=0}^{j} a_k \, b_{j-k} \right] p^j$$

$$= \sum_{j=0}^{\infty} p_j p^j \quad ,$$

where

$$\sum_{k=0}^{j} a_k \, b_{j-k} + c_j = p_j + c_{j+1} p \quad (j \geq 0) \quad ,$$

such that $0 \leq p_j < p$, c_{j+1} is the carry out of position j, and $c_0 = 0$. Via Theorem 4, the p-adic sum $\sum_{j=0}^{\infty} p_j p^j$ is the unique p-adic expansion of $\alpha\beta$. We may therefore define the mapping $\phi \mid \hat{Q} \rightarrow Q_p$ as the operation of p-adically expanding the elements of the ring $\hat{Q}$. We are of course at liberty to write $\phi(\alpha) = \alpha$ because of Theorem 4 and the way in which the addition and multiplication of p-adic numbers is defined. In fact we have

Lemma 1 • $\phi \mid \hat{Q} \rightarrow Q_p$ is a ring homomorphism. That is, for any $\alpha, \beta \in \hat{Q}$,

$$\phi(\alpha + \beta) = \phi(\alpha) + \phi(\beta) \ ,$$

$$\phi(\alpha\beta) = \phi(\alpha)\phi(\beta) \ .$$

Proof • Follows automatically from the definition of p-adic addition and multiplication defined above, and Theorem 4.

As well, we have

Lemma 2 • Let $\phi(\hat{Q}) = \{\alpha \in Q_p \mid \alpha = \phi(a)$ for *some* $a \in \hat{Q}\}$, then $\phi(\hat{Q})$ is a subring of the field Q_p.

Proof • It is a basic property of all ring homomorphisms that the image set of the ring homomorphism is itself a ring.

Because of Theorem 4 $\phi \mid \hat{Q} \rightarrow \phi(\hat{Q})$ is one- to-one as well as onto (isomorphism).

Division in Q_p is discussed in [4,6]. We will not consider it here as our primary concern is with the operations of addition and multiplication in Q_p. The addition and multiplication operations on Hensel codes, which we shall define in the next section, are directly inherited from the corresponding operations on the p-adic numbers that we have defined in this section.

2.3 Hensel Codes (Finite Segment p-Adic Numbers)

In this section we define the Hensel codes, or finite-segment p-adic numbers as they are alternatively called, and we define arithmetic on the Hensel codes.

We begin by defining the mapping $\psi \mid \phi(\hat{Q}) \to Z_{p^r}$, where $\phi(\hat{Q})$ is the image of $\hat{Q}$ in Q_p under ϕ (see section 2.2). Let $\alpha \in \hat{Q}$, then $\phi(\alpha) = \sum_{j=0}^{\infty} a_j p^j \in \phi(\hat{Q})$, and

$$\psi(\phi(\alpha)) = \sum_{j=0}^{r-1} a_j p^j = a_0 + a_1 p + \cdots + a_{r-1} p^{r-1} \in Z_{p^r} \quad , \qquad (3)$$

where $\| \alpha \|_p = p^{-n}$, and if $n \geq r$ then $\psi(\phi(\alpha)) = 0$. In other words, the mapping ψ formally truncates the p-adic sum (expansion) of $\alpha \in \hat{Q}$. We have

Lemma 3 • $\psi \mid \phi(\hat{Q}) \to Z_{p^r}$ is a ring homomorphism. That is, for any $\alpha, \beta \in \phi(\hat{Q})$,

$$\psi(\alpha + \beta) = \psi(\alpha) + \psi(\beta) \quad ,$$

$$\psi(\alpha\beta) = \psi(\alpha)\psi(\beta) \quad .$$

Proof • $\phi(\hat{Q})$ is a ring (Lemma 2). Z_{p^r} is a ring as defined in section 2.1. Let $\alpha, \beta \in \phi(\hat{Q})$ such that

$$\alpha = \sum_{j=0}^{\infty} a_j p^j \; , \quad \beta = \sum_{j=0}^{\infty} b_j p^j \quad .$$

Thus,

$$\psi(\alpha)+\psi(\beta) = \sum_{j=0}^{r-1} a_j p^j + \sum_{j=0}^{r-1} b_j p^j = \sum_{j=0}^{r-1} (a_j + b_j) p^j = \sum_{j=0}^{r-1} s_j p^j = \psi(\alpha+\beta) \; ,$$

since

$$a_j + b_j + c_j = s_j + c_{j+1} p \quad (0 \leq j \leq r-1) \quad ,$$

where $0 \leq s_j < p$, c_{j+1} is the carry out of position j, $c_0 = 0$ and we ignore c_r. We have employed the definition of addition in the ring Z_{p^r} when the members of Z_{p^r} are given the radix-p representation in (3). This representation is readily proven to be unique. Similarly,

$$\psi(\alpha)\psi(\beta) = \left[\sum_{j=0}^{r-1} a_j p^j\right]\left[\sum_{j=0}^{r-1} b_j p^j\right] = \sum_{j=0}^{r-1}\left[\sum_{k=0}^{j} a_k b_{j-k}\right] = \sum_{j=0}^{r-1} p_j p^j = \psi(\alpha\beta) ,$$

since

$$\sum_{k=0}^{j} a_k b_{j-k} + c_j = p_j + c_{j+1}p \quad (0 \le j \le r-1) ,$$

where $0 \le p_j < p$, c_{j+1} is the carry out of position j, $c_0 = 0$ and we ignore c_r. We have employed the definition of multiplication in Z_{p^r} when the members of Z_{p^r} are given in a radix-p representation as in (3).

We also have

Lemma 4 ∘ ψ maps $\phi(\hat{Q})$ onto Z_{p^r} .

Proof ∘ Let $\alpha \in Z_{p^r}$, then we may write

$$\alpha = \sum_{j=0}^{r-1} a_j p^j .$$

If $a = \sum_{j=0}^{\infty} \overline{a}_j p^j$ then $\psi(a) = \alpha$, where $\overline{a}_j = a_j$ for $0 \le j \le r-1$, but $\overline{a}_j$ can be arbitrary for $j \ge r$. That is, for any $\alpha \in Z_{p^r}$ there is an $a \in \phi(\hat{Q})$ such that $\psi(a) = \alpha$.

We may define another mapping $\Psi \mid \phi(\hat{Q}) \rightarrow H_\Psi$ as follows. Let $\alpha \in \phi(\hat{Q})$ where

$$\alpha = .a_0 a_1 a_2 \cdots a_{r-1} a_r a_{r+1} \cdots$$

and it is of course possible that $a_i = 0$ for $0 \le i \le n-1$ with $n \ge r$. We write

$$\Psi(\alpha) = .a_0 a_1 a_2 \cdots a_{r-1} .$$

We let $H_\Psi = \{\Psi(\alpha) \mid \alpha \in \phi(\hat{Q})\}$. We may define addition and multiplication on this set and so make it into a commutative ring with identity. This we do as follows.

Let $a = .a_0 \cdots a_{r-1}$, and $b = .b_0 \cdots b_{r-1}$ belong to H_Ψ. Addition is defined as

$$a + b = .a_0 \cdots a_{r-1} + .b_0 \cdots b_{r-1} = .s_0 \cdots s_{r-1} ,$$

where

$$a_i + b_i + c_i = s_i + c_{i+1}p \quad (0 \le i \le r-1)$$

such that $0 \le s_i < p$, c_{i+1} is the carry out of position i, $c_0 = 0$ and we ignore c_r. Multiplication is defined as

$$ab = (.a_0 \cdots a_{r-1})(.b_0 \cdots b_{r-1}) = .p_0 \cdots p_{r-1} ,$$

where

$$\sum_{k=0}^{i} a_k b_{i-k} + c_i = p_i + c_{i+1}p \quad (0 \le i \le r-1)$$

such that $0 \le p_i < p$, c_{i+1} is the carry out of position i, $c_0 = 0$ and we ignore c_r. Thus, we may state

Lemma 5 ∘ $\Psi \mid \phi(\hat{Q}) \to H_\Psi$ is an onto ring homomorphism.

Proof ∘ Similar to the proof of Lemmas 3 and 4.

Lemma 6 ∘ $\theta \mid H_\Psi \to Z_{p^r}$ as defined by

$$\theta(.a_0 a_1 \cdots a_{r-1}) = \sum_{i=0}^{r-1} a_i p^i$$

is a ring isomorphism.

Proof ∘ Addition and multiplication in H_Ψ is the same as addition and multiplication in Z_{p^r} except for trivial notational differences: H_Ψ uses positional notation, and Z_{p^r} uses radix-p representations. As well, θ is clearly a one-to-one onto mapping.

We have $F_N \subset \hat{Q}$, $\phi(F_N) \subset \phi(\hat{Q})$, and we may define

$$H = \left\{ \Psi(\alpha) \mid \alpha \in \phi(F_N) \right\}$$

which gives us $H \subset H_\Psi$. It may be readily seen that $H = \{H(p,r,\alpha) \mid \alpha \in F_N\}$ as well. $H(p,r,\alpha)$ denotes the *Hensel code* (see [4]) for α, where r is the number of p-adic digits after the p-adic point that we retain (the digits are the a_i in (2a,b)). Thus, $\frac{1}{3} \in F_N, p = 5$, and $r = 4$, then

$$\phi(\frac{1}{3}) = .23131313 \cdots$$

and $H(5,4,1/3) = .2313$. $\phi(10) = .02000 \cdots$, and $H(5,4,10) = .0200$. It is actually possible to find Hensel codes for any $\alpha \in Q$ as specified by Theorem 4.5 in [4] (see pp. 80-81). We will let

$$H_X = \left\{ H(p,r,\alpha) \mid \alpha \in X \right\}$$

denote the Hensel codes for the invalid order-N Farey fractions X. When $\alpha \in X$ is written as in (1) we get $n < 0$. As a result, the p-adic expansion of α contains negative powers of p (see Theorem 4). H_X is obtained by truncating the p-adic expansions of all of the members of X as described in [4]. Thus, $H \cap H_X = \emptyset$.

The following lemma makes it possible to equivalently add, subtract, or multiply in F_N, $\hat{Z}_{p^r}$, or H.

Lemma 7 ∘ Mapping $| \cdot |_{p^r} \mid F_N \rightarrow \hat{Z}_{p^r}$ is equivalent to the composite mapping $\psi \circ \phi \mid F_N \rightarrow \hat{Z}'_{p^r}$ ($= \psi \circ \phi(F_N)$). That is, $\psi \circ \phi(\alpha) = | \alpha |_{p^r}$ for all $\alpha \in F_N$, both mappings are one-to-one and onto (so $\hat{Z}_{p^r} = \hat{Z}'_{p^r}$).

Proof ∘ $| \cdot |_{p^r} \mid F_N \rightarrow \hat{Z}_{p^r}$ is one-to-one and onto (Theorem 5.17, page 28 of [4]). If $\alpha = \frac{a}{b} \in F_N$, then

$$\alpha = \frac{a}{b} = \sum_{j=0}^{\infty} a_j p^j = \phi(\alpha) ,$$

and so

$$\frac{a}{b} = (a_0 + a_1 p + \cdots + a_{r-1} p^{r-1}) + p^r R_r$$

or

$$a = b(a_0 + a_1 p + \cdots + a_{r-1} p^{r-1}) + b p^r R_r .$$

Hence

$$| a |_{p^r} = | b(a_0 + a_1 p + \cdots + a_{r-1} p^{r-1}) |_{p^r}$$

which implies that

$$| ab^{-1} |_{p^r} = a_0 + a_1 p + \cdots + a_{r-1} p^{r-1} = | \alpha |_{p^r} .$$

But $\psi \circ \phi(\alpha) = \sum_{j=0}^{r-1} a_j p^j$ and so $\psi \circ \phi(\alpha) = | \alpha |_{p^r}$ for all $\alpha \in F_N$. Thus, $\psi \circ \phi \mid F_N \rightarrow \hat{Z}'_{p^r}$ is one-to-one and onto, and $\hat{Z}_{p^r} = \hat{Z}'_{p^r}$.

The proof of Lemma 7 is similar to the proof of Theorem 4.5, pages 80-81 of [4].

We are now in a position to state

Theorem 5 • Arithmetic with the elements of F_N can be equivalently performed with the elements of $\hat{Z}_{p^r}$, or with the elements of H, provided that the input data and final results of arithmetic with the elements of F_N lie entirely within F_N. By arithmetic we mean addition, subtraction, and multiplication.

Proof • Since $F_N \subset \hat{Q}$, $\phi(F_N) \subset \phi(\hat{Q})$, $\hat{Z}_{p^r} \subset Z_{p^r}$ and $H \subset H_\Psi$, the result follows from the use of the previous lemmas. We omit the details.

We have defined many sets and mappings between sets. These sets and mappings are summarized in Figure 1 for the convenience of the reader.

What we have proven is that the ring Z_{p^r} and the ring H_Ψ are the same. As a result, there is little reason to map addition, subtraction or multiplication in the ring Z_{p^r} to similar operations in H_Ψ. It would seem then that the concept of Hensel codes is redundant from a practical standpoint. As we have already noted in the Introduction, Gorgui-Naguib and King [5] have shown that it is hazardous to compute with the members of the set $H \cup H_X$, unless special steps are taken. Specifically, they have shown how to redefine the operations of addition and multiplication of Hensel codes so that the correct results are obtained even though the input data and final results may correspond to invalid order-N Farey fractions. Unfortunately, their methods are not very practical since one is effectively forced (in general) to map from $F_N \cup X$ to $H \cup H_X$ and vice versa, perhaps several times, during the course of a computation. This effectively precludes using any form of special purpose hardware to implement the methods described in [5]. Thus, from a practical standpoint one is forced to shun

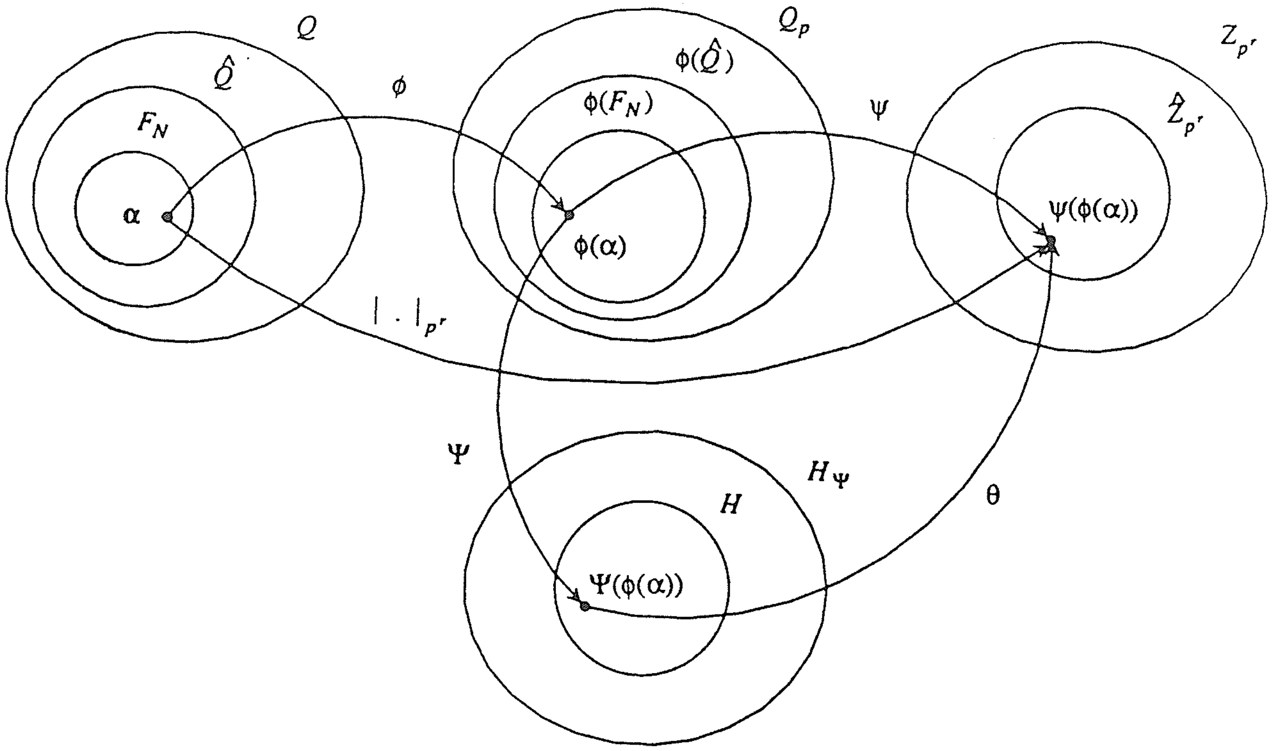


Figure 1: A summary of the various sets and mappings in sections 2.1 - 2.3.

the use of invalid Farey fractions as operands, but this eliminates the usefulness of the Hensel code concept. We are thus led to ask if it is possible to include the elements of the set X as operands but eliminate the need to map back and forth between the rationals and the Hensel codes during the course of a computation. It is possible to do this, as we'll soon see, although not in a very practical way in the general case.

To conclude this section, it is possible to divide with the elements of F_N ($F_N \subset Q$ the field of rational numbers), but this is potentially troublesome, since if $a,b \in F_N$, then $\frac{a}{b} \in X$ is possible. We note that addition, subtraction, and multiplication of the elements of F_N will never produce an element of X ($F_N \subset \hat{Q}$ the commutative ring such that $X \cap \hat{Q} = \varnothing$). Thus, division with the elements of H, as defined in [4] (pp. 92-93), is potentially troublesome too, although addition, subtraction and multiplication are not.

2.4 Restoring the Elements of X as Valid Operands

In this section we show how to compute with the Hensel codings of $H' \cup H_X'$ (these sets are defined below), such that the results of these computations correspond to the correct element of $F_N \cup X$. By computation we mean, as usual, addition, subtraction and multiplication.

We may naturally extend the mapping $\phi \mid \hat{Q} \to Q_p$ of section 2.2 to $\phi \mid Q \to Q_p$ (via Theorem 4) except that $n < 0$ is now possible. Once again, $\phi \mid Q \to Q_p$ is a ring homomorphism (in fact, $\phi \mid Q \to \phi(Q)$ is an isomorphism). We must of course modify (slightly) the operations of negation, addition and multiplication on the members of $\phi(\hat{Q}) \subset Q_p$ that were defined in section 2.2 in order to accommodate the fact that $n < 0$ may now hold.

Actually, negation need not be modified: simply allow $n < 0$ to hold. As for addition, let $\alpha, \beta \in \phi(Q)$ such that

$$\alpha = \sum_{j=n}^{\infty} a_j p^j \ , \ \ \beta = \sum_{j=m}^{\infty} b_j p^j \ , \tag{4}$$

and assume, without loss of generality, that $n \le m$. Note that $a_n \ne 0$, and $b_m \ne 0$ according to the notational conventions of this section. Thus,

$$\alpha + \beta = \sum_{j=n}^{\infty} a_j p^j + \sum_{j=m}^{\infty} b_j p^j = \sum_{j=n}^{\infty} (a_j + b_j) p^j = \sum_{j=n}^{\infty} s_j p^j \ ,$$

where $b_j = 0$ for $n \le j \le m-1$ and

$$a_j + b_j + c_j = s_j + c_{j+1} p \quad (j \ge n)$$

such that $0 \le s_j < p$, c_{j+1} is the carry out of position j, and $c_n = 0$. It is clear that if $n \ge 0$ then the above definition of addition coincides with the previous one. Similarly, for multiplication

$$\alpha\beta = \left(\sum_{j=n}^{\infty} a_j p^j \right) \left(\sum_{j=m}^{\infty} b_j p^j \right) = p^{n+m} \sum_{j=0}^{\infty} \left(\sum_{k=0}^{j} a_{n+k} b_{m+j-k} \right) p^j = \sum_{i=n+m}^{\infty} p_i p^i$$

where $i = j+n+m$ and

$$\sum_{k=0}^{j} a_{n+k} b_{m+j-k} + c_{j+n+m} = p_{j+n+m} + c_{j+n+m+1} p \quad (j \geq 0)$$

such that $0 \leq p_i < p$, $c_{j+n+m+1}$ is the carry out of position $j+n+m$, and $c_{n+m} = 0$. If $n \geq 0$, $m \geq 0$ then this definition of multiplication coincides with the previous one.

Since $n < 0$ is possible in (4), we want to know the largest value of $| n |$ for all $\alpha \in X$. This is determined by N, which specifies the maximum size of the input operands and final results. Knowing N fixes r (for a given p) via Theorem 1. Thus, if $\frac{a}{b} \in F_N \cup X$, then

$$| a |, | b | \leq N = \left\lfloor \sqrt{\frac{p^r - 1}{2}} \right\rfloor < p^{\frac{r}{2}} .$$

Therefore, the largest p^k which divides $| a |$ or $| b |$ is $p^{\frac{r}{2}-1}$, assuming that r is even which we shall do from now on for simplicity. Naturally, $r \geq 2$, and if $r = 2$ then $k = 0$. Thus, we shall need $\frac{r}{2} - 1$ digits to the left of the p-adic point in order to find the full p-adic expansion of any $\alpha \in X$. As well, there will never be more than $\frac{r}{2} - 1$ zeros to the right of the p-adic point.

The set H_X corresponds to the Hensel codes of X is defined in [3,4]. If $\alpha = a_n a_{n+1} \cdots a_{-1}.a_0 \cdots a_{r-1} a_r \cdots$, and $\alpha \in X$, then the Hensel code for it in H_X is obtained by taking the first r nonzero digits of its p-adic expansion. As a result, the first digit of the Hensel code will be a_n. The position of the p-adic point must of course be maintained. Hence

$$H_X = \left\{ a_n \cdots a_{-1}.a_0 \cdots a_{n+r-1} \mid \alpha = \sum_{j=n}^{\infty} a_j p^j \in \phi(X) \right\},$$

where $-\frac{r}{2} + 1 \leq n < 0$. We have

Lemma 8 • The elements of H_X are in one-to-one correspondence with the elements of X.

Proof ∘ This result follows from Theorem 4.5, pages 80-81 of [4].

We may extend the mapping $\Psi \mid \phi(\hat{Q}) \to H_\Psi$. Let

$$\alpha = \sum_{j=n}^{\infty} a_j p^j \in \phi(Q)$$

and so let $\hat{\Psi} \mid \phi(Q) \to H_{\hat{\Psi}}$ be defined by

$$\hat{\Psi}(\alpha) = a_n a_{n+1} \cdots a_{-1}.a_0 \cdots a_{\hat{r}-1} \, ,$$

where $n \in Z$ and $\hat{r} \geq 1$. Specifically, if $\alpha \in \phi(\hat{Q})$ then $n \geq 0$, otherwise $n < 0$. As a result, $\hat{\Psi}(\alpha)$ is $\mid n \mid + \hat{r}$ digits long if $n < 0$, otherwise it is $\hat{r}$ digits long. If $\hat{r} = r$ then $\hat{\Psi} \mid \phi(\hat{Q}) \to H_\Psi$ and $\hat{\Psi} = \Psi$ in this case. As well, $\hat{\Psi} \circ \phi \mid F_N \to H$ holds.

Let us define addition and multiplication on $H_{\hat{\Psi}}$ as follows. Negation is defined in the obvious way. First, consider addition. Let

$$\alpha = a_n \cdots a_{-1}.a_0 \cdots a_{\hat{r}-1} \, , \beta = b_m \cdots b_{-1}.b_0 \cdots b_{\hat{r}-1} \in H_{\hat{\Psi}} \, , \qquad (5)$$

and assume without loss of generality that $n \leq m$. Then,

$$\alpha + \beta = s_n \cdots s_{-1}.s_0 \cdots s_{\hat{r}-1}$$

where $b_j = 0$ for $n \leq j \leq m-1$, and

$$a_j + b_j + c_j = s_j + c_{j+1}p \quad (n \leq j \leq \hat{r}-1) \cdot$$

such that $0 \leq s_j < p$, c_{j+1} is the carry out of position j, $c_n = 0$ and we ignore $c_{\hat{r}}$. As for multiplication,

$$\alpha\beta = p_{n+m}p_{n+m+1} \cdots p_{-1}.p_0 \cdots p_{\hat{r}-1}$$

where

$$\sum_{k=0}^{j} a_{n+k}b_{m+j-k} + c_{j+n+m} = p_{j+n+m} + c_{j+n+m+1}p \quad (0 \leq j \leq \hat{r}-n-m-1)$$

such that $0 \leq p_j < p$, $c_{j+n+m+1}$ is the carry out of position $j+n+m$, $c_{n+m} = 0$, and we ignore $c_{\hat{r}}$.

H_Ψ is not a commutative ring with identity under the above operations since there is no associative law for multiplication. For example, the reader should try multiplying $a = a_{-1}.a_0a_1$, $b = .b_0b_1$, $c = c_{-1}.c_0c_1$ where $\hat{r} = 2$ and it will be seen that $(ab)c \neq a(bc)$. As well, even if H_Ψ were a ring, $\Psi \mid \phi(Q) \rightarrow H_\Psi$ is not a homomorphism since $\Psi(\alpha\beta) \neq \Psi(\alpha)\Psi(\beta)$. For example, let $p = 5$, $\hat{r} = r = 4$, with

$$\Psi \circ \phi(\frac{5}{4}) = .0433 \ , \quad \Psi \circ \phi(\frac{1}{15}) = 2.3131 \ ,$$

then $.0433 \times 2.3131 = .3423$. But $\Psi \circ \phi(\frac{1}{12}) = .3424$ and $\frac{5}{4} \cdot \frac{1}{15} = \frac{1}{12}$ and so .3423 is the wrong answer.

Let us define two new sets, namely

$$H' = \left\{ \Psi \circ \phi(\alpha) \mid \alpha \in F_N \right\} \subset H_\Psi \text{ , and}$$

$$H_X' = \left\{ \Psi \circ \phi(\alpha) \mid \alpha \in X \right\} \subset H_\Psi \ .$$

Note that $H = H'$ if $\hat{r} = r$.

Because of Lemma 8, the elements of H_X' are in one-to-one correspondence with the elements of X. In fact we have

Lemma 9 • $\Psi \circ \phi \mid F_N \cup X \rightarrow H' \cup H_X'$ with $\hat{r} = r$ is one-to-one and onto.

Proof • $\Psi \circ \phi \mid F_N \rightarrow H$ is one-to-one and onto, and use Lemma 8. Clearly, $H' \cap H_X' = \varnothing$, $F_N \cap X = \varnothing$.

Since $\Psi \mid \phi(Q) \rightarrow H_\Psi$ is not a homomorphism, $\Psi \circ \phi \mid Q \rightarrow H_\Psi$ is obviously not a homomorphism either. Thus, despite Lemma 9, we cannot perform arithmetic in $H' \cup H_X'$ such that it corresponds to arithmetic in $F_N \cup X$, unless $\hat{r}$ is sufficiently large with respect to r. How large $\hat{r}$ should be depends upon the number and kind of arithmetic operations to be performed.

For example, if we wish to find $\alpha\beta$ (α,β as in (5)), then $\hat{r} = \frac{3r}{2} - 1$ is needed, since $n,m \geq -\frac{r}{2}+1$, and to correctly compute digits $n+m$ to $r-1$ of $\alpha\beta$ it is necessary that we know the digits of the p-adic expansion to position $\frac{3r}{2}-1$. Consider $p = 5$, $r = 4$ so $\hat{r} = 5$, with

$$\Psi \circ \phi(\frac{5}{4}) = .04333\ ,\ \Psi \circ \phi(\frac{1}{15}) = 2.31313$$

and so $.04333 \times 2.31313 = .3424x$. We have omitted to compute the digit labeled x since .3424 is enough to identify the product as being $\frac{1}{12}$. In fact, with $\hat{r} = \frac{3r}{2}-1$ we cannot in general compute x correctly. Thus, any succeeding products involving $.3424x$ as an operand cannot be guaranteed to produce the correct result (unless we increase $\hat{r}$). If we wish to find $\alpha + \beta$, then $\hat{r} = r$ will work. Consider $p = 5$, $\hat{r} = r = 4$ with

$$\Psi \circ \phi(\frac{11}{10}) = 3.3222\ ,\ \Psi \circ \phi(\frac{16}{15}) = 2.4131$$

and so $3.3222 + 2.4131 = 0.3404$ and $\Psi \circ \phi(\frac{13}{6}) = .3404$ which is the correct result since $\frac{11}{10} + \frac{16}{15} = \frac{13}{6}$. In general, if $\hat{r} = r$, then

$$\Psi \circ \phi(\alpha + \beta) = \Psi \circ \phi(\alpha) + \Psi \circ \phi(\beta)\ ,$$

and so it is multiplication that causes difficulties, as we've already noted.

Thus, we must select $\hat{r}$ so that if $n < 0$ for some result $\alpha \in H' \cup H_X'$, the first r digits of α must be correct, and if $n \geq 0$ then the first r digits to the right of the p-adic point must be correct. Thus, it will be necessary in practice to select a large value for $\hat{r}$, especially if multiplication is to be performed often.

The discussions and analyses of this and the preceding sections underscores the importance of the following principle. Given two rings R and S such that we wish to perform addition and multiplication in S so that it corresponds to the same operations in R, it is necessary that we have a mapping $\phi \mid R \to S$ such that ϕ is a ring homomorphism. Since ϕ is many-to-one and into in general, it is also necessary that we identify subsets $A \subset R$ and $B \subset S$ such that $\phi \mid A \to B$ is one-to-one and onto.

Thus, our attempt at including the elements of X as valid operands and final results is limited since H_{Ψ} is not a ring, and $\Psi \circ \phi \mid Q \rightarrow H_{\Psi}$ is not a homomorphism. Furthermore, there appears to be no reasonable way of augmenting F_N with X and yet obtain a suitable ring structure in the sense specified in the previous paragraph.

2.5 Conclusions

We have demonstrated that arithmetic with Hensel codes as defined in Krishnamurthy, Rao and Subramanian [3] and in Gregory and Krishnamurthy [4] will work provided that the input data and final results correspond to order-N Farey fractions. By arithmetic we mean negation, addition, subtraction (via negation), and multiplication.

We have also shown that by excluding the invalid Farey fractions from consideration we eliminate the need for the Hensel code concept. This is due to the fact that Z_{p^r} and H_{Ψ} are isomorphic rings, and that the differences between them are purely notational.

Finally, we have shown how to include the invalid Farey fractions as operands and results such that in computing with their Hensel codes, it is not necessary to map back and forth between the Hensel codes and the rationals. Unfortunately, the proposed solution is generally impractical as it requires the use of Hensel codes of very large size.

3. Finite Rings and Fields

We have seen in section 2 that mapping computations involving rational data into Hensel codes (finite segment p-adic numbers) is of little, if any, practical use. Thus, having also rejected rational arithmetic, we are compelled to conclude that error-free computation should only be performed in finite rings or fields. We shall see in this and the next section that various choices exist when it comes to selecting a suitable ring or field. We have already seen one choice in section 2 where, via Theorem 2, arithmetic (meaning addition, subtraction and multiplication only) with a suitable subset of the rationals can be performed in the finite ring Z_m, where $m = p^r$ (p is a positive prime, r is a positive integer).

Thus, in general, the set $Z_m = \{0,1,...,m-1\}$ forms a finite ring under modulo m addition and multiplication. It is possible for m to be composite, but if it is prime then Z_m forms a finite field (more usually denoted by $GF(m)$). A ring Z_m may be called a *single modulus residue number system (SMRNS)* [4], where m is the *modulus*. If we are interested in computing with a subset S_m of the integers Z, where

$$S_m = \left\{ -\frac{m-1}{2}, \ldots, -1,0,1, \ldots, \frac{m-1}{2} \right\}, \tag{6}$$

(m odd) then such a computation may be mapped into the ring Z_m, provided that the input data and final answer are from S_m (see [4]). If $x \in S_m$ and $x \geq 0$ then x maps to x in Z_m. If $x \in S_m$ but $x < 0$ then x maps to $x + m$ in Z_m.

Suppose now that $m = m_1 m_2 \cdots m_k$, and $gcd(m_i, m_j) = 1$ $(i \neq j)$ for all i and j, then Z_m can be made isomorphic to the product ring

$$Z_{m_1} \times Z_{m_2} \times \cdots \times Z_{m_k} . \tag{7}$$

We call m_i the *ith modulus*. The *product ring* in (7) is often refered to as a *multiple modulus residue number system (MMRNS)*. If $x \in Z_m$ then x maps to a unique k-tuple $(x_1, x_2, ..., x_k) \in Z_{m_1} \times \cdots \times Z_{m_k}$, where $x_i = x \ (mod \ m_i)$. We will discuss the inverse mapping from the product ring back to Z_m below, but let us now consider arithmetic in (7). If $(x_1, \ldots, x_k), (y_1, \ldots, y_k) \in Z_{m_1} \times \cdots \times Z_{m_k}$, then

$$(x_1, \ldots, x_k) \circledcirc (y_1, \ldots, y_k) = (z_1, \ldots, z_k) \in Z_{m_1} \times \cdots \times Z_{m_k} , \tag{8a}$$

where '$\circledcirc$' represents addition, subtraction or multiplication in the product ring, and

$$z_i = x_i \circ y_i , \tag{8b}$$

where '$\circ$' represents addition, subtraction or multiplication modulo m_i. Thus, arithmetic in the product ring is performed component-wise. That is, no carry information is propagated from one component to the next, in contrast with conventional weighted binary arithmetic schemes. Note that it is carry propagation that often dominates the time taken to perform basic arithmetic operations. It is primarily for this reason that the MMRNS has been proposed as a means of implementing certain digital signal processing (DSP) algorithms (see [7,8]), and error-free computation algorithms (see [4]) at high speed. DSP algorithms that have been implemented with MMRNS schemes include FIR filtering [9], IIR filtering [10], and number theoretic transforms (NTTs) [11], which may be used to perform error-free circular convolutions and correlations.

An element of the product ring (7) may be mapped back to its corresponding unique member of Z_m via either:

(i) the Chinese Remainder Theorem (CRT), or

(ii) the mixed-radix number representation (MRNR).

The CRT is explained in such books as McClellan and Rader [8], and Blahut [12], and the MRNR is explained in Gregory and Krishnamurthy [4], and Taylor [7], but we shall review the main points here.

Let us begin with the CRT. Suppose $M = \prod_{i=1}^{k} m_i$ (so $M = m$), and that $M_i = M/m_i$, and if we know the residues $x_i = x \ (mod\ m_i)$ $(x \in Z_m)$ for all $i = 1,\ldots,k$, then we can recover x via

$$x = \sum_{i=1}^{k} x_i M_i N_i \quad (mod\ M) \tag{9a}$$

such that N_i satisfies the diophantine equation

$$N_i M_i + n_i m_i = 1 \ . \tag{9b}$$

The Euclidean algorithm [4,8,12] can be used to solve (9b) for n_i and N_i. The proof of (9a,b) may be found in [12]. It is clear that the inverse mapping of (9a) is quite complex from a computational standpoint. It is also complex from the viewpoint of a special purpose hardware implementation. As is argued in [12] (see page 60), it is only worthwhile mapping a computation in S_m (or, equivalently, in Z_m) to the product ring if the computation in S_m is complicated (i.e., has many steps) in relation to the complexity of the inverse mapping in (9a). Intermediate results can remain in the product ring and only the final answer must be mapped back to S_m.

The MRNR [4,7] is an alternative means of mapping a product ring element back to Z_m. This mapping is generally accepted as being superior to the CRT (see [4,7]), either in terms of computational speed or because special purpose hardware implementations of the MRNR are more efficient than equivalent CRT implementations. The MRNR is known to be particularly convenient to work with for certain combinations of moduli, such as $m_1 = 2^n - 1$, $m_2 = 2^n$, and $m_3 = 2^n + 1$ (see [7]). We may describe the MRNR as follows. Let $\{r_1, r_2, \cdots, r_k\}$ be a set of *radices*. Let $R = \prod_{i=1}^{k} r_i$. Every integer y such that $0 \le y < R$ can be uniquely expressed as

$$y = y_1 + y_2 r_1 + y_3 r_1 r_2 + \cdots + y_k r_1 r_2 \cdots r_{k-1} , \tag{10a}$$

where $y_1, y_2, \cdots, y_k$ are the *mixed-radix digits* (see [4], pp. 17-18), such that

$$0 \le y_i < r_i \quad i = 1,\ldots,k . \tag{10b}$$

If $(x_1, \ldots, x_k) \in Z_{m_1} \times \cdots \times Z_{m_k}$ then by assigning $r_i = m_i$ it is possible to map $(x_1, \ldots, x_k)$ to a representation like that in (10a). Such a mapping can be performed with residue arithmetic. This mapping is explained in detail in [4] (see pp. 19-23) where examples are also given. Special purpose hardware structures for the MRNR may be viewed in [7].

Even though the MRNR may represent a better means of carrying out the inverse mapping from the product ring of (7) to Z_m compared to the CRT, it is clear that the MRNR is still quite complicated from a computational and implementational standpoint. Thus, it is primarily the complexity of the inverse mapping problem that limits the applicability of MMRNSs. However, we have already noted that certain moduli and combinations of moduli make the MRNR relatively easy to work with, thus making MMRNS arithmetic practical, at least in certain applications.

4. The Quadratic Residue Number System and Farey Fractions

Herein we extend the definition of a quadratic residue number system (QRNS) to accommodate complex numbers with real and imaginary parts that are members of the set of order-N Farey fractions, a special subset of the rationals (see section 2.1). On the basis of Theorem 6 (below), and some computer search results, we conjecture that $j \notin \hat{Z}_{p^r}$, i.e., that j does not correspond to an order-N Farey fraction, where $j \in Z_{p^r}$ is a solution to $j^2 = -1 \ (mod\ p^r)$. This presumes that p is prime, and $p = 4k + 1 \ (k > 0)$. The results of sections 4.2 and 4.3 below are from Zarowski and Card [2].

4.1 The Conventional Quadratic Residue Number System

We will begin by describing what is classically understood to be the quadratic residue number system.

We may define a Gaussian ring $Z_{p^r}[i] = \{a + ib \mid a, b \in Z_{p^r}\}$, where $i = \sqrt{-1}$, and where the ring operations are

addition: $(a + ib) + (c + id) = (a + c) \ mod\ p^r + i(b + d) \ mod\ p^r$,

multiplication: $(a + ib)(c + id) = (ac - bd) \ mod\ p^r + i(ad + bc) \ mod\ p^r$.

This ring is often called a *complex residue number system (CRNS).* Note that multiplication in $Z_{p^r}[i]$ involves four multiplications in the ring Z_{p^r} in general. If there is no $j \in Z_{p^r}$ such that $j^2 = -1 \ (mod\ p^r)$, then -1 is called a *quadratic nonresidue,* otherwise it is called a *quadratic residue* (see Taylor [13], Jenkins and Krogmeier [14], Jullien, Krishnan and Miller [15], Dudley [16] (p. 85), or Hillman and Alexanderson [17] (p. 422)). It is well known that -1 is a quadratic residue if p is prime and of the form $4k$+1 ($k > 0$) (see [13-17]). Similarly, -1 is a quadratic nonresidue if p is prime but of the form $4k$+3. If $p = 4k + 1$ and is prime, then it is called a *Gaussian prime.*

If -1 is a quadratic residue then there are precisely two distinct solutions to $x^2 = -1 \ (mod\ p^r)$, and these solutions are additive and multiplicative inverses of each other. The theorem in the appendix of [14] shows how to solve this equation, given the two solutions two $x^2 = -1 \ (mod\ p)$. This latter equation is easy to solve if $p = 2^n + 1 \ (n > 1)$ since then $x = 2^{n/2}$ or $x = 2^n - 2^{n/2} + 1$. This makes sense if and only if n is even (see Theorem 2 in [14]).

We may define the product ring $Z_{p^r} \times Z_{p^r} = \{(a, b) \mid a, b \in Z_{p^r}\}$, where the ring operations are addition: $(a,b) + (c,d) = (a + c \ (mod\ p^r)\ ,\ b + d \ (mod\ p^r))$,

multiplication: $(a,b)(c,d) = (ac \ (mod\ p^r)\ ,\ bd \ (mod\ p^r))$.

Note that multiplication in $Z_{p^r} \times Z_{p^r}$ only involves two multiplications in the ring Z_{p^r} . When $p = 4k + 1$, we may define the mapping $\phi \mid Z_{p^r}[i] \rightarrow Z_{p^r} \times Z_{p^r}$ as

$$\phi(a + ib) = (a + jb \quad (mod\ p^r), a - jb \quad (mod\ p^r)) \ , \tag{11a}$$

where

$$j^2 = -1 \quad (mod\ p^r) \ . \tag{11b}$$

It turns out that ϕ is a ring isomorphism, and so a computation in $Z_{p^r}[i]$ can be mapped into an equivalent one in $Z_{p^r} \times Z_{p^r}$. It is clear that the main advantage in doing so is that the number of multiplications in Z_{p^r} is reduced from four to two.

