

