Since ϕ is an isomorphism, $\phi^{-1} \mid Z_{p^r} \times Z_{p^r} \rightarrow Z_{p^r}[i]$ exists and is

$$\phi^{-1}((a,b)) = x + iy \quad , \tag{12a}$$

where

$$\begin{aligned} x &= 2^{-1}(a + b) \quad (mod\ p^r) \ , \\ y &= 2^{-1}j^{-1}(a - b) \quad (mod\ p^r) \ . \end{aligned} \tag{12b}$$

When p is a Gaussian prime, $Z_{p^r} \times Z_{p^r}$ forms what is commonly called a *quadratic residue number system (QRNS)*. The properties and applications of the QRNS are further explored in Leung [18], Vanwormhoudt [19], Baraniecka and Jullien [11], and Krogmeier and Jenkins [20], in addition to [13-15].

4.2 How To Include Rational Data

We now propose extensions to the results summarized in the previous section. The extensions allow efficient computation with complex numbers that have rational-valued real and imaginary parts from the order-N Farey fractions.

We may define the Gaussian ring $Q[i] = \{a + ib \mid a,b \in Q \ , i = \sqrt{-1}\}$. It is straightforward to verify that $\hat{Q}[i] = \{a + ib \mid a,b \in \hat{Q} \ , i = \sqrt{-1}\}$ is a subring of $Q[i]$. The operations defined on $Q[i]$ and inherited by $\hat{Q}[i]$ are analogous to those for $Z_{p^r}[i]$ in section 4.1. Sets $F_N[i]\ (\subset \hat{Q}[i])$, $\hat{Z}_{p^r}[i]\ (\subset Z_{p^r}[i])$, and $\hat{Z}_{p^r} \times \hat{Z}_{p^r}\ (\subset Z_{p^r} \times Z_{p^r})$ are defined in the obvious way.

It is necessary to extend the mapping $|\cdot|_{p^r} \mid \hat{Q} \rightarrow Z_{p^r}$ of section 2 to $|\cdot|_{p^r} \mid \hat{Q}[i] \rightarrow Z_{p^r}[i]$. This extended mapping is defined as

$$|x + iy|_{p^r} = |x|_{p^r} + i\,|y|_{p^r}\ , \tag{13}$$

where $x + iy \in \hat{Q}[i]$, so $x, y \in \hat{Q}$. This extended mapping for modulo p^r reduction is clearly a ring homomorphism. It is also straightforward to verify that $|\cdot|_{p^r} \mid F_N[i] \rightarrow \hat{Z}_{p^r}[i]$ is one-to-one and onto. Hence, a computation with input data and final results in $F_N[i]$ may be mapped equivalently into the finite ring $Z_{p^r}[i]$.

It is also clear that $\phi \mid \hat{Z}_{p^r}[i] \rightarrow \hat{Z}_{p^r} \times \hat{Z}_{p^r}$ is one-to-one and onto, since $\phi \mid Z_{p^r}[i] \rightarrow Z_{p^r} \times Z_{p^r}$ is a ring isomorphism. We can map a computation with inputs and final results in $\hat{Z}_{p^r}[i]$ to an equivalent computation in $Z_{p^r} \times Z_{p^r}$. Thus, we have formally extended the QRNS to accommodate complex rational data. Figure 2 summarizes the mappings and sets involved in the extension process.

We note that the ability to accommodate rational data is valuable as, for example, the Schur algorithm for Hermitian Toeplitz matrices requires the division operation. Even with Hermitian Toeplitz matrices with Gaussian integer entries only, the Schur variables and reflection coefficients will generally be rational in their real and imaginary parts.

4.3 What Elements are in $\hat{Z}_{p^r}$?

It appears that there are no results concerning what elements of Z_{p^r} are to be found in $\hat{Z}_{p^r}$. Certainly, no results are to be found in [3,4,21]. Naturally, given a particular member of a particular Z_{p^r}, one could use the inverse mapping algorithm in [4,21] to determine whether or not it maps to a Farey fraction. However, this is highly inefficient if p^r is large, and it offers no insight into the structure of set $\hat{Z}_{p^r}$.

We do not propose to solve this problem completely here, but nevertheless we can state

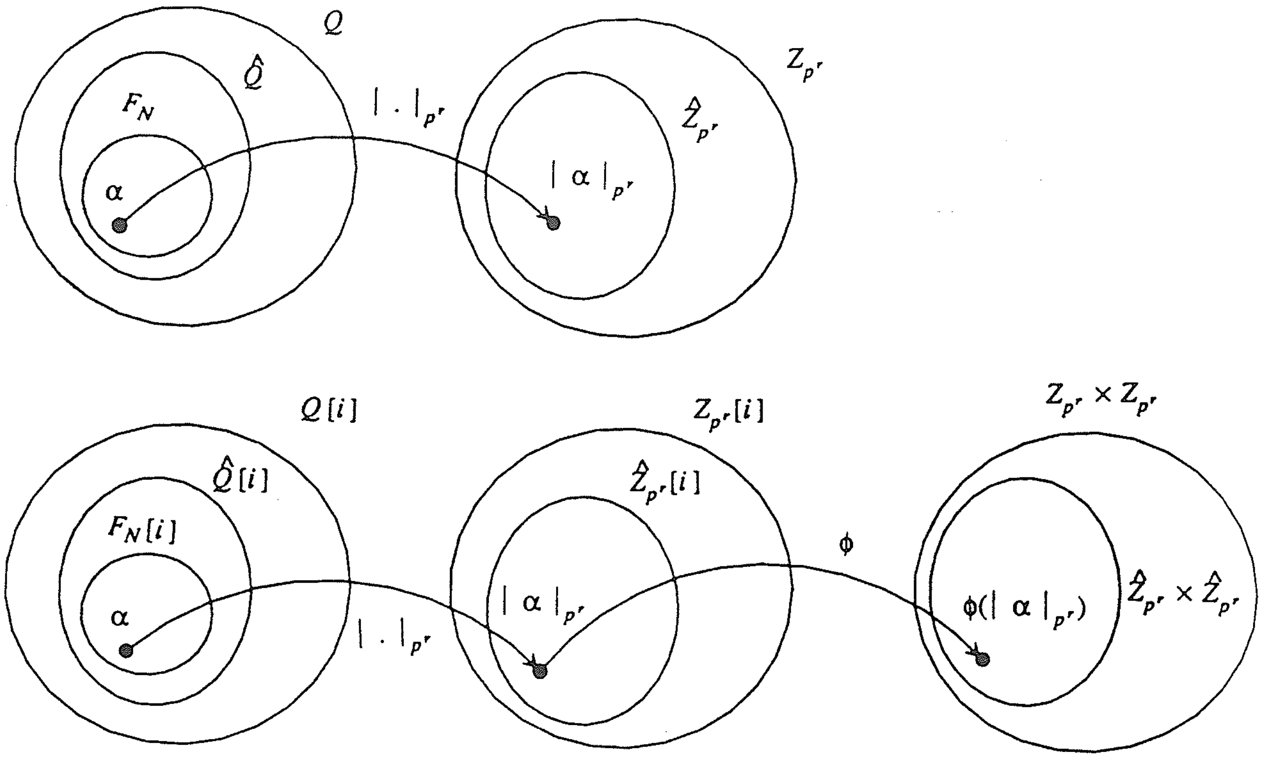


Figure 2: A summary of sets and mappings in sections 2.1, 4.1 and 4.2.

Theorem 6 ∘ $j \notin \hat{Z}_{p^r}$ where $p = 2^{2n} + 1$, $r = 1$, $n \in \{1,2,3,...\}$, and j satisfies (11b).

Proof ∘ According to the theorem in the appendix of [14], there are precisely two distinct solutions to (11b). These are readily shown to be

$$j = 2^n \ , \ 2^{2n} - 2^n + 1 \ .$$

From Theorem 1 of section 2.1 we have

$$N < 2^{n-1}\sqrt{2} < 2^n \quad .$$

We can carry out the inverse mapping algorithm on pp. 42-46 of [4] in order to ascertain whether or not the above values of j belong to $\hat{Z}_{p^r}$. Thus, for $j = 2^n$ we have the table

	$2^{2n} + 1$	0
	2^n	1
2^n	1	-2^n
2^n	0	$2^{2n} + 1$

Hence $j = 2^n$ maps to the rationals $\frac{2^n}{1}$ or $-\frac{1}{2^n}$. Neither of these rationals is a valid order-N Farey fraction. Thus, $j = 2^n$ is not in $\hat{Z}_{p^r}$. Similarly, for $j = 2^{2n} - 2^n + 1$ we have the table

	$2^{2n} + 1$	0
	$2^{2n} - 2^n + 1$	1
1	2^n	-1
$2^n - 1$	1	2^n
2^n	0	$-(1 + 2^{2n})$

Therefore, $j = 2^{2n} - 2^n + 1$ maps to the rationals $\frac{2^{2n} - 2^n + 1}{1}$, $-\frac{2^n}{1}$ or $\frac{1}{2^n}$. None of these rationals is a valid order-N Farey fraction. Thus, $j = 2^{2n} - 2^n + 1$ is not in $\hat{Z}_{p^r}$.

It is possible to construct an alternate proof with the aid of

Theorem 7 • Let $j_1, j_2 \in Z_{p^r}$ be the solutions to (11b). If $j_1 \notin \hat{Z}_{p^r}$, then $j_2 \notin \hat{Z}_{p^r}$.

Proof • Without loss of generality we may consider j_1. Assume that $j_1 \notin \hat{Z}_{p^r}$. We have $| Q_{j_1} |_{p^r} = j_1$, i.e., all members of the generalized residue class Q_{j_1} maps to $j_1 \in Z_{p^r}$. The inverse mapping algorithm in [4,21], that maps the elements of $\hat{Z}_{p^r}$ to F_N, produces a finite sequence of elements from Q_{j_1}, given j_1 and p^r. If it were true that $j_1 \in \hat{Z}_{p^r}$ then the finite sequence would contain the element of F_N that maps to j_1. But since $j_1 \notin \hat{Z}_{p^r}$, no such element will be found by the inverse mapping. Since $j_2 = j_1^{-1}$ $(mod\ p^r)$, applying the inverse mapping to j_2 will produce a finite sequence of elements from Q_{j_2} that are reciprocals of elements from Q_{j_1}. None of the elements of Q_{j_1} is an order-N Farey fraction, and so therefore none of the elements of Q_{j_2} is an order-N Farey fraction. Thus, if $j_1 \notin \hat{Z}_{p^r}$, then $j_2 \notin \hat{Z}_{p^r}$.

Appendix I contains the PASCAL program used to obtain the results in Table I of Appendix J. Table I shows several sets of p and r illustrating that $j \notin \hat{Z}_{p^r}$. Note that the theorem in the appendix of [14] was used to compute j satisfying (11b) in all cases. The inverse mapping of [4,21] was then used to test whether or not $j \in \hat{Z}_{p^r}$. No case of $j \in \hat{Z}_{p^r}$ was ever found. On the basis of this admittedly scant computer search evidence, and upon Theorem 6, we conjecture that $j \notin \hat{Z}_{p^r}$, where p is a Gaussian prime, r is a positive integer, and j satisfies (11b).

REFERENCES

[1] C. J. Zarowski, H. C. Card, "On Addition and Multiplication with Hensel Codes," submitted to the IEEE Trans. on Comp.

[2] C. J. Zarowski, H. C. Card, "Quadratic Residue Number Systems and Farey Fractions," submitted to the IEEE Trans. on Acoust., Speech, and Signal Proc.

[3] E. V. Krishnamurthy, T. M. Rao, K. Subramanian, "p-adic Arithmetic Procedures for Exact Matrix Computations," Proc. Indian Acad. Sci., vol. 82A, 1975b, pp. 165-175.

[4] R. T. Gregory, E. V. Krishnamurthy, *Methods and Applications of Error-Free Computation*. New York, New York: Springer-Verlag, 1984.

[5] R. N. Gorgui-Naguib, R. A. King, "Comments on "Matrix Processors Using p-Adic Arithmetic for Exact Linear Computations"," IEEE Trans. on Comp., vol. C-35, Oct. 1986, pp. 928-930.

[6] N. Koblitz, *p-adic Numbers, p-adic Analysis, and Zeta-Functions*. New York, New York: Springer-Verlag, 1977.

[7] F. J. Taylor, "Residue Arithmetic: A Tutorial with Examples," IEEE Computer Magazine, May 1984, pp. 50-62.

[8] J. H. McClellan, C. M. Rader, *Number Theory in Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[9] W. K. Jenkins, B. J. Leon, "The Use of Residue Number Systems in the Design of Finite Impulse Response Digital Filters," IEEE Trans. on Circ. and Syst., vol. CAS-24, April 1977, pp. 191-201.

[10] W. K. Jenkins, "Recent Advances in Residue Number Techniques for Recursive Digital Filtering," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-27, Feb. 1979, pp. 19-30.

[11] A. Z. Baraniecka, G. A. Jullien, "Residue Number System Implementations of Number Theoretic Transforms in Complex Residue Rings," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-28, June 1980, pp. 285-291.

[12] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, Massachusetts: Addison-Wesley, 1985.

[13] F. J. Taylor, "On the Complex Residue Arithmetic System (CRNS)," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-34, Dec. 1986, pp. 1675-1677.

[14] W. K. Jenkins, J. V. Krogmeier, "The Design of Dual-Mode Complex Signal Processors Based on Quadratic Modular Number Codes," IEEE Trans. on Circ. and Syst., vol. CAS-34, April 1987, pp. 354-364.

[15] G. A. Jullien, R. Krishnan, W. C. Miller, "Complex Digital Signal Processing Over Finite Rings," IEEE Trans. on Circ. and Syst., vol. CAS-34, April 1987, pp. 365-377.

[16] U. Dudley, *Elementary Number Theory,* 2nd ed. New York, New York: Freeman, 1978.

[17] A. P. Hillman, G. L. Alexanderson, *A First Undergraduate Course in Abstract Algebra,* 3rd ed. Belmont, California: Wadsworth, 1983.

[18] S.-H. Leung, "Application of Residue Number Systems to Complex Digital Filters," Proc. of the 15th Asilomar Conf. on Circ., Syst., and Comp., Pacific Grove, California, Nov. 1981, pp. 70-74.

[19] M. C. Vanwormhoudt, "Structural Properies of Complex Residue Rings Applied to Number Theoretic Fourier Transforms," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-26, Feb. 1978, pp. 99-104.

[20] J. V. Krogmeier, W. K. Jenkins, "Error Detection and Correction in Quadratic Residue Number Systems," Proc. of the 26th Midwest Symp. on Circ. and Syst., Puebla, Mexico, Aug. 1983, pp. 408-411.

[21] P. Kornerup, R. T. Gregory, "Mapping Integers and Hensel Codes Onto Farey Fractions," BIT, vol. 23, 1983, pp. 9-20.

Chapter VII

AN ERROR-FREE FORM OF THE SCHUR ALGORITHM

In Chapter V we found that although the Schur algorithm is numerically stable, applying the algorithm to ill-conditioned input data can give poor results. In Chapter VI we considered various arithmetics for error-free computation, as error-free computation can be used to successfully handle ill-conditioned data. In Chapter VI the term "arithmetic" generally meant addition, subtraction and multiplication, but not division. This is because division is potentially quite troublesome. For example, if $a, b \in F_N$ then $\frac{a}{b} \notin F_N$ was possible (section 2.3 of Chapter VI). In addition, if the input data is integer-valued and from S_m (defined in Chapter VI, section 3), then division will yield non-integer (rational) results. Since we wish to map computations with data from S_m to residue number systems (SMRNS or MMRNS), the division operation will conflict with this requirement. Thus, we seek to modify the Schur algorithm of Chapter II to defer the division operation such that it may be performed under more convenient circumstances. The meaning of this will become clear as the reader studies this chapter. In this chapter we shall focus exclusively on the problem of LDU factorizing the Toeplitz matrix T_n, and the computation of its reflection coefficients. These results are taken from Zarowski and Card [1].

1. Options

The type of error-free computation number system to employ will depend at least in part on the elements of the Toeplitz matrix T_n. In a digital signal processing context, T_n will usually consist of elements that are fixed-point weighted binary numbers. Such numbers, if they contain fractional parts, can be scaled to become integers. Hence, in this case we can map the elements of T_n to $S_m \subset Z$, for m sufficiently large. A computation with the elements of S_m can then be mapped into a suitable residue number system (see Chapter VI, section 3).

On the other hand, if the elements of T_n are rational numbers that have no exact finite precision weighted binary number representation, it may be useful to map the elements of T_n to a suitable set of order-N Farey fractions. This will simply involve a suitable choice for p and r (subject to the constraint in Theorem 1 of Chapter VI, and the definition of F_N (Definition 1, Chapter VI)). The computation with elements of F_N may then be mapped into the finite ring Z_{p^r} (see Chapter VI, section 2). The division operation required by the Schur algorithm may be carried out by the computation of modulo p^r inverses, since we are mapping rational operands into the finite ring Z_{p^r}. However, such inverses won't always exist in Z_{p^r} as the number to be inverted might contain the factor p, although the likelihood of encountering a noninvertible operand decreases as p increases (see Thomas and Parker [2]). When mapping results in Z_{p^r} back to the Farey fractions, the answer will be in lowest terms (i.e., no common factor between numerator and denominator). However, if p^r is large, mapping from Z_{p^r} back to F_N will be a very onerous task. Fortunately, it is still possible to scale the entries of T_n such that it becomes a matrix of integers, and so the option in the previous paragraph remains open to matrices with rational entries. Note that to scale the entries of T_n ideally requires knowledge of the least common multiple of the denominators of t_i.

Recall that, as always, when mapping any computation into a finite ring or field it is necessary that the ring or field be large enough to contain the input data and the final results. It should also be clear that, if T_n is complex-valued, then the use of a QRNS (Chapter VI, section 4) becomes possible.

2. Error-Free Schur Algorithm: Nonsymmetric Toeplitz Matrix Input

We will assume with little loss of generality that T_n is a Toeplitz matrix of integers from S_m. This is reasonable given the discussion of the previous section. We are also assuming that T_n is not complex-valued, as this case will be discussed later on

(Hermitian Toeplitz case only). In addition, T_n will be nonsymmetric. Since we wish to LDU factorize T_n in an error-free manner with residue arithmetic, and division in a residue number system is difficult, we should like to rearrange the computations involved in executing the Schur algorithm so that division is avoided. We shall show how to do this here.

Since T_n is nonsymmetric we are considering the Schur algorithm for this case in Chapter II (see the pseudocode immediately following equation (69a,b)). Since the entries of T_n are integers we may write

$$v_i^{(k)} = \frac{\hat{v}_i^{(k)}}{c_k} \; , \quad u_i^{(k)} = \frac{\hat{u}_i^{(k)}}{d_k} \; , \tag{1}$$

where $\hat{v}_i^{(k)}$, $\hat{u}_i^{(k)}$, c_k and d_k are integers. It is therefore possible to write the inner-most For-do loop of the pseudocode as

$$\begin{bmatrix} \dfrac{\hat{v}_i^{(k+1)}}{c_{k+1}} \\ \dfrac{\hat{u}_i^{(k+1)}}{d_{k+1}} \end{bmatrix} = \begin{bmatrix} 1 & -\dfrac{d_k \hat{v}_{-k}^{(k)}}{c_k \hat{u}_{-k+1}^{(k)}} \\ -\dfrac{c_k \hat{u}_1^{(k)}}{d_k \hat{v}_0^{(k)}} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\hat{v}_i^{(k)}}{c_k} \\ \dfrac{\hat{u}_{i+1}^{(k)}}{d_k} \end{bmatrix} \; , \tag{2}$$

or equivalently as

$$\begin{bmatrix} \hat{v}_i^{(k+1)} \\ \hat{u}_i^{(k+1)} \end{bmatrix} = \begin{bmatrix} \hat{u}_{-k+1}^{(k)} & -\hat{v}_{-k}^{(k)} \\ -\hat{u}_1^{(k)} & \hat{v}_0^{(k)} \end{bmatrix} \begin{bmatrix} \hat{v}_i^{(k)} \\ \hat{u}_{i+1}^{(k)} \end{bmatrix} \; , \tag{3a}$$

$$c_{k+1} = c_k \hat{u}_{-k+1}^{(k)} \; , \quad d_{k+1} = d_k \hat{v}_0^{(k)} \; . \tag{3b}$$

It is clear that $c_1 = d_1 = 1$ and that

$$v_i^{(1)} = \hat{v}_i^{(1)} = t_{-i} \; , \quad u_i^{(1)} = \hat{u}_i^{(1)} = t_{-i} \; . \tag{3c}$$

It is also true that

$$K_k^f = -\frac{u_1^{(k)}}{v_0^{(k)}} = -\frac{c_k \hat{v}_1^{(k)}}{d_k \hat{v}_0^{(k)}} \; , \quad K_k^r = -\frac{v_{-k}^{(k)}}{u_{-k+1}^{(k)}} = -\frac{d_k \hat{v}_{-k}^{(k)}}{c_k \hat{u}_{-k+1}^{(k)}} \; . \tag{4}$$

Equation (3a,b,c) constitutes the unsimplified error-free form of the Schur algorithm. The final simplified form appears below. Equation (4) can be used to obtain the reflection coefficients as they are needed. This of course involves division, but this operation can be carried out after c_k , d_k , $\hat{v}_i^{(k)}$, and $\hat{u}_i^{(k)}$ are determined. It is clear that division has been deferred, but at the expense of additional multiplications.

It is possible to show that $c_k = d_k$ for all k, and so (1) and (3b) simplify. We have the following

Lemma ∘ $v_0^{(k)} = u_{-k+1}^{(k)}$ for all k.

Proof ∘ Follows simply from the fact that $T_n = L_n D_n^{-1} U_n$, where $L_n = L_n^* D_n$ and $U_n = D_n U_n^*$. The quantities $v_0^{(k)}$ and $u_{-k+1}^{(k)}$ lie on the main diagonals of L_n and U_n, respectively (see Chapter II, section 2.3).

From this Lemma we immediately deduce that $c_k = d_k$ for all k, and that $\hat{v}_0^{(k)} = \hat{u}_{-k+1}^{(k)}$. Thus, the *error-free form of the Schur algorithm* becomes:

```
$c_1 := 1$ ;
For $i := -n$ to $n$ do begin
    $\hat{u}_i^{(1)} := t_{-i}$ ; $\hat{v}_i^{(1)} := t_{-i}$ ;
    end;
For $k := 1$ to $n$ do begin
    $c_{k+1} := c_k \hat{u}_{-k+1}^{(k)}$ ;
    For $i := -n$ to $n$ do begin
        $\hat{u}_i^{(k+1)} := -\hat{u}_1^{(k)} \hat{v}_i^{(k)} + \hat{u}_{-k+1}^{(k)} \hat{u}_{i+1}^{(k)}$ ;
        $\hat{v}_i^{(k+1)} := \hat{u}_{-k+1}^{(k)} \hat{v}_i^{(k)} - \hat{v}_{-k}^{(k)} \hat{u}_{i+1}^{(k)}$ ;
        end;
    end;
For $k := 1$ to $n$ do begin
    $K_k^f := -\frac{\hat{u}_1^{(k)}}{\hat{v}_0^{(k)}}$ ; $K_k^r := -\frac{\hat{v}_{-k}^{(k)}}{\hat{u}_{-k+1}^{(k)}}$ ;
    end;
For $k := 1$ to $n$ do begin
    For $i := -n$ to $n$ do begin
        $v_i^{(k+1)} := \frac{\hat{v}_i^{(k+1)}}{c_{k+1}}$ ; $u_i^{(k+1)} := \frac{\hat{u}_i^{(k+1)}}{c_{k+1}}$ ;
```

```
            end;
        end;
```

Note that if we are not interested in obtaining the Schur variables $u_i^{(k)}$ and $v_i^{(k)}$, then we can omit the computation of c_k, and the computation of the last nested For-do loop. It may even be appropriate to keep the reflection coefficients in the form $\frac{a}{b}$, $(a, b \in S_m)$, and so division can be eliminated entirely in this case.

3. Error-Free Schur Algorithm: Symmetric Toeplitz Matrix Input

We shall assume that T_n is as it was in section 2, except that now it is symmetric.

Because T_n is symmetric, we must now consider the Schur algorithm for this special case. The relevant Schur algorithm is in Chapter II (section 2.3) between equations (77) and (78) (ignore the complex conjugation operations). Via the straightforward extension of the results in section 2 above, the *error-free form of the Schur algorithm* is:

```
$c_1 := 1$ ;
For $i := 0$ to $n$ do begin
    $\hat{a}_i^{(1)} := t_i$ ; $\hat{a}_{-i}^{(1)} := t_i$ ;
    end;
For $k := 1$ to $n$ do begin
    $c_{k+1} := c_k \hat{a}_{-k+1}^{(k)}$ ;
    For $i := 0$ to $n - k$ do begin
        $\hat{a}_{-(k+i)}^{(k+1)} := \hat{a}_{-k+1}^{(k)} \hat{a}_{-(k+i)+1}^{(k)} - \hat{a}_1^{(k)} \hat{a}_{i+1}^{(k)}$ ;
        $\hat{a}_i^{(k+1)} := -\hat{a}_1^{(k)} \hat{a}_{-(k+i)+1}^{(k)} + \hat{a}_{-k+1}^{(k)} \hat{a}_{i+1}^{(k)}$ ;
        end;
    end;
For $k := 1$ to $n$ do begin
    $K_k := - \frac{\hat{a}_1^{(k)}}{\hat{a}_{-k+1}^{(k)}}$ ;
```

```
        end;
    For $k$ := 1 to $n$ do begin
        For $i$ := 0 to $n - k$ do begin
            $u_i^{(k+1)} := \frac{\hat{a}_i^{(k+1)}}{c_{k+1}}$ ; $u_{-(k+i)}^{(k+1)} := \frac{\hat{a}_{-(k+i)}^{(k+1)}}{c_{k+1}}$ ;
        end;
    end;
```

The operations in the nested For-do loop used to compute $\hat{a}_i^{(k)}$ can be written in matrix form as

$$\begin{bmatrix} \hat{a}_{-(k+i)}^{(k+1)} \\ \hat{a}_i^{(k+1)} \end{bmatrix} = \begin{bmatrix} \hat{a}_{-k+1}^{(k)} & -\hat{a}_1^{(k)} \\ -\hat{a}_1^{(k)} & \hat{a}_{-k+1}^{(k)} \end{bmatrix} \begin{bmatrix} \hat{a}_{-(k+i)+1}^{(k)} \\ \hat{a}_{i+1}^{(k)} \end{bmatrix} . \tag{5}$$

Note that the 2×2 matrix in (5) is circulant. It can therefore be diagonalized by a 2-point DFT (discrete Fourier transform) and we can eliminate two multiplications. Thus,

$$\begin{bmatrix} \hat{a}_{-k+1}^{(k)} & -\hat{a}_1^{(k)} \\ -\hat{a}_1^{(k)} & \hat{a}_{-k+1}^{(k)} \end{bmatrix} = 2^{-1} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \hat{a}_{-k+1}^{(k)} - \hat{a}_1^{(k)} & 0 \\ 0 & \hat{a}_{-k+1}^{(k)} + \hat{a}_1^{(k)} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} . \tag{6}$$

If $\hat{a}_i^{(k)} \in Z_m$, then $2^{-1} \in Z_m$ if m contains no factor of value two. The benefit in using the factorization in (6) can be maximized if m is chosen such that 2^{-1} is a "nice" number, such as an integer power of two. For example, if $m = 2^{2n} + 1$, then $2^{-1} = -2^{2n-1}$. In a case like this, multiplication by 2^{-1} becomes shifting modulo m.

4. Error-Free Schur Algorithm: Hermitian Toeplitz Matrix Input

We will now assume that T_n is Hermitian with Gaussian integer entries of the form $x + iy$, $x, y \in S_m$. Straightforwardly, the *error-free form of the Schur algorithm* is now:

```
    $c_1 := 1$ ;
    For $i$ := 0 to $n$ do begin
        $\hat{a}_{-i}^{(1)} := t_i$ ; $\hat{a}_i^{(1)} := \overline{t_i}$ ;
    end;
```

```
For $k$ := 1 to $n$ do begin
    $c_{k+1} := c_k \hat{u}^{(k)}_{-k+1}$ ;
    For $i$ := 0 to $n - k$ do begin
        $\hat{u}^{(k+1)}_{-(k+i)} := \hat{u}^{(k)}_{-(k+i)+1} \hat{u}^{(k)}_{-k+1} - \bar{\hat{u}}^{(k)}_{i+1} \hat{u}^{(k)}_{1}$ ;
        $\hat{u}_i^{(k+1)} := -\bar{\hat{u}}^{(k)}_{-(k+i)+1} \hat{u}^{(k)}_{1} + \hat{u}^{(k)}_{i+1} \hat{u}^{(k)}_{-k+1}$ ;
    end;
end;
For $k$ := 1 to $n$ do begin
    $K_k := - \dfrac{\hat{u}^{(k)}_1}{\hat{u}^{(k)}_{-k+1}}$ ;
end;
For $k$ := 1 to $n$ do begin
    For $i$ := 0 to $n - k$ do begin
        $u_i^{(k+1)} := \dfrac{\hat{u}_i^{(k+1)}}{c_{k+1}}$ ; $u^{(k+1)}_{-(k+i)} := \dfrac{\hat{u}^{(k+1)}_{-(k+i)}}{c_{k+1}}$ ;
    end;
end;
```

Note that $c_k \in S_m$ for all k, as $\hat{u}^{(k)}_{-k+1} \in S_m$ for all k. This latter fact is consistent with

$$\hat{u}^{(k+1)}_{-k} = (\hat{u}^{(k)}_{-k+1})^2 - | \hat{u}^{(k)}_1 |^2 , \ \hat{u}^{(1)}_0 = t_0 \in S_m \ .$$

This is obtained from the above pseudocode. It is clear that computation in a QRNS will result in a significant savings in the number of multiplications to be performed. The nested For-do loop where $\hat{u}_i^{(k)}$ is computed involves 12 multiplications in the ring Z_{p^r} per iteration of the loop, if the computation is performed in a CRNS. Only 8 multiplications are required if the computation is performed in a QRNS.

The problem of developing an error-free form the of the Schur algorithm for Hermitian Toeplitz matrices of any rank profile (Chapter III) remains open. For the present, one can consider the elements of T_n to be of the form $x + iy$, $x,y \in F_N$, and map the computation into an extended QRNS (extended as in Chapter VI, section

4.2). As we have noted, this carries the risk of not being able to divide, as not all members of $Z_{p^r} \times Z_{p^r}$ have modulo p^r inverses. However, as we have also noted, this risk declines as p increases in value. Another open problem is that the back-substitution algorithms of Chapter IV must somehow be "integrated" into the Schur algorithm in an error-free manner. This is necessary to facilitate the error-free solution of Toeplitz systems of equations.

5. Size of Modulus Needed

How big should modulus m be ? Not only must m be big enough so that $t_k \in S_m$ for all k, but m must also be big enough so that $\hat{u}_i^{(k)}$, $\hat{v}_i^{(k)}$, $c_k \in S_m$ for all i and k. In this section we consider T_n to satisfy the assumptions of section 2.

We can use (3a) to arrive at a pessimistic upper bound for the values c_k , $\hat{v}_i^{(k)}$, and $\hat{u}_i^{(k)}$. From (3a)

$$| \hat{v}_i^{(k+1)} | \leq | \hat{u}_{-k+1}^{(k)} | \, | \hat{v}_i^{(k)} | + | \hat{v}_{-k}^{(k)} | \, | \hat{u}_{i+1}^{(k)} | \ , \tag{7a}$$

$$| \hat{u}_i^{(k+1)} | \leq | \hat{u}_1^{(k)} | \, | \hat{v}_i^{(k)} | + | \hat{v}_0^{(k)} | \, | \hat{u}_{i+1}^{(k)} | \ . \tag{7b}$$

Define

$$t = \max \left\{ | \, t_k \, | \right\} , \ t_k \in S_m \ . \tag{8}$$

Thus, $| \hat{v}_i^{(1)} |$, $| \hat{u}_i^{(1)} | \leq t$ for all i. From (7a,b) it is evident that

$$| \hat{v}_i^{(2)} | \leq t \; t + t \; t = 2t^2 \ ,$$

$$| \hat{u}_i^{(2)} | \leq t \; t + t \; t = 2t^2 \ ,$$

and similarly

$$| \hat{v}_i^{(3)} | \ , \ | \hat{u}_i^{(3)} | \leq 2 \ (2t^2)^2 \ ,$$

$$| \hat{v}_i^{(4)} | \ , \ | \hat{u}_i^{(4)} | \leq 2 \ (2 \ (2t^2)^2)^2 \ ,$$

and so on. In general it may be shown that

$$| \hat{v}_i^{(k)} | \, , \, | \hat{u}_i^{(k)} | \le t^{2^{i-1}} \prod_{l=0}^{k-2} 2^{2^l} = t^{2^{k-1}} \, 2^{2^{k-1}-1} \tag{9}$$

for $k \ge 2$. Since $c_{n+1} = \prod_{i=1}^{n} \hat{u}_{-i+1}^{(i)}$ (using the section 2 pseudocode) we can use (9) to write

$$| c_{n+1} | \le t \prod_{i=2}^{n} t^{2^{i-1}} \, 2^{2^{i-1}-1} \; . \tag{10}$$

We must have $| c_{n+1} | \le \dfrac{m-1}{2}$ and $| \hat{v}_k^{(n+1)} | \, , \, | \hat{u}_k^{(n+1)} | \le \dfrac{m-1}{2}$ as well. This gives

$$m > \max \left\{ 2t \prod_{i=2}^{n} t^{2^{i-1}} \, 2^{2^{i-1}-1} \, , \, 2t^{2^n} \, 2^{2^n-1} \right\} \; . \tag{11}$$

If we assume $n = 9$, $t = 2^7$ (not unusual values) then $m > 2^{4080}$ which implies that we need a dynamic range of at least 4080 bits. This is a very large value, and is likely to be larger than what is actually necessary. Thus, there is a need for a much tighter bound on m. An alternative means of determining m involves the use of computer simulations of the error-free form of the Schur algorithm. One can produce randomly generated sets of autocorrelation coefficients for the classes of input signals of interest, and run the error-free Schur algorithm using these data sets to estimate how large the Schur variables and parameters c_k actually are.

It is obvious that the large numbers produced by the error-free versions of the Schur algorithm will limit the size of matrix (n) that can be considered. Unfortunately, there is no way to avoid this problem. This is especially true of problems involving Hermitian Toeplitz matrices with singular leading principal submatrices, since in this case quantization error is completely intolerable (the problem is extremely ill-conditioned (reflection coefficients with unity magnitude are present) due to the need to test for equality to zero). In the case of Toeplitz matrices without singular leading principal submatrices, the conventional form of the Schur algorithm (Chapter II) applies. In this case, one could compromise by using error-free computation until the numbers reach some intolerably large size, and then the results could be scaled down to a reasonable size. Error-free computation could then resume until the numbers become intolerably large again. Clearly, quantization errors will be introduced in a scheme such as this, but the rate at which such errors accumulate would be greatly diminished, relative to conventional finite precision arithmetic implementations, when the input is badly ill-conditioned (reflection coefficients with a magnitude close to, but not equalling unity).

REFERENCES

[1] C. J. Zarowski, H. C. Card, "An Error-Free Form of the Schur Algorithm," to be submitted to the IEEE Trans. on Acoust., Speech, and Signal Proc.

[2] J. J. Thomas, S. R. Parker, "Implementing Exact Calculations in Hardware," IEEE Trans. on Comp., vol. C-36, June 1987, pp. 764-768.

Chapter VIII

SOME DESIGNS FOR COMPUTATION IN Z_{p^r}

In Chapters VI and VII we have often refered to computation in the finite ring Z_{p^r} . Since we are interested in the VLSI implementation of the error-free forms of the algorithms in Chapter VII, and these are to be implemented in the form of parallel processing systems similar to those in Chapter IV, we shall consider the hardware implementation of addition and multiplication in Z_{p^r} . Results on this subject are taken from Zarowski and Card [1] (see section 1 below). The problem of mapping integer data (i.e., data from Z) into Z_{p^r} was considered in Zarowski and Card [2] and will be presented here as well (see section 2).

1. Serial and Parallel Architectures for Addition and Multiplication in Z_{p^r}

Recall that Z_m denotes the finite ring of integers $\{0,1,...,m-1\}$ under modulo m addition and multiplication, and that it forms a so-called *single modulus residue number system* (SMRNS). As well, if $m = m_1 m_2 \cdots m_k$ and $gcd(m_i,m_j) = 1$ $(i \neq j)$ for every i and j, then Z_m can be made isomorphic to the direct product ring

$$Z_{m_1} \times Z_{m_2} \times \cdots \times Z_{m_k} \; . \tag{1}$$

The product ring in (1) is often refered to as a *multiple modulus residue number system* (MMRNS). If $x \in Z_m$ then x maps to a unique $(x_1, \cdots ,x_k) \in Z_{m_1} \times \cdots \times Z_{m_k}$, where $x_i = x \; (mod \; m_i)$. An element of the product ring may be mapped back to its corresponding unique member of Z_m via either the Chinese Remainder Theorem (CRT) (see [3,4]), or the mixed-radix number representation (MRNR) (see [5]).

If $(x_1, \cdots, x_k), (y_1, \cdots, y_k) \in Z_{m_1} \times \cdots \times Z_{m_k}$ then

$$(x_1, \cdots, x_k) \circledcirc (y_1, \cdots, y_k) = (z_1, \cdots, z_k) \in Z_{m_1} \times \cdots \times Z_{m_k}, \qquad (2a)$$

where '$\circledcirc$' represents addition, subtraction or multiplication in the product ring, and

$$z_i = x_i \circ y_i , \qquad (2b)$$

where '$\circ$' represents addition, subtraction or multiplication modulo m_i. Thus arithmetic in the product ring is performed component-wise, and no carry information is propagated from one component to the next. Recall that for this reason the multiple modulus residue number system has been proposed as a means of implementing certain digital signal processing (DSP) algorithms at high speed (such as in [6,7,8]).

We also know that the possibility of the high speed implementation of algorithms is certainly not the only reason to perform computations in a finite ring. The other main reason is that computation in any finite ring is exact; there is no quantization error. Thus, it is possible to implement algorithms with reduced, or even without, quantization error, depending upon the algorithm. For example, circular convolution algorithms based on NTTs terminate in a finite number of steps, and so it is possible to completely eliminate all quantization error with suitable scaling of the input data and suitable choice of ring (see [8]). IIR filtering algorithms never terminate in principle, and so intermediate results must be scaled to prevent overflow (see [7]). Overflow occurs when a result is produced that cannot be uniquely represented in the chosen ring. Overflow is refered to as pseudo-overflow in [5].

Some algorithms are numerically unstable in that quantization errors tend to accumulate as the computation proceeds, regardless of the nature of the input. Such unstable algorithms can be stabilized by using error-free computation with a finite ring. Other problems are ill-conditioned in that the final solution is highly sensitive to small perturbations in the input data. In particular, many linear algebra problems are of this class, such as the problems discussed in Chapter V. The use of error-free computation with finite rings applied to other linear algebra problems may be found in [5].

We have already alluded to the fact that overflow in an RNS system must be avoided. That is, we must select the ring Z_m such that all input data and final results are uniquely representable in Z_m (or its product ring equivalent in (1)). As a result of this requirement, it is possible that m may be extremely large, as we've seen in Chapter VII. Any number in Z_m can be represented by an η-bit binary number, where

$$\eta = \lceil \log_2 m \rceil \ . \tag{3}$$

As a rough guide, extremely large may be taken to mean that $\eta \geq 100$.

The need to compute in a large ring poses serious problems. Clearly, one solution is to isomorphically map our computational problem in Z_m to an equivalent problem in the product ring of (1). We thus conveniently break the problem into k independent, and smaller parts. Arithmetic modulo m_i is likely to be much simpler and faster than arithmetic modulo m since m_i is typically much smaller than m, and it is often possible to select convenient values for m_i.

Mapping a computation in Z_m to an equivalent computation in the product ring is not a panacea, however. For one thing, the inverse mapping from the product ring back to Z_m via the CRT, or even via the MRNR, is a difficult operation. The hardware complexity of these inverse mappings is rather high. Hardware implementations of the MRNR and the CRT are described in [9] and [6], respectively. In addition, if k is large (i.e., there are many mutually prime moduli), then many different modulo m_i arithmetic units must be designed. This is inconvenient, and potentially costly, in the context of VLSI designs, since it will be necessary to maintain a large library of different computational circuit blocks.

Many of the hardware designs proposed as a means of implementing arithmetic modulo m involve the use of lookup tables. These tables may be implemented using RAM or ROM (usually ROM). We refer the reader to references [6,11] for examples of this practice. As well, ways have been suggested of avoiding or at least reducing the use of lookup tables. We refer the reader to references [12,13] for examples. Arguments against the use of lookup tables are to be found in Taylor [12]. Although memory density and speed are high, the cost of fast and dense memory is also high enough to discourage their widespread use. Power dissipation is also a drawback. It is also possible to argue that, despite the existence of high density memory, the memory intensive approach to RNS system implementation is very expensive in terms of chip area requirements when compared with combinatorial logic implementations. This is likely to be significant in the context of the VLSI implementation of RNS systems.

In view of the preceding remarks it has been suggested that computation should occur in rings of the form Z_{p^r} (see Ramnarayan and Taylor [14], or Thomas and Parker [15,16] for example). Clearly, for a fixed p, unlimited dynamic range can be achieved by increasing r (a positive integer). If $x \in Z_{p^r}$ then x can be uniquely written as a radix-p number

$$x = x_0 + x_1 p + x_2 p^2 + \cdots + x_{r-1} p^{r-1} \quad , \tag{4}$$

where $x_i \in \{0,1,\ldots,p-1\}$. When the members of Z_{p^r} are expanded as in (4), arithmetic with such numbers is similar to arithmetic in a weighted binary arithmetic system, as has been noted in [14,15]. This is due to the propagation of carry information from digit to digit. The digits are the x_i variables in (4). We shall describe addition, negation and multiplication in Z_{p^r} in a later section of this chapter.

Because of the need to manage carry information, special purpose hardware for computation in Z_{p^r} is likely to be slower than equivalent special purpose hardware for computation in a product ring of similar size, at least provided that m is not too large. However, the relative ease of obtaining a large dynamic range in Z_{p^r} makes computation in Z_{p^r} highly attractive.

In the present chapter we propose serial and parallel architectures for the addition, subtraction (via negation), and multiplication modulo p^r of numbers in the form of (4). By serial architectures we mean processors that accept operands one digit x_i at a time, and which produce outputs one digit at a time. These architectures are similar to those presented in Lyon [17], and in Jackson, Kaiser and McDonald [18] for two's complement arithmetic. The resulting machines are highly modular, readily cascadeable, and the flow of data through them is highly regular. In the case of serial modulo

p^r multiplication, the cells making up the multiplier only communicate with their nearest neighbors. Thus, the machines that we propose are highly suitable for VLSI implementation. By virtue of their serial design, their main drawback is a high latency and low throughput (see section 1.5). That is, it will take $O(r)$ clock cycles to compute any new sum or product (the length of a clock cycle will depend upon p). It is possible, however, to obtain a reasonable throughput via pipelining, at least for some applications. As well, one should note that present technology places rather stringent upper limits on the number of I/O pins that a chip may have, and on the size of parallel-input, parallel-output ALU that can realistically be built. These constraints will cause us to favour the use of serial machines despite their slow speed, especially if m is to be extremely large, and the desire for numerical stability and the suppression of ill-conditioned data effects, takes precedence over operating speed.

In this thesis we shall consider p to have the form $2^n \pm 1$ since it is generally accepted that such numbers yield the most practical modulo p arithmetic units, other than the choice of $p = 2^n$ (which is trivial). Furthermore, none of our designs make use of lookup tables. Thus, we eliminate the drawbacks of lookup table based designs previously cited.

We shall also present asymptotic area and time complexity estimates for the designs proposed, and compare them to the asymptotic area and time complexities of certain parallel modulo p^r arithmetic processors; the latter are also to be described in this chapter.

1.1 Hardware for Modulo p Arithmetic

It is necessary to have hardware that can perform modulo p arithmetic before one can construct modulo p^r arithmetic hardware. In this section we present hardware structures for addition, negation and multiplication modulo p when $p = 2^n \pm 1$ (n is a positive integer) and $p \geq 3$.

Let us first consider modulo p addition. Let a_i , $b_i \in Z_p$ and $c_i \in \{0,1\}$. We can write

$$a_i + b_i + c_i = s_i + c_{i+1}p \quad , \tag{5}$$

where $c_{i+1} \in \{0,1\}$, and $0 \le s_i < p$. This follows from the division algorithm for integers. Thus, $s_i = a_i + b_i + c_i \ (mod\ p)$. Later on, c_i and c_{i+1} shall represent carries into and out of the ith position of a parallel modulo p^r adder, respectively. This is the reason for the use of the i subscript.

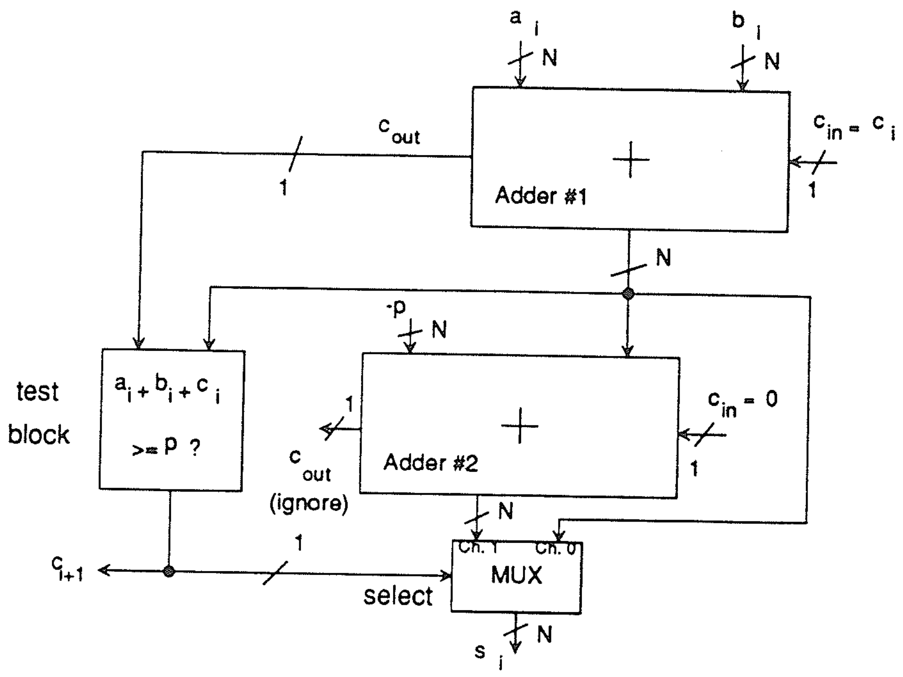


Figure 1: Modulo p adder with carry output (c_{i+1}).

If $p = 2^n + 1$ then a_i, b_i and s_i can be represented as $N = n+1$ bit positive integers. Similarly, if $p = 2^n - 1$ then a_i, b_i and s_i can be represented as $N = n$ bit positive integers. In this chapter all operand and result digits shall be so represented.

Figure 1 depicts a modulo p adder similar to the offset adders described in Taylor [13]. The modulo p adder is composed of two ordinary adders (Adder #1 and Adder #2) which are symbolized in this and all following figures by a box with a plus sign (+). The carry out of Adder #1 and the N-bit output of it are tested by the test block to determine whether or not $a_i+b_i+c_i \ge p$. If $a_i+b_i+c_i \ge p$ then $c_{i+1} = 1$, otherwise $c_{i+1} = 0$. The N-bit output of Adder #1 is added to $-p$ (= two's complement of p) using Adder #2. The output of Adder #2 feeds into the Channel 1 input of the multiplexer (MUX box). The Channel 0 input of the MUX is the N-bit output of

Adder #1. Channel 1 is selected if $c_{i+1} = 1$, otherwise Channel 0 is selected. The MUX output is s_i, while the test block output is of course c_{i+1}.

$a_i + b_i + c_i$	$a_i + b_i + c_i \ (mod\ p)$
0000	000
0001	001
0010	010
0011	011
0100	100
0101	000
0110	001
0111	010
1000	011
1001	100

Table I: Table of all possible values taken on by a_i+b_i+c_i for $p = 2^2 + 1$.

A study of Tables I and II yields the test blocks depicted in Figure 2: Figure 2(a) shows the test block for $p = 2^n + 1$, and Figure 2(b) shows the test block for $p = 2^n - 1$. If $p = 2^n + 1$ then a_i+b_i+$c_i \geq p$ if bit n+1 is 1, or bit n is 1, provided that a_i+b_i+$c_i \neq 2^n$. Note that the bits are indexed from 0 (LSB) to n+1 (MSB). This convention shall be followed throughout this chapter. The large n+1 bit AND gate with the inputs inverted detects the special case where a_i+b_i+$c_i = 2^n$ and forces $c_{i+1} = 0$. Input n+1 of the test block is the carry output c_{out} from Adder #1. If $p = 2^n - 1$ then a_i+b_i+$c_i \geq p$ if bit n is 1, or if a_i+b_i+$c_i = p$. The n-input AND gate detects the special case where a_i+b_i+$c_i = p$ and forces $c_{i+1} = 1$. Input n of the test block is c_{out} from Adder #1.

Let us now consider modulo p multiplication. Figure 3 depicts a modulo p multiplier for $p = 2^n + 1$. Figure 4 depicts a modulo p multiplier for $p = 2^n - 1$.

$a_i + b_i + c_i$	$a_i + b_i + c_i \ (mod\ p)$
0000	000
0001	001
0010	010
0011	011
0100	100
0101	101
0110	110
0111	000
1000	001
1001	010
1010	011
1011	100
1100	101
1101	110

Table II: Table of all possible values taken on by $a_i+b_i+c_i$ for $p = 2^3 - 1$.

Any product of $a, b \in Z_p$ may be written in the form

$$ab = c + xp \qquad (6)$$

where $0 \leq c < p$, and $0 \leq x \leq p-2$. Thus, we shall want a circuit that produces x and c as outputs for the inputs a and b. The circuits of Figures 3 and 4 perform this function.

Let us consider Figure 3 first. The modulo p multiplier consists of an ordinary positive integer multiplier, which is symbolized by a box with a multiplication sign ($\times$), a modulo p adder of the type in Figure 1, which is symbolized by a box with a ring-sum sign ($\oplus$), a modulo p negator (to be described later on), which is symbolized by a box with NEG written in it, an ordinary subtractor, which is a box with a

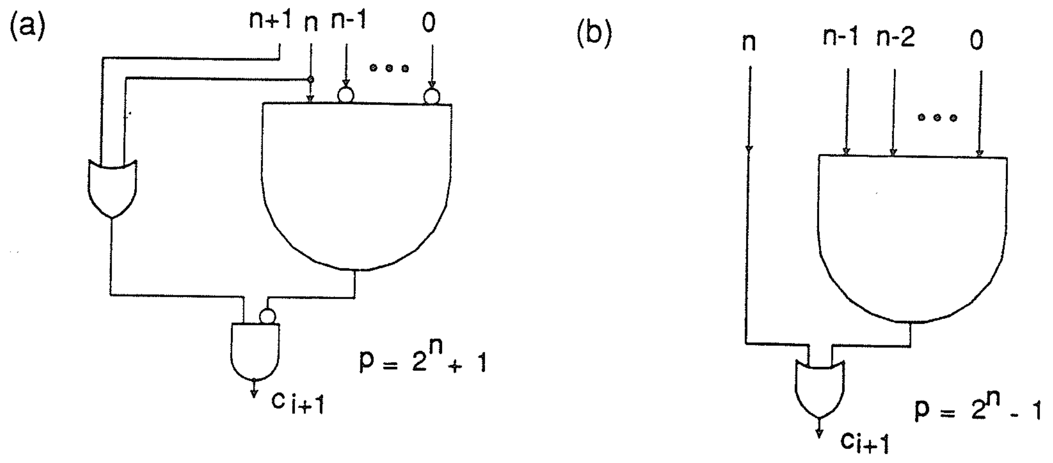


Figure 2: (a) Test block for the adder of Figure 1 ($p = 2^n + 1$); (b) Test block for the adder of Figure 1 ($p = 2^n - 1$).

minus sign (-), and some additional logic circuitry. We may write the binary (radix-2) expansion of product ab as

$$ab = \rho = \rho_{2n+1}\rho_{2n}\rho_{2n-1} \cdots \rho_{n+1}\rho_n\rho_{n-1} \cdots \rho_1\rho_0 , \tag{7}$$

where $\rho_i \in \{0,1\}$. We may define certain positive integers

$$P_H = \rho_{2n} , \tag{8a}$$

$$P_M = \rho_{2n-1} \cdots \rho_n , \tag{8b}$$

$$P_L = \rho_{n-1} \cdots \rho_0 . \tag{8b}$$

We interpret ρ_n to be the LSB of positive integer P_M and ρ_{2n-1} to be the MSB of it. A similar interpretation applies to P_L. From McClellan and Rader [3] (see pp. 14-15)

$$c = P_L - P_M \ \ (mod\ p) , \tag{9}$$

unless $P_H = 1$ in which case $c = 1$. The combinational logic of Figure 3 accounts for this latter special case. Clearly, $x = \lfloor \frac{ab}{p} \rfloor$ and

$$xp = ab - ab\ (mod\ p) = ab - c\ . \tag{10}$$

It is straightforward to show that $x = P_L - c$. The multiplier of Figure 3 is similar to the multiplier in Figure 6 of [13].

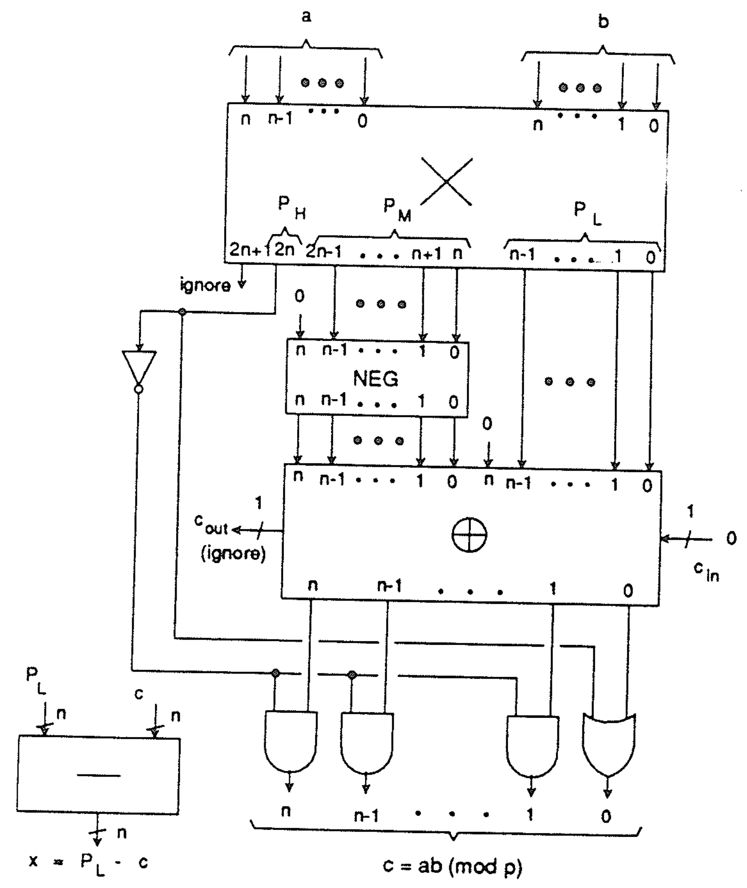


Figure 3: Modulo $p = 2^n + 1$ multiplier with quotient output (x).

Let us now consider Figure 4. The notational conventions of this figure are the same as those of Figure 3. We may write the weighted binary expansion of ab as

$$ab = \rho = \rho_{2n}\rho_{2n-1} \cdots \rho_{n+1}\rho_n\rho_{n-1} \cdots \rho_0 \tag{11}$$

and, as in (8a,b,c), we may define the positive integers

$$P_H = \rho_{2n-1} \cdots \rho_n \ , \tag{12a}$$

$$P_L = \rho_{n-1} \cdots \rho_0 \ . \tag{12b}$$

Once again, from pp. 14-15 of [3],

$$c = P_L + P_H \ \ (mod\ p) \ , \tag{13}$$

and it is straightforward to show that $x = c - P_L$.

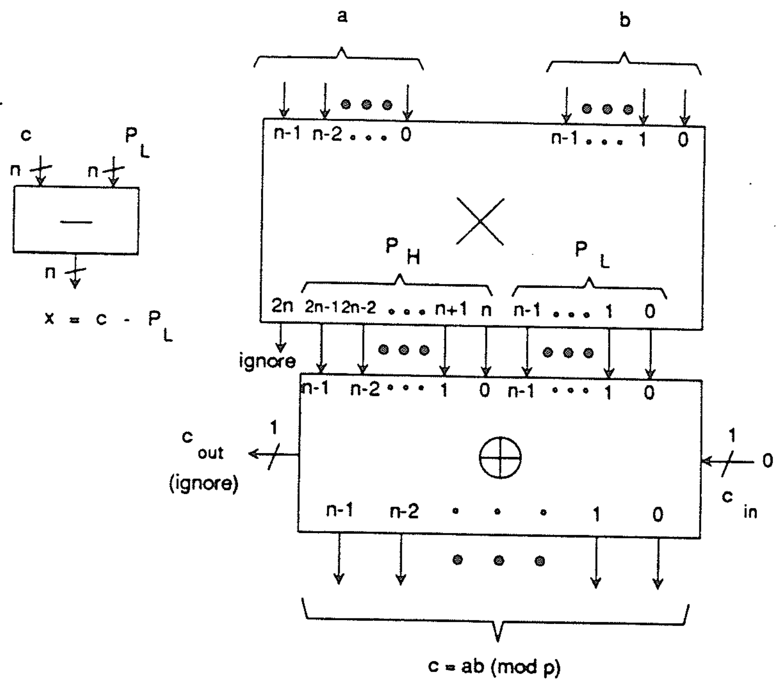


Figure 4: Modulo $p = 2^n - 1$ multiplier with quotient output (x).

Finally, let us consider modulo p negation since it is needed by the multiplier of Figure 3. Let $A \in Z_p$ and its binary expansion is

$$A = A_n A_{n-1} \cdots A_1 A_0 \ . \tag{14}$$

We have

$$-A \ (mod \ p) = p - A = (2^n + 1) + \overline{A} + 1 = 2^n + \overline{A} + 2 \ , \tag{15}$$

where $\overline{A}$ is the bit-wise complement of A in (14). If we let $S = \overline{A} + 2$, then it may be readily seen that, for $n = 4$, and $A \neq 0$,

$$\begin{aligned} S_0 &= \overline{A}_0 \ , \\ S_1 &= \overline{A}_1 \oplus 1 = A_1 \ , \\ S_2 &= \overline{A}_2 \oplus \overline{A}_1 \ , \\ S_3 &= \overline{A}_2 \overline{A}_1 \oplus \overline{A}_3 \ , \\ S_4 &= \overline{A}_3 \overline{A}_2 \overline{A}_1 \oplus \overline{A}_4 \ , \end{aligned} \tag{16}$$

where ' $\oplus$ ' denotes the exclusive-OR operation here. If $A = 0$ we want $-A = 0$, and

and $0 \le s_i < p$, c_{i+1} is the carry out of digit position i, $c_0 = 0$, and we ignore c_r. As in (5), $c_i \in \{0,1\}$ for all i.

Now let us consider negation. Let a be as in (18) but where $a_i = 0$ for $i = 0,\ldots,e-1$ (if $e = 0$ then $a_0 \ne 0$). Let b be as in (18) but where $b = -a \ (mod\ p^r)$. Thus, $b_i = 0$ for $i < e$, $b_e = p - a_e$, and $b_i = (p-1) - a_i$ for all $i > e$.

Finally, let us consider multiplication. With a and b as in (18),

$$ab\ (mod\ p^r) = \left[\sum_{j=0}^{r-1} a_j p^j\right] \left[\sum_{j=0}^{r-1} b_j p^j\right] = \sum_{j=0}^{r-1} p_j p^j \ , \tag{20a}$$

where

$$\sum_{k=0}^{j} a_k b_{j-k} + r_j = p_j + r_{j+1} p \quad (0 \le j \le r-1) \ , \tag{20b}$$

and $0 \le p_j < p$, r_{j+1} is the carry out of digit position j, $r_0 = 0$ and we ignore r_r.

1.3 Parallel Hardware for Modulo p^r Arithmetic

In this section we describe certain parallel modulo p^r adders and multipliers. Specifically, we describe a parallel modulo p^r adder that is analogous to a parallel binary adder, where the latter is essentially the same as the adder depicted in Figure 2(a) of [14]. However, we also present a carry-lookahead array for it, and we show that it is possible to develop a Brent-Kung [19] modulo p^r adder. In addition we present a ripple-through modulo p^r array multiplier, and a pipelined modulo p^r array multiplier of a design similar to the pipelined positive integer multiplier in Figure 1 of McCanny and McWhirter [20]. These designs will later be compared, in terms of asymptotic area and time complexities, with the serial designs of the next section.

From the modulo p^r addition algorithm of (19a,b) we obtain the ripple-through parallel modulo p^r adder array of Figure 6(a). The structure of Figure 6(a) is a linear array of modulo p adders of the type shown in Figure 1. As in Figures 3 and 4, modulo p adders are depicted as a box with a ring-sum sign ($\oplus$) written in it. It is evident that the ripple-through adder will be quite slow. Thus, a carry-lookahead unit may prove to be a useful way of speeding up the summation process. Naturally, one so this yields the logic network of Figure 5(a). The generalization to arbitrary n is straightforward. The negator of Figure 5(a) is essentially that of Figure 4 in [13].

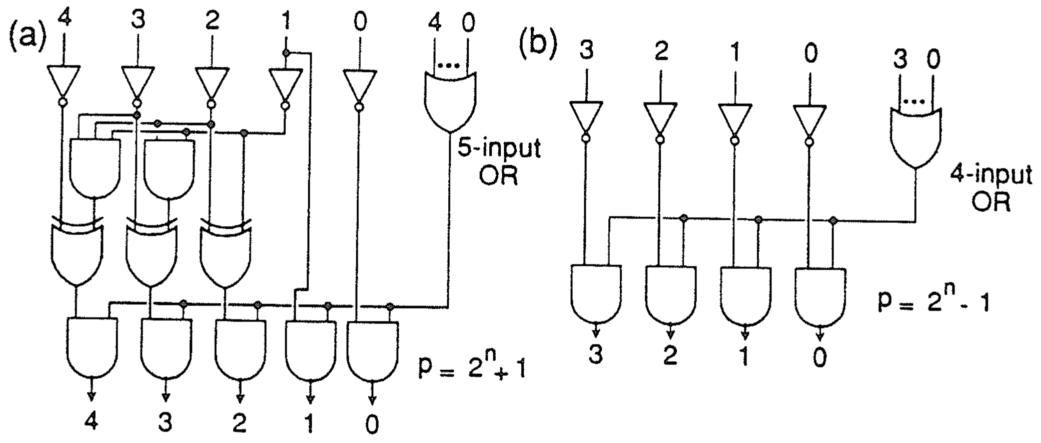


Figure 5: (a) Modulo $p = 2^4 + 1$ negator; (b) Modulo $p = 2^4 - 1$ negator.

For the sake of completeness, a modulo $(2^n - 1)$ negator is illustrated in Figure 5(b) for the special case of $n = 4$. Once again, if $A \in Z_p$, then

$$-A \ (mod\ p) = p - A = (2^n - 1) + \bar{A} + 1 = 2^n + \bar{A} \tag{17}$$

which implies that $-A = \bar{A}$, unless $A = 0$ in which case $-A = 0$.

1.2 Arithmetic in Z_{p^r}

In this section we describe addition, negation and multiplication modulo p^r when the members of Z_{p^r} are represented as in (4). It will then become possible to discuss hardware structures for modulo p^r arithmetic.

First we consider addition. Let

$$a = \sum_{i=0}^{r-1} a_i p^i \ , \ b = \sum_{i=0}^{r-1} b_i p^i \tag{18}$$

be elements of Z_{p^r}. Then

$$a + b \ (mod\ p^r) = s = \sum_{i=0}^{r-1} s_i p^i \ , \tag{19a}$$

where

$$a_i + b_i + c_i = s_i + c_{i+1} p \tag{19b}$$

could also use carry-lookahead units of conventional design to speed up the summation process within the modulo p adder cells themselves.

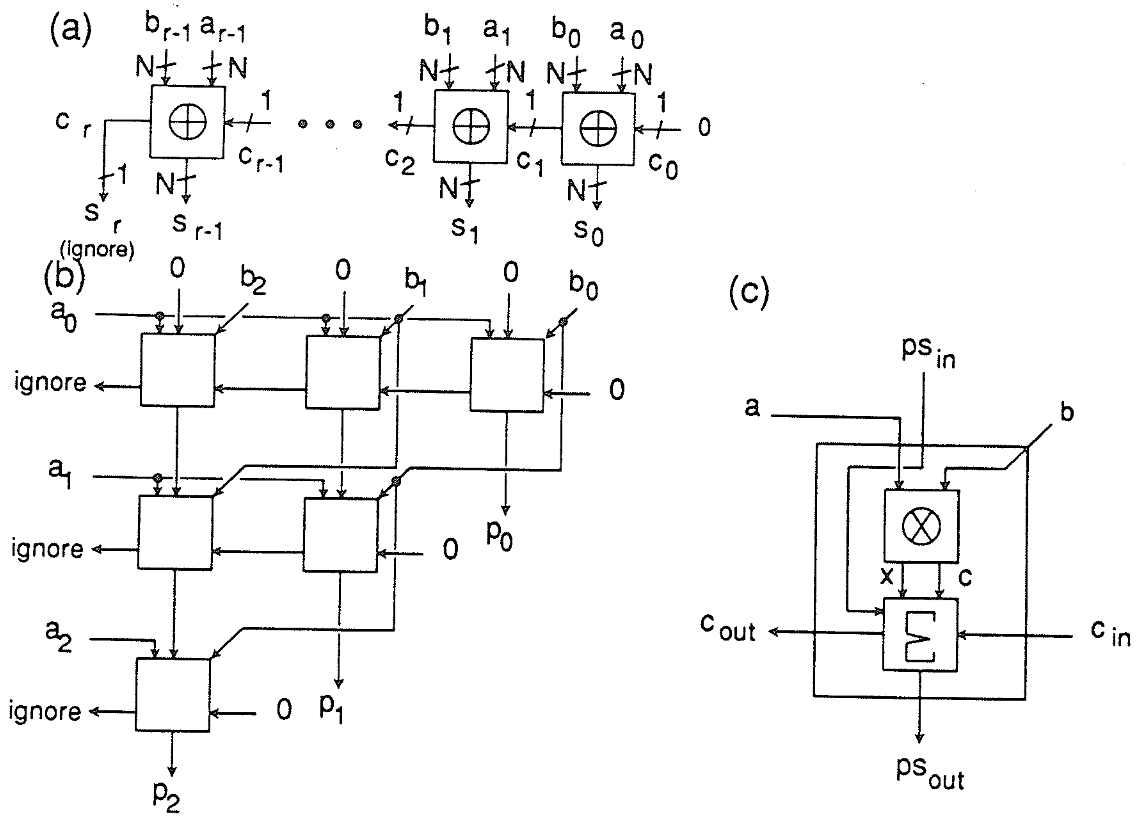


Figure 6: (a) Ripple-through parallel modulo p^r adder; (b) Unpipelined modulo p^r array multiplier; (c) basic cell of (b).

Define the test function

$$t_\alpha(x,y) = \begin{cases} 1, & x+y \geq \alpha \\ 0, & x+y < \alpha. \end{cases} \tag{21}$$

This function will be used to produce signals analogous to carry propagate and carry generate signals. We have

$$c_0 = 0$$

$$c_1 = t_p(a_0,b_0)$$

$$c_2 = t_p(a_1,b_1) \cup (t_{p-1}(a_1,b_1) \cap c_1) \tag{22}$$

$$c_{i+1} = t_p(a_i,b_i) \cup (t_{p-1}(a_i,b_i) \cap c_i)$$

where a and b are as in (18) and we are of course computing $s = a + b$. In the expressions for c_i, $\cup$ means logical OR and $\cap$ means logical AND. Thus,

$$g_i = t_p(a_i,b_i) \ , \qquad (23)$$

$$p_i = t_{p-1}(a_i,b_i) \ ,$$

are the carry generate and propagate conditions at position $i+1$, respectively. If $r = 4$, then

$$c_0 = 0$$

$$c_1 = t_p(a_0,b_0)$$

$$c_2 = t_p(a_1,b_1) \cup (t_{p-1}(a_1,b_1) \cap t_p(a_0,b_0))$$

$$c_3 = t_p(a_2,b_2) \cup (t_{p-1}(a_2,b_2) \cap t_p(a_1,b_1)) \cup (t_{p-1}(a_2,b_2) \cap t_{p-1}(a_1,b_1) \cap t_p(a_0,b_0))$$

which gives us the carry-lookahead array of Figure 7.

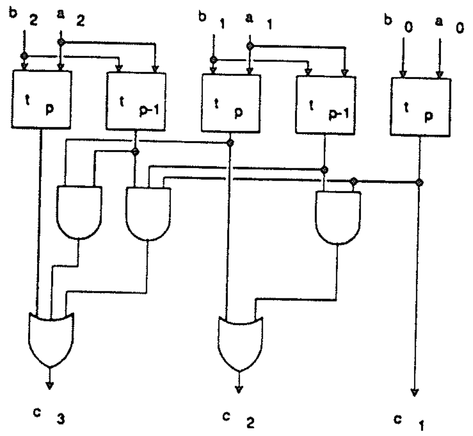


Figure 7: Carry-lookahead circuit for the adder in Figure 6(a).

The Brent-Kung [19] parallel adder can be adapted to the present modulo p^r summation problem. The o-operator of [19] is

$$(g,p)\text{o}(g',p') = (g \cup (p \cap g'), p \cap p') .$$

Note that the prime ($'$) does not denote logical complement. Symbols $\cup$ and $\cap$ have the same meanings that they did in (22). There is little danger of confusion between the carry propagate p and the p in Z_{p^r} . The following lemmas apply to this operator:

Lemma 1 • Let

$$(G_i,P_i) = \begin{cases} (g_0,p_0) & \text{if } i = 0 \\ (g_i,p_i)\text{o}(G_{i-1},P_{i-1}) & \text{if } 1 \le i \le r-1 \; . \end{cases}$$

Then

$$c_{i+1} = G_i$$

for $i = 0,\ldots,r-1$.

Lemma 2 • The o-operator is associative.

These lemmas may be proven in exactly the same manner as their counterparts in [19] and so we shall not present the proofs here.

The above lemmas yield the structure of Figure 8 (which is essentially the same as Figure 5 in [19]) for $r = 8$. Note, however, that the left-most column of processors that produce c_8 may be eliminated as c_8 is not needed when $r \doteq 8$. Variables g_i and p_i are as defined in (23). The black and white processors, described at the bottom of Figure 8, perform the same operations as their counterpart black and white processors in Figure 4 of [19].

Figure 6(b) is a ripple-through modulo p^r array multiplier (for $r = 3$). The product output is $\sum_{i=0}^{2} p_i p^i$. This multiplier is adapted from the positive integer array multiplier depicted in Figure 8.30 of Rabiner and Gold [21]. However, its cell complexity is much higher than that of the positive integer array multiplier of [21]. The cell that makes up the modulo p^r multiplier is shown in Figure 6(c). This cell is itself composed of a modulo p multiplier of the type shown in Figures 3 or 4, and of a cell denoted by a box with a sigma sign (Σ). This sigma-cell will be described in greater

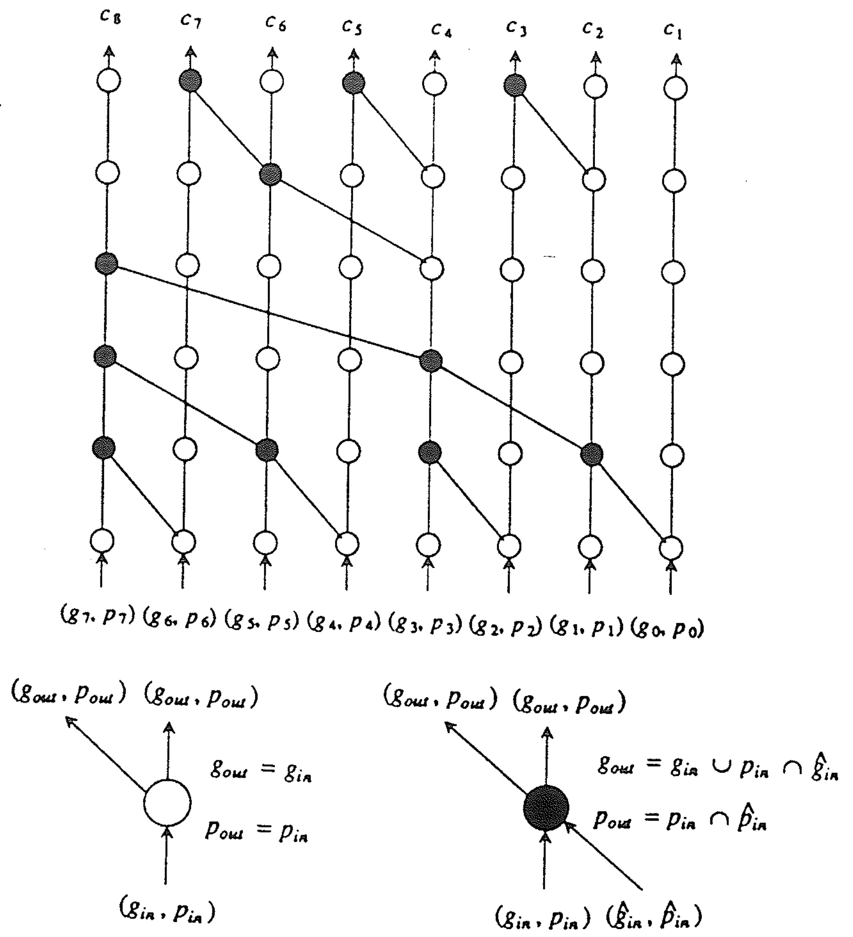

c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1
(g_7, p_7) (g_6, p_6) (g_5, p_5) (g_4, p_4) (g_3, p_3) (g_2, p_2) (g_1, p_1) (g_0, p_0)
(g_{out}, p_{out}) (g_{out}, p_{out})
$g_{out} = g_{in}$
$p_{out} = p_{in}$
(g_{in}, p_{in})
(g_{out}, p_{out}) (g_{out}, p_{out})
$g_{out} = g_{in} \cup p_{in} \cap \hat{g}_{in}$
$p_{out} = p_{in} \cap \hat{p}_{in}$
(g_{in}, p_{in}) $(\hat{g}_{in}, \hat{p}_{in})$

Figure 8: Brent-Kung carry generation array for the adder in Figure 6(a).

detail shortly.

It is obvious that the ripple-through array multiplier will be very slow in terms of throughput. However, the throughput can be considerably increased (and made independent of r) by pipelining the array multiplier in the manner depicted in Figure 9. This pipelined modulo p^r array multiplier is similar to the pipelined positive integer multiplier of Figure 1 in [20]. The darkened circles (•) of Figure 9 are the latches (delays) used to achieve pipelining. It is clear that the throughput of the pipelined multiplier is determined by the propagation delay of signals through the cells of which it is composed. This delay is a function of p but is independent of r. The cells making up the multiplier of Figure 9 are identical to those making up the multiplier in Figure 6(b).

It is now appropriate to discuss the sigma-cell in Figure 6(c). Let us consider the case where a and b are as in (18) but $r = 3$. We can write the modulo p^3 product of

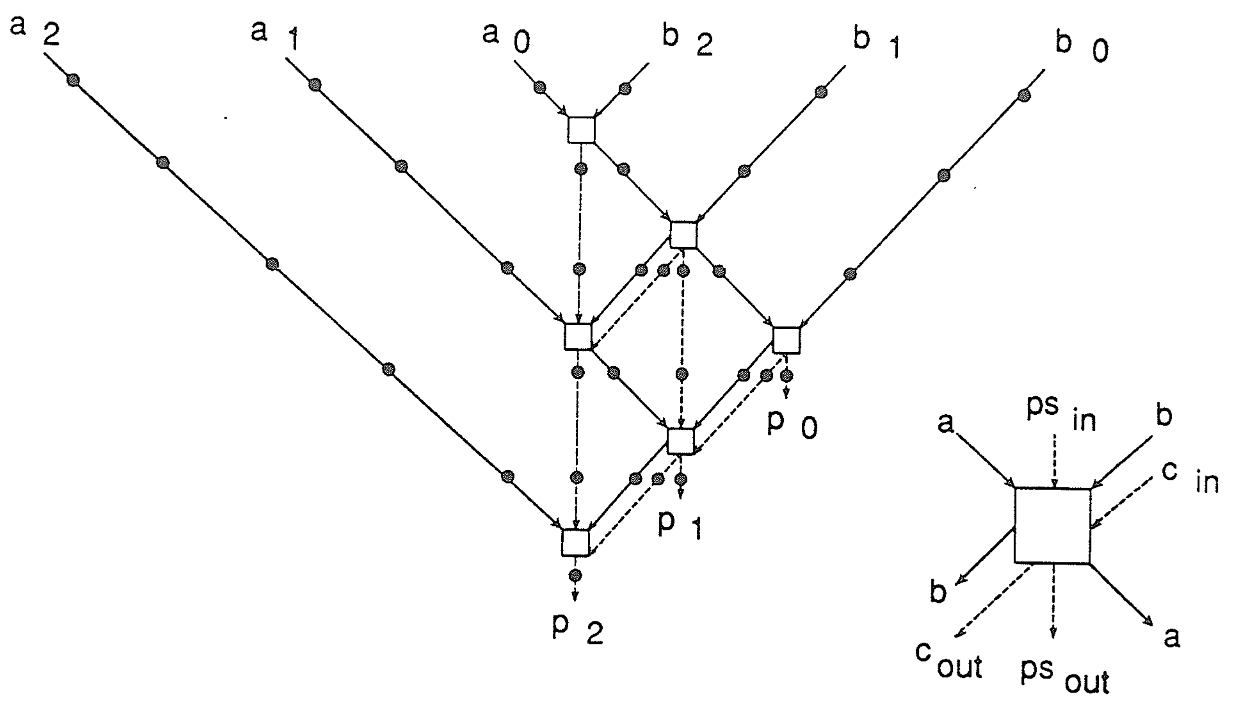


Figure 9: Pipelined modulo p^r array multiplier.

a and b in "pencil- and-paper" fashion as follows:

	b_2	b_1	b_0
$\otimes$	a_2	a_1	a_0
	a_0b_2	a_0b_1	a_0b_0
	a_1b_1	a_1b_0	
	a_2b_0		
	p_2	p_1	p_0

This product should be compared with the structure of the array multiplier in Figure 6(b). This is because the array multiplier approach to multiplication closely follows the pencil-and-paper method. The product components p_i are obtained as described in (20b). The sigma-cell sums the product ab, the carry out from the column of cells to the right c_{in}, and the partial sum out from the row of cells immediately above, ps_{in}. The sum of all c_{in}'s that contribute to p_j is equal to r_j in (20b). Thus, the sum computed by the sigma-cell is

$$ab + c_{in} + ps_{in} = ps_{out} + c_{out}p \ , \tag{24}$$

where ps_{out} and c_{out} are the output from the sigma-cell. As in section 2, the modulo p multiplier, symbolized by a box with a ring-product sign ($\otimes$), produces outputs x and c from the inputs a and b, where as before

$$ab = c + xp \ . \tag{25}$$

We have

$$ps_{out} = ps_{in} + c_{in} + c \quad (mod\ p) \ , \tag{26a}$$

$$c_{out} = x + \lfloor \frac{ps_{in} + c_{in} + c}{p} \rfloor \ . \tag{26b}$$

The sigma-cell computes ps_{out} and c_{out} according to (26a,b), and is depicted in Figure 10. In Figure 10 we have the following correspondences between variables:

$$\begin{aligned} x &= x_i \ , \\ ps_{out} &= ps_{i+1} \ , \\ ps_{in} &= ps_i \ , \\ c &= p_i \ , \\ c_{in} &= c_i \ , \\ c_{out} &= c_{i+1} \ . \end{aligned} \tag{27}$$

Note that the box with the HA written in it is an ordinary half-adder. It uses the carry outputs of the two modulo p adders to compute $\lfloor \frac{ps_{in} + c_{in} + c}{p} \rfloor$. The ordinary $\overline{N}$-bit adder then sums the half-adder output and x in order to compute c_{out} (c_{i+1} in the figure). We can have $\overline{N} = N$ or, alternatively, if $p = 2^n + 1$ we can have $\overline{N} = n$ (instead of $\overline{N} = n + 1$) and use the carry out of the ordinary adder.

1.4 Serial Hardware for Modulo p^r Arithmetic

In this section we present architectures for serial modulo p^r addition, negation and multiplication. The machines to be described are modifications of the serial weighted binary arithmetic schemes presented in [17] and [18].

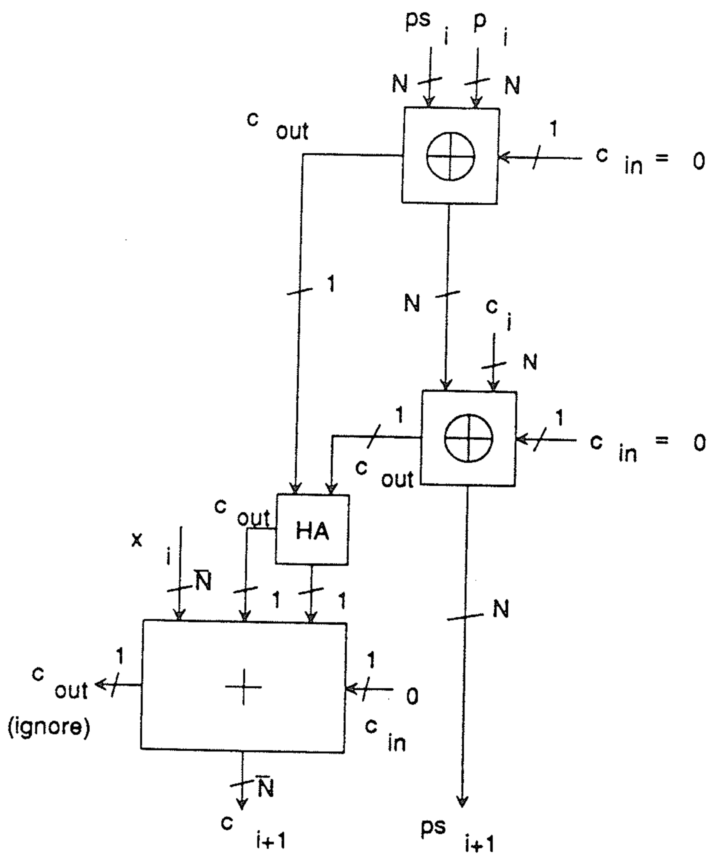


Figure 10: Sigma-cell of Figure 6(c).

Figure 11(a) depicts the serial modulo p^r adder. It is composed of a single modulo p adder of the type in Figure 1, and of a single bit of storage to hold carry outputs from the current digit position. The delay element is denoted by a box with a triangle (Δ). Naturally, the operands enter the machine in a digit serial fashion, least significant digit (a_0,b_0) first. The output also appears in digit serial fashion, least significant digit (s_0) first.

Figure 11(b) depicts the serial modulo p^r negator. It is based upon the negation algorithm of section 3. Table III specifies the operation of the control box. The operands enter the negator in digit serial fashion, least significant digit (a_0) first, and the result leaves the negator in digit serial fashion, least significant digit (b_0) first. The initial state of the control unit is zero (0).

Figure 12 illustrates the serial modulo p^r multiplier. Figure 12(a) shows the inner details of the r identical cells making up the serial multiplier, which are

Z	*present state*	*switch position* (A,B,C)	*next state*
0	0	C	0
0	1	B	1
1	0	A	1
1	1	B	1

Table III: State transition table for the control box of the serial modulo p^r negator.

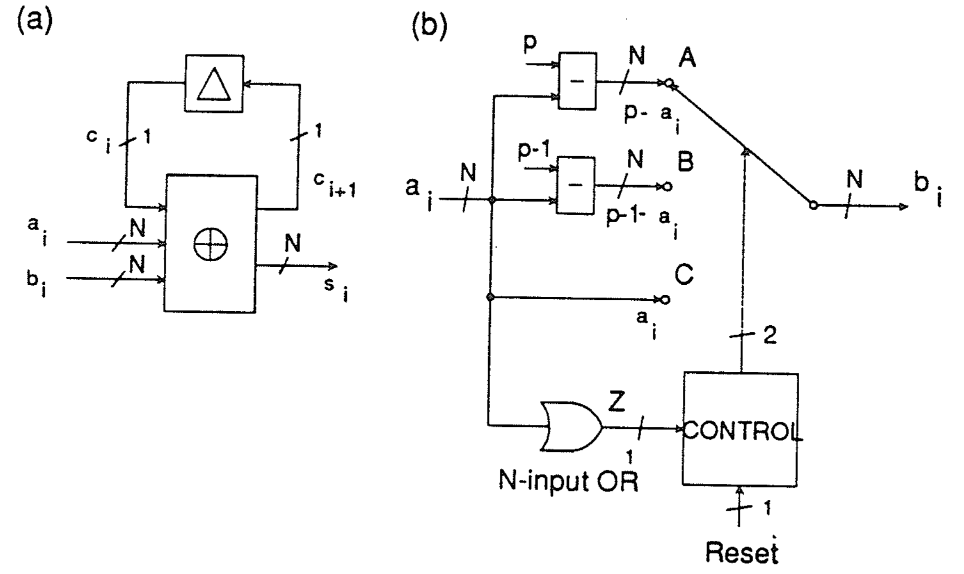


Figure 11: (a) Serial modulo p^r adder; (b) Serial modulo p^r negator.

connected in the manner indicated in Figure 12(b). The darkened circles of Figure 12(b) denote the optional latches (delays) for pipelining.

Each cell consists of a modulo p multiplier of the type in Figures 3 or 4, a sigma-cell of the type in Figure 10, plus some switching circuitry and storage. Registers A_i and B_i save the digits of the operands which are input digit-wise, least significant digit first, at the ports labeled α_0 and β_0 $(ps_0 = 0)$. The product output appears in digit serial fashion least significant digit first at ps_r. Register Q_i is a single bit control register, the state of which determines the position of the switches indicated in the cell. Register C_i saves the carry outputs that make up the variables labeled r_j

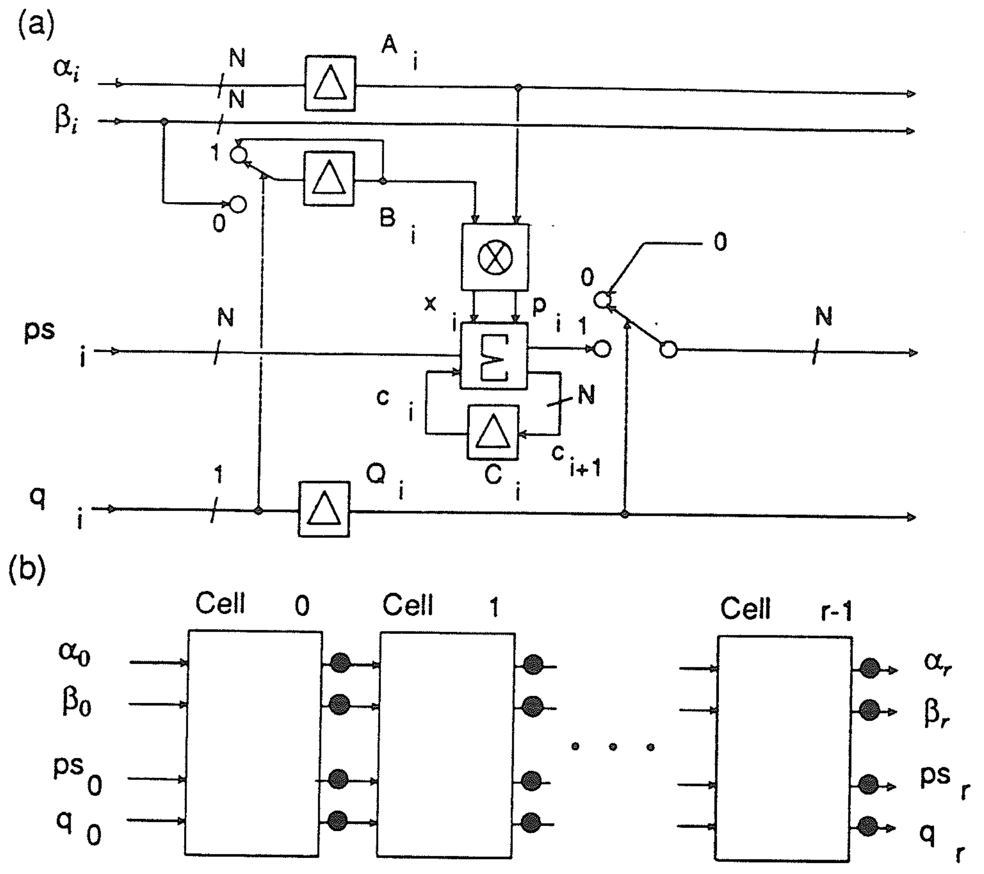


Figure 12: (a) Basic cell of the serial modulo p^r multiplier; (b) Serial modulo p^r multiplier as a cascade of the cells in Figure 12(a).

in (20b). Figure 13 shows the flow of data and control through an $r = 3$ cell machine. The contents of registers C_0, C_1 and C_2 are not shown. Figure 13 assumes ripple-through operation. Because of the relatively high time complexities of the modulo p multiplier and sigma-cells, pipelining of the type depicted in Figure 12(b) will likely be essential in practice.

1.5 Asymptotic Area and Time Complexities

In this section we evaluate the relative costs of the previously described designs in terms of their asymptotic area and time complexities. We shall use the definitions of *functional latency* (T), and *functional period* (P) found in Capello and Steiglitz [22,23]. We refer the reader to references [22,23] for the definition statements. We shall denote the area by A.

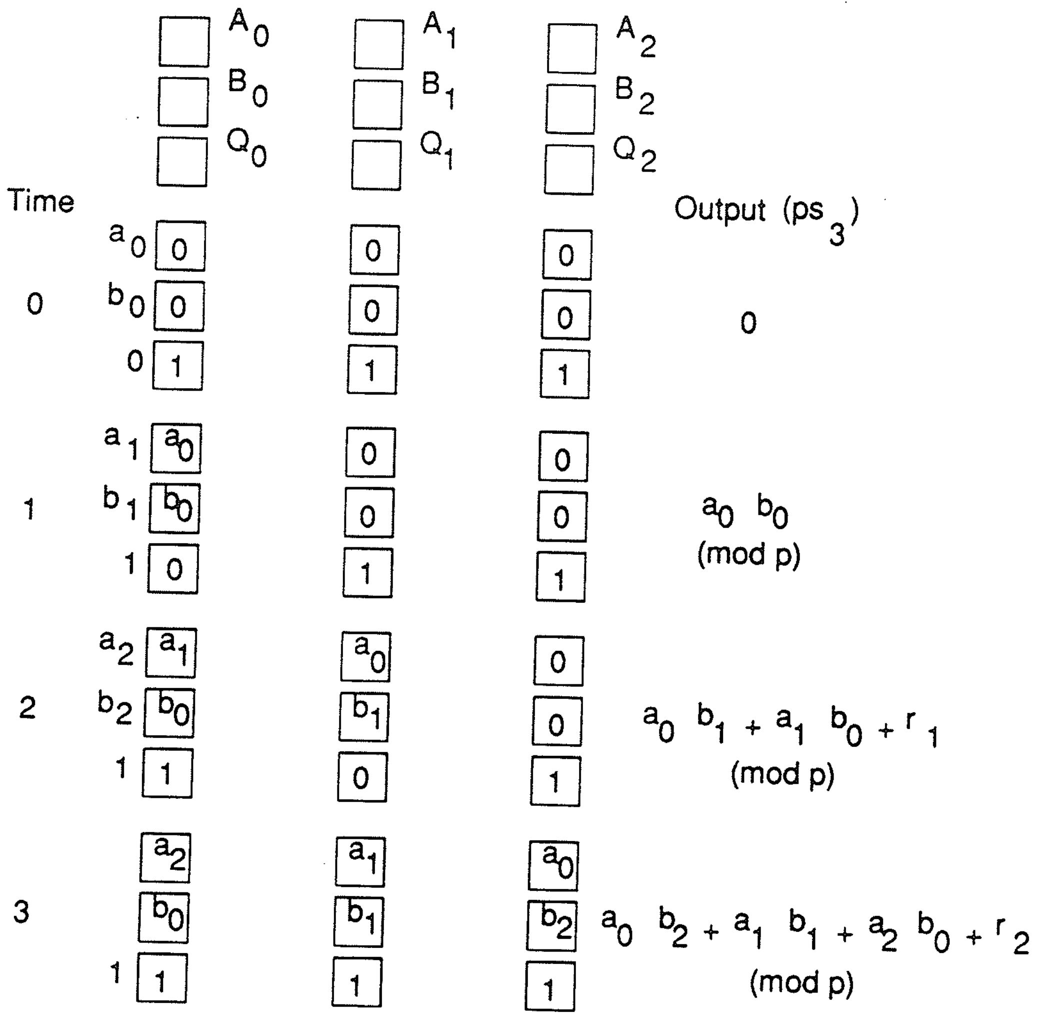

A0
B0
Q0
A1
B1
Q1
A2
B2
Q2
Time
Output (ps3)
0
1
2
3
a0 b0 (mod p)
a0 b1 + a1 b0 + r1 (mod p)
a0 b2 + a1 b1 + a2 b0 + r2 (mod p)

Figure 13: Flow of data through the serial modulo p^r multiplier of Figure 12.

The modulo p arithmetic units are used to construct modulo p^r arithmetic units, and the modulo p arithmetic units are in turn constructed using "conventional" binary arithmetic units. Thus, Table IV lists the asymptotic area and time complexities of the Brent-Kung adder [19], the McCanny-McWhirter pipelined array multiplier [20], the Luk recursive multiplier ([24], pp.317-326), and the pipelined Dadda multiplier [23,25]. Other multiplier designs could have been added to the list: for example, Lyon's serial multiplier [17], or the various DFT-FFT based designs of Brent and Kung [26], and of Preparata ([24], pp. 311-316). However, we regard these designs as being either too slow (serial multiplier), or not practical (DFT-FFT based multipliers) for the reasons discussed in [24,26]. We shall assume that the serial and parallel modulo p^r arithmetic units of the previous sections are composed exclusively of the

Binary circuits	A	T	P
Brent–Kung adder	$O(n \log n)$	$O(\log n)$	$O(1)$
Multipliers:			
McCanny–McWhirter	$O(n^2)$	$O(n)$	$O(1)$
Luk	$O(n^2 \log^2 n)$	$O(\log^2 n)$	$O(1)$
Dadda	$O(n^2 \log n)$	$O(\log n)$	$O(1)$

Table IV: Asymptotic area and time complexities of the basic building blocks making up the modulo p and modulo p^r arithmetic units.

binary arithmetic units in Table IV.

Mod p^r arith. circs.	A	T	P
Brent–Kung adder	$O(rn\log n + r\log r)$	$O(\log n + \log r)$	$O(1)$
McCanny et al. array:			
McCanny–McWhirter	$O(r^2n^2)$	$O(rn)$	$O(1)$
Luk	$O(r^2n^2\log^2 n)$	$O(r \log^2 n)$	$O(1)$
Dadda	$O(r^2n^2\log n)$	$O(r \log n)$	$O(1)$
Serial adder	$O(n \log n)$	$O(r \log n)$	$O(r \log n)$
Serial mult. (pipelined):			
Array	$O(rn^2)$	$O(rn)$	$O(rn)$
Luk	$O(rn^2\log^2 n)$	$O(r \log^2 n)$	$O(r \log^2 n)$
Dadda	$O(rn^2\log n)$	$O(r \log n)$	$O(r \log n)$

Table V: Asymptotic area and time complexities of the serial and parallel modulo p^r arithmetic units.

Therefore, from the entries of Table IV, we can construct Table V. This table contains the asymptotic area and time complexity expressions for the serial and parallel modulo p^r arithmetic units. The Brent-Kung adder entry of Table V is the modulo p^r adder of Figure 6(a), but with the carries generated by the Brent-Kung array of Figure 8. As we have already noted, the modulo p adders themselves are constructed using ordinary Brent-Kung adders (of size $n + 1$ bits if $p = 2^n + 1$, and of size n bits if $p = 2^n - 1$).

The McCanny-McWhirter array multiplier entry of Table V is the pipelined modulo p^r array multiplier of Figure 9. The three subentries labeled McCanny-McWhirter, Luk and Dadda assume that the basic cells of Figure 9 are composed of $O(n)$ bit binary McCanny-McWhirter, Luk and Dadda multipliers, respectively. Pipelining to the fullest extent possible is assumed so that in all cases $P = O(1)$. The resulting machines are *completely pipelined* as defined in Capello and Steiglitz [22,23].

The serial adder entry of Table V has $P = O(r \log n)$, and so it is not completely pipelined. This is because the *cycle time* (see [22,23]) is $O(log\ n)$, and the number of cycles separating corresponding bits of successive inputs (elements of Z_{p^r}), or outputs, of the addition function is $O(r)$.

The serial multiplier (pipelined) entry of Table V has three subentries labeled Array, Luk and Dadda. The serial modulo p^r multiplier is pipelined as in Figure 12(b). The Array subentry assumes that the modulo p cell is composed of an ordinary $O(n)$ bit unpipelined array multiplier, since pipelining it would serve no useful purpose. Similarly, the Luk and Dadda subentries assume that the modulo p multiplier cell is composed of unpipelined $O(n)$ bit Luk and Dadda multipliers. Thus, for the serial modulo p^r multiplier, $P = O(r \log n)$ is the smallest possible period. In this case the cycle time takes on its smallest possible value of $O(log\ n)$. Hence, the serial modulo p^r multiplier is not completely pipelined either. Note that the serial modulo p^r multiplier, with a Dadda multiplier cell, has the same T and P as the serial modulo p^r adder (to within a constant factor).

Some comparisons are worth making. The parallel modulo p^r arithmetic units, which are based upon the McCanny-McWhirter [20] design, have a high throughput ($P = O(1)$), but they also have a large area ($O(r^2 n^2 \log n) \geq A \geq O(r^2 n^2)$, depending upon which multiplier from Table IV is used). The serial modulo p^r arithmetic units, which are based upon Lyon's serial multiplier design [17], have a low throughput (i.e., high P) since P can vary from $O(r \log n)$ to $O(rn)$ (see Table V), but they have a relatively low area since A varies from $O(rn^2 \log n)$ to $O(rn^2)$ (see Table V). As well, their latency is comparable to the parallel modulo p^r arithmetic units ($O(r \log n) \leq T \leq O(rn)$). It seems reasonable to conclude that the parallel units are the most desirable when a high throughput is needed and one is willing to pay the price in chip area.

The digit serial units are most desirable when a fast response (low T) is needed, a low area design is required, and one is willing to compromise on throughput. Note that the digit serial approach to modulo p^r arithmetic can have an asymptotically lower T than a completely bit serial approach. A completely bit serial modulo p^r multiplier would have $T_{bs} = O(nr)$, but a digit serial modulo p^r multiplier, using a Dadda multiplier cell, has $T_{ds} = O(r \log n)$ and thus

$$\frac{T_{ds}}{T_{bs}} = c_T \frac{\log n}{n} < 1$$

for sufficiently large n. Note that the asymptotic upper bounds for T (and for A and P) are tight in all cases that we have covered. That is, they are accurate for large n and r to within a constant factor. Similarly, $P_{bs} = O(nr)$ and $P_{ds} = O(r \log n)$, and so

$$\frac{P_{ds}}{P_{bs}} = c_P \frac{\log n}{n} < 1$$

for sufficiently large n. In addition, $A_{bs} = O(nr)$ and $A_{ds} = O(r\ n^2 \log n)$ which gives

$$\frac{A_{ds}}{A_{bs}} = c_A \ n \ \log n \quad .$$

Hence, in return for an improvement in P and T, the digit serial modulo p^r multiplier requires a larger area than the completely bit serial modulo p^r multiplier. Similar results can be obtained for completely bit serial modulo p^r addition and digit serial addition.

1.6 A Note Concerning the Quadratic Residue Number System

From Chapter VI, the quadratic residue number system (QRNS) of Leung [27] (see also Jenkins and Krogmeier [28], Jullien, Krishnan, and Miller [29], or Taylor [13]) may be constructed out of prime numbers of the form $4k + 1$. Recall that if $p = 2^n + 1$, then p has the form $4k + 1$ if and only if n is even (see [28], Theorem 2), and in fact a QRNS can be constructed using Z_{p^r} for any such p even if $r > 1$ holds (see [28]). Thus, the architectures for arithmetic in Z_{p^r} have potential applications in the construction of QRNSs with a large dynamic range. Furthermore, if the parallel modulo p^r arithmetic unit designs are employed, then high throughputs can be expected.

2. Mapping From the Integers to the Finite Ring Z_{p^r}

In this section we consider the problem of mapping an integer to a number of the form in (4). That is, we consider the problem of mapping certain elements of Z (ring of integers under the usual operations) to the elements of Z_{p^r} . The method described herein does not use integer division. We only consider $p = 2^n \pm 1$. The case $p = 2^n$ is trivial. In addition, the problem of mapping the elements of Z_{p^r} back to the integers is briefly considered.

It is useful to begin by defining certain sets and ideas which we shall use in succeeding sections.

The input data to a computational problem in a finite ring often originate from a finite subset of the integers Z. This subset is usually rather small. Let this subset be denoted by S. In computing with the elements of S, the final solution(s) may lie in a larger subset of Z. Let this subset be denoted by

$$S_{p^r} = \left\{ -\frac{p^r-1}{2}, \cdots, -1,0,1, \cdots, \frac{p^r-1}{2} \right\} . \tag{28}$$

Thus we have $S \subset S_{p^r} \subset Z$.

We shall therefore map the computation in S_{p^r} to an equivalent one in Z_{p^r} in the usual way (see Gregory and Krishnamurthy [5], pp. 9-10). Thus, if $x \in S_{p^r}$ and $x \geq 0$ then $x \in Z_{p^r}$, but if $x \in S_{p^r}$ and $x < 0$, then x maps to $x + p^r \in Z_{p^r}$. We shall want to place the elements of S, the input data set, in the form shown in (4).

If we want to express $x \in S$, with $x \geq 0$, in the form of (4), we need only compute the first few (2 or 3 usually) digits, since the remaining digits will be zero. This is so since the members of S consist of relatively small integers, often in the range of 8 to 16 bits. Thus, we can expect simplifications in the mapping from S to Z_{p^r}.

2.1 Case $p = 2^n + 1$

Let us assume $X \in S$ with $X \geq 0$ (case $X < 0$ will be considered later). As we have stated, we want to map X to $x \in Z_{p^r}$ with x in the form shown in (4). We can write

$$X = X_0 + 2^n X_1 + 2^{2n} X_2 + \cdots + 2^{kn} X_k + \cdots . \tag{29}$$

Thus, $X_i \in \{0,\ldots,2^n-1\}$.

From now on we shall proceed by example. Let us assume that

$$X = X_0 + 2^n X_1 + 2^{2n} X_2 , \tag{30}$$

so $X_i = 0$ for $i \geq 3$ holds. Since $p = 2^n + 1$, X will map to

$$x = x_0 + x_1 p + x_2 p^2 , \tag{31}$$

since $x_i = 0$ for $i \geq 3$.

It is straightforward to verify that

$$X = [(X_1 - 2X_2)2^n + (X_0 - X_2)] + X_2 p^2 \text{ , and}$$

$$(X_1-2X_2)2^n + (X_0-X_2) = (X_0-X_1+X_2) + (X_1-2X_2)p$$

via the rule for integer division. Thus,

$$X = (X_0-X_1+X_2) + (X_1-2X_2)p + X_2p^2 \ . \qquad (32)$$

It is clear that further work is needed in order to obtain x_0, x_1 and x_2. Clearly,

$$\begin{aligned} X_0 - X_1 + X_2 &= x_0 + r_0p \ , \\ r_0 + X_1 - 2X_2 &= x_1 + r_1p \ , \\ r_1 + X_2 &= x_2 + r_2p \ , \end{aligned} \qquad (33)$$

but we ignore r_2 as it will be zero. Thus, we must find r_0 and r_1 in order to compute x_0, x_1 and x_2. Let $<x>_p$ denote the remainder (residue) of x divided by p. Hence,

$$\begin{aligned} x_0 &= <X_0 - X_1 + X_2>_p \ , \\ x_1 &= <r_0 + X_1 - 2X_2>_p \ , \\ x_2 &= <r_1 + X_2>_p \ . \end{aligned} \qquad (34)$$

It is evident that r_0 and r_1 will be small integer multiples of p. In fact, it is readily shown that

$$r_0 \in \left\{-1,0,1\right\} \ , r_1 \in \left\{-3,-2,-1,0,1,2\right\} \ . \qquad (35)$$

in the worst possible case.

Since x_i is $n + 1$ bits long, we make X_i $n + 1$ bits long by juxtaposing a zero (0) to the most significant bit position of X_i. Clearly, this does not affect the value of X_i. As well, we consider r_i to be $n + 1$ bits long. Define

$$t_0 = (X_0 - X_1 + X_2) - x_0 = r_0p \ , \qquad (36a)$$

$$t_1 = (r_0 + X_1 - 2X_2) - x_1 = r_1p \ ,$$

where

$$t_0 = t_{0,n}t_{0,n-1} \ \cdots \ t_{0,1}t_{0,0} \ , \qquad (36b)$$

$$t_1 = t_{1,n}t_{1,n-1} \cdots t_{1,1},t_{1,0} \;,$$

and $t_{0,i}$, $t_{1,i} \in \{0,1\}$. Equation (36b) represents the binary expansion of t_0 and t_1. We will assume a twos complement representation in particular.

By examining the possible values for $r_0 p$ at the bit level, knowing that r_0 satisfies (35), we conclude that

$$r_0 = \hat{t}_{0,n}t_{0,n-1} \cdots t_{0,1}t_{0,0} \;, \hat{t}_{0,n} = t_{0,0}\overline{t}_{0,n} \;, \qquad (37a)$$

where the juxtaposition of $t_{0,0}$ and $\overline{t}_{0,n}$ is the logical AND of the arguments, and $\overline{t}_{0,n}$ is the logical complement of $t_{0,n}$. Similarly, by examining the possible values for $r_1 p$ at the bit level, knowing that r_1 satisifies (35), we conclude that

$$r_1 = \hat{t}_{1,n}t_{1,n-1} \cdots t_{1,1}t_{1,0} \;, \hat{t}_{1,n} = \overline{t}_{1,n}t_{1,0} + t_{1,n}\overline{t}_{1,0} \;, \qquad (37b)$$

where '+' is the logical OR operation. Thus, simple logical operations on t_0 and t_1 suffice to determine r_0 and r_1.

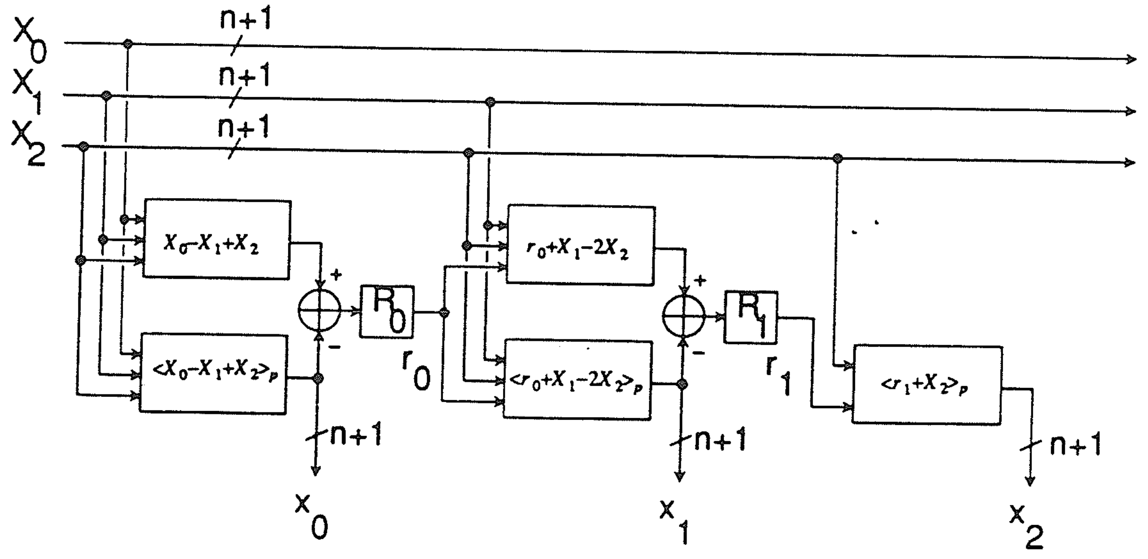


Figure 14: Machine to map $X \in S$ for $X \geq 0$ to Z_{p^r} for $p = 2^n + 1$.

Figure 14 depicts an architecture that may be used to implement the preceding mapping algorithm. The boxes labeled R_0 and R_1 produce r_0 and r_1, respectively, according to (37a,b). Notice that the least significant digits are produced first. This is useful when the modulo p^r arithmetic operations are to be performed in a digit serial fashion.

2.2 Case $p = 2^n - 1$

We will again proceed by example. As before, assume that $X \in S$ and that $X \geq 0$ such that (30) is satisfied. Since $p = 2^n - 1$, X will map to

$$x = x_0 + x_1 p + x_2 p^2 + x_3 p^3 . \qquad (38)$$

Via the method used to obtain (32) we can write

$$X = (X_0 + X_1 + X_2) + (X_1 + 2X_2)p + X_2 p^2 . \qquad (39)$$

As in (33) we have

$$\begin{aligned} X_0 + X_1 + X_2 &= x_0 + r_0 p \ , \\ r_0 + X_1 + 2X_2 &= x_1 + r_1 p \ , \\ r_1 + X_2 &= x_2 + r_2 p \ , \\ r_2 &= x_3 + r_3 p \ , \end{aligned} \qquad (40)$$

but we ignore r_3 as it will be zero. As in (34),

$$\begin{aligned} x_0 &= \langle X_0 + X_1 + X_2 \rangle_p \ , \\ x_1 &= \langle r_0 + X_1 + 2X_2 \rangle_p \ , \\ x_2 &= \langle r_1 + X_2 \rangle_p \ , \\ x_3 &= r_2 \ . \end{aligned} \qquad (41)$$

In the worst case

$$r_0 , r_1 \in \left\{ 0,1,2,3 \right\} , r_2 \in \left\{ 0,1 \right\} . \qquad (42)$$

X_i and r_i are n bits long, since x_i is n bits long. Define by analogy with (36a)

$$\begin{aligned} t_0 &= (X_0 + X_1 + X_2) - x_0 = r_0 p \ , \\ t_1 &= (r_0 + X_1 + 2X_2) - x_1 = r_1 p \ , \\ t_2 &= (r_1 + X_2) - x_2 = r_2 p \ , \end{aligned} \qquad (43a)$$

where

$$t_0 = t_{0,n-1}t_{0,n-2} \cdots t_{0,1}t_{0,0} \ ,$$

$$t_1 = t_{1,n-1}t_{1,n-2} \cdots t_{1,1}t_{1,0} \ , \qquad (43b)$$

$$t_2 = t_{2,n-1}t_{2,n-2} \cdots t_{2,1}t_{2,0} \ ,$$

and $t_{0,i}$, $t_{1,i}$, $t_{2,i} \in \{0,1\}$. The n bit binary representation of p is $11 \cdots 1$ (n ones).

By examining the possible values for r_0p , r_1p , and r_2p at the bit level, knowing that they satisfy (42), we conclude that

$$r_0 = \textit{twos complement of } t_0 \ ,$$

$$r_1 = \textit{twos complement of } t_1 \ , \qquad (44)$$

$$r_2 = \textit{twos complement of } t_2 \ .$$

Note that t_i is n bits long.

Figure 15 depicts an architecture, like that of Figure 14, that may be used to implement the preceding mapping algorithm. The boxes labeled NEG compute r_0 , r_1 and r_2 according to (44). As in Figure 14, the least significant digits are produced first.

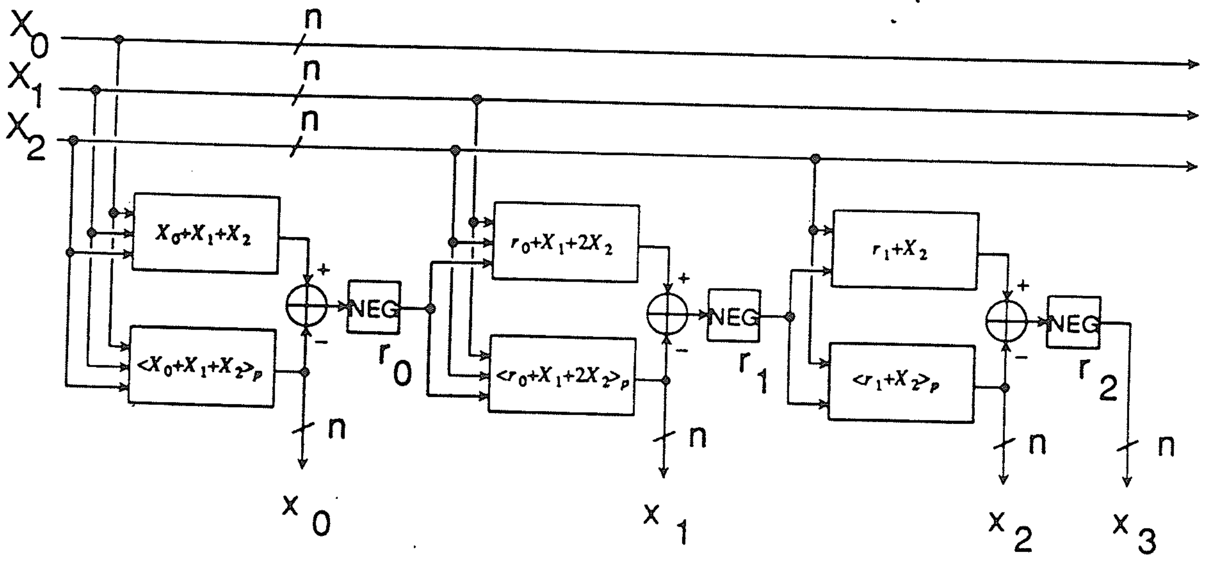


Figure 15: Machine to map $X \in S$ for $X \geq 0$ to Z_p, for $p = 2^n - 1$.

If $X \in S$ and $X < 0$, then simply map $|X|$ (absolute value of X) to Z_{p^r} according to the methods described in this and the previous section. Finally, negate the result modulo p^r. Modulo p^r negation is straightforward (see [15]).

2.3 Mapping from Z_{p^r} to Z

Thus far we have considered how to map from Z to Z_{p^r}. Now we consider the inverse problem of mapping from Z_{p^r} back to Z.

In principle one could use Horner's rule for polynomial evaluation to map x in (4) to an integer in Z, i.e.,

$$x = x_0 + p(x_1 + p(x_2 + px_3)) \tag{45}$$

if $r = 4$, for example. Equation (45) is evaluated starting with the innermost parentheses. However, since $x > \frac{p^r-1}{2}$ corresponds to a negative integer in Z, this approach must be modified.

One way would be to test if $x > \frac{p^r-1}{2}$ holds, and if this is so, compute $|x| = -x \pmod{p^r}$, and so map $-x \pmod{p^r}$ to a positive integer via (45). The sign could then be readily corrected. It may be readily shown that

$$y = \sum_{i=0}^{r-1} y_i p^i = \frac{p^r - 1}{2} \tag{46a}$$

for

$$y_i = \begin{cases} 2^{n-1}, & p = 2^n + 1 \ , \\ 2^{n-1} - 1, & p = 2^n - 1 \ , \end{cases} \tag{46b}$$

(all i). Thus, to check if $x \in Z_{p^r}$ corresponds to a negative integer, we must compare it to y in (46a) whose digits satisfy (46b).

Magnitude comparison in Z_{p^r} is straightforward, provided that we interpret all of the elements of Z_{p^r} as nonnegative integers. This will be the case when we compare any $x \in Z_{p^r}$ against y in (46a,b) to see if x corresponds to a negative integer. Let us consider arbitrary magnitude comparison with an example. Let

$$x = x_0 + x_1 p + x_2 p^2 + x_3 p^3 \ , \tag{47}$$

$$y = y_0 + y_1 p + y_2 p^2 + y_3 p^3 \ ,$$

where $x, y \in Z_{p^r}$. Define test functions

$$e(a,b) = \begin{cases} 1, & a = b, \\ 0, & a \neq b, \end{cases} \tag{48a}$$

$$g(a,b) = \begin{cases} 1, & a > b, \\ 0, & a \leq b. \end{cases} \tag{48b}$$

We are assuming that $a, b \in \{0, 1, \ldots, p-1\}$. Such test functions could be constructed in the usual manner (see Mano [30], pp. 164-167). Let

$$T_{r-1} = \begin{cases} 1, & x > y, \\ 0, & x \leq y, \end{cases} \tag{49}$$

where $x, y \in Z_{p^r}$, and so x and y are r digits long (in our current example $r = 4$). It is straightforward to verify that for x and y in (47)

$$T_3 = g_3 + e_3 g_2 + e_3 e_2 g_1 + e_3 e_2 e_1 g_0 \ , \tag{50a}$$

where

$$g_i = g(x_i, y_i) \ \ , e_i = e(x_i, y_i) \ . \tag{50b}$$

T_{r-1} can be recursively computed via

$$T_i = g(x_i, y_i) + e(x_i, y_i)\, T_{i-1} \ \ , T_0 = g(x_0, y_0) \ , \tag{51}$$

where $0 \leq i \leq r-1$.

Equation (50a,b) is analogous to the expression for $(A > B)$ at the bottom of page 165 in [30]. Equation (51) makes it possible to compare x and y in digit serial fashion beginning with the least significant digit. Since y satisfies (46a,b) in our problem, the comparator structure will simplify. This is especially true of the structure to implement $g(a,b)$. However, we omit the details. Note that once we have obtained $|\ x\ |$, scaling by a power of p is trivial.

REFERENCES

[1] C. J. Zarowski, H. C. Card, "Serial and Parallel Architectures for Addition and Multiplication in the Ring Z_{p^r} ," submitted to the IEEE Trans. on Circ. and Syst.

[2] C. J. Zarowski, H. C. Card, "Mapping From the Integers to the Finite Ring Z_{p^r} ($p = 2^n \pm 1$)," submitted to the IEEE Trans. on Comp.

[3] J. H. McClellan, C. M. Rader, *Number Theory in Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[4] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, Massachusetts: Addison-Wesley, 1985.

[5] R. T. Gregory, E. V. Krishnamurthy, *Methods and Applications of Error-Free Computation*. New York, New York: Springer-Verlag, 1984.

[6] W. K. Jenkins, B. J. Leon, "The Use of Residue Number Systems in the Design of Finite Impulse Response Digital Filters," IEEE Trans. on Circ. and Syst., vol. CAS-24, April 1977, pp. 191-201.

[7] W. K. Jenkins, "Recent Advances in Residue Number Techniques for Recursive Digital Filtering," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-27, Feb. 1979, pp. 19-30.

[8] A. Z. Baraniecka, G. A. Jullien, "Residue Number System Implementations of Number Theoretic Transforms in Complex Residue Rings," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-28, June 1980, pp. 285-291.

[9] A. Baraniecka, G. A. Jullien, "On Decoding Techniques for Residue Number System Realizations of Digital Signal Processing Hardware," IEEE Tran. on Circ. and Syst., vol. CAS-25, Nov. 1978, pp. 935-936.

[10] G. A. Jullien, "Residue Number Scaling and Other Operations Using ROM Arrays," IEEE Trans. on Comp., vol. C-27, April 1978, pp. 325-336.

[11] G. A. Jullien, "Implementation of Multiplication, Modulo a Prime Number, with Applications to Number Theoretic Transforms," IEEE Trans. on Comp., vol. C-29, Oct. 1980, pp. 899-905.

[12] F. J. Taylor, "A VLSI Residue Arithmetic Multiplier," IEEE Trans. on Comp., vol. C-31, June 1982, pp. 540-546.

[13] F. J. Taylor, "A Single Modulus Complex ALU for Signal Processing," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Oct. 1985, pp. 1302-1315.

[14] R. Ramnarayan, F. J. Taylor, "RNS Cellular Arrays," IEEE Trans. on Circ. and Syst., vol. CAS-33, May 1986, pp. 526-532.

[15] J. J. Thomas, S. R. Parker, "Implementing Exact Calculations in Hardware," IEEE Trans. on Comp., vol. C-36, June 1987, pp. 764-768.

[16] J. J. Thomas, S. R. Parker, "A Viable Technique for Calculating Algorithms to any Specified Accuracy," EUSIPCO-83, Erlangen, West Germany, Sept. 12-16, 1983.

[17] R. F. Lyon, "Two's Complement Pipeline Multipliers," IEEE Trans. on Communications, vol. COM-24, April 1976, pp. 418-425.

[18] L. B. Jackson, J. F. Kaiser, H. S. McDonald, "An Approach to the Implementation of Digital Filters," IEEE Trans. on Audio and Electroacoustics, vol. AU-16, Sept. 1968, pp. 413-421.

[19] R. P. Brent, H. T. Kung, "A Regular Layout for Parallel Adders," IEEE Trans. on Comp., vol. C-31, March 1982, pp. 260-264.

[20] J. V. McCanny, J. G. McWhirter, "Completely Iterative, Pipelined Multiplier Array Suitable for VLSI," IEE Proc., vol. 129, Pt. G, April 1982, pp. 40-46.

[21] L. R. Rabiner, B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[22] P. R. Capello, K. Steiglitz, "Completely-Pipelined Architectures for Digital Signal Processing," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-31, Aug. 1983, pp. 1016-1023.

[23] P. R. Capello, K. Steiglitz, "A VLSI Layout for a Pipelined Dadda Multiplier," ACM Trans. on Comp. Syst., vol. 1, May 1983, pp. 157-174.

[24] H. T. Kung, B. Sproull, G. Steele (eds.), *VLSI Systems and Computations*. Rockville, Maryland: Computer Science Press, 1981.

[25] L. Dadda, "Some Schemes for Parallel Multipliers," Alta Frequenza, vol. 34, 1965, pp. 349-356.

[26] R. P. Brent, H. T. Kung, "The Area-Time Complexity of Binary Multiplication," J. Assoc. Comp. Mach., vol. 28, July 1981, pp. 521-534.

[27] S. H. Leung, "Application of Residue Number Systems to Complex Digital Filters," Proc. 15th Asilomar Conf. Circuit Syst., Pacific Grove, California, Nov. 1982, pp. 70-74.

[28] W. K. Jenkins, J. V. Krogmeier, "The Design of Dual-Mode Complex Signal Processors Based on Quadratic Modular Number Codes," IEEE Trans. on Circ. and Syst., vol. CAS-34, April 1987, pp. 354-364.

[29] G. Jullien, R. Krishnan, W. C. Miller, "Complex Digital Signal Processing Over Finite Rings," IEEE Trans. on Circ. and Syst., vol. CAS-34, April 1987, pp. 365-377.

[30] M. M. Mano, *Digital Logic and Computer Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

Chapter IX

FAULT-TOLERANT DESIGN

When Z_{p^r} is a large ring (i.e., large p^r), the arithmetic units used to perform modulo p^r operations, and the units to map from Z to Z_{p^r} and vice versa will occupy a large chip area. Indeed, the area required may be so large that such technologies as wafer-scale integration (WSI) may be necessary in order to successfully construct such systems. We know that errors in finite ring computations are intolerable in general, especially if the ring is a product ring. Yet it is also known that the yield (fraction of chips or wafers fabricated that are fully functional) falls rapidly as areas increase, and can be close to zero for large area systems. Clearly then, unless special steps are taken, it will not be possible to compute in a very large ring at all since it will not be possible to build working hardware. Furthermore, even if such hardware could be built, it is likely to have reliability problems. Thus, we must be able to design fault-tolerant systems. However, it is beyond the scope of the present work to consider this problem in detail. Instead we will outline the problem and attempt to evaluate the prospects of arriving at a successful solution to it.

1. Types of Faults

An integrated circuit (IC) may be afflicted by two classes of faults. These are:

(i) hard faults, and

(ii) soft faults.

Hard faults are due to physical failures in the circuitry itself. These faults can occur during fabrication, or when the IC is operating in the field (i.e., wear-out failures). Physical failures in ICs are discussed in greater detail in Pradhan [1] (see pp. 6-15).

Soft faults (errors) can be caused in at least three ways (Savaria et al. [2]):

(i) ionizing radiation (e.g., alpha particles),

(ii) electromagnetic interference,

(iii) electrical noise (e.g., thermal noise, shot noise, etc.).

Soft faults are of a transient nature, and so are hard to detect. We shall consider the phenomenon of charge leakage to be a hard fault. Note that hard faults (such as charge leakage) can be of an intermittent nature as well, and so can mimic soft faults.

In designing a system for computation in a large ring, it will be necessary in general to design the system to tolerate both hard and soft faults.

2. Fault Tolerance in VLSI Based Systems

The motivation for incorporating fault tolerance into a system is actually twofold (see [1], pp. 547-549):

(i) yield enhancement, and

(ii) reliability improvement.

Fault-tolerant system design involves the use of hardware redundancy. In this section we briefly outline various strategies for fault tolerance, some of which are applicable to arbitrary design problems, and some of which are peculiar to residue number systems.

2.1 Testing and Restructuring

Testing and restructuring strategies are discussed in [1] (see Chapter 7, section 7.9), and we summarize a few concepts from [1] on this subject here. Note that design for testability (DFT) is a vital consideration if the test and restructure philosophy is to be employed in the construction of fault-tolerant hardware. DFT is discussed in [1] (Chapters 1 and 2) and so we will not consider it here. However, we emphasize that testing does not indicate that *no* faults are present. It can only indicate if faults are present. This means that some faults will inevitably escape detection. The challenge then is to reduce the level of undetected faults to an acceptable minimum.

Since faults can occur during fabrication, *yield enhancement* can be achieved by incorporating spare elements on the chip or wafer to replace those elements that are tested and found to be defective. Such testing and replacement is performed during production testing (i.e., before the chip or wafer is sent out into the field for use). Yield enhancement is therefore achieved by making it possible to use defective circuits.

Reliability improvement is achieved by replacing defective elements with properly operating spares while the chip or wafer is out in the field. Such replacements may even occur while the chip or wafer is in operation. Note that in the field, the chip or wafer is not directly accessible as it is at the production facility prior to assembly. Thus, strategies for fault tolerance as a means of achieving reliability improvement are generally different from strategies for achieving yield enhancement. Fault tolerance for reliability improvement will often require the chip or wafer to test itself.

An important point concerning the test and restructure strategy is that there is a basic tradeoff between yield and speed/performance, as is argued in [1] (see pp. 565-567). When defective elements are replaced (bypassed), the path lengths between the working elements often increases, and this causes a degradation in the system's speed/performance. Thus, given a fixed speed/ performance requirement, one cannot permit arbitrarily large numbers of elements on a wafer without reducing the total yield. Conversely, given a fixed total yield requirement, the allowable speed/performance degradation due to element replacement may have to increase with the number of wafer elements.

2.2 Triple Redundancy With Voting

Figure 1 illustrates the principle of triple redundancy with voting, also called triple modular redundancy (TMR). Suppose $f(x) \in \{0,1\}$ is a Boolean function, where x is a Boolean input vector. The circuit implementing f is duplicated three times (blocks labeled A,B, and C). If the output of one of the three circuits disagrees with that of the other two, then the voter circuit will select the output to be that of the two circuits that agree with each other. A VLSI bit serial adder and multiplier by

Kanopoulos [3] uses this method of achieving fault tolerance. Clearly, the method has disadvantages. For one thing, the voter is a critical circuit; it cannot be allowed to fail. Faults at the input x cannot always be corrected. The method has a high area overhead since it occupies an area of more than three times that of the nonredundant implementation of function f. The circuit is also somewhat slower than that of the nonredundant implementation of f.

TMR can be used for reliability improvement. We will examine its possible role in yield enhancement here.

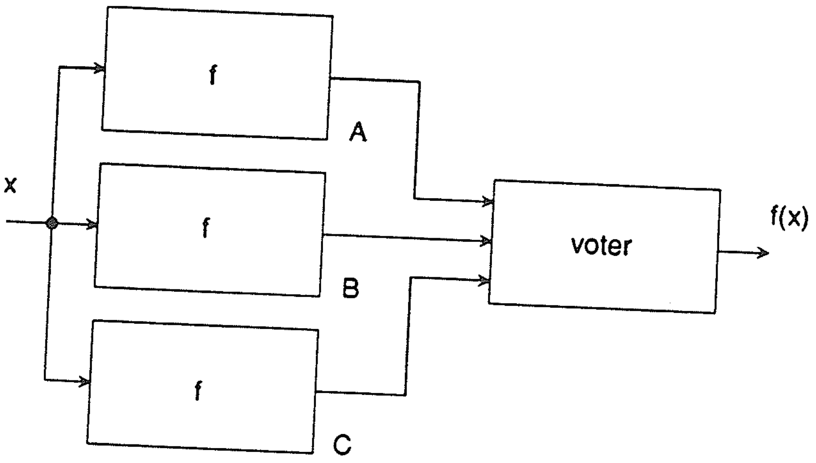


Figure 1: Triply redundant implementation of Boolean function $f(x)$ with a voting circuit.

Suppose A_f is the area of the circuit that implements f nonredundantly, and A_v is the area of the voting circuit. We can reasonably model yield in the following way, at least to a first approximation. If there are N fatal flaws per unit area on a chip of area A, then the probability of a bad chip is (see [4], pp. 45-46)

$$P = 1 - e^{-NA} \quad . \tag{1}$$

It is assumed that the fatal flaws are randomly distributed over the wafer. Thus, if the circuit for implementing f is implemented nonredundantly, it has a failure probability of

$$P_I = 1 - e^{-NA_f} \quad . \tag{2a}$$

If f is implemented redundantly as in Fig. 1, it has a failure probability of

$$P_{II} = 1 - 3(1 - e^{-NA_f})e^{-2NA_f}e^{-NA_v} - e^{-3NA_f}e^{-NA_v}$$

$$= 1 - 3e^{-N(2A_f + A_v)} + 2e^{-N(3A_f + A_v)} \quad , \tag{2b}$$

where we assume that the three copies of f and the voter circuit fail independently. For simplicity, let $A_v = 0$, and let $\alpha = e^{-NA_f}$ (= probability that the circuit which implements f nonredundantly works). Thus

$$P_I = 1 - \alpha \ , \tag{3a}$$

$$P_{II} = 1 - 3\alpha^2 + 2\alpha^3 \ . \tag{3b}$$

A sketch of P_I and P_{II} versus α may be seen in Figure 2. It is clear that $P_{II} < P_I$ for $\frac{1}{2} < \alpha < 1$. Thus yield is enhanced if and only if $\frac{1}{2} < \alpha < 1$ in our present model. In fact, if $\alpha \to 1$ then $\frac{P_I}{P_{II}} \to \infty$, and $\alpha \to 1$ if $NA_f \to 0$.

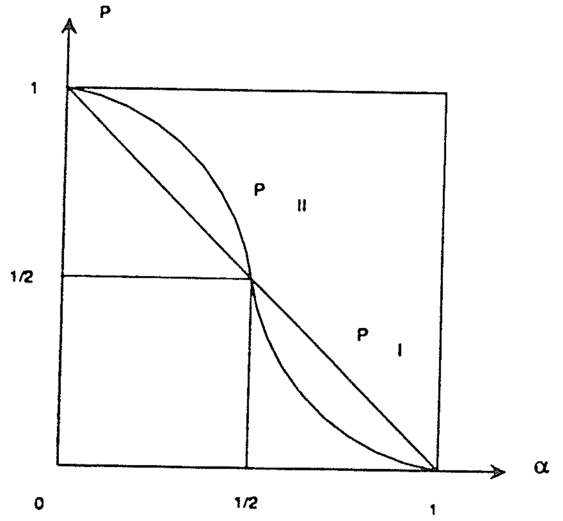


Figure 2: P_I and P_{II} versus α.

Assume that NA_f and NA_v are small enough so that we may accurately approximate e^x by $1 + x$. Then, using (2a) and (2b),

$$\frac{P_I}{P_{II}} = \frac{A_f}{A_v} \ . \tag{4}$$

Thus, since we want $P_I > P_{II}$, we must have $A_f > A_v$. Thus, a significant enhancement of yield will occur if NA_f is small and A_v is much less than A_f. This suggests that TMR as a means of achieving enhanced yield should only be used at low levels in a design, i.e., at the level of circuits with a small area relative to the area of the entire design.

Another point of importance is that, on a wafer of a given size, the number of redundantly implemented functions f that can fit onto it is much less than the number of nonredundantly implemented functions. In the present case, the redundantly implemented function is more than three times larger than the nonredundant function. Clearly then, the increase in the yield of functions per wafer due to the use of redundancy must offset the loss in yield of functions per wafer due to the larger area of the redundantly implemented functions. Thus the large area of a redundantly implemented function will likely discourage the use of TMR as a yield enhancement strategy.

We emphasize that the yield model of (1) is known to be rather simplistic. For example, it is known that yield varies radially with distance from the edge of the wafer (see Ferris-Prabhu et al. [5]). A better model is based upon the use of the so-called generalized negative binomial distribution [6,7], since it accounts for the tendency of faults to cluster.

As a historical note, TMR was originally conceived by Von Neumann [8]. The method is an example of an *error masking strategy*. Another error masking strategy is called *quadded logic*, which is a quadruple redundancy method. It was developed by Tryon ([9], pp. 205-228; also see Kohavi [10]).

2.3 Error Control Coding Theory Approaches

Error control coding theory is known to be useful in digital communications systems since errors that occur in data transmitted over a communications channel may be corrected at the receiver provided that the data was suitably coded at the transmitter and the channel capacity is not exceeded (see Blahut [11] or MacWilliams and Sloane [12]). It was realized in the 1950's (e.g., see [8] or Peterson and Rabin [13]) that coding theory could also be applied to the problem of fault-tolerant system design, since a processing unit resembles a communications channel, and in particular, a faulty processor resembles a noisy communications channel. Thus, Figure 3 illustrates the principle behind the error control coding theory approach to fault-tolerant system design.

In Fig. 3 output z is some function of the inputs x and y. Inputs x and y are encoded as $U(x)$ and $V(y)$, respectively. Instead of processing x and y directly, the processor operates on the coded inputs $U(x)$ and $V(y)$ giving coded output $W(z)$. $W(z)$ is then decoded as z, the true output, by the decoder block. Clearly, for this scheme to succeed, the processor must produce a valid codeword output for a valid codeword input. This naturally constrains the coding methods that can be employed.

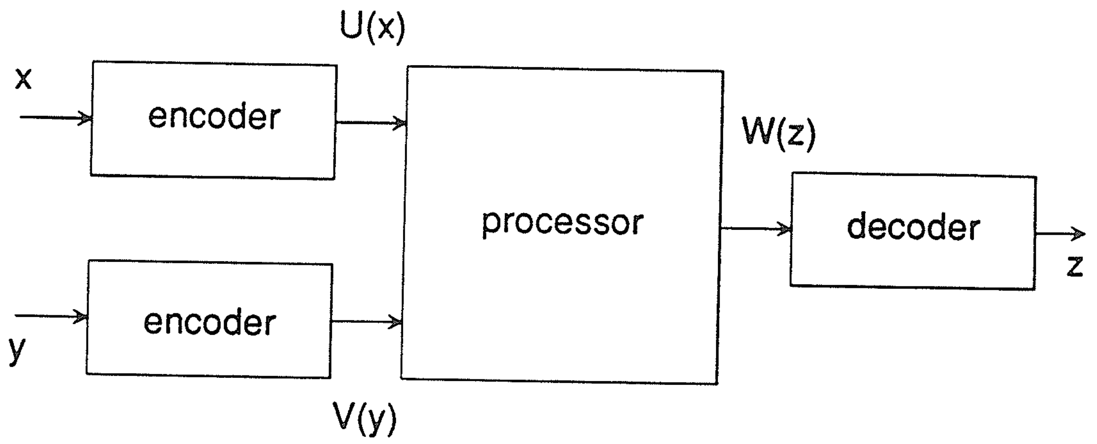


Figure 3: Illustration of the error control coding approach to fault-tolerant sytem design.

Perhaps the simplest example of the coding theory approach to fault-tolerant system design is *parity prediction* (see [1], pp. 344-359). In this method the parity of the result of some operation is predicted by a parity prediction circuit, and compared with the true parity of the result that is actually produced. If there is a disagreement between the predicted parity and the actual parity then an error signal is generated. This concept can be employed in adders, multipliers, dividers, and even in arbitrary combinational logic circuits. The area of a circuit with parity prediction is roughly twice that of a circuit without it.

Arithmetic codes are error control codes used to check arithmetic operations (see [1], pp. 312-319, and pp. 337-344). Since the algorithms considered in this thesis are very arithmetic intensive, such codes may be very useful in solving the reliability enhancement problem. One type of arithmetic code involves the use of residue arithmetic. Let N represent a positive integer. Let

$$R = N \ mod \ A \quad ,$$

so R is the residue of N modulo A. The concatenation of N and R, denoted (NR), is called a *residue code* (see [1], pp. 339-340). The modulus A is called a *check base.* With this code errors may be detected in the following way. Suppose that N_1 and N_2 are positive integers with modulo A residues R_1 and R_2, respectively. Then

$$(N_1 \pm N_2) \ mod \ A = (R_1 \pm R_2) \ mod \ A \quad ,$$

$$(N_1 N_2) \ mod \ A = (R_1 R_2) \ mod \ A \quad .$$

Checking involves the comparison of the residues of the operations on N_1 and N_2 with the residues of the operations on R_1 and R_2. Any discrepancy indicates the presence of an error. The choice of check base influences the implementational complexity of the residue code. So-called *low-cost codes* arise when one considers a check base of the form $A = 2^b - 1$ (see [1], pp. 340-341, or Avizienis [34]).

Reed-Muller codes (see [11,12]) may be employed in the construction of fault-tolerant combinational logic (see [1] pp. 319-321, Pradhan and Reddy [14], and Pradhan [15]). A Reed-Muller code of blocklength 2^m and of order i $(0 \le i \le m)$ has a minimum Hamming distance of 2^{m-i} and so it can correct $\lfloor \frac{(2^{m-i} - 1)}{2} \rfloor$ errors and detect 2^{m-i-1} errors. The central result of [14] is a lemma:

Lemma (Pradhan and Reddy [14]): If x and y exist in an ith order Reed-Muller code, then $x * y$ exists in a $2i$th order Reed-Muller code, where $*$ is any two variable function, performed bit-by-bit.

Thus, the technique in [14] can correct up to $\lfloor \frac{(2^{m-2i} - 1)}{2} \rfloor$ errors and detect 2^{m-2i-1} errors in the output of the processor (see [14], Theorem 2). Note that with this approach the decoder is critical and so a failure in it is not tolerable. The decoding operation involves the use of majority logic and exclusive-OR gates (see [11,12]).

Based upon certain assumptions (see [14], section II), the maximum efficiency of any error control scheme that can either detect or correct errors in $W(z)$ is $\frac{1}{2}$ $(= \frac{k}{n},$

$k =$ number of information bits, $n =$ blocklength of the code). The Reed-Muller scheme of Pradhan and Reddy [14] asymptotically approaches this efficiency, for a code with a given minimum distance specification, as $n = 2^m \to \infty$. The area overhead of the Pradhan and Reddy strategy for fault tolerance is not evaluated in either [14] or [15]. It will depend upon the areas of the encoding and decoding circuits, and upon the efficiency of the particular code chosen. However, it seems reasonable to assume that the area of a processor with Reed-Muller coding for fault tolerance will be at least twice as large as a processor without such a scheme. It is also clear that the speed of a circuit that uses Reed-Muller coding will be lower than that of one which does not use it.

Checksum techniques can be applied to error detection and correction in processor arrays for the solution of certain linear algebra problems. This fact is demonstrated in Huang and Abraham [16], and in Jou and Abraham [17].

If A is an $n \times m$ matrix such that $A = [a_{ij}]_{n \times m}$, where a_{ij} may be integer or floating-point, and $1 \le i \le n$, $1 \le j \le m$, then the *column checksum matrix* A_c of A may be defined as the matrix A augmented with an $(n+1)$th row $e^T A$ (T is transpose), where $e^T = [1\ 1\ \cdots\ 1]$ is a $1 \times n$ vector all of ones. Symbolically,

$$A_c = \left| \frac{A}{e^T A} \right| . \tag{5a}$$

Similarly, the *row checksum matrix* of A is

$$A_r = \left| A \mid Ae \right| , \tag{5b}$$

where Ae is the $(m+1)$th column used to augment A, and the *full checksum matrix* A_f of A is defined as the column checksum matrix of A_r. These definitions are from [16]. In [16] it is shown that certain operations performed on checksum matrices result in valid checksum matrices. For example, from Theorem 4.1 in [16], if $C = AB$ (matrix product), then

$$C_f = A_c B_r , \tag{6a}$$

since

$$A_c B_r = \left| \frac{A}{e^T A} \right| \left| B \mid Be \right| = \left| \frac{AB}{e^T AB} \mid \frac{ABe}{e^T ABe} \right| . \qquad (6b)$$

A more general version of this checksum encoding scheme, called the *weighted checksum coding (WCC) scheme,* may be found in Jou and Abraham [17]. The WCC is more powerful than the scheme in [16] since it is more general in that it can correct and detect multiple errors. It is important to note that these techniques are applicable at high levels in a system, i.e., at the processor level. As such, the method is to be used in solving the reliability enhancement problem, rather than the yield enhancement problem. WCC is only applicable to a limited (though vital) class of matrix algebra problems. Parity prediction and the Reed-Muller coding method are applicable at lower levels in a design, and so are more universally applicable.

There is of course a time and area overhead involved in the use of checksum schemes for fault-tolerant system design. For example, consider the problem of multiplying two $n \times n$ matrices on a mesh-connected processor array (see Fig. 2 in [16]). The array without the simple checksum scheme of [16] requires n^2 processors. If the simple checksum scheme of (6a,b) is used, then an additional $2n + 1$ processors are needed. On such an array, the two matrices are multiplied in $O(n)$ time, but an additional $O(\log n)$ time units are needed in order to detect any errors that may have occurred [16].

2.4 Techniques Peculiar to Residue Number Systems

Certain techniques have been developed for the design of fault-tolerant RNS computers. Often these methods are based upon the use of *redundant residue digits.* Examples of this practice may be found in Mandelbaum [18], Barsi and Maestrini [19], or Ramachandran [20]. The redundant residue concept is extendable to computing in quadratic residue number systems (see Krogmeier and Jenkins [21]). Hardware structures to implement the redundant residue schemes are discussed in Jenkins [22,23]. The convolution of finite length sequences over finite fields can be made fault-tolerant using results from the theory of cyclic error control codes (see Redinbo [24,25] and LaMacchia and Redinbo [26]).

A *redundant residue number system (RRNS)* may be described in the following way (see [18]). Ordinarily, the nonnegative integers x in the range $[0,M)$, where $M = m_1 m_2 \cdots m_n$ and $gcd(m_i, m_j) = 1$ for all $i \neq j$, are mapped to the n-tuple $(x_1, x_2, \cdots, x_n)$, where $x_i = | x |_{m_i}$ (= residue (remainder) obtained upon dividing x by modulus m_i). In an RRNS the *nonredundant moduli* $m_1, \ldots, m_n$ are augmented by r additional *redundant moduli* $m_{n+1}, \ldots, m_{n+r}$. Thus, $x \in [0,M)$ is mapped to the $(n+r)$- tuple $(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+r})$. The digits x_i for $1 \leq i \leq n$ are called *nonredundant digits,* while the digits for $n+1 \leq i \leq n+r$ are called redundant digits. Note that, from [18],

$$m_{n+r} > \cdots > m_{n+1} > m_n > \cdots > m_1 , \tag{7}$$

but this restriction is actually relaxed in Barsi and Maestrini [19]. In [18] it is shown that an RRNS with r redundant moduli will detect r errors and correct $\lfloor \frac{r}{2} \rfloor$ errors in any ith residue digit $(1 \leq i \leq n+r)$. It is important to note that input data and the results of computations in an RRNS must be restricted to range $[0,M)$, not $[0, Mm_{n+1} \cdots m_{n+r})$, as numbers in the range from M to $Mm_{n+1} \cdots m_{n+r} - 1$ will be considered as the result of errors by the RRNS error detection and correction circuits. The implementation techniques discussed in [18-23] are all quite complex from the hardware implementational standpoint.

We shall not consider the fault-tolerant convolution schemes in [24-26] since we are not directly concerned with convolution in this work. However, we note that the methods in [24-26] are applicable to the design of fault- tolerant correlators for sequences over finite fields. Recall that correlation coefficients often constitute the input data to the Schur (and other) algorithms. It may be worth noting that binary multiplication can be considered as the convolution of finite length sequences over a finite field with a "carry release" operation (see Brent and Kung [27], or Preparata in [28] on pp. 311-316). Thus, Redinbo's cyclic coding schemes could conceivably be used in the design of fault-tolerant binary multipliers, except that some scheme other than a cyclic coding one would be needed to make the carry release circuitry fault-tolerant. This is likely to be a difficult problem, however.

2.5 Soft-Error Filtering (SEF)

In Savaria, Hayes, Rumin and Agarwal [2], and in Savaria, Rumin, Hayes and Agarwal [29], it is argued that soft errors (faults) will be the dominant factor determining the reliability of VLSI and WSI circuits with submicron feature sizes. In fact, they argue that soft errors will exceed errors due to hard faults. They propose a method, called *soft-error filtering (SEF)*, as a means of controling errors caused by ionizing radiation, and electrical noise or interference.

The finite state machine of Figure 4 is a widely applicable model for digital machines, and so when SEF is applied to this circuit, the method can be readily extended to other circuits (e.g., pipelined machines). The model in Fig. 4 is taken from Fig. 1 in [2,29] and assumes a two-phase nonoverlapping clock scheme (clock signals are $\phi 1$ and $\phi 2$). The outputs of the latches follow the inputs when the clock signals are high.

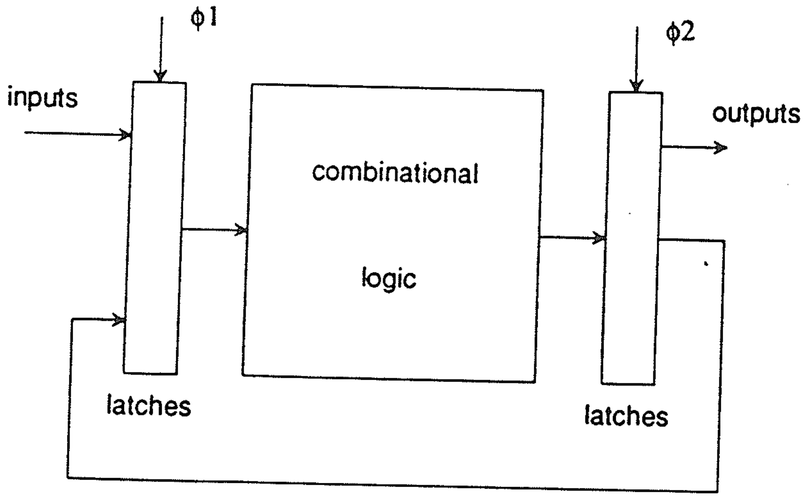


Figure 4: Illustration of the soft-error filtering (SEF) concept circuit model.

A soft error can occur in either the latches, or the combinational logic network. SEF assumes intrinsically soft error tolerant latches (static latches), and so soft errors can only result from transients injected into the combinational logic. Such injected transients will appear as transients in the output of that logic (i.e., at the inputs to the output latches, which are clocked with $\phi 2$). It is readily seen that if the transient is of a duration less than the setup time of the output latches, then the transient will not pass through the output latches. Thus, the principle of SEF is to increase the latch setup times sufficiently to reduce soft error rates to acceptable levels. Evaluation of the effectiveness of SEF, the design of SEF latches for CMOS technology, and an evaluation of the area and time overheads involved in the use of the method may be found in [2,29]. It is concluded in [2,29] that SEF is likely to be a superior way of controlling soft errors when compared with classical approaches based upon such methods as TMR, and error control coding theory approaches. It is important to note that SEF does not work with dynamic logic; it is intended for use with static circuits only. As well, SEF obviously gives no protection against hard faults, unless their effects are of sufficiently short duration (which is unlikely of course).

3. The Prospects for Reliable Computation in Large Finite Rings

On the basis of what we have seen so far, is it *likely* that *reliable computation* in *large finite rings* can be achieved ? We shall attempt to answer this question here. The phrase "reliable computation" shall be taken to mean the following. If we can build hardware for computation in large finite rings with an acceptable yield, and if the resulting hardware has a sufficiently high probability of working for some prespecified length of time without failures (hard or soft) that result in errors, then we have achieved reliable computation. For us, a finite ring will be large if hardware for computation in it must be built with WSI technology, or at least with VLSI technology possibly employing submicron feature sizes, multiple layers of metal, polysilicon, etc., and/or large die sizes. The use of the vague term "likely" is deliberate. We use it because it is not possible, in the present work, to definitively prove that reliable computation in a large finite ring can be achieved. Definitive proof will necessarily require the design, fabrication, and testing of real physical systems, and this will be a very arduous task.

Let us first address the yield enhancement problem. From the arguments of section 2.2 it is unlikely that such approaches as TMR, quadded logic, or even the error control coding methods (with the possible exception of low-cost coding schemes) can be used successfully in solving the yield enhancement problem. This is due to the relatively large area overhead involved in using these methods. The redundant residue schemes of section 2.4 are not useful either since they are used at the system level (i.e., at a high level in the design) and occupy a large area. SEF obviously can't enhance yield. Thus, it appears that we are left with the test and restructure methodology of section 2.1. This strategy is advocated in Koren and Pradhan [7], and is reviewed in Moore [30]. In [30] it is concluded that VLSI chips with a one-dimensional array architecture (the structures of interest to us here) could successfully exploit the test and restructure methodology. This conclusion is partly based upon results presented in Manning [31], and in Finnila and Love [32]. In [32] a whole-wafer linear array processor was proposed for radar tracking and general arithmetic applications. Discretionary wiring was used to interconnect working processing elements on the wafer. Discretionary wiring involves the testing of processing elements and the use of a metallization layer dedicated to the task of interconnecting working processors. Since faults can occur in this metallization layer, electronic switching was also incorporated in order to isolate faulty cells. In [31], cellular grids that can be programmed to link themselves into working systems were investigated, and the systems either had a one-dimensional topology (i.e., were linear arrays) or had a tree topology. In both cases the operations performed by the cells were not highly complex. Thus, the success of these test and restructure methods seems to hinge on the simplicity of the constituent cells (processors) since simple and regular processing elements are more testable than complex and irregular processing elements. Since the cells composing the modulo p^r arithmetic units are simple and regular, the prospects of successfully employing such test and reconfigure schemes appears to be reasonably good. References to more recent examples of the successful exploitation of test and reconfigure methods in systolic processors may be found in [1] (see page 567), and also in Kuhn [33].

As we have seen, many approaches exist for solving the reliability problem (e.g., TMR, error control schemes, etc.). Many of these methods yield systems tolerant to both soft and hard faults. However, it seems the best method of achieving tolerance to soft faults (in static circuits) is the soft-error filtering (SEF) method of Savaria et al.

[2,29]. Error control coding methods may well be the most efficient way of achieving tolerance to hard faults, especially since this approach can be applied at all levels in a system. The overheads appear reasonable when compared with classical "brute force" strategies such as TMR and quadded logic. The redundant residue schemes of section 2.4 are likely to be of little use. This is because the redundant moduli are so large compared with the nonredundant moduli thus adding enormously to the area and time overhead.

Thus, we tentatively conclude that reliable computation in large finite rings is now, or will soon be, possible. Acceptable yields will be achieved mainly through the use of test and restructure schemes. Reliable operation will be achieved through SEF (for soft faults) and through error control coding methods (for soft or hard faults). In view of the large areas required by implementations of the architectures in this thesis, it is recommended that a systematic study of the schemes for fault tolerance outlined above be included in further work on this topic.

REFERENCES

[1] D. K. Pradhan (ed.), *Fault-tolerant Computing Theory and Techniques*, vols. I & II. Englewood-Cliffs, New Jersey: Prentice-Hall, 1986.

[2] Y. Savaria, J. F. Hayes, N. C. Rumin, V. K. Agarwal, "A Theory for the Design of Soft-Error-Tolerant VLSI Circuits," IEEE Jour. on Select. Areas in Comm., vol. SAC-4, Jan. 1986, pp. 15-23.

[3] N. Kanopoulos, "A Bit-Serial Architecture for Digital Signal Processing," IEEE Trans. on Circ. and Syst., vol. CAS-32, Mar. 1985, pp. 289-291.

[4] C. Mead, L. Conway, *Introduction to VLSI Systems*. Reading, Massachusetts: Addison-Wesley, 1980.

[5] A. V. Ferris-Prabhu, L. D. Smith, H. A. Bonges, J. K. Paulien, "Radial Yield Variations in Semiconductor Wafers," IEEE Circ. and Dev. Magazine, vol. 3, March 1987, pp. 42-47.

[6] C. H. Stapper, A. N. McLaren, M. Dreckman, "Yield Model for Productivity Optimization of VLSI Memory Chips with Redundancy and Partially Good Product," IBM J. of Res. Devel., vol. 24, May 1980, pp. 398-409.

[7] I. Koren, D. K. Pradhan, "Yield and Performance Enhancement Through Redundancy in VLSI and WSI Multiprocessor Systems," Proc. IEEE, vol. 74, May 1986, pp. 699-711.

[8] J. Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Automata Studies, no. 34, pp. 43-49. Princeton, New Jersey: Princeton Univ. Press, 1956.

[9] Wilcox and Mann (eds.), *Redundancy Techniques for Computing Systems*. Washington, D.C.: Spartan Books, 1962.

[10] Z. Kohavi, *Switching and Finite Automata Theory*. New York, New York: McGraw-Hill, 1978.

[11] R. E. Blahut, *Theory and Practice of Error Control Codes*. Reading, Massachusetts: Addison-Wesley, 1983.

[12] F. J. Williams, N. J. A. Sloane, *The Theory of Error-Correcting Codes*. New York, New York: North-Holland, 1977.

[13] W. W. Peterson, M. O. Rabin, "On Codes for Checking Logical Operations," IBM J. Res. Devel., vol. 3, April 1959, pp. 163-168.

[14] D. K. Pradhan, S. M. Reddy, "Error-Control Techniques for Logic Processors," IEEE Trans. on Comp., vol. C-21, Dec. 1972, pp. 1331-1336.

[15] D. K. Pradhan, "Fault-Tolerant Carry-Save Adders," IEEE Trans. on Comp., vol. C-23,Dec. 1974, pp. 1320-1322.

[16] K.-H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," IEEE Trans. on Comp., vol. C-33, June 1984, pp. 518-528.

[17] J.-Y. Jou, J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," IEEE Proc., vol. 74, May 1986, pp. 732-741.

[18] D. Mandelbaum, "Error Correction in Residue Arithmetic," IEEE Trans. on Comp., vol. C-21, June 1972, pp. 538-545.

[19] F. Barsi, P. Maestrini, "Error Correcting Properties of Redundant Residue Number Systems," IEEE Trans. on Comp., vol. C-22, March 1973, pp. 307-315.

[20] V. Ramachandran, "Single Residue Error Correction in Residue Number Systems," IEEE Trans. on Comp., vol. C-32, May 1983, pp. 504-507.

[21] J. V. Krogmeier, W. K. Jenkins, "Error Detection and Correction in Quadratic Residue Number Systems," Proc. of the 26th Midwest Symp. on Circ. and Syst., Puebla, Mexico, Aug. 1983, pp. 408-411.

[22] W. K. Jenkins, "The Design of Error Checkers for Self-Checking Residue Number Arithmetic," IEEE Trans. on Comp., vol. C-32, April 1983, pp. 388-396.

[23] W. K. Jenkins, "A Technique for the Efficient Generation of Projections for Error Correcting Residue Codes," IEEE Trans. on Circ. and Syst., vol. CAS-31, Feb. 1984, pp. 223-226.

[24] G. R. Redinbo, "Finite Field Fault-Tolerant Digital Filtering Architectures," IEEE Trans. on Comp., vol. C-36, Oct. 1987, pp. 1236-1242.

[25] G. R. Redinbo, "Fault-Tolerant Digital Filtering Architectures Using Fast Finite Field Transforms," Signal Proc., vol. 9, 1985, pp. 37-50.

[26] B. W. LaMacchia, G. R. Redinbo, "RNS Digital Filtering Structures for Wafer-Scale Integration," IEEE J. on Select. Areas in Comm., vol. SAC-4, Jan. 1986, pp. 67-79.

[27] R. P. Brent, H. T. Kung, "The Area-Time Complexity of Binary Multiplication," J. Assoc. Comp. Mach., vol. 28, July 1981, pp. 521-534.

[28] H. T. Kung, B. Sproull, G. Steele (eds.), *VLSI Systems and Computations*. Rockville, Maryland: Computer Science Press, 1981.

[29] Y. Savaria, N. C. Rumin, J. F. Hayes, V. K. Agarwal, "Soft-Error Filtering: A Solution to the Reliability Problem of Future VLSI Digital Circuits," Proc. IEEE, vol. 74, May 1986, pp. 669-683.

[30] W. R. Moore, "A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield," Proc. IEEE, vol. 74, May 1986, pp. 684-698.

[31] F. B. Manning, "An Approach to Highly Integrated, Computer-Maintained Cellular Arrays," IEEE Trans. on Comp., vol. C-26, June 1977, pp. 536-552.

[32] C. A. Finnila, H. H. Love, Jr., "The Associative Linear Array Processor," IEEE Trans. on Comp., vol.C-26, Feb. 1977, pp. 112-125.

[33] R. H. Kuhn, "Yield Enhancement by Fault-Tolerant Systolic Arrays," in *VLSI and Modern Signal Processing,* (S. Y. Kung, H. J. Whitehouse, T. Kailath, eds.). Englewood Cliffs, New Jersey: Prentice-Hall, 1985, pp. 178-184.

[34] A. Avizienis, "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," IEEE Trans. on Comp., vol. C-20, Nov. 1971, pp. 1322-1331.

Chapter X

CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

1. Summary and Conclusions

We have investigated the solution of certain problems involving Toeplitz matrices. These problems were: (i) Toeplitz matrix inversion and/or *LDU* factorization; (ii) Toeplitz system solution; (iii) reflection coefficient computation. The Toeplitz matrix problems were examined from several different viewpoints. The main points may be summarized as follows. We have considered the Schur and split Schur algorithms for the solution of Toeplitz matrix problems, and in the process a Schur algorithm for Hermitian Toeplitz matrices of any rank profile (i.e., the singular leading principal submatrix case) was developed. VLSI/WSI implementable linear parallel-pipelined processor arrays consisting of $O(n)$ processors (n is the order of the matrix) were presented to implement the Schur and split Schur algorithms. These had a time complexity of $O(n)$, except in the singular leading principal submatrix case where the time complexity may be as high as $O(n^2)$. The split Schur algorithms, when implemented on a sequential processing system, represent a more efficient means of computing reflection coefficients than the Schur algorithm, but are not well suited to the problem of *LDU* factorization. This is because the inverse mapping from the split Schur variables to the Schur variables increases the number of multiplications needed, and this makes the split algorithms no more efficient for this application than the Schur algorithm. However, the inverse mapping presents no problem in the context of a parallel processor implementation. The fixed-point arithmetic properties of the Schur and split Schur algorithms were considered, and the algorithms were found to be numerically stable. Only ill-conditioned input data gives poor results. To cope with the ill-conditioned data cases, error-free computation was advocated. In particular, computation in finite rings was shown to be better than computation with Hensel codes or with rational numbers. The finite ring of integers modulo p^r, denoted Z_{p^r}, was studied for the cases where $p = 2^n \pm 1$. The ring Z_{p^r} was advocated mainly because of the ease with which its size can be increased; simply increase r while holding p fixed. As well, large quadratic residue number systems (QRNSs) can be created when p is a Gaussian prime of the form $p = 2^n + 1$. This is potentially useful in the complex data case. To handle complex-valued data with rational-valued real and imaginary parts, the conventional QRNS was extended. Hardware structures for modulo p^r arithmetic (addition, subtraction, multiplication), and for mapping integer data into Z_{p^r} without integer division were presented. These structures are VLSI/WSI implementable. Error-free forms of the Schur algorithm were presented to facilitate their implementation with arithmetic in finite rings.

We conclude that error-free computation must be used to handle the ill-conditioned Toeplitz matrix cases. These cases arise when reflection coefficients have magnitudes at or near unity. Thus, the singular leading principal submatrix case is severely ill-conditioned, and error-free computation is essential to successfully solve such a problem. Furthermore, to solve Toeplitz matrix problems as rapidly as possible, parallel-pipelined processor arrays of the type discussed in this thesis must be employed. Because rings of large size are needed, the ring Z_{p^r} is a logical choice. Finally, we tentatively conclude that fault-tolerant design methods are, or will soon be, sufficiently advanced to permit the actual VLSI/WSI implementation of the Toeplitz matrix problem solution methods that appear in this thesis.

2. Suggestions for Future Research

There is much potential for further work. We have already noted certain open problems, the most important of which are:

(1) It is necessary to investigate means of multiplying matrices by vectors other than the linear systolic array such that the Schur algorithm for Hermitian Toeplitz

matrices of any rank profile will have a time complexity of no worse than $O(n)$ (see Chapter IV, section 4).

(2) An error-free form of the Schur algorithm for Hermitian Toeplitz matrices of any rank profile would be useful.

(3) Error-free forms of the First and Second Back-Substitution Algorithms (Chapter IV) should be "integrated" with the error-free forms of the Schur algorithm (Chapter VII).

(4) The problem of fault-tolerant system design in the context of the problems considered in this thesis needs to be intensively investigated, and this includes design for testability issues as well.

However, many other open problems exist. We list a few more here:

(5) The Levinson-Durbin algorithm of Delsarte, Genin and Kamp [1] produces Levinson polynomials $a_k(z)$, and predictor polynomials $x_k(z)$. The Schur algorithm of Chapter III, section 3 can be modified to produce these polynomials without the need for inner product computation. A parallel processor implementation of the resulting modified Schur algorithm would be desirable.

(6) Split Schur algorithms for the Hermitian Toeplitz case have yet to be developed. The results in Krishna and Morgera [2] may help to solve this problem.

(7) Rissanen's algorithm [3] is applicable to both Hankel and Toeplitz matrices, but it has not yet been investigated to see if it is amenable to parallel processing system implementation. Note that this algorithm applies to the singular submatrix case.

(8) It would be worthwhile to find Schur-like algorithms for the near-to-Toeplitz matrix problems noted in section 2.7 of Chapter II.

(9) An error-free form of the Bareiss algorithm would be useful.

(10) Can the Bareiss algorithm be modified to cope with the singular leading principal submatrix case ?

(11) Investigate error-free forms of Jain's algorithm [4] for banded Toeplitz matrices.

(12) An inverse scattering theory framework has been developed by Bruckstein and Kailath [5] for the systematic development of Schur and Levinson-Durbin algorithms to solve one-dimensional inverse scattering problems (the geophysical example of Chapter II, section 3.5 fits this description). The Gohberg-Semencul formula (see Chapter II, section 2.2) fits into this framework (see Kailath, Bruckstein, Morgan [6]), and so it is reasonable to suspect that Trench's algorithm does so too (this is not shown in [6], and so it needs to be verified formally). Brent and Luk [7] observe that the Trench and Bareiss algorithms are "related", but they do not define this precisely. Thus, is it possible that the Bareiss algorithm may be derived in the inverse scattering theory framework of Bruckstein and Kailath ?

(13) Can abstract algebra reveal methods of solving Toeplitz matrix problems in an error-free manner when the entries of the matrix are *irrational numbers ?*

(14) Prove (or disprove) the conjecture of Chapter VI.

(15) How about extending the results of this thesis to handle block-Toeplitz matrix problems ?

(16) Suppose a parallel processing array is too small to handle the Toeplitz matrix that it is given, and that we cannot make the array larger (due to economic or other constraints). How can we partition the problem to fit the available array ?

(17) It would be interesting to apply the finite precision arithmetic analysis method of Cybenko [8] to the Schur and split Schur algorithms, and to Bareiss's algorithm as well.

(18) Do the split algorithms of Delsarte and Genin [9,10] and of Krishna and Morgera [2] fit into the inverse scattering theory framework of Bruckstein and Kailath ? If so, then how ?

(19) Develop a method to synthesize Toeplitz matrices of any rank profile. This would be a useful means of generating test matrices.

(20) Can Dadda's multipliers [11] be adapted to the problem of modulo p^r multiplication ?

(21) Hardware structures for division (mod p^r inverses) in Z_{p^r} need to be obtained.

(22) The problem of mapping from the product ring

$$Z_{p_1^r} \times Z_{p_2^r} \times Z_{p_3^r}$$

where $p_1 = 2^n + 1$, $p_2 = 2^n$, $p_3 = 2^n - 1$, back to the ring of integers needs to be studied, as does the problem of scaling in such a product ring.

(23) Computation in Z_{p^r} for $p = 2^n \pm k$ ($k =$ small and odd positive integer other than unity) should be examined.

(24) The constant factors and lower order terms of the asymptotic area and time complexity expression of Chapter VIII, section 1.5 should be worked out. A good first approximation can be found by counting gates and levels of gates.

(25) Hardware structures for the inverse mapping from Z_{p^r} to F_N require development.

(26) It might be best to implement any error-free form of the Schur algorithm for Hermitian Toeplitz matrices of any rank profile in a multimicroprocessor-based parallel processing system. Traditional microprocessor designs are not optimized to support error-free computation. Thus, it might prove useful to develop a microprocessor optimized for error-free computation problems. Such a microprocessor would be useful in many other kinds of problems where computation in finite rings and fields is performed.

(27) Can free accumulation (defined in Cappello and Steiglitz [12]) be used to remove the parallel processing bottleneck that the inner product operation in the Levinson-Durbin algorithm represents ?

This concludes our list of future research topics. No doubt many others could be added to the list.

REFERENCES

[1] P. Delsarte, Y. V. Genin, Y. Kamp, "A Generalization of the Levinson Algorithm for Hermitian Toeplitz Matrices of Any Rank Profile," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-33, Aug. 1985, pp. 964-971.

[2] H. Krishna, S. D. Morgera, "The Levinson Recurrence and Fast Algorithms for Solving Toeplitz Systems of Linear Equations," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, June 1987, pp. 839-848.

[3] J. Rissanen, "Solution of Linear Equations with Hankel and Toeplitz Matrices," Numerische Mathematik, vol. 22, 1974, pp. 361-366.

[4] A. K. Jain, "Fast Inversion of Banded Toeplitz Matrices by Circular Decompositions," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-26, April 1978, pp. 121-126.

[5] A. Bruckstein, T. Kailath, "An Inverse Scattering Framework for Several Problems in Signal Processing," IEEE ASSP Magazine, vol. 4, Jan. 1987, pp. 6-20.

[6] T. Kailath, A. Bruckstein, D. Morgan, "Fast Matrix Factorizations Via Discrete Transmission Lines," Lin. Alg. and its Appl., vol. 75, 1986, pp. 1-25.

[7] R. P. Brent, F. T. Luk, "A Systolic Array for the Linear-Time Solution of Toeplitz Systems of Equations," J. of VLSI and Comp. Syst., vol. 1, 1983, pp. 1-22.

[8] G. Cybenko, "The Numerical Stability of the Levinson-Durbin Algorithm for Toeplitz Systems of Equations," SIAM J. of Sci. and Stat. Comp., vol. 1, Sept. 1980, pp. 303-319.

[9] P. Delsarte, Y. V. Genin, "The Split Levinson Algorithm," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-34, June 1986, pp. 47-478.

[10] P. Delsarte, Y. V. Genin, "On the Splitting of Classical Algorithms in Linear Prediction Theory," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, May 1987, pp. 645-653.

[11] L. Dadda, "Some Schemes for Parallel Multipliers," Alta Frequenza, vol. 34, 1965, pp. 349-356.

[12] P. R. Cappello, K. Steiglitz, "A Note on "Free Accumulation" in VLSI Filter Architectures," IEEE Trans. on Circ. and Syst., vol. CAS-32, March 1985, pp. 291-296.

APPENDIX A
Finite Precision Arithmetic Simulator
for the Schur Algorithm

```
#nolist
#include <stdio.h>
#include <math.h>
#list

#define     CLIM        10      /* Max. no. of autocor. coeffs.     */
#define     LIMIT     12000     /* Maximum number of data points    */

#define     finpar    "PAR"     /* Input data to the simulator      */
#define     ffout    "FOUT"     /* Output data sequence             */

FILE        *inpp;  /* declare pointer to input data file          */
FILE        *oupp;  /* declare pointer to output data file         */


double btof();



siggen(signal,length,rho,theta)
                 /* This function models a 2nd order AR process     */
                 /* with a single complex pole pair (modulus rho,   */
                 /* argument theta (degrees)).  A 2nd order all-pole*/
                 /* filter is driven by a                           */
                 /* Gaussian noise generator routine based on the   */
                 /* algorithm described in:                         */
                 /*   Rabiner and Gold,"Theory and Application of   */
                 /*   Digital Signal Processing"                    */
double signal[];
double *rho,*theta;
int length;
{
  double var,x,y,w;
  double a1,a2;
  double pie;
  int i;

  printf("      - SIGNAL MODEL PARAMETERS -      \n");
  printf("   Enter desired noise variance: \n");
  scanf("%f",&var);
  printf("   Enter desired pole modulus: \n");
  scanf("%f",rho);
  printf("   Enter desired pole argument (degrees): \n");
  scanf("%f",theta);

  fprintf(oupp,"%s"," - 2nd Order AR Process Parameters - \n");
  fprintf(oupp,"\n");
  fprintf(oupp,"%s %12.6f \n","  pole modulus = ",*rho);
  fprintf(oupp,"%s %12.6f \n","  pole angle (degrees) = ",*theta);
  fprintf(oupp,"%s %12.6f \n","  noise variance = ",var);
  fprintf(oupp,"\n\n");

  pie = 3.141592654;
  *theta = *theta * pie /180.0; /* convert to radians */
  a1 = -2.0*cos(*theta)* *rho;
  a2 = *rho * *rho;

 for (i=0; i<=(length - 1); ++i)
   {
  x = randl();
  y = sqrt(2.0*var*log(1.0/x));
  w = y*cos(2.0*pie*randl()); /* w is a noise point */
  if ( i == 0 )
    {
    signal[0] = w;
    }
  if ( i == 1 )
    {
    signal[1] = w - a1 * signal[0];
    }
  if ( i > 1 )
    {
    signal[i] = w - a1 * signal[i-1] - a2 * signal[i-2];
    }
  }
}


autocorrelate(s,r,l,nco,nseg)
               /* Compute nominal ensemble of normalized         */
               /* autocorrelation coefficients                   */
```

```
                     /* (must have length >= l * nseg).                 */
  int l,nco,nseg;
  double s[],r[][CLIM];
  {
    double sum1,sum2;
    int i,j,n;

    for (i=0; i<=(nseg-1) ; i++)
      {
      sum2 = 0.0;
      for (n=0; n<=(l-1) ; n++)
        {
        sum2 = sum2 + s[n + i*l]*s[n + i*l];
        }
      sum2 = sum2/l;
      for (j=1; j<=nco; j++)
        {
        sum1 = 0.0;
        for (n=0; n<=(l-j-1) ; n++)
          {
          sum1 = sum1 + s[n + i*l]*s[n + i*l + j];
          }
        sum1 = sum1/(l - j);
        r[i][j] = sum1/sum2;
        }
     r[i][0] = 1.0;
      }
   }


 quant(x,n)        /* Argument x is a 2n-bit number that we want to   */
                   /* round off to n-bits.  Argument x is a "standard */
                   /* format" 2n-bit number.  This function assumes   */
                   /* that the computer uses 2s complement arithmetic */
                   /* for integer arithmetic itself.  This function   */
                   /* can be used for "double-precision" multiply-    */
                   /* accumulate operations.                          */
 int x,n;
 {
   int q,mask,roun;

   if (x >= 0)
     {
     mask = 01;
     mask = mask << (n-2);
     roun = 0;
     if ( (mask & x) != 0 )
       {
       roun = 1;
       }
     q = x >> (n-1);
     q = q + roun;
     }
  else
     {
     q = ~x;
     q = q + 1; /* q = 2s compl. of x now. */
     mask = 01;
     mask = mask << (n-2);
     roun = 0;
     if ( (mask & q) != 0 )
       {
       roun = 1;
       }
     q = q >> (n-1);
     q = q + roun;
     q = ~q;
     q = q + 1; /* restore true sign of q */
     }
  return(q);
}


ftob(x,n)              /* Convert the double precision floating */
                       /* point number x into a standard format */
                       /* word.                                 */
int n;
double x;
{
  int i,c,mask,masks,sign,ix;
  double fx;
```

```
    sign = 0;
    if (x < 0.0)
      {
      sign = -1;
      x = -x;
      }
    ix = x;                   /* find integer part of x    */
    fx = x - (double)ix;   /* find fractional part of x */
    c = 0;
    mask = 01;
    for (i=(n-2); i>=0; i--)
      {
      fx = 2.0*fx;
      masks = mask << i;
      if (fx >= 1.0)
        {
        fx = fx - 1.0;
        c = c | masks;
        }
      }
    fx = 2.0 * fx;
    if (fx >= 1.0)
      {  /* Add unity to effect the rounding operation */
      c = c + 1;
      }
    ix = ix << (n-1);
    c = c + ix;
    if ( sign == -1 )
      {
      c = -c;
      }
    return(c);
  }


  double btof(y,n)              /* Convert standard format y into float */
                                /* type.                                */
  int n,y;
  {
    int i,j,sign,mask,masks;
    double c;

    c = 0.0;
    sign = 0;
    mask = 01;

    if ( ((mask << (n-1)) & y) != 0 )
      {
      sign = -1;
      }
    for (i=(n-2); i>=0; i--)
      {
      masks = mask << i;
      if ( (masks & y) != 0 )
        {
        j = n - i - 1;
        c = c + pow(2.0,(double) -j);
        }
      }
    if (sign == -1)
      {
      c = -1.0 + c;
      }
    return(c);
  }


divide(x,y,n)     /* Find the n-bit, 2s complement (standard format)  */
                  /* coding of x/y.  x and y are standard format nos. */
int x,y,n;
{
  double xf,yf;

  xf = btof(x,n);
  yf = btof(y,n);
  return(ftob(xf/yf,n));
}
```

```
main ()  /*                      FP_SCHUR_SIM                          */
         /*        (Finite Precision Schur Algorithm Simulator)         */
         /*                                                             */
         /* We use fixed-point 2s complement arithmetic (n-bits,        */
         /* including sign) with format (standard format):              */
         /*                     x | x x x  ... x   .                     */
         /*                     0   1 2 3      k                         */
         /* where x  is the sign bit, and n = k + 1.                    */
         /*        0                                                     */
         /* We use integer  types to contain standard format binary     */
         /* numbers.  The rightmost (least significant) bit of an       */
         /* integer  type corresponds to x .                            */
         /*                                k                             */
         /*                                                             */
         /* This program simulates the Schur algorithm for symmetric    */
         /* Toeplitz matrices under finite precision arithmetic         */
         /* conditions.  The purpose of this program is to test the     */
         /* theoretical predictions derived in the paper:               */
         /*                                                             */
         /*    C. J. Zarowski, H. C. Card, "Finite Precision            */
         /*    Arithmetic and the Schur Algorithm," to be submitted     */
         /*    to the IEEE Trans. on Acoust., Speech, and Signal        */
         /*    Proc.                                                    */

{
  int fp_ul[CLIM][CLIM];       /* negatively indexed u-parameters        */
                               /* (n-bit, 2s complement)                 */
  int fp_uu[CLIM][CLIM];       /* nonnegatively indexed u-parameters     */
                               /* (n-bit, 2s complement)                 */
  double ul[CLIM][CLIM];       /* negatively indexed u-parameters        */
                               /* (double precision floating-point)      */
  double uu[CLIM][CLIM];       /* nonnegatively indexed u-parameters     */
                               /* (double precision floating-point)      */
  double r[CLIM][CLIM];        /* ensemble of normalized autocorrel.     */
                               /* coefficients (nominal) constructed     */
                               /* from segments of signal[];             */
                               /*  r[i][j] - i = segment index           */
                               /*          - j = jth coeff. of seg. i    */
 double mean[CLIM];            /* mean of reflection coefficients in     */
                               /* array fp_k[][]                         */
 double vari[CLIM];            /* variance of reflection coefficients    */
                               /* in array fp_k[][] about their mean     */
                               /* values in mean[]                       */
 double signal[LIMIT];         /* signal generated by siggen function    */
                               /* (double precision floating-point)      */
 double no_k[CLIM][CLIM];      /* ensemble of nominal reflection         */
                               /* coefficients                           */
 double bigk;                  /* double precision floating-point        */
                               /* reflection coefficient                 */
 int fp_k[CLIM][CLIM];         /* ensemble of finite precision (n-bit,   */
                               /* 2s complement) reflection coeffs.      */
 double rho;                   /* pole modulus of 2nd order AR model     */
 double theta;                 /* pole argument of 2nd order AR model    */
 double r1,r2;                 /* theoretical values for r[1],r[2]       */
                               /* assuming 2nd order AR process          */
 double var_bigk_3;            /* theoretical variance of index 3        */
                               /* reflection coefficient assuming no     */
                               /* error in our knowledge of r[1],r[2]    */
 double t1,t2,denom;           /* temporary variables                    */
 int i,j,k;                    /* loop counter variables                 */
 int n;                        /* number of bits (including sign bit)    */
 int length;                   /* length of signal[]                     */
 int l;                        /* number of points used to get r[]       */
 int nco;                      /* largest lag value AND also the         */
                               /* number of reflection coeffs. to be     */
                               /* computed                               */
 int prod1,prod2;              /* temporary integers                     */
 int nseg;                     /* number of segments of signal[] used    */
                               /* to compute r[][] (length >=l*nseg)     */

inpp = fopen(finpar,"r");
oupp = fopen(ffout,"w");

fscanf(inpp,"%d %d %d %d %d",&n,&length,&l,&nco,&nseg);

siggen(signal,length,&rho,&theta);  /* construct the test signal    */
autocorrelate(signal,r,l,nco,nseg); /* find floating-pt. autocorr.  */
                                    /* coefficients                 */
```

```
    fprintf(oupp,"%s","  Theoretical Index 3 Reflection Coefficient \n");
    fprintf(oupp,"%s","                        Variance                \n");
    fprintf(oupp,"\n");
    r1 = 2*rho*cos(theta)/(1 + rho*rho);
    denom = (1 + rho*rho) * sin(theta);
    r2 = rho*rho*(sin(3*theta)-rho*rho*sin(theta))/denom;
    fprintf(oupp,"%s %10.6f \n"," theoretical r[1] = ",r1);
    fprintf(oupp,"%s %10.6f \n"," theoretical r[2] = ",r2);
    t1 = r1*r1*(r2-1)*(r2-1) + (r2-r1*r1)*(r2-r1*r1);
    t2 = (r1*r1 - 1)*(r1*r1 - 1)*(1 + (r1*r1 - 1)*(r1*r1 - 1));
    var_bigk_3 = (t1 + t2) * pow(2.0,-2.0*(float)n)/3.0;
    t1 = (r1*r1 - 1) * (r1*r1 - 1) * (r1*r1 - 1);
    var_bigk_3 = var_bigk_3/t1;
    t1 = (r1*r1 - 1)*(r1*r1 - 1);
    t2 = (t1 + 4*r1*r1*(r2 - 1)*(r2 - 1))/(t1 * t1);
    t2 = t2 * pow(2.0,-2.0*(float)n )/3.0;
                    /* t2 is the component of var_bigk_3 that is due to */
                    /* rounding of the nominal normalized autocorrel.   */
                    /* coefficients                                     */
   var_bigk_3 = var_bigk_3 + t2;
   fprintf(oupp,"%s %14.10f \n"," variance = ",var_bigk_3);
   fprintf(oupp,"\n\n");

   fprintf(oupp,"%s","  Ensemble of Nominal and Normalized       \n");
   fprintf(oupp,"%s","      Autocorrelation Coefficients         \n");
   fprintf(oupp,"\n");
   for (i=0; i < nseg; i++)
     {
     for (k=0; k<=nco; k++)
       {
       fprintf(oupp,"%s %d %s %9.6f ","r[",k,"] = ",r[i][k]);
       }
     fprintf(oupp,"\n");
     }
  fprintf(oupp,"\n\n");

     /* compute ensemble of nominal reflection coefficients */

  fprintf(oupp,"%s"," Ensemble of Nominal Reflection Coefficients \n");
  fprintf(oupp,"\n");
  for (j=0; j < nseg; j++)
    {
    for (k=0; k<=nco; k++)
      {
      ul[1][k] = r[j][k];
      uu[1][k] = r[j][k];
      }
    for (i=1; i<=nco; i++)
      {
      bigk = -uu[i][1]/ul[i][i-1];
      no_k[j][i+1] = bigk;
      fprintf(oupp,"%s%d%s%10.6f"," K(",i+1,") = ",bigk);
      for (k=0; k<=(nco-i); k++)
        {
        ul[i+1][k+i] = ul[i][k+i-1] + bigk*uu[i][k+1];
        uu[i+1][k] = bigk*ul[i][k+i-1] + uu[i][k+1];
        }
      }
    fprintf(oupp,"\n");
    }
 fprintf(oupp,"\n\n");

    /* compute ensemble of fixed-point reflection coefficients */
    /* assuming that there is no quantization error in the     */
    /* autocorrelation coefficient estimates except that due   */
    /* to rounding to n-bit, 2s complement numbers              */

fprintf(oupp,"%s","    Ensemble of Fixed-Point Reflection  \n");
fprintf(oupp,"%s","                 Coefficients           \n");
fprintf(oupp,"%s","   (error-free autocorrelation coeffs.) \n");
fprintf(oupp,"\n");
t1 = 1.0 - pow(2.0,-( (float)n - 1.0 ));
for (j=0; j < nseg; j++)
  {
  for (k=0; k<=nco; k++)
    {
    fp_ul[1][k] = ftob(r[j][k],n);
    fp_uu[1][k] = ftob(r[j][k],n);
    fp_ul[1][0] = ftob(t1,n);
    fp_uu[1][0] = ftob(t1,n);
```

```
        }
      for (i=1; i<=nco; i++)
        {
        fp_k[j][i+1] = divide(-fp_uu[i][1],fp_ul[i][i-1],n);
       fprintf(oupp,"%s%d%s%10.6f"," K(",i+1,") = ",btof(fp_k[j][i+1],n));
        for (k=0; k<=(nco-i); k++)
          {
          prod1 = quant(fp_k[j][i+1]*fp_uu[i][k+1],n);
          prod2 = quant(fp_k[j][i+1]*fp_ul[i][k+i-1],n);
          fp_ul[i+1][k+i] = fp_ul[i][k+i-1] + prod1;
          fp_uu[i+1][k] = prod2 + fp_uu[i][k+1];
          }
        }
     fprintf(oupp,"\n");
     }
  fprintf(oupp,"\n\n");
  fprintf(oupp,"%s %d %s \n"," Wordsize = ",n," bits ");
  fprintf(oupp,"\n\n");

      /* compute and output the means and variances of the   */
      /* reflection coefficients                             */

  fprintf(oupp,"%s","    Fixed-Point Reflection Coefficient    \n");
  fprintf(oupp,"%s","              Means and Variances         \n");
  fprintf(oupp,"\n");
  for (j=2; j<=(nco+1) ; j++)
    {
    mean[j] = 0.0;
    for (i=0; i < nseg; i++)
      {
      mean[j] = mean[j] + btof(fp_k[i][j],n);
      }
    mean[j] = mean[j]/nseg;
    }
  for (j=2; j<=(nco+1) ; j++)
    {
    vari[j] = 0.0;
    for (i=0; i < nseg; i++)
      {
      bigk = btof(fp_k[i][j],n);
      vari[j] = vari[j] + (no_k[i][j] - bigk)*(no_k[i][j] - bigk);
      }
    vari[j] = vari[j]/nseg;
    }
 fprintf(oupp,"%s\n","         Mean         Variance          ");
 for(j=2; j<=(nco+1); j++)
   {
   fprintf(oupp,"%s%d%s %11.8f %14.8f \n","K(",j,")",mean[j],vari[j]);
   }
}
```

```
File fout printed on Wed Aug 19 12:24:33 1987
                                                                              page 1 of 1
------------------------------------------------------------------------------------------
 - 2nd Order AR Process Parameters -

  pole modulus =       0.750000
  pole angle (degrees) =       5.000000
  noise variance =       1.000000


  Theoretical Index 3 Reflection Coefficient
                 Variance

 theoretical r[1] =    0.956346
 theoretical r[2] =    0.866561
 variance =    0.0005879122


  Ensemble of Nominal and Normalized
     Autocorrelation Coefficients

r[ 0 ] =    1.000000 r[ 1 ] =    0.959886 r[ 2 ] =    0.876623
r[ 0 ] =    1.000000 r[ 1 ] =    0.952875 r[ 2 ] =    0.860052
r[ 0 ] =    1.000000 r[ 1 ] =    0.958227 r[ 2 ] =    0.871039
r[ 0 ] =    1.000000 r[ 1 ] =    0.955963 r[ 2 ] =    0.864753
r[ 0 ] =    1.000000 r[ 1 ] =    0.956819 r[ 2 ] =    0.864804
r[ 0 ] =    1.000000 r[ 1 ] =    0.957402 r[ 2 ] =    0.870561
r[ 0 ] =    1.000000 r[ 1 ] =    0.952390 r[ 2 ] =    0.856361
r[ 0 ] =    1.000000 r[ 1 ] =    0.954367 r[ 2 ] =    0.864344
r[ 0 ] =    1.000000 r[ 1 ] =    0.948484 r[ 2 ] =    0.840988
r[ 0 ] =    1.000000 r[ 1 ] =    0.955099 r[ 2 ] =    0.865224


 Ensemble of Nominal Reflection Coefficients

 K(2) =  -0.959886 K(3) =   0.569323
 K(2) =  -0.952875 K(3) =   0.520708
 K(2) =  -0.958227 K(3) =   0.576527
 K(2) =  -0.955963 K(3) =   0.570178
 K(2) =  -0.956819 K(3) =   0.600031
 K(2) =  -0.957402 K(3) =   0.552373
 K(2) =  -0.952390 K(3) =   0.545299
 K(2) =  -0.954367 K(3) =   0.521081
 K(2) =  -0.948484 K(3) =   0.584127
 K(2) =  -0.955099 K(3) =   0.535295


   Ensemble of Fixed-Point Reflection
              Coefficients
  (error-free autocorrelation coeffs.)

K(2) =  -0.960937 K(3) =   0.589843
K(2) =  -0.955078 K(3) =   0.578125
K(2) =  -0.960937 K(3) =   0.666015
K(2) =  -0.957031 K(3) =   0.582031
K(2) =  -0.958984 K(3) =   0.658203
K(2) =  -0.958984 K(3) =   0.585937
K(2) =  -0.955078 K(3) =   0.623046
K(2) =  -0.957031 K(3) =   0.582031
K(2) =  -0.951171 K(3) =   0.632812
K(2) =  -0.957031 K(3) =   0.582031


Wordsize =  10  bits


  Fixed-Point Reflection Coefficient
          Means and Variances

          Mean            Variance
K(2) -0.95722656       0.00000469
K(3)  0.60800781       0.00306908
```

APPENDIX B

Summary Tables for Output from the

Program in Appendix A

Wordlength (bits)	$var[\Delta K^{(3)}]\ 10^4$	
	Theoretical	Experimental
6	38.1	34300
8	2.38	15.5
10	0.149	0.960
12	0.0093	0.0768
14	0.00058	0.0042
16	0.000036	0.0001

Table I: Comparison of theoretical and experimental results for the finite precision Schur algorithm. Here $\rho = 0.9375$ and $\theta = 45°$.

Wordlength (bits)	$var[\Delta K^{(3)}]\ 10^4$	
	Theoretical	Experimental
10	89.9	24100
12	5.62	20.5
14	0.351	1.63
16	0.022	0.106

Table II: Comparison of theoretical and experimental results for the finite precision Schur algorithm. Here $\rho = 0.875$ and $\theta = 5°$.

Wordlength (bits)	$var[\Delta K^{(3)}]\ 10^4$	
	Theoretical	Experimental
8	94.1	5060
10	5.88	30.7
12	0.367	2.00
14	0.023	0.0633
16	0.0014	0.0078

Table III: Comparison of theoretical and experimental results for the finite precision Schur algorithm. Here $\rho = 0.75$ and $\theta = 5°$.

APPENDIX C

Finite Precision Arithmetic Simulator

for the Symmetric Split Schur Algorithm

(Checks Expression for $Var[\Delta K_2]$)

```
#nolist
#include <stdio.h>
#include <math.h>
#list

#define     CLIM        12      /* Max. no. of autocor. coeffs.      */
#define    LIMIT     12000      /* Maximum number of data points     */

#define    finpar    "PAR"      /* Input data to the simulator       */
#define    ffout    "FOUT"      /* Output data sequence              */

FILE       *inpp;  /* declare pointer to input data file --           */
FILE       *oupp;  /* declare pointer to output data file             */


double btof();



siggen(signal,length,rho,theta)
                 /* This function models a 2nd order AR process     */
                 /* with a single complex pole pair (modulus rho,   */
                 /* argument theta (degrees)).  A 2nd order all-pole*/
                 /* filter is driven by a                           */
                 /* Gaussian noise generator routine based on the   */
                 /* algorithm described in:                         */
                 /*   Rabiner and Gold,"Theory and Application of   */
                 /*   Digital Signal Processing"                    */
double signal[];
double *rho,*theta;
int length;
{
  double var,x,y,w;
  double a1,a2;
  double pie;
  int i;

  printf("      - SIGNAL MODEL PARAMETERS -      \n");
  printf("   Enter desired noise variance: \n");
  scanf("%f",&var);
  printf("   Enter desired pole modulus: \n");
  scanf("%f",rho);
  printf("   Enter desired pole argument (degrees): \n");
  scanf("%f",theta);

  fprintf(oupp,"%s"," - 2nd Order AR Process Parameters - \n");
  fprintf(oupp,"\n");
  fprintf(oupp,"%s %12.6f \n","  pole modulus = ",*rho);
  fprintf(oupp,"%s %12.6f \n","  pole angle (degrees) = ",*theta);
  fprintf(oupp,"%s %12.6f \n","  noise variance = ",var);
  fprintf(oupp,"\n\n");

  pie = 3.141592654;
  *theta = *theta * pie /180.0; /* convert to radians */
  a1 = -2.0*cos(*theta)* *rho;
  a2 = *rho * *rho;

 for (i=0; i<=(length - 1); ++i)
   {
   x = randl();
   y = sqrt(2.0*var*log(1.0/x));
   w = y*cos(2.0*pie*randl()); /* w is a noise point */
   if ( i == 0 )
     {
     signal[0] = w;
     }
   if ( i == 1 )
     {
     signal[1] = w - a1 * signal[0];
     }
   if ( i > 1 )
     {
     signal[i] = w - a1 * signal[i-1] - a2 * signal[i-2];
     }
  }
}


autocorrelate(s,r,l,nco,nseg)
               /* Compute nominal ensemble of normalized          */
               /* autocorrelation coefficients                    */
```

```
                  /* (must have length >= l * nseg).                    */
int l,nco,nseg;
double s[],r[][CLIM];
{
  double sum1,sum2;
  int i,j,n;

  for (i=0; i<=(nseg-1) ; i++)
    {
    sum2 = 0.0;
    for (n=0; n<=(l-1) ; n++)
      {
      sum2 = sum2 + s[n + i*l]*s[n + i*l];
      }
    sum2 = sum2/l;
    for (j=1; j<=nco; j++)
      {
      sum1 = 0.0;
      for (n=0; n<=(l-j-1) ; n++)
        {
        sum1 = sum1 + s[n + i*l]*s[n + i*l + j];
        }
      sum1 = sum1/(l - j);
      r[i][j] = sum1/sum2;
      }
   r[i][0] = 1.0;
   }
 }


quant(x,n) /* Argument x is a 2(n+m)-bit number that we want to      */
           /* round off to 2m+n+1-bits.                              */
           /* The fractional part of the quantized product is        */
           /* n-1-bits long.  The integer part is not altered.       */
           /* Argument x is a "standard format"                      */
           /* 2(n+m)-bit product.  This function assumes             */
           /* that the computer uses 2s complement arithmetic        */
           /* for integer arithmetic itself.  This function          */
           /* can be used for "double-precision" multiply-           */
           /* accumulate operations.                                 */
int x,n;
{
  int q,mask,roun;

  if (x >= 0)
    {
    mask = 01;
    mask = mask << (n-2);
    roun = 0;
    if ( (mask & x) != 0 )
      {
      roun = 1;
      }
    q = x >> (n-1);
    q = q + roun;
    }
 else
    {
    q = ~x;
    q = q + 1; /* q = 2s compl. of x now. */
    mask = 01;
    mask = mask << (n-2);
    roun = 0;
    if ( (mask & q) != 0 )
      {
      roun = 1;
      }
    q = q >> (n-1);
    q = q + roun;
    q = ~q;
    q = q + 1; /* restore true sign of q */
    }
  return(q);
}


ftob(x,n)           /* Convert the double precision floating */
                    /* point number x into a standard format */
                    /* word.                                 */
int n;
double x;
```

```
  {
    int i,c,mask,masks,sign,ix;
    double fx;

    sign = 0;
    if (x < 0.0)
      {
       sign = -1;
       x = -x;
       }
    ix = x;                    /* find integer part of x    */
    fx = x - (double)ix;       /* find fractional part of x */
    c = 0;
    mask = 01;
    for (i=(n-2); i>=0; i--)
      {
      fx = 2.0*fx;
      masks = mask << i;
      if (fx >= 1.0)
        {
        fx = fx - 1.0;
        c = c | masks;
        }
      }
   fx = 2.0 * fx;
   if ( fx >= 1.0)
     { /* Add unity to effect the rounding operation */
     c = c + 1;
     }
   ix = ix << (n-1);
   c = c + ix;
   if ( sign == -1 )
     {
     c = -c;
     }
   return(c);
}


double btof(x,n,m)   /* Convert standard format x into double */
                     /* precision floating-point number.      */
int n,m,x;
{
  int i,j,sign,mask,masks;
  double c;

  c = 0.0;
  sign = 0;
  mask = 01;
  if (x < 0)
    {
    sign = -1;
    x = -x;
    }
  for (i=(n-2); i >= 0; i--)
    {
    masks = mask << i;
    if ( (masks & x) != 0 )
      {
      j = n - i - 1;
      c = c + pow(2.0,(double)-j);
      }
    }
  j = 0;
  for ( i=(n-1); i <= (n+m-1); i++)
    {
    masks = mask << i;
    if ( (masks & x) != 0)
      {
      c = c + pow(2.0,(double)j);
      }
    j = j + 1;
    }
  if ( sign == -1 )
    {
    c = -c;
    }
  return(c);
}
```

```
divide(x,y,n,m) /* Find the n+m-bit, 2s complement (standard    */
                /* format) coding of x/y.  x and y are standard */
                /* format n+m-bit binary numbers.               */
int x,y,n,m;
{
  double xf,yf;

  xf = btof(x,n,m);
  yf = btof(y,n,m);
  return(ftob(xf/yf,n));
}



main ()   /*                  FP_SPLIT_SCHUR_SIM                    */
          /*  (Finite Precision Split Schur Algorithm Simulator)    */
          /*                                                        */
          /* We use fixed-point 2s complement arithmetic (n+m - bits, */
          /* including sign) with format (standard format):         */
          /*              x   ... x | x x x  ... x  ,               */
          /*               -m       0   1 2 3       k               */
          /* where x  is the sign bit, and n = k + 1.               */
          /*        -m                                              */
          /* We use integer  types to contain standard format binary */
          /* numbers.  The rightmost (least significant) bit of an  */
          /* integer  type corresponds to x .                       */
          /*                                k                       */
          /*                                                        */
          /* This program simulates the split Schur algo. for symm. */
          /* Toeplitz matrices under finite precision arithmetic    */
          /* conditions.                                            */
          /* The split Schur algorithm simulated is taken from the  */
          /* paper:                                                 */
          /*                                                        */
          /*    P. Delsarte, Y. Genin, "On the Splitting of Classical */
          /*    Algorithms in Linear Prediction Theory," IEEE Trans. */
          /*    on Acoust., Speech, and Signal Proc., vol. ASSP-35, */
          /*    pp. 645 - 653, May 1987.                            */
          /*                                                        */

{
  int fp_v[CLIM][CLIM];          /* nonnegatively indexed v-parameters   */
                                 /* (n+m-bit, 2s complement)             */
  double v[CLIM][CLIM];          /* nonnegatively indexed v-parameters   */.
                                 /* (double precision floating-point)    */
  double r[CLIM][CLIM];          /* ensemble of normalized autocorrel.   */
                                 /* coefficients (nominal) constructed   */
                                 /* from segments of signal[];           */
                                 /*  r[i][j] - i = segment index         */
                                 /*          - j = jth coeff. of seg. i  */
  double mean[CLIM];             /* mean of reflection coefficients in   */
                                 /* array fp_k[][]                       */
  double vari[CLIM];             /* variance of reflection coefficients  */
                                 /* in array fp_k[][] about their mean   */
                                 /* values in mean[]                     */
  double signal[LIMIT];          /* signal generated by siggen function  */
                                 /* (double precision floating-point)    */
  double no_k[CLIM][CLIM];       /* ensemble of nominal reflection       */
                                 /* coefficients                         */
  double no_a[CLIM][CLIM];       /* ensemble of nominal split Schur      */
                                 /* reflection coefficients              */
  double bigk;                   /* double precision floating-point      */
                                 /* reflection coefficient               */
  double alphak;                 /* split Schur reflection coefficient   */
                                 /* (double precision floating-point)    */
  int fp_k[CLIM][CLIM];          /* ensemble of finite precision (n+m-bit*/
                                 /* , 2s complement) reflection coeffs.  */
  int fp_a[CLIM][CLIM];          /* ensemble of finite precision (n+m-bit*/
                                 /* , 2s complement) split Schur         */
                                 /* reflection coefficients              */
  int fp_bigk;                   /* fixed-point, 2s comp. version of     */
                                 /* variable bigk                        */
  int fp_alphak;                 /* fixed-point, 2s comp. version of     */
                                 /* variable alphak                      */
  double rho;                    /* pole modulus of 2nd order AR model   */
  double theta;                  /* pole argument of 2nd order AR model  */
  double r1,r2;                  /* theoretical values for r[1],r[2]     */
                                 /* assuming 2nd order AR process        */
  int one;                      /* standard format representation of   */
                                /* the number 1 (one)                  */
  int i,j,k;                    /* loop counter variables              */
  int n,m;                      /* number of bits (including sign bit) */
                                /* in the fixed-point word is n+m      */
  int length;                   /* length of signal[]                  */
  int l;                        /* number of points used to get r[]    */
  int nco;                      /* largest lag value AND also the      */
                                /* number of reflection coeffs. to be  */
                                /* computed                            */
  int prod;                     /* temporary integer product           */
  double denom;                 /* temporary variable                  */
  double tr1,tr2,t1,t2,t3;      /* more temporary variables            */
  double t4,t5;                 /* still more temporary variables      */
  double var_bigk_2;            /* theoretical variance of the 2nd     */
                                /* reflection coefficient              */
  int nseg;                     /* number of segments of signal[] used */
                                /* to compute r[][] (length >=l*nseg)  */

  inpp = fopen(finpar,"r");
  oupp = fopen(ffout,"w");

  fscanf(inpp,"%d %d %d %d %d %d",&n,&m,&length,&l,&nco,&nseg);

  siggen(signal,length,&rho,&theta);  /* construct the test signal    */
  autocorrelate(signal,r,l,nco,nseg); /* find floating-pt. autocorr. */
                                      /* coefficients                 */

  fprintf(oupp,"%s","  Theoretical 2nd Reflection Coefficient      \n");
  fprintf(oupp,"%s","                Error Variance                \n");
  fprintf(oupp,"\n");
  r1 = 2*rho*cos(theta)/(1 + rho*rho);
  denom = (1 + rho*rho) * sin(theta);
  r2 = rho*rho*(sin(3*theta)-rho*rho*sin(theta))/denom;
  fprintf(oupp,"%s %10.6f \n"," theoretical r[1] = ",r1);
  fprintf(oupp,"%s %10.6f \n"," theoretical r[2] = ",r2);
  denom = (1.0-r1)*(1.0-r1*r1)*(1.0+r1)*(1.0+r1);
  tr1 = (2.0*r1*r1-r2-1.0)*(1.0+r1)*(1.0+r1) +
        (1.0-r1*r1)*(2.0*r1*r1+4.0*r1+r2+1.0);
  tr2 = (1.0-r1*r1)*(1.0+r1);
  t1 = (2.0*r1*r1-r2-1.0)*(1.0+r1)*(1.0+r1) +
       2.0*r1*(1.0+r1)*(1.0-r1*r1);
  t2 = (1.0+r1)*(1.0+r1)*(1.0-r1*r1);
  t3 = (1.0+r2-2.0*r1*r1)*(1.0+r1)*(1.0+r1);
  t4 = (1.0-r1)*(1.0-r1*r1)*(1.0+r1)*(1.0+r1);
  t5 = (1.0+r1)*(1.0-r1*r1);
  var_bigk_2 = (tr1*tr1+tr2*tr2+t1*t1+t2*t2+t3*t3+t4*t4+t5*t5)
               /(denom*denom);
  var_bigk_2 = var_bigk_2 * pow(2.0,-2.0*(double)n) / 3.0;
  fprintf(oupp,"%s %14.8f \n"," Variance of K(2) = ",var_bigk_2);
  fprintf(oupp,"\n\n");

  fprintf(oupp,"%s","  Ensemble of Nominal and Normalized      \n");
  fprintf(oupp,"%s","     Autocorrelation Coefficients         \n");
  fprintf(oupp,"\n");
  for (i=0; i < nseg; i++)
    {
    for (k=0; k<=nco; k++)
      {
      fprintf(oupp,"%s %d %s %9.6f ","r[",k,"] = ",r[i][k]);
      }
    fprintf(oupp,"\n");
    }
  fprintf(oupp,"\n\n");

    /* compute ensemble of nominal reflection coefficients */

  fprintf(oupp,"%s"," Ensemble of Nominal Reflection Coefficients \n");
  fprintf(oupp,"\n");
  for (j=0; j < nseg; j++)
    {
    v[0][0] = r[j][0];
    bigk = 0.0;
    for (k=1; k<=nco; k++)
      {
      v[0][k] = 2.0 * r[j][k];
      }
    for (k=0; k<nco ; k++)
      {
      v[1][k] = r[j][k] + r[j][k+1];
      }
      for (i=1; i<=nco; i++)
        {
        alphak = v[i][0]/v[i-1][0];
        bigk = 1.0 - (alphak / (1.0 + bigk));
        no_a[j][i] = alphak;
        no_k[j][i] = bigk;
        fprintf(oupp,"%s%d%s%10.6f"," K(",i,") = ",bigk);
        for (k=0; (k <= (nco-i-1)) && (k > -1) ; k++)
          {
          v[i+1][k] = v[i][k] + v[i][k+1] - alphak * v[i-1][k+1];
          }
        }
      fprintf(oupp,"\n");
      }
    fprintf(oupp,"\n\n");

      /* output ensemble of nominal split Schur reflection       */
      /* coefficients                                            */

    fprintf(oupp,"  Ensemble of Nominal Split Schur Reflection \n");
    fprintf(oupp,"                    Coefficients              \n");
    fprintf(oupp,"\n");
    for (j=0; j < nseg ; j++)
      {
      for (i=1; i<=nco ; i++)
        {
        fprintf(oupp,"%s%d%s%10.6f"," a(",i,") = ",no_a[j][i]);
        }
      fprintf(oupp,"\n");
      }
    fprintf(oupp,"\n\n");

      /* compute ensemble of fixed-point reflection coefficients */
      /* assuming that there is no quantization error in the     */
      /* autocorrelation coefficient estimates except that due   */
      /* to rounding to n-bit, 2s complement numbers             */

    fprintf(oupp,"%s","    Ensemble of Fixed-Point Reflection  \n");
    fprintf(oupp,"%s","               Coefficients             \n");
    fprintf(oupp,"%s","  (error-free autocorrelation coeffs.) \n");
    fprintf(oupp,"\n");
    one = 01;
    one = one << (n-1);
    for (j=0; j < nseg; j++)
      {
      fp_v[0][0] = ftob(r[j][0],n);
      fp_bigk = 0;
      for (k=1; k<=nco; k++)
        {
        fp_v[0][k] = 2*ftob(r[j][k],n);
        }
      for (k=0; k<nco ; k++)
        {
        fp_v[1][k] = ftob(r[j][k],n) + ftob(r[j][k+1],n);
        }
      for (i=1; i<=nco; i++)
        {
        fp_alphak = divide(fp_v[i][0],fp_v[i-1][0],n,m);
        fp_bigk = one - divide(fp_alphak,one + fp_bigk,n,m);
        fp_a[j][i] = fp_alphak;
        fp_k[j][i] = fp_bigk;
        fprintf(oupp,"%s%d%s%10.6f"," K(",i,") = ",btof(fp_bigk,n,m));
        for (k=0; (k <= (nco-i-1)) && ( k > -1) ; k++)
          {
          prod = quant(fp_alphak*fp_v[i-1][k+1],n);
          fp_v[i+1][k] = fp_v[i][k] + fp_v[i][k+1] - prod;
          }
        }
      fprintf(oupp,"\n");
      }
  fprintf(oupp,"\n\n");

      /* output ensemble of fixed-point split Schur reflection  */
      /* coefficients                                           */

    fprintf(oupp,"  Ensemble of Fixed-Point Split Schur        \n");
    fprintf(oupp,"            Reflection Coefficients           \n");
    fprintf(oupp,"\n");
    for (j=0; j < nseg ; j++)
      {
      for (i=1; i<=nco ; i++)
      {
      fprintf(oupp,"%s%d%s%10.6f"," a(",i,") = ",btof(fp_a[j][i],n,m));
      }
    fprintf(oupp,"\n");
    }
  fprintf(oupp,"\n\n");
fprintf(oupp,"%s %d %s \n"," Wordsize = ",n+m," bits ");
fprintf(oupp,"%s %d \n"," No. of fractional bits = ",n-1);
fprintf(oupp,"\n\n");

     /* compute and output the means and variances of the  */
     /* reflection coefficients                            */

fprintf(oupp,"%s","    Fixed-Point Reflection Coefficient    \n");
fprintf(oupp,"%s","             Means and Variances          \n");
fprintf(oupp,"\n");
for (j=1; j <= nco ; j++)
  {
  mean[j] = 0.0;
  for (i=0; i < nseg; i++)
    {
    mean[j] = mean[j] + btof(fp_k[i][j],n,m);
    }
  mean[j] = mean[j]/nseg;
  }
for (j=1; j <= nco ; j++)
  {
  vari[j] = 0.0;
  for (i=0; i < nseg; i++)
    {
    bigk = btof(fp_k[i][j],n,m);
    vari[j] = vari[j] + (no_k[i][j] - bigk)*(no_k[i][j] - bigk);
    }
  vari[j] = vari[j]/nseg;
  }
fprintf(oupp,"%s\n","        Mean          Variance           ");
for(j=1; j <= nco ; j++)
  {
  fprintf(oupp,"%s%d%s %11.8f %14.8f \n","K(",j,")",mean[j],vari[j]);
  }
}
```

```
-----------------------------------------------------------------------------------------------
 - 2nd Order AR Process Parameters -

  pole modulus =        0.750000
  pole angle (degrees) =    175.000000
  noise variance =       1.000000


  Theoretical 2nd Reflection Coefficient
            Error Variance

 theoretical r[1] =    -0.956346
 theoretical r[2] =     0.866561
 Variance of K(2) =       0.00000249


 Ensemble of Nominal and Normalized
    Autocorrelation Coefficients

r[ 0 ] =    1.000000 r[ 1 ] =  -0.959196 r[ 2 ] =    0.876658
r[ 0 ] =    1.000000 r[ 1 ] =  -0.954644 r[ 2 ] =    0.858507
r[ 0 ] =    1.000000 r[ 1 ] =  -0.955647 r[ 2 ] =    0.865001
r[ 0 ] =    1.000000 r[ 1 ] =  -0.953469 r[ 2 ] =    0.857209
r[ 0 ] =    1.000000 r[ 1 ] =  -0.954328 r[ 2 ] =    0.863378
r[ 0 ] =    1.000000 r[ 1 ] =  -0.958454 r[ 2 ] =    0.871223
r[ 0 ] =    1.000000 r[ 1 ] =  -0.957220 r[ 2 ] =    0.868155
r[ 0 ] =    1.000000 r[ 1 ] =  -0.960356 r[ 2 ] =    0.876835
r[ 0 ] =    1.000000 r[ 1 ] =  -0.954882 r[ 2 ] =    0.858790
r[ 0 ] =    1.000000 r[ 1 ] =  -0.959615 r[ 2 ] =    0.876313


Ensemble of Nominal Reflection Coefficients

K(1) =    0.959196 K(2) =    0.542869
K(1) =    0.954644 K(2) =    0.596007
K(1) =    0.955647 K(2) =    0.556391
K(1) =    0.953469 K(2) =    0.570931
K(1) =    0.954328 K(2) =    0.530663
K(1) =    0.958454 K(2) =    0.582704
K(1) =    0.957220 K(2) =    0.574666
K(1) =    0.960356 K(2) =    0.584826
K(1) =    0.954882 K(2) =    0.601037
K(1) =    0.959615 K(2) =    0.562908


 Ensemble of Nominal Split Schur Reflection
               Coefficients

a(1) =    0.040803 a(2) =    0.895608
a(1) =    0.045355 a(2) =    0.789661
a(1) =    0.044352 a(2) =    0.867540
a(1) =    0.046530 a(2) =    0.838171
a(1) =    0.045671 a(2) =    0.917237
a(1) =    0.041545 a(2) =    0.817254
a(1) =    0.042779 a(2) =    0.832471
a(1) =    0.039643 a(2) =    0.813889
a(1) =    0.045117 a(2) =    0.779925
a(1) =    0.040384 a(2) =    0.856530


   Ensemble of Fixed-Point Reflection
              Coefficients
   (error-free autocorrelation coeffs.)

K(1) =    0.959228 K(2) =    0.543090
K(1) =    0.954589 K(2) =    0.594360
K(1) =    0.955688 K(2) =    0.557739
K(1) =    0.953491 K(2) =    0.571411
K(1) =    0.954345 K(2) =    0.530761
K(1) =    0.958496 K(2) =    0.583984
K(1) =    0.957275 K(2) =    0.576660
K(1) =    0.960327 K(2) =    0.583984
K(1) =    0.954833 K(2) =    0.598999
K(1) =    0.959594 K(2) =    0.562133


 Ensemble of Fixed-Point Split Schur
      Reflection Coefficients

a(1) =    0.040771 a(2) =    0.895263
a(1) =    0.045410 a(2) =    0.792968
a(1) =    0.044311 a(2) =    0.864990
```

```
 a(1) =   0.046508 a(2) =   0.837280
 a(1) =   0.045654 a(2) =   0.917114
 a(1) =   0.041503 a(2) =   0.814697
 a(1) =   0.042724 a(2) =   0.828613
 a(1) =   0.039672 a(2) =   0.815429
 a(1) =   0.045166 a(2) =   0.783813
 a(1) =   0.040405 a(2) =   0.858032

 Wordsize =  16  bits
 No. of fractional bits =  13

     Fixed-Point Reflection Coefficient
            Means and Variances

          Mean           Variance
K(1)  0.95678710      0.00000000
K(2)  0.57031250      0.00000158
```

APPENDIX D

Summary Tables for Output from the

Program in Appendix C

b	$var[\Delta K^{(2)}]\ 10^4$	
	Theoretical	Experimental
8	1670	5930
10	104	66.8
12	6.51	6.95
14	0.407	0.574

Table I: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho = 0.875$ and $\theta = 5°$.

b	$var[\Delta K^{(2)}]\ 10^4$	
	Theoretical	Experimental
8	116	47
10	7.24	9.48
12	0.453	0.654
14	0.0282	0.0111

Table II: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho = 0.75$ and $\theta = 5°$.

b	$var[\Delta K^{(2)}]\ 10^4$	
	Theoretical	Experimental
8	2.44	2.60
10	0.153	0.151
12	0.0095	0.0111
14	0.0005	0.0002

Table III: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho = 0.875$ and $\theta = 45°$.

b	$var[\Delta K^{(2)}]\ 10^4$	
	Theoretical	Experimental
8	2.09	0.742
10	0.131	0.170
12	0.0081	0.0045
14	0.0005	0.0003

Table IV: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho = 0.875$ and $\theta = 135°$.

b	$var[\Delta K^{(2)}]\ 10^4$	
	Theoretical	Experimental
8	1460	1440
10	91.3	62.6
12	5.71	11.4
14	0.357	0.366

Table V: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho = 0.875$ and $\theta = 175°$.

b	$var[\Delta K^{(2)}]\ 10^4$	
	Theoretical	Experimental
8	102	27.2
10	6.40	3.96
12	0.400	0.170
14	0.0249	0.0158

Table VI: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho = 0.75$ and $\theta = 175°$.

APPENDIX E

Program to Compute the Theoretical Value of $Var\,[\Delta K_k]$ for the Symmetric Split Schur Algorithm

Program SYM_SCHUR_VAR

I. Introduction

The finite precision arithmetic (twos complement) implementation of the symmetric split Schur algorithm of Delsarte and Genin [1] produces estimates $\hat{K}_k$ of the true (infinite precision) reflection coefficients K_k with error ΔK_k (i.e., $\hat{K}_k = K_k + \Delta K_k$). Equations (10)-(12) of Zarowski and Card [2] provide iterative expressions for ΔK_k. Closed-form expressions for ΔK_k are hard to obtain, so a recursive PASCAL program called SYM_SCHUR_VAR is used to calculate ΔK_k for any k, and any suitable sequence of normalized autocorrelation coefficients as input. Thus, we have produced a crude symbolic computation solution to the problem of estimating $Var[\Delta K_k]$.

II. The Data Structures and Program

We begin by stating the recursions evaluated by the program SYM_SCHUR_VAR. Since we are not interested in $Var[\Delta\alpha_k]$, we can eliminate $\Delta\alpha_k$ from (11) and (12) using (10). This gives us

$$\Delta K_k = \frac{1-K_k}{1+K_{k-1}}\Delta K_{k-1} - \frac{1}{v_{k-1,0}(1+K_{k-1})}\Delta v_{k,0} \qquad (1a)$$

$$+\frac{\alpha_k}{v_{k-1,0}(1+K_{k-1})}\Delta v_{k-1,0} - \frac{1}{1+K_{k-1}}\eta_{\alpha,k} + \eta_{K,k} \ ,$$

$$\Delta v_{k+1,j} = \Delta v_{k,j} + \Delta v_{k,j+1} - \alpha_k \Delta v_{k-1,j+1} - \frac{v_{k-1,j+1}}{v_{k-1,0}}\Delta v_{k,0} \qquad (1b)$$

$$+\alpha_k \frac{v_{k-1,j+1}}{v_{k-1,0}}\Delta v_{k-1,0} + \eta_{v_{k+1,j},j} - v_{k-1,j+1}\eta_{\alpha,k} \ ,$$

where $1 \le k \le n$ and $0 \le j \le n-k-1$. Initially,

$$\Delta v_{0,0} = \Delta r_0 = 0 \ ,$$

$$\Delta v_{0,j} = 2\Delta r_j \quad (1 \le j \le n) \ ,$$
$$\Delta v_{1,j} = \Delta r_j + \Delta r_{j+1} \quad (0 \le j \le n-1) \ . \tag{2}$$

A PASCAL record type called DELTA is defined as follows:

```
DELTA = RECORD
    K,J:INTEGER;                         { = $k$ and $j$, respectively }.
    SUCC_DK:PT_DELTA;                    { = $\Delta K_{k-1}$ }
    SUCC_DVK_0:PT_DELTA;                 { = $\Delta v_{k,0}$ }
    SUCC_DVKM1_0:PT_DELTA;               { = $\Delta v_{k-1,0}$ }
    SUCC_DVK_J:PT_DELTA;                 { = $\Delta v_{k,j}$ }
    SUCC_DVK_JP1:PT_DELTA;               { = $\Delta v_{k,j+1}$ }
    SUCC_DVKM1_JP1:PT_DELTA;             { = $\Delta v_{k-1,j+1}$ }
    SUCC_ETA_ALPHA_K:PT_DELTA;           { = $\eta_{\alpha,k}$ }
    SUCC_ETA_K_K:PT_DELTA;               { = $\eta_{K,k}$ }
    SUCC_ETA_VKP1_J:PT_DELTA;            { = $\eta_{v_{k+1},j}$ }
    DK:DOUBLE_REAL;
    DVK_0:DOUBLE_REAL;
    DVKM1_0:DOUBLE_REAL;
    DVK_J:DOUBLE_REAL;
    DVK_JP1:DOUBLE_REAL;
    DVKM1_JP1:DOUBLE_REAL;
    ETA_ALPHA_K:DOUBLE_REAL;
    ETA_K_K:DOUBLE_REAL;
    ETA_VKP1_J:DOUBLE_REAL;
    END;
```

PT_DELTA is a pointer to a DELTA record. Notice that each field of type PT_DELTA represents a term in one or both of the equations (1a,b). The DOUBLE_REAL types (double precision floating-point) are the coefficients in front of the variables (e.g., DK $= (1-K_k)/(1+K_{k-1})$ is the coefficient of term ΔK_{k-1} in (1a) which is represented by the pointer SUCC_DK). Nominal parameters such as K_k, α_k or $v_{k,j}$ are computed by the procedure called NOMINALS. Nominal parameters are made globally available for convenience. K ($= k$) is the level of the recursion, and J ($= j$) is the jth coefficient at the kth level. The user specified value of k (which is read in by procedure READ_PARAMETERS) is the top-most level of the recursion, and so corresponds to the root node of the tree (see below).

The pointers PT_DELTA may be set to nil for one of three reasons:

(1) The variable to which the pointer corresponds is unused (e.g., equation (1b) does not have a ΔK_{k-1} term), in which case the associated coefficient is set to zero as well.

(2) When $\bar{K}$ (k) is of such a value that we have terms like $\Delta v_{0,0}$, $\Delta v_{0,j}$, or $\Delta v_{1,j}$ (which depend upon Δr_j (see (2))), then the associated pointer is set to nil.

(3) SUCC_ETA_ALPHA_K, SUCC_ETA_K_K, and SUCC_ETA_VKP1_J are always set to nil since they are not recursively dependent upon anything.

Thus, nil pointers primarily indicate that we've reached terms in the expression for ΔK_k of the form Δr_j , $\eta_{\alpha,k}$, $\eta_{K,k}$ or $\eta_{v_{k+1},j}$. Like terms in these variables may then be collected and saved by a suitable tree traversal (see below).

Each of (1a) and (1b) requires a recursive procedure to represent it. Procedure MAKE_DK represents equation (1a) and procedure MAKE_DV represents equation (1b). Procedure MAKE_DK calls itself (since (1a) contains a term with ΔK_{k-1} in it) and procedure MAKE_DV (since (1a) contains terms with $\Delta v_{k,0}$ and $\Delta v_{k-1,0}$ in them). Procedure MAKE_DV only calls itself since there is no term containing ΔK_{k-1} in (1b). A comparison of the MAKE_DK and MAKE_DV procedures with (1a,b) should reveal to the reader how they work in detail.

The tree constructed by MAKE_DK and MAKE_DV is traversed by the procedure TRAVERSE, which collects and saves the like terms in the variables of the form $\eta_{X,Y}$ and Δr_j as previously described. The first call of TRAVERSE (in the MAINLINE part of the program) is

TRAVERSE (*TOP* ,1.0,0,0,0);

where TOP points to the top of the tree, and VALUE = 1.0 is the initial value of any term in the final expression for ΔK_k (before all like terms in a variable have been collected by TRAVERSE). IID,IK, and IJ are all arbitrarily set to zero (0) initially. If PT is nil then the IID number determines where VALUE is to be saved (by accumulation (adding)). Note that as TRAVERSE traverses the tree, VALUE is multiplied by the appropriate coefficient until a nil pointer is reached (PT = nil) whence VALUE is added to a suitable location in one of the following ways:

DELTA_R[.] (= Δr_j) if IID = 1,

ETA_ALPHA[.] (= $\eta_{\alpha,k}$) if IID = 2,

ETA_K[.] (= $\eta_{K,k}$) if IID = 3,

ETA_VKP1[.,.] (= $\eta_{v_{k+1},j}$) if IID = 4.

Thus, the arrays DELTA_R[.], ETA_ALPHA[.], etc., contain the terms in the final formula for ΔK_k as a function of the variables Δr_j , $\eta_{\alpha,k}$, $\eta_{K,k}$ and $\eta_{v_{k+1},j}$ as we wish. Squaring and summing the entries of these arrays yields

$$\frac{Var\,[\Delta K_k]}{\sigma_\eta^2}$$

which corresponds to the variable VARIANCE in the MAINLINE part of the program. The program prints out VARIANCE $\times\ \sigma_\eta^2$, where $\sigma_\eta^2 = \dfrac{2^{-2b}}{3}$ (see [2]). It does so for b = 8,10,12,14. The entries of the arrays DELTA_R[.], ETA_ALPHA[.], ETA_K[.] and ETA_VKP1[.,.] are also printed out.

III. Sources of Inefficiency

Procedure TRAVERSE wastefully accumulates VALUE and then multiplies VALUE by zero when it encounters an unused pointer. One way to avoid this would be to use two different record types; one for each of the equations (1a) and (1b). However, this greatly complicates the traversal process and so is not a viable alternative. A better solution might be to flag unused pointers as such and so avoid using them during the traversal. For small to moderate values of k there seems to be little advantage in doing this.

A much more significant source of inefficiency is the fact that a given term in the expansion of the expression for ΔK_k may appear many times. This corresponds to "repeated branches" in the tree representing the expansion of the equation for ΔK_k. For example, when k = 3, the term $\Delta v_{2,0}$ appears 4 times. If k is large then much storage could be consumed and much time wasted in constructing tree branches that already exist somewhere else. A solution to this problem might involve flagging the repeated terms (repeated branches) somehow and creating them once only. However, this complicates the program (though probably not enormously). The simple, though inefficient, solution presented works well enough for the purposes of the paper [2].

REFERENCES

[1] P. Delsarte, Y. Genin, "On the Splitting of Classical Algorithms in Linear Prediction Theory," IEEE Trans. on Acoust., Speech, and Signal Proc., vol. ASSP-35, pp. 645-653, May 1987.

[2] C. J. Zarowski, H. C. Card, "Finite Precision Arithmetic and the Split Schur Algorithms," submitted to the IEEE ASSP Transactions.

```
PROGRAM SYM_SCHUR_VAR(INSTUFF,OUTSTUFF);

  (* This program computes theoretical reflection coefficient     *)
  (* error variances due to fixed-point 2s complement arithmetic  *)
  (* with quantization due to rounding.                           *)
  (* The underlying algorithm that creates the reflection         *)
  (* coefficients is the symmetric split Schur algorithm of       *)
  (* Delsarte and Genin:                                          *)
  (*                                                              *)
  (*   P. Delsarte, Y. Genin, "On the Splitting of Classical      *)
  (*   Algorithms," IEEE Trans. on Acoust., Speech, and Signal    *)
  (*   Proc., vol. ASSP-35, pp. 645-653, May 1987.                *)
  (*                                                              *)
  (* Because of the complexity of the recursions for the          *)
  (* quantization error expressions of the reflection             *)
  (* coefficients, it is not possible to obtain closed-form       *)
  (* error variance expressions.  Hence, it is necessary to       *)
  (* construct a recursive program to produce these reflection    *)
  (* coefficient error variances.  This is accomplished by this   *)
  (* program.                                                     *)

CONST
  CLIM = 12;               (* Maximum number of autocorrel. coeffs.    *)
TYPE
  VECTOR = ARRAY[0..CLIM] OF DOUBLE_REAL;
  MATRIX = ARRAY[0..CLIM,0..CLIM] OF DOUBLE_REAL;
  PT_DELTA = ^DELTA;
  DELTA = RECORD
    K,J:INTEGER;
    SUCC_DK:PT_DELTA;
    SUCC_DVK_0:PT_DELTA;
    SUCC_DVKM1_0:PT_DELTA;
    SUCC_DVK_J:PT_DELTA;
    SUCC_DVK_JP1:PT_DELTA;
    SUCC_DVKM1_JP1:PT_DELTA;
    SUCC_ETA_ALPHA_K:PT_DELTA;
    SUCC_ETA_K_K:PT_DELTA;
    SUCC_ETA_VKP1_J:PT_DELTA;
    DK:DOUBLE_REAL;
    DVK_0:DOUBLE_REAL;
    DVKM1_0:DOUBLE_REAL;
    DVK_J:DOUBLE_REAL;
    DVK_JP1:DOUBLE_REAL;
    DVKM1_JP1:DOUBLE_REAL;
    ETA_ALPHA_K:DOUBLE_REAL;
    ETA_K_K:DOUBLE_REAL;
    ETA_VKP1_J:DOUBLE_REAL;
    END;

VAR
  INSTUFF,OUTSTUFF:TEXT;
  V:MATRIX;                      (* Matrix of v-parameters produced by *)
                                 (* the split algorithm.               *)
  BIG_K,ALPHA,R:VECTOR;          (* Vector of reflection coefficients, *)
                                 (* split Schur reflection coeffs., &  *)
                                 (* normalized autocorrelation coeffs. *)
                                 (* , respectively.                    *)
  N:INTEGER;                     (* Number of correlation coeffs.      *)
  IX,JX,B:INTEGER;               (* Loop counters.                     *)
  TOP:PT_DELTA;                  (* Top of tree representing the       *)
                                 (* coupled recursive error formulae.  *)
  VARIANCE:DOUBLE_REAL;          (* Nth reflection coefficient error   *)
                                 (* variance.                          *)
  DELTA_R,ETA_ALPHA,ETA_K:VECTOR;
                                 (* Terms in the formula for VARIANCE. *)
  ETA_VKP1:MATRIX;               (* More terms in formula for VARIANCE.*)


PROCEDURE READ_PARAMETERS(VAR R:VECTOR;VAR N:INTEGER);
  (* This procedure reads in the normalized autocorrelation      *)
  (* coefficients from a file along with the number of such      *)
  (* coefficients.  The zero lag coefficient is always one and   *)
  (* so is not read in.                                          *)
VAR
  I:INTEGER;
BEGIN (* READ_PARAMETERS *)
RESET(INSTUFF,'CORREL_COEFFS');
READ(INSTUFF,N);
R[0] := 1.0;
FOR I := 1 TO N DO
  BEGIN
```

```
   READ(INSTUFF,R[I]);
   END;
 END;  (* READ_PARAMETERS *)


 PROCEDURE NOMINALS(VAR BIG_K,ALPHA,R:VECTOR;VAR V:MATRIX;N:INTEGER);
   (* This procedure computes the nominal values of BIG_K, ALPHA, and *)
   (* of V using the normalized autocorrelation coefficients in R.    *)
 VAR
   I,K:INTEGER;
 BEGIN (* NOMINALS *)
 V[0,0] := R[0];
 BIG_K[0] := 0.0;
 FOR K := 1 TO N DO
   BEGIN
   V[0,K] := 2.0 * R[K];
   END;
 FOR K := 0 TO N-1 DO
   BEGIN
   V[1,K] := R[K] + R[K+1];
   END;
 FOR I := 1 TO N DO
   BEGIN
   ALPHA[I] := V[I,0]/V[I-1,0];
   BIG_K[I] := 1.0 - ALPHA[I]/(1.0 + BIG_K[I-1]);
   K := 0;
   WHILE ( ( K <= N - I - 1 ) AND ( K > -1) ) DO
     BEGIN
     V[I+1,K] := V[I,K] + V[I,K+1] - ALPHA[I] * V[I-1,K+1];
     K := K + 1;
     END;
   END;
 END;  (* NOMINALS *)


 PROCEDURE MAKE_DV(PT_DV:PT_DELTA;KK,JJ:INTEGER);
 BEGIN (* MAKE_DV *)
 WITH PT_DV^ DO
   BEGIN
   K := KK;
   J := JJ;
   IF KK = 1 THEN
     BEGIN
     SUCC_DVK_0 := NIL;
     SUCC_DVKM1_0 := NIL;
     SUCC_DVK_J := NIL;
     SUCC_DVK_JP1 := NIL;
     SUCC_DVKM1_JP1 :=NIL;
     SUCC_ETA_ALPHA_K := NIL;
     SUCC_ETA_VKP1_J := NIL;
     DVK_0 := -V[KK-1,JJ+1]/V[KK-1,0];
     DVKM1_0 := ALPHA[KK]*V[KK-1,JJ+1]/V[KK-1,0];
     DVK_J := 1.0;
     DVK_JP1 := 1.0;
     DVKM1_JP1 := -ALPHA[KK];
     ETA_ALPHA_K := -V[KK-1,JJ+1];
     ETA_VKP1_J := 1.0;
     END
   ELSE
     BEGIN
     NEW(SUCC_DVK_0);
     DVK_0 := -V[KK-1,JJ+1]/V[KK-1,0];
     MAKE_DV(SUCC_DVK_0,KK-1,0);
     IF KK-1 = 1 THEN
       BEGIN
       SUCC_DVKM1_0 := NIL;
       DVKM1_0 := ALPHA[KK] * V[KK-1,JJ+1] / V[KK-1,0];
       END
     ELSE
       BEGIN
       NEW(SUCC_DVKM1_0);
       DVKM1_0 := ALPHA[KK] * V[KK-1,JJ+1] / V[KK-1,0];
       MAKE_DV(SUCC_DVKM1_0,KK-1,0);
       END;
   NEW(SUCC_DVK_J);
   DVK_J := 1.0;
   MAKE_DV(SUCC_DVK_J,KK-1,JJ);
   NEW(SUCC_DVK_JP1);
   DVK_JP1 := 1.0;
   MAKE_DV(SUCC_DVK_JP1,KK-1,JJ+1);
   IF KK-1 = 1 THEN
```

```
            BEGIN
            SUCC_DVKM1_JP1 := NIL;
            DVKM1_JP1 := -ALPHA[KK];
            END
          ELSE
            BEGIN
            NEW(SUCC_DVKM1_JP1);
            DVKM1_JP1 := -ALPHA[KK];
            MAKE_DV(SUCC_DVKM1_JP1,KK-1,JJ+1);
            END;
          SUCC_ETA_ALPHA_K := NIL;
          SUCC_ETA_VKP1_J := NIL;
          ETA_ALPHA_K := -V[KK-1,JJ+1];
          ETA_VKP1_J := 1.0;
          END;
        SUCC_DK := NIL;
        SUCC_ETA_K_K := NIL;
        DK := 0.0;
        ETA_K_K := 0.0;
        END;
    END;  (* MAKE_DV *)


    PROCEDURE MAKE_DK(PT_DK:PT_DELTA;KK:INTEGER);
    BEGIN (* MAKE_DK *)
    WITH PT_DK^ DO
      BEGIN
      K := KK;
      J := 0;
      IF KK = 1 THEN
        BEGIN
        SUCC_DK := NIL;
        SUCC_DVK_0 := NIL;
        SUCC_DVKM1_0 := NIL;
        SUCC_ETA_ALPHA_K := NIL;
        SUCC_ETA_K_K := NIL;
        DK := 0.0;
        DVK_0 := -1.0/V[KK-1,0];
        DVKM1_0 := ALPHA[KK]/V[KK-1,0];
        ETA_ALPHA_K := -1.0;
        ETA_K_K := 1.0;
        END
    ELSE
      BEGIN
      NEW(SUCC_DK);
      DK := (1.0 - BIG_K[KK])/(1.0 + BIG_K[KK-1]);
      MAKE_DK(SUCC_DK,KK-1);
      NEW(SUCC_DVK_0);
      DVK_0 := -1.0/(V[KK-1,0]*(1.0 + BIG_K[KK-1]));
      MAKE_DV(SUCC_DVK_0,KK-1,J);
      IF KK-2 = 0 THEN
        BEGIN
        SUCC_DVKM1_0 := NIL;
        DVKM1_0 := ALPHA[KK]/(V[KK-1,0]*(1.0 + BIG_K[KK-1]));
        END
      ELSE
        BEGIN
        NEW(SUCC_DVKM1_0);
        DVKM1_0 := ALPHA[KK]/(V[KK-1,0]*(1.0 + BIG_K[KK-1]));
        MAKE_DV(SUCC_DVKM1_0,KK-2,J);
        END;
      SUCC_ETA_ALPHA_K := NIL;
      ETA_ALPHA_K := -1.0/(1.0 + BIG_K[KK-1]);
      SUCC_ETA_K_K := NIL;
      ETA_K_K := 1.0;
      END;
    SUCC_DVK_J := NIL;
    SUCC_DVK_JP1 := NIL;
    SUCC_DVKM1_JP1 := NIL;
    SUCC_ETA_VKP1_J := NIL;
    DVK_J := 0.0;
    DVK_JP1 := 0.0;
    DVKM1_JP1 := 0.0;
    ETA_VKP1_J := 0.0;
    END;
  END;  (* MAKE_DK *)


  PROCEDURE TRAVERSE(PT:PT_DELTA;VALUE:DOUBLE_REAL;IID,IK,IJ:INTEGER);
  VAR
    TVALUE:DOUBLE_REAL;
```

```
BEGIN (* TRAVERSE *)
IF PT <> NIL THEN
  BEGIN
  WITH PT^ DO
    BEGIN
    TVALUE := VALUE * DK;
    TRAVERSE(SUCC_DK,TVALUE,0,K,J);
    TVALUE := VALUE * DVK_0;
    TRAVERSE(SUCC_DVK_0,TVALUE,1,K,0);
    TVALUE := VALUE * DVKM1_0;
    TRAVERSE(SUCC_DVKM1_0,TVALUE,1,K-1,0);
    TVALUE := VALUE * DVK_J;
    TRAVERSE(SUCC_DVK_J,TVALUE,1,K,J);
    TVALUE := VALUE * DVK_JP1;
    TRAVERSE(SUCC_DVK_JP1,TVALUE,1,K,J+1);
    TVALUE := VALUE * DVKM1_JP1;
    TRAVERSE(SUCC_DVKM1_JP1,TVALUE,1,K-1,J+1);
    TVALUE := VALUE * ETA_ALPHA_K;
    TRAVERSE(SUCC_ETA_ALPHA_K,TVALUE,2,K,J);
    TVALUE := VALUE * ETA_K_K;
    TRAVERSE(SUCC_ETA_K_K,TVALUE,3,K,J);
    TVALUE := VALUE * ETA_VKP1_J;
    TRAVERSE(SUCC_ETA_VKP1_J,TVALUE,4,K+1,J);
    END;
  END
ELSE
  BEGIN
  IF IID = 1 THEN
    BEGIN
    IF ((IK=0) AND (IJ<>0)) THEN
      BEGIN
      DELTA_R[IJ] := DELTA_R[IJ] + 2.0 * VALUE;
      END;
    IF IK=1 THEN
      BEGIN
      IF IJ = 0 THEN
        BEGIN
        DELTA_R[IJ+1] := DELTA_R[IJ+1] + VALUE;
        END
      ELSE
        BEGIN
        DELTA_R[IJ] := DELTA_R[IJ] + VALUE;
        DELTA_R[IJ+1] := DELTA_R[IJ+1] + VALUE;
        END;
      END;
    END;
  IF IID = 2 THEN
    BEGIN
    ETA_ALPHA[IK] := ETA_ALPHA[IK] + VALUE;
    END;
  IF IID = 3 THEN
    BEGIN
    ETA_K[IK] := ETA_K[IK] + VALUE;
    END;
  IF IID = 4 THEN
    BEGIN
    ETA_VKP1[IK,IJ] := ETA_VKP1[IK,IJ] + VALUE;
    END;
  END;
END;  (* TRAVERSE *)


          (* ..... MAINLINE PROGRAM ..... *)


BEGIN
REWRITE(OUTSTUFF,'VARIANCES');
READ_PARAMETERS(R,N);
NOMINALS(BIG_K,ALPHA,R,V,N);
NEW(TOP);
MAKE_DK(TOP,N);  (* Construct tree representing reflection error *)
                 (* variance formula.                            *)
VARIANCE := 0.0;
FOR IX := 0 TO N DO
  BEGIN (* Initiallization loop. *)
  DELTA_R[IX] := 0.0;
  ETA_ALPHA[IX] := 0.0;
  ETA_K[IX] := 0.0;
  FOR JX := 0 TO N DO
    BEGIN
    ETA_VKP1[IX,JX] := 0.0;
```

```
        END;
      END;
    TRAVERSE(TOP,1.0,0,0,0);
    WRITELN(OUTSTUFF,'   DELTA_R   ');
    FOR IX := 0 TO N DO
      BEGIN
      WRITE(OUTSTUFF,DELTA_R[IX]:16:10);
      END;
    WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);
    WRITELN(OUTSTUFF,'   ETA_ALPHA   ');
    FOR IX := 0 TO N DO
      BEGIN
      WRITE(OUTSTUFF,ETA_ALPHA[IX]:16:10);
      END;
    WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);
    WRITELN(OUTSTUFF,'   ETA_K   ');
    FOR IX := 0 TO N DO
      BEGIN
      WRITE(OUTSTUFF,ETA_K[IX]:16:10);
      END;
    WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);
    WRITELN(OUTSTUFF,'   ETA_VKP1   ');
    FOR IX := 0 TO N DO
      BEGIN
      FOR JX := 0 TO N DO
        BEGIN
        WRITE(OUTSTUFF,ETA_VKP1[IX,JX]:16:10);
        END;
      WRITELN(OUTSTUFF);
      END;
    WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);
    FOR IX := 0 TO N DO
      BEGIN
      VARIANCE := VARIANCE + DELTA_R[IX]*DELTA_R[IX] + ETA_K[IX]*ETA_K[IX]
                  + ETA_ALPHA[IX]*ETA_ALPHA[IX];
      FOR JX := 0 TO N DO
        BEGIN
        VARIANCE := VARIANCE + ETA_VKP1[IX,JX]*ETA_VKP1[IX,JX];
        END;
      END;
    WRITELN(OUTSTUFF,'            Reflection Coefficient Error Variances ');
    WRITELN(OUTSTUFF);
    B := 8;
    WRITE(OUTSTUFF,B:21);
    WHILE B <= 12 DO
      BEGIN
      B := B + 2;
      WRITE(OUTSTUFF,B:14);
      END;
    WRITELN(OUTSTUFF);
    WRITE(OUTSTUFF,' Var[DK(',N,')] = ');
    B := 8;
    WHILE B <= 14 DO
      BEGIN
      WRITE(OUTSTUFF,(VARIANCE * POWER(2.0,-2.0*B) / 3.0):14:8);
      B := B + 2;
      END;
    WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);
END.
```

APPENDIX F

Program to Produce Nominal Normalized Autocorrelation Coefficients for 4th Order AR Processes

```
PROGRAM AR_4TH(INSTUFF,OUTSTUFF);

  (* This program computes the normalized autocorrelation      *)
  (* coefficients for a 4th order autoregressive process with  *)
  (* two complex conjugate pole-pairs.  The resulting numbers  *)
  (* are written to the input file of the program called       *)
  (* SYM_SCHUR_VAR which uses them to compute reflection       *)
  (* coefficient error variances.                              *)

CONST
  CLIM = 12;        (* Maximum number of autocorrel. coeffs.    *)
  PIE = 3.141592654;
TYPE
  VECTOR = ARRAY [0..CLIM] OF DOUBLE_REAL;
  COMPLEX = RECORD   (* Complex number type. *)
    REL:DOUBLE_REAL; (* Real part.           *)
    IMG:DOUBLE_REAL; (* Imaginary part.      *)
    END;
VAR
  INSTUFF,OUTSTUFF:TEXT;


FUNCTION CADD(X,Y:COMPLEX):COMPLEX;
  (* This function adds two complex types together.         *)
BEGIN (* CADD *)
CADD.REL := X.REL + Y.REL;
CADD.IMG := X.IMG + Y.IMG;
END;  (* CADD *)


FUNCTION CSUB(X,Y:COMPLEX):COMPLEX;
  (* This function subtracts two complex numbers.           *)
BEGIN (* CSUB *)
CSUB.REL := X.REL - Y.REL;
CSUB.IMG := X.IMG - Y.IMG;
END;  (* CSUB *)


FUNCTION CMUL(X,Y:COMPLEX):COMPLEX;
  (* This function multiplies two complex types together.   *)
BEGIN (* CMUL *)
CMUL.REL := X.REL*Y.REL - X.IMG*Y.IMG;
CMUL.IMG := X.REL*Y.IMG + X.IMG*Y.REL;
END;  (* CMUL *)


FUNCTION CDIV(X,Y:COMPLEX):COMPLEX;
  (* This function divides two complex types.               *)
VAR
  DENOM:DOUBLE_REAL;
BEGIN (* CDIV *)
DENOM := Y.REL*Y.REL + Y.IMG*Y.IMG;
CDIV.REL := ( X.REL*Y.REL + X.IMG*Y.IMG )/ DENOM;
CDIV.IMG := ( X.IMG*Y.REL - X.REL*Y.IMG )/ DENOM;
END;  (* CDIV *)


FUNCTION CEXP(X,Y:DOUBLE_REAL):COMPLEX;
  (* This function computes x exp(jy), j = sqrt(-1), x,y are real.  *)
VAR
  SINE:DOUBLE_REAL;
BEGIN (* CEXP *)
SINE := COS(PIE/2.0 - Y);
CEXP.REL := X * COS(Y);
CEXP.IMG := X * SINE;
END;  (* CEXP *)


FUNCTION REAL_PART(X:COMPLEX):DOUBLE_REAL;
  (* This function takes the real part of x.                *)
BEGIN (* REAL_PART *)
REAL_PART := X.REL;
END;  (* REAL_PART *)


FUNCTION IMAG_PART(X:COMPLEX):DOUBLE_REAL;
  (* This function takes the imaginary part of x.           *)
BEGIN (* IMAG_PART *)
IMAG_PART := X.IMG;
END;  (* IMAG_PART *)
```

```
PROCEDURE FIND_KS(VAR RHO1,RHO2,THETA1,THETA2:DOUBLE_REAL;
                  VAR K11,K13:COMPLEX;VAR N:INTEGER);
VAR
  T1,T2,T3,T4,T5,T6,T7,T8:COMPLEX;
  X11,X13:COMPLEX;
  SINE:DOUBLE_REAL;
BEGIN (* FIND_KS *)
WRITELN('           Signal Model Parameters           ');
WRITELN(' Enter pole moduli (rho1 & rho2):   ');
READ(RHO1,RHO2);
WRITELN(' Enter pole arguments (theta1,theta2) in degrees: ');
READ(THETA1,THETA2);
WRITELN(' Enter the number of coefficient lags desired: ');
READ(N);

THETA1 := THETA1 * PIE /180.0;
THETA2 := THETA2 * PIE /180.0;

SINE := COS(PIE/2.0 - THETA1);
T1.REL := 0.0;
T1.IMG := -RHO1*RHO1/(2.0*SINE*(1.0-RHO1*RHO1));
T2 := CEXP(1.0,3.0*THETA1);
T3 := CSUB(CEXP(RHO1,THETA1),CEXP(RHO2,THETA2));
T4 := CSUB(CEXP(RHO1,THETA1),CEXP(RHO2,-THETA2));
T5.REL := 1.0;
T5.IMG := 0.0;
T6 := CSUB(T5,CEXP(RHO1*RHO1,2.0*THETA1));
T7 := CSUB(T5,CEXP(RHO1*RHO2,THETA1+THETA2));
T8 := CSUB(T5,CEXP(RHO1*RHO2,THETA1-THETA2));
X11 := CMUL(T1,T2);
T1 := CMUL(T3,T4);
T1 := CMUL(T1,T6);
T1 := CMUL(T1,T7);
T1 := CMUL(T1,T8);
K11 := CDIV(X11,T1);

SINE := COS(PIE/2.0 - THETA2);
T1.REL := 0.0;
T1.IMG := -RHO2*RHO2/(2.0*SINE*(1.0-RHO2*RHO2));
T2 := CEXP(1.0,3.0*THETA2);
T3 := CSUB(CEXP(RHO2,THETA2),CEXP(RHO1,THETA1));
T4 := CSUB(CEXP(RHO2,THETA2),CEXP(RHO1,-THETA1));
T5.REL := 1.0;
T5.IMG := 0.0;
T6 := CSUB(T5,CEXP(RHO2*RHO2,2.0*THETA2));
T7 := CSUB(T5,CEXP(RHO2*RHO1,THETA2+THETA1));
T8 := CSUB(T5,CEXP(RHO2*RHO1,THETA2-THETA1));
X13 := CMUL(T1,T2);
T1 := CMUL(T3,T4);
T1 := CMUL(T1,T6);
T1 := CMUL(T1,T7);
T1 := CMUL(T1,T8);
K13 := CDIV(X13,T1);
END;  (* FIND_KS *)


PROCEDURE AUTOCORRELATE;
VAR
  K11,K13:COMPLEX;
  RHO1,RHO2,THETA1,THETA2:DOUBLE_REAL;
  I,N:INTEGER;
  R:VECTOR;
  T1,T3:COMPLEX;
BEGIN (* AUTOCORRELATE *)
REWRITE(OUTSTUFF,'CORREL_COEFFS');
FIND_KS(RHO1,RHO2,THETA1,THETA2,K11,K13,N);
WRITELN(OUTSTUFF,N);
R[0] := 2.0 * (REAL_PART(K11) + REAL_PART(K13));
FOR I := 1 TO N DO
  BEGIN
  T1 := CMUL(K11,CEXP(1.0,THETA1*I));
  T3 := CMUL(K13,CEXP(1.0,THETA2*I));
  R[I] := 2.0 * ( REAL_PART(T1) * POWER(RHO1,I) +
                  REAL_PART(T3) * POWER(RHO2,I) );
  WRITELN(OUTSTUFF,(R[I]/R[0]):16:10);
  END;
END;  (* AUTOCORRELATE *)


          (* ..... MAINLINE PROGRAM ..... *)
```

```
BEGIN
AUTOCORRELATE;
END.
```

APPENDIX G

Program to Compute the Experimental Values of $Var[\Delta K_k]$ for the Symmetric Split Schur Algorithm

```
#nolist
#include <stdio.h>
#include <math.h>
#list

#define    CLIM       12     /* Max. no. of autocor. coeffs.      */
#define    LIMIT   12000     /* Maximum number of data points     */

#define    finpar  "PAR"     /* Input data to the simulator       */
#define    ffout   "FOUT"    /* Output data sequence              */

FILE       *inpp;  /* declare pointer to input data file          */
FILE       *oupp;  /* declare pointer to output data file         */


double btof();


siggen(signal,length,rho1,rho2,theta1,theta2)
                  /* This function models a 4th order AR process    */
                  /* with two complex pole pairs (moduli rho1,rho2  */
                  /* arguments theta1,theta2 (degrees)). A 4th order */
                  /* all-pole filter is driven by a                 */
                  /* Gaussian noise generator routine based on the  */
                  /* algorithm described in:                        */
                  /*   Rabiner and Gold,"Theory and Application of  */
                  /*   Digital Signal Processing"                   */
double signal[];
double *rho1,*rho2,*theta1,*theta2;
int length;
{
  double tsig[LIMIT];
  double var,x,y,w;
  double a1,a2;
  double pie;
  int i;

  printf("     - SIGNAL MODEL PARAMETERS -      \n");
  printf("   Enter desired noise variance: \n");
  scanf("%f",&var);
  printf("   Enter desired pole moduli (rho1 and rho2): \n");
  scanf("%f%f",rho1,rho2);
  printf("   Enter desired pole arguments (theta1 and theta2) \n");
  printf("   in degrees:                                      \n");
  scanf("%f%f",theta1,theta2);

  fprintf(oupp,"%s"," - 4th Order AR Process Parameters - \n");
  fprintf(oupp,"\n");
  fprintf(oupp,"%s%12.6f\n","  pole modulus rho1 = ",*rho1);
  fprintf(oupp,"%s%12.6f\n","  pole modulus rho2 = ",*rho2);
  fprintf(oupp,"%s%12.6f\n","  pole angle (degrees) theta1 = ",*theta1);
  fprintf(oupp,"%s%12.6f\n","  pole angle (degrees) theta2 = ",*theta2);
  fprintf(oupp,"%s %12.6f \n","  noise variance = ",var);
  fprintf(oupp,"\n\n");

  pie = 3.141592654;
  *theta1 = *theta1 * pie /180.0; /* convert to radians */
  *theta2 = *theta2 * pie /180.0; /* convert to radians */

  a1 = -2.0*cos(*theta1)* *rho1;
  a2 = *rho1 * *rho1;

for (i=0; i < length; ++i)
  {
  x = rand1();
  y = sqrt(2.0*var*log(1.0/x));
  w = y*cos(2.0*pie*rand1()); /* w is a noise point */
  if ( i == 0 )
    {
    signal[0] = w;
    }
  if ( i == 1 )
    {
    signal[1] = w - a1 * signal[0];
    }
  if ( i > 1 )
    {
    signal[i] = w - a1 * signal[i-1] - a2 * signal[i-2];
    }
 }
```

```
    a1 = -2.0*cos(*theta2)* *rho2;
    a2 = *rho2 * *rho2;

  for (i=0; i < length; ++i)
    {
    tsig[i] = signal[i];
    }
  for (i=0; i < length; ++i)
    {
     w = tsig[i];
     if ( i == 0 )
       {
       signal[0] = w;
       }
     if ( i == 1 )
       {
       signal[1] = w - a1 * signal[0];
       }
     if ( i > 1 )
       {
       signal[i] = w - a1 * signal[i-1] - a2 * signal[i-2];
       }
    }
 }


 autocorrelate(s,r,l,nco,nseg)
                 /* Compute nominal ensemble of normalized          */
                 /* autocorrelation coefficients                    */
                 /* (must have length >= l * nseg).                 */
 int l,nco,nseg;
 double s[],r[][CLIM];
 {
   double sum1,sum2;
   int i,j,n;

   for (i=0; i < nseg; i++)
     {
     sum2 = 0.0;
     for (n=0; n<=(l-1) ; n++)
       {
       sum2 = sum2 + s[n + i*l]*s[n + i*l];
       }
     sum2 = sum2/l;
     for (j=1; j<=nco; j++)
       {
       sum1 = 0.0;
       for (n=0; n<=(l-j-1) ; n++)
         {
         sum1 = sum1 + s[n + i*l]*s[n + i*l + j];
         }
       sum1 = sum1/(l - j);
       r[i][j] = sum1/sum2;
       }
   r[i][0] = 1.0;
   }
}


quant(x,n) /* Argument x is a 2(n+m)-bit number that we want to    */
           /* round off to 2m+n+1-bits.                            */
           /* The fractional part of the quantized product is      */
           /* n-1-bits long.  The integer part is not altered.     */
           /* Argument x is a "standard format"                    */
           /* 2(n+m)-bit product.  This function assumes           */
           /* that the computer uses 2s complement arithmetic      */
           /* for integer arithmetic itself.  This function        */
           /* can be used for "double-precision" multiply-         */
           /* accumulate operations.                               */
int x,n;
{
  int q,mask,roun;

  if (x >= 0)
    {
    mask = 01;
    mask = mask << (n-2);
    roun = 0;
    if ( (mask & x) != 0 )
      {
      roun = 1;
```

```
        }
      q = x >> (n-1);
      q = q + roun;
      }
    else
      {
      q = ~x;
      q = q + 1; /* q = 2s compl. of x now. */
      mask = 01;
      mask = mask << (n-2);
      roun = 0;
      if ( (mask & q) != 0 )
        {
        roun = 1;
        }
      q = q >> (n-1);
      q = q + roun;
      q = ~q;
      q = q + 1; /* restore true sign of q */
      }
   return(q);
  }


  ftob(x,n)           /* Convert the double precision floating */
                      /* point number x into a standard format */
                      /* word.                                 */
  int n;
  double x;
  {
    int i,c,mask,masks,sign,ix;
    double fx;

    sign = 0;
    if (x < 0.0)
      {
      sign = -1;
      x = -x;
      }
    ix = x;                  /* find integer part of x    */
    fx = x - (double)ix;     /* find fractional part of x */
    c = 0;
    mask = 01;
    for (i=(n-2); i>=0; i--)
      {
      fx = 2.0*fx;
      masks = mask << i;
      if (fx >= 1.0)
        {
        fx = fx - 1.0;
        c = c | masks;
        }
      }
    fx = 2.0 * fx;
    if (fx >= 1.0)
      {  /* Add unity to effect the rounding operation */
      c = c + 1;
      }
    ix = ix << (n-1);
    c = c + ix;
    if ( sign == -1 )
      {
      c = -c;
      }
    return(c);
}


double btof(x,n,m)   /* Convert standard format x into double */
                     /* precision floating-point number.      */
int n,m,x;
{
  int i,j,sign,mask,masks;
  double c;

  c = 0.0;
  sign = 0;
  mask = 01;
  if (x < 0)
    {
    sign = -1;
```

```
      x = -x;
      }
    for (i=(n-2); i >= 0; i--)
      {
      masks = mask << i;
      if ( (masks & x) != 0 )
        {
        j = n - i - 1;
        c = c + pow(2.0,(double)-j);
        }
      }
    j = 0;
    for ( i=(n-1); i <= (n+m-1); i++)
      {
      masks = mask << i;
      if ( (masks & x) != 0)
        {
        c = c + pow(2.0,(double)j);
        }
      j = j + 1;
      }
    if ( sign == -1 )
      {
      c = -c;
      }
    return(c);
  }


divide(x,y,n,m) /* Find the n+m-bit, 2s complement (standard    */
                /* format) coding of x/y.  x and y are standard */
                /* format n+m-bit binary numbers.               */
int x,y,n,m;
{
  double xf,yf;

  xf = btof(x,n,m);
  yf = btof(y,n,m);
  return(ftob(xf/yf,n));
}



main ()  /*                    SIM_SPLIT                          */
         /*  (Finite Precision Split Schur Algorithm Simulator)   */
         /*                                                       */
         /* We use fixed-point 2s complement arithmetic (n+m - bits, */
         /* including sign) with format (standard format):        */
         /*             x   ... x | x x x  ... x  ,               */
         /*              -m     0   1 2 3      k                  */
         /* where x  is the sign bit, and n = k + 1.              */
         /*        -m                                             */
         /* We use integer  types to contain standard format binary */
         /* numbers.  The rightmost (least significant) bit of an */
         /* integer  type corresponds to x .                      */
         /*                                k                      */
         /*                                                       */
         /* This program simulates the split Schur algo. for symm. */
         /* Toeplitz matrices under finite precision arithmetic   */
         /* conditions.                                           */
         /* The split Schur algorithm simulated is taken from the */
         /* paper:                                                */
         /*                                                       */
         /*    P. Delsarte, Y. Genin, "On the Splitting of Classical */
         /*    Algorithms in Linear Prediction Theory," IEEE Trans. */
         /*    on Acoust., Speech, and Signal Proc., vol. ASSP-35, */
         /*    pp. 645 - 653, May 1987.                           */
         /*                                                       */

{
 int fp_v[CLIM][CLIM];        /* nonnegatively indexed v-parameters   */
                              /* (n+m-bit, 2s complement)             */
 double v[CLIM][CLIM];        /* nonnegatively indexed v-parameters   */
                              /* (double precision floating-point)    */
 double r[CLIM][CLIM];        /* ensemble of normalized autocorrel.   */
                              /* coefficients (nominal) constructed   */
                              /* from segments of signal[];           */
                              /*  r[i][j] - i = segment index         */
                              /*          - j = jth coeff. of seg. i  */
  double mean[CLIM];          /* mean of reflection coefficients in   */
                              /* array fp_k[][]                       */
  double vari[CLIM];          /* variance of reflection coefficients  */
                              /* in array fp_k[][] about their mean   */
                              /* values in mean[]                     */
  double signal[LIMIT];       /* signal generated by siggen function  */
                              /* (double precision floating-point)    */
  double no_k[CLIM][CLIM];    /* ensemble of nominal reflection       */
                              /* coefficients                         */
  double no_a[CLIM][CLIM];    /* ensemble of nominal split Schur      */
                              /* reflection coefficients              */
  double bigk;                /* double precision floating-point      */
                              /* reflection coefficient               */
  double alphak;              /* split Schur reflection coefficient   */
                              /* (double precision floating-point)    */
  int fp_k[CLIM][CLIM];       /* ensemble of finite precision (n+m-bit*/
                              /* , 2s complement) reflection coeffs.  */
  int fp_a[CLIM][CLIM];       /* ensemble of finite precision (n+m-bit*/
                              /* , 2s complement) split Schur         */
                              /* reflection coefficients              */
  int fp_bigk;                /* fixed-point, 2s comp. version of     */
                              /* variable bigk                        */
  int fp_alphak;              /* fixed-point, 2s comp. version of     */
                              /* variable alphak                      */
  double rho1,rho2;           /* pole moduli of 4th order AR model    */
  double theta1,theta2;       /* pole arguments of 4th order AR model */
  int half;                   /* standard format representation of    */
                              /* the number 0.5                       */
  int one;                    /* standard format representation of    */
                              /* the number 1 (one)                   */
  int i,j,k;                  /* loop counter variables               */
  int n,m;                    /* number of bits (including sign bit)  */
                              /* in the fixed-point word is n+m       */
  int length;                 /* length of signal[]                   */
  int l;                      /* number of points used to get r[]     */
  int nco;                    /* largest lag value AND also the       */
                              /* number of reflection coeffs. to be   */
                              /* computed                             */
  int prod;                   /* temporary integer product            */
  int nseg;                   /* number of segments of signal[] used  */
                              /* to compute r[][] (length >=l*nseg)   */

 inpp = fopen(finpar,"r");
 oupp = fopen(ffout,"w");

 fscanf(inpp,"%d %d %d %d %d %d",&n,&m,&length,&l,&nco,&nseg);

 siggen(signal,length,&rho1,&rho2,&theta1,&theta2);
                                    /* construct the test signal      */
 autocorrelate(signal,r,l,nco,nseg); /* find floating-pt. autocorr.   */
                                    /* coefficients                   */

 fprintf(oupp,"%s %d %s \n"," Wordsize = ",n+m," bits ");
 fprintf(oupp,"%s %d \n"," No. of fractional bits = ",n-1);
 fprintf(oupp,"\n\n");

 fprintf(oupp,"%s","  Ensemble of Nominal and Normalized       \n");
 fprintf(oupp,"%s","     Autocorrelation Coefficients          \n");
 fprintf(oupp,"\n");
 for (i=0; i < nseg; i++)
   {
   for (k=0; k<=nco; k++)
     {
     fprintf(oupp,"%s %d %s %9.6f ","r[",k,"] = ",r[i][k]);
     }
   fprintf(oupp,"\n");
   }
 fprintf(oupp,"\n\n");

    /* compute ensemble of nominal reflection coefficients */

 fprintf(oupp,"%s"," Ensemble of Nominal Reflection Coefficients \n");
 fprintf(oupp,"\n");
 for (j=0; j < nseg; j++)
   {
   v[0][0] = r[j][0];
   bigk = 0.0;
   for (k=1; k<=nco; k++)
     {
     v[0][k] = 2.0 * r[j][k];
```

```
        }
      for (k=0; k<nco ; k++)
        {
        v[1][k] = r[j][k] + r[j][k+1];
        }
      for (i=1; i<=nco; i++)
        {
        alphak = v[i][0]/v[i-1][0];
        bigk = 1.0 - (alphak / (1.0 + bigk));
        no_a[j][i] = alphak;
        no_k[j][i] = bigk;
        fprintf(oupp,"%s%d%s%10.6f"," K(",i,") = ",bigk);
        for (k=0; (k <= (nco-i-1)) && (k > -1) ; k++)
          {
          v[i+1][k] = v[i][k] + v[i][k+1] - alphak * v[i-1][k+1];
          }
        }
      fprintf(oupp,"\n");
      }
    fprintf(oupp,"\n\n");

      /* output ensemble of nominal split Schur reflection      */
      /* coefficients                                           */

    fprintf(oupp,"  Ensemble of Nominal Split Schur Reflection \n");
    fprintf(oupp,"                       Coefficients               \n");
    fprintf(oupp,"\n");
    for (j=0; j < nseg ; j++)
      {
      for (i=1; i<=nco ; i++)
        {
        fprintf(oupp,"%s%d%s%10.6f"," a(",i,") = ",no_a[j][i]);
        }
      fprintf(oupp,"\n");
      }
    fprintf(oupp,"\n\n");

      /* compute ensemble of fixed-point reflection coefficients */
      /* assuming that there is no quantization error in the     */
      /* autocorrelation coefficient estimates except that due   */
      /* to rounding to n-bit, 2s complement numbers             */

    fprintf(oupp,"%s","    Ensemble of Fixed-Point Reflection  \n");
    fprintf(oupp,"%s","                Coefficients             \n");
    fprintf(oupp,"%s","   (error-free autocorrelation coeffs.) \n");
    fprintf(oupp,"\n");
    one = 01;
    one = one << (n-1);
    for (j=0; j < nseg; j++)
      {
      fp_v[0][0] = ftob(r[j][0],n);
      fp_bigk = 0;
      for (k=1; k<=nco; k++)
        {
        fp_v[0][k] = 2 * ftob(r[j][k],n);
        }
      for (k=0; k<nco ; k++)
        {
        fp_v[1][k] = ftob(r[j][k],n) + ftob(r[j][k+1],n);
        }
      for (i=1; i<=nco; i++)
        {
        fp_alphak = divide(fp_v[i][0],fp_v[i-1][0],n,m);
        fp_bigk = one - divide(fp_alphak,one + fp_bigk,n,m);
        fp_a[j][i] = fp_alphak;
        fp_k[j][i] = fp_bigk;
        fprintf(oupp,"%s%d%s%10.6f"," K(",i,") = ",btof(fp_bigk,n,m));
        for (k=0; (k <= (nco-i-1)) && ( k > -1) ; k++)
          {
          prod = quant(fp_alphak*fp_v[i-1][k+1],n);
          fp_v[i+1][k] = fp_v[i][k] + fp_v[i][k+1] - prod;
          }
        }
      fprintf(oupp,"\n");
      }
fprintf(oupp,"\n\n");

      /* output ensemble of fixed-point split Schur reflection   */
      /* coefficients                                            */

  fprintf(oupp,"  Ensemble of Fixed-Point Split Schur        \n");
```

```
   fprintf(oupp,"        Reflection Coefficients              \n");
   fprintf(oupp,"\n");
   for (j=0; j < nseg ; j++)
     {
     for (i=1; i<=nco ; i++)
       {
       fprintf(oupp,"%s%d%s%10.6f"," a(",i,") = ",btof(fp_a[j][i],n,m));
       }
     fprintf(oupp,"\n");
     }
   fprintf(oupp,"\n\n");

      /* compute and output the means and variances of the  */
      /* reflection coefficients                             */

 fprintf(oupp,"%s","    Fixed-Point Reflection Coefficient    \n");
 fprintf(oupp,"%s","             Means and Variances          \n");
 fprintf(oupp,"\n");
 for (j=1; j <= nco ; j++)
   {
   mean[j] = 0.0;
   for (i=0; i < nseg; i++)
     {
     mean[j] = mean[j] + btof(fp_k[i][j],n,m);
     }
   mean[j] = mean[j]/nseg;
   }
 for (j=1; j <= nco ; j++)
   {
   vari[j] = 0.0;
   for (i=0; i < nseg; i++)
     {
     bigk = btof(fp_k[i][j],n,m);
     vari[j] = vari[j] + (no_k[i][j] - bigk)*(no_k[i][j] - bigk);
     }
   vari[j] = vari[j]/nseg;
   }
 fprintf(oupp,"%s\n","        Mean          Variance          ");
 for(j=1; j <= nco ; j++)
   {
   fprintf(oupp,"%s%d%s %11.8f %14.8f \n","K(",j,")",mean[j],vari[j]);
   }
}
```

```
------------------------------------------------------------------------------------------
 - 4th Order AR Process Parameters -

  pole modulus rho1 =        0.500000
  pole modulus rho2 =        0.500000
  pole angle (degrees) theta1 =     10.000000
  pole angle (degrees) theta2 =     15.000000
  noise variance =      1.000000


 Wordsize =  10  bits
 No. of fractional bits =  7


  Ensemble of Nominal and Normalized
     Autocorrelation Coefficients

r[ 0 ] =    1.000000 r[ 1 ] =    0.938136 r[ 2 ] =    0.788623 r[ 3 ] =    0.609614 r[ 4 ] =    0.444046
r[ 0 ] =    1.000000 r[ 1 ] =    0.931751 r[ 2 ] =    0.771818 r[ 3 ] =    0.588631 r[ 4 ] =    0.426595
r[ 0 ] =    1.000000 r[ 1 ] =    0.938758 r[ 2 ] =    0.789428 r[ 3 ] =    0.607712 r[ 4 ] =    0.435048
r[ 0 ] =    1.000000 r[ 1 ] =    0.937809 r[ 2 ] =    0.786584 r[ 3 ] =    0.601930 r[ 4 ] =    0.423253
r[ 0 ] =    1.000000 r[ 1 ] =    0.937701 r[ 2 ] =    0.786034 r[ 3 ] =    0.602787 r[ 4 ] =    0.429033
r[ 0 ] =    1.000000 r[ 1 ] =    0.938130 r[ 2 ] =    0.788147 r[ 3 ] =    0.604250 r[ 4 ] =    0.428190
r[ 0 ] =    1.000000 r[ 1 ] =    0.933054 r[ 2 ] =    0.774047 r[ 3 ] =    0.583591 r[ 4 ] =    0.408362
r[ 0 ] =    1.000000 r[ 1 ] =    0.931928 r[ 2 ] =    0.773845 r[ 3 ] =    0.592044 r[ 4 ] =    0.433188
r[ 0 ] =    1.000000 r[ 1 ] =    0.929390 r[ 2 ] =    0.759295 r[ 3 ] =    0.549666 r[ 4 ] =    0.348103
r[ 0 ] =    1.000000 r[ 1 ] =    0.930963 r[ 2 ] =    0.769325 r[ 3 ] =    0.578942 r[ 4 ] =    0.409944


 Ensemble of Nominal Reflection Coefficients

 K(1) =  -0.938136 K(2) =   0.762937 K(3) =  -0.420445 K(4) =   0.099673
 K(1) =  -0.931751 K(2) =   0.730756 K(3) =  -0.404517 K(4) =   0.117999
 K(1) =  -0.938758 K(2) =   0.773485 K(3) =  -0.409445 K(4) =   0.120979
 K(1) =  -0.937809 K(2) =   0.770888 K(3) =  -0.379434 K(4) =   0.149952
 K(1) =  -0.937701 K(2) =   0.772476 K(3) =  -0.425361 K(4) =   0.172044
 K(1) =  -0.938130 K(2) =   0.766736 K(3) =  -0.349100 K(4) =   0.051207
 K(1) =  -0.933054 K(2) =   0.746018 K(3) =  -0.324859 K(4) =  -0.003518
 K(1) =  -0.931928 K(2) =   0.719686 K(3) =  -0.355830 K(4) =   0.011541
 K(1) =  -0.929390 K(2) =   0.766865 K(3) =  -0.276909 K(4) =   0.020219
 K(1) =  -0.930963 K(2) =   0.730403 K(3) =  -0.314886 K(4) =  -0.056566


  Ensemble of Nominal Split Schur Reflection
                 Coefficients

 a(1) =    1.938136 a(2) =   0.014665 a(3) =    2.504156 a(4) =    0.521788
 a(1) =    1.931751 a(2) =   0.018375 a(3) =    2.430877 a(4) =    0.525215
 a(1) =    1.938758 a(2) =   0.013872 a(3) =    2.499631 a(4) =    0.519110
 a(1) =    1.937809 a(2) =   0.014248 a(3) =    2.442823 a(4) =    0.527510
 a(1) =    1.937701 a(2) =   0.014174 a(3) =    2.526420 a(4) =    0.475774
 a(1) =    1.938130 a(2) =   0.014432 a(3) =    2.383504 a(4) =    0.617568
 a(1) =    1.933054 a(2) =   0.017002 a(3) =    2.313228 a(4) =    0.677516
 a(1) =    1.931928 a(2) =   0.019081 a(3) =    2.331603 a(4) =    0.636735
 a(1) =    1.929390 a(2) =   0.016461 a(3) =    2.256127 a(4) =    0.708470
 a(1) =    1.930963 a(2) =   0.018611 a(3) =    2.275284 a(4) =    0.723868


   Ensemble of Fixed-Point Reflection
                Coefficients
  (error-free autocorrelation coeffs.)

 K(1) =  -0.937500 K(2) =   0.750000 K(3) =  -0.429687 K(4) =   0.125000
 K(1) =  -0.929687 K(2) =   0.664062 K(3) =  -0.101562 K(4) =  -0.414062
 K(1) =  -0.937500 K(2) =   0.750000 K(3) =  -0.429687 K(4) =   0.304687
 K(1) =  -0.937500 K(2) =   0.750000 K(3) =  -0.289062 K(4) =  -0.101562
 K(1) =  -0.937500 K(2) =   0.750000 K(3) =  -0.289062 K(4) =  -0.250000
 K(1) =  -0.937500 K(2) =   0.750000 K(3) =  -0.289062 K(4) =  -0.250000
 K(1) =  -0.929687 K(2) =   0.664062 K(3) =  -0.101562 K(4) =  -0.109375
 K(1) =  -0.929687 K(2) =   0.664062 K(3) =  -0.203125 K(4) =  -0.046875
 K(1) =  -0.929687 K(2) =   0.781250 K(3) =  -0.406250 K(4) =   0.156250
 K(1) =  -0.929687 K(2) =   0.664062 K(3) =  -0.320312 K(4) =   0.195312


 Ensemble of Fixed-Point Split Schur
       Reflection Coefficients

 a(1) =    1.937500 a(2) =   0.015625 a(3) =    2.500000 a(4) =    0.500000
 a(1) =    1.929687 a(2) =   0.023437 a(3) =    1.835937 a(4) =    1.273437
 a(1) =    1.937500 a(2) =   0.015625 a(3) =    2.500000 a(4) =    0.398437
 a(1) =    1.937500 a(2) =   0.015625 a(3) =    2.250000 a(4) =    0.781250
 a(1) =    1.937500 a(2) =   0.015625 a(3) =    2.250000 a(4) =    0.890625
 a(1) =   1.937500 a(2) =   0.015625 a(3) =   2.250000 a(4) =   0.890625
 a(1) =   1.929687 a(2) =   0.023437 a(3) =   1.835937 a(4) =   1.000000
 a(1) =   1.929687 a(2) =   0.023437 a(3) =   2.000000 a(4) =   0.835937
 a(1) =   1.929687 a(2) =   0.015625 a(3) =   2.500000 a(4) =   0.500000
 a(1) =   1.929687 a(2) =   0.023437 a(3) =   2.203125 a(4) =   0.546875

    Fixed-Point Reflection Coefficient
            Means and Variances

          Mean          Variance
K(1) -0.93359375      0.00000248
K(2)  0.71875000      0.00208076
K(3) -0.28593750      0.02125650
K(4) -0.03906250      0.07461526
```

APPENDIX H

Summary Tables for Output from the

Programs in Appendices E and G

Theoretical Values ($\times 10^4$)				
b	8	10	12	14
$Var[\Delta K_4]$	5480	342	21.4	1.34
$Var[\Delta K_3]$	1440	90.0	5.63	0.352
$Var[\Delta K_2]$	63.1	3.94	0.246	0.0154
$Var[\Delta K_1]$	0.153	0.0095	0.0006	~0

Experimental Values ($\times 10^4$)				
b	8	10	12	14
$Var[\Delta K_4]$	746	415	10.5	1.10
$Var[\Delta K_3]$	213	136	4.88	0.377
$Var[\Delta K_2]$	20.8	2.85	0.189	0.0125
$Var[\Delta K_1]$	0.0248	0.0023	0.0002	~0

Table VII: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho_1 = 0.5$, $\rho_2 = 0.5$, $\theta_1 = 10°$, and $\theta_2 = 15°$.

Theoretical Values ($\times 10^4$)				
b	8	10	12	14
$Var[\Delta K_4]$	58200	3640	227	14.2
$Var[\Delta K_3]$	10700	666	41.6	2.60
$Var[\Delta K_2]$	238	14.9	0.930	0.0581
$Var[\Delta K_1]$	0.153	0.0095	0.0006	~0

Experimental Values ($\times 10^4$)				
b	8	10	12	14
$Var[\Delta K_4]$	-	6450	429	27.0
$Var[\Delta K_3]$	-	705	30.5	2.18
$Var[\Delta K_2]$	-	21.0	0.601	0.0406
$Var[\Delta K_1]$	-	0.0041	0.0001	~0

Table VIII: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho_1 = 0.75$, $\rho_2 = 0.5$, $\theta_1 = 5°$, and $\theta_2 = 45°$.

Theoretical Values ($\times 10^4$)				
b	8	10	12	14
$Var[\Delta K_4]$	4.51	0.281	0.0176	0.0011
$Var[\Delta K_3]$	8.81	0.551	0.0344	0.0022
$Var[\Delta K_2]$	0.212	0.0133	0.0008	0.0001
$Var[\Delta K_1]$	0.153	0.0095	0.0006	~0

Experimental Values ($\times 10^4$)				
b	8	10	12	14
$Var[\Delta K_4]$	10.5	0.728	0.0350	0.0024
$Var[\Delta K_3]$	8.4	0.494	0.0338	0.0034
$Var[\Delta K_2]$	0.0881	0.0105	0.0012	~0
$Var[\Delta K_1]$	0.0660	0.0026	0.0001	~0

Table IX: Comparison of theoretical and experimental results for the finite precision symmetric split Schur algorithm. Here $\rho_1 = 0.75$, $\rho_2 = 0.75$, $\theta_1 = 85°$, and $\theta_2 = 90°$.

APPENDIX I

Program to Test if $j \in \hat{Z}_{p^r}$

```
PROGRAM CONJECTURE(INSTUFF,OUTSTUFF);

   (* This program tests my conjecture concerning whether or not  *)
   (* it is possible to have a " Quadratic Finite Segment p-adic  *)
   (* Number System ".  It tests whether quadratic residues, i.e. *)
   (* values j that satisfy                                       *)
   (*                          2          r                       *)
   (*                         j  = -1 (mod p ) .                  *)
   (* are such that j belongs to the "hatted" subset of the ring  *)
   (*                                Z  .                         *)
   (*                                 r                           *)
   (*                                p                            *)
   (* If p is a Gaussian prime then precisely two values of j     *)
   (* exist to satisfy the above congruence and that belong to    *)
   (* above ring.                                                 *)

CONST
  LIMIT = 100;
TYPE
  VECTOR = ARRAY [-1 .. LIMIT] OF INTEGER;
VAR
  INSTUFF,OUTSTUFF:TEXT;


FUNCTION POW(X,Y:INTEGER):INTEGER;
  (* Raise positive integer X to the Yth power (Y is positive    *)
  (* integer).                                                   *)
VAR
  I,TEMP:INTEGER;
BEGIN (* POW *)
IF Y = 0 THEN
  BEGIN
  POW := 1;
  END;
IF Y = 1 THEN
  BEGIN
  POW := X;
  END;
IF Y > 1 THEN
  BEGIN
  TEMP := X;
  FOR I := 1 TO Y-1 DO
    BEGIN
    TEMP := TEMP * X;
    END;
  POW := TEMP;
  END;
END;  (* POW *)

FUNCTION MPOW(X,Y,M:INTEGER):INTEGER;
  (* Raise X to the Yth power modulo M.                            *)
VAR
  I,TEMP:INTEGER;
BEGIN (* MPOW *)
TEMP := X MOD M;
FOR I := 1 TO Y-1 DO
  BEGIN
  TEMP := (TEMP * X) MOD M;
  END;
MPOW := TEMP;
END;  (* MPOW *)

PROCEDURE READIN(VAR P,R,J1,J2:INTEGER);
  (* Read in the following parameters:                             *)
  (*     P - Gaussian prime number p.                              *)
  (*     R - the rth power of p (R = r).                           *)
  (*    J1 - first quadratic residue if r = 1.                     *)
  (*    J2 - second quadratic residue if r = 1.                    *)
BEGIN (* READIN *)
RESET(INSTUFF,'RESIDUES');
READ(INSTUFF,P,R,J1,J2);
END;  (* READIN *)

FUNCTION QUAD_RES(P,R,J:INTEGER):INTEGER;
  (* This function finds the quadratic residue for p (= P) raised *)
  (* to the rth (r = R) power given the quadratic residue         *)
  (* j (= J) when r = 1.                                          *)
  (*                                                              *)
  (* This function utilises the theorem in the appendix of the    *)
  (* paper:                                                       *)
  (*        W. K. Jenkins, J. V. Krogmeier, "The Design of Dual-   *)
```

```
    (*       Mode Complex Signal Processors Based on Quadratic      *)
    (*       Modular Number Codes," IEEE Trans. on Circ. and Syst., *)
    (*       Vol. CAS-34, pp. 354-364, April 1987.                  *)
   VAR
    M,K,A,T1,T2,T3,T4:INTEGER;
   BEGIN (* QUAD_RES *)
   A := J;
   M := POW(P,R);
   FOR K := 1 TO R-1 DO
     BEGIN
     T1 := (A * A) MOD M;
     T1 := (T1 + 1) MOD M;
     T2 := MPOW(2,P-2,M);
     T3 := MPOW(A,P-2,M);
     T4 := (T2 * T1) MOD M;
     T4 := (T4 * T3) MOD M;
     A  := (A - T4) MOD M;
     END;
   QUAD_RES := A;
   END;  (* QUAD_RES *)

  FUNCTION FIND_N(P,R:INTEGER):INTEGER;
    (* Find the largest possible N satisfying                    *)
    (*                    2          r                            *)
    (*                  2N  + 1 <= p                              *)
  VAR
    X:DOUBLE_REAL;
  BEGIN (* FIND_N *)
  X := POW(P,R);
  X := SQRT( (X - 1.0)/2.0 );
  FIND_N := TRUNC(X);
  END;  (* FIND_N *)

  FUNCTION TEST(REM,D:VECTOR;N:INTEGER):BOOLEAN;
    (* If REM[I]/D[I-1] is an order-N Farey fraction then return  *)
    (* TRUE, otherwise return FALSE.                              *)
  VAR
    I:INTEGER;
  BEGIN (* TEST *)
  TEST := FALSE;
  I := 1;
  REPEAT
    IF ( ( 0 <= ABS(REM[I]) ) AND ( ABS(REM[I]) <= N ) AND
         ( 0 <  ABS(D[I-1]) ) AND ( ABS(D[I-1]) <= N ) ) THEN
      BEGIN
      TEST := TRUE;
      END;
    I := I + 1;
    UNTIL REM[I] = 0
  END;  (* TEST *)

 PROCEDURE EUCLID(P,R,X1,X2:INTEGER);
   (* Herein the Kornerup-Gregory algorithm for finding the Farey *)
   (* fraction corresponding to a finite ring element is found.   *)
 VAR
   I:INTEGER;
   N,J1,J2:INTEGER;
   REM,QUOT,D:VECTOR;
 BEGIN (* EUCLID *)
 N := FIND_N(P,R);
 IF R > 1 THEN
   BEGIN
   J1 := QUAD_RES(P,R,X1);
   J2 := QUAD_RES(P,R,X2);
   END
 ELSE
   BEGIN
   J1 := X1;
   J2 := X2;
   END;
 WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);
 WRITELN(OUTSTUFF,' p = ',P:4,' r = ',R:4);
 WRITELN(OUTSTUFF,' N = ',N:8,' p ** r = ',POW(P,R):8);
 WRITELN(OUTSTUFF,'   2               ');
 WRITELN(OUTSTUFF,' 2N  + 1      = ',(2 * N * N + 1):8);
 WRITELN(OUTSTUFF,'         2        ');
 WRITELN(OUTSTUFF,' 2(N+1)   + 1 = ',(2*(N+1)*(N+1) + 1):8);
 WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);

 WRITELN(OUTSTUFF,' j = ',J1:8);
 WRITELN(OUTSTUFF);
```

```
REM[0] := POW(P,R);
REM[1] := J1;
D[0] := 1;
D[-1] := 0;
WRITELN(OUTSTUFF,'          ',REM[0]:8,' ',D[-1]:8);
WRITELN(OUTSTUFF,'          ',REM[1]:8,' ',D[0]:8);
WRITELN(OUTSTUFF);
I := 1;
WHILE  REM[I] <> 0 DO
  BEGIN
  REM[I + 1] := REM[I - 1] MOD REM[I];
  QUOT[I] := REM[I - 1] DIV REM[I];
  D[I] := D[I - 2] - QUOT[I] * D[I - 1];
  WRITELN(OUTSTUFF,QUOT[I]:8,REM[I+1]:8,D[I]:8);
  I := I + 1;
  END;
WRITELN(OUTSTUFF);
IF TEST(REM,D,N) THEN
  BEGIN
  WRITELN(OUTSTUFF,'  j has a Hensel code ! ');
  END
ELSE
  BEGIN
  WRITELN(OUTSTUFF,'  j has no Hensel code. ');
  END;
WRITELN(OUTSTUFF);WRITELN(OUTSTUFF);

WRITELN(OUTSTUFF,' j = ',J2:8);
WRITELN(OUTSTUFF);
REM[1] := J2;
WRITELN(OUTSTUFF,'          ',REM[0]:8,' ',D[-1]:8);
WRITELN(OUTSTUFF,'          ',REM[1]:8,' ',D[0]:8);
WRITELN(OUTSTUFF);
I := 1;
WHILE REM[I] <> 0 DO
  BEGIN
  REM[I + 1] := REM[I - 1] MOD REM[I];
  QUOT[I] := REM[I - 1] DIV REM[I];
  D[I] := D[I - 2] - QUOT[I] * D[I - 1];
  WRITELN(OUTSTUFF,QUOT[I]:8,REM[I+1]:8,D[I]:8);
  I := I + 1;
  END;
WRITELN(OUTSTUFF);
IF TEST(REM,D,N) THEN
  BEGIN
  WRITELN(OUTSTUFF,'  j has a Hensel code ! ');
  END
ELSE
  BEGIN
  WRITELN(OUTSTUFF,'  j has no Hensel code. ');
  END;
END;  (* EUCLID *)

PROCEDURE INVERSE_MAP;
  (* This procedure uses the Kornerup-Gregory Euclidean algorithm  *)
  (* approach to mapping an element of the ring to an order-N      *)
  (* Farey fraction.  Whether or not quadratic residue j has a     *)
  (* Hensel code is output by this program.                        *)
VAR
  I,P,R,J1,J2:INTEGER;
BEGIN (* INVERSE_MAP *)
REWRITE(OUTSTUFF,'FAREY');
READIN(P,R,J1,J2);
FOR I := 1 TO R DO
  BEGIN
  EUCLID(P,I,J1,J2);
  END;
END;  (* INVERSE_MAP *)


                (* ..... MAINLINE PROGRAM ..... *)


BEGIN
INVERSE_MAP;
END.
```

```
p =   13 r =    1
N =        2 p ** r =       13
  2
2N  + 1        =        9
         2
2(N+1)   + 1 =        19

j =        5

               13        0
                5        1

      2         3       -2
      1         2        3
      1         1       -5
      2         0       13

 j has no Hensel code.

j =        8

               13        0
                8        1

      1         5       -1
      1         3        2
      1         2       -3
      1         1        5
      2         0      -13

 j has no Hensel code.


p =   13 r =    2
N =        9 p ** r =      169
  2
2N  + 1        =      163
         2
2(N+1)   + 1 =        201

j =       70

              169        0
               70        1

      2        29       -2
      2        12        5
      2         5      -12
      2         2       29
      2         1      -70
      2         0      169

 j has no Hensel code.

j =       99

              169        0
               99        1

      1        70       -1
      1        29        2
      2        12       -5
      2         5       12
      2         2      -29
      2         1       70
      2         0     -169

 j has no Hensel code.


p =   13 r =    3
N =       33 p ** r =     2197
```

```
  2
2N  + 1      =     2179
      2
2(N+1)  + 1 =     2313

j =       239

              2197        0
               239        1

        9      46        -9
        5       9        46
        5       1      -239
        9       0      2197

j has no Hensel code.

j =      1958

              2197        0
              1958        1

        1     239        -1
        8      46         9
        5       9       -46
        5       1       239
        9       0     -2197

j has no Hensel code.


p =    13 r =     4
N =        119 p ** r =     28561
  2
2N  + 1      =     28323
      2
2(N+1)  + 1 =     28801

j =       239

             28561        0
               239        1

      119     120      -119
        1     119       120
        1       1      -239
      119       0     28561

j has no Hensel code.

j =     28322

             28561        0
             28322        1

        1     239        -1
      118     120       119
        1     119      -120
        1       1       239
      119       0    -28561

j has no Hensel code.
```

APPENDIX J

Summary Table for Output from the

Program in Appendix I

p	*range of* r	j *values* (r=1 *case only*)
5	$1 \leq r \leq 6$	2,3
13	$1 \leq r \leq 4$	5,8
17	$1 \leq r \leq 3$	4,13
29	$1 \leq r \leq 3$	12,17
37	$1 \leq r \leq 2$	6,31
41	$1 \leq r \leq 2$	9,32
53	$1 \leq r \leq 2$	23,30
257	$1 \leq r \leq 2$	16,241

Table I: Sets of r and p showing that $j \notin \hat{Z}_{p^r}$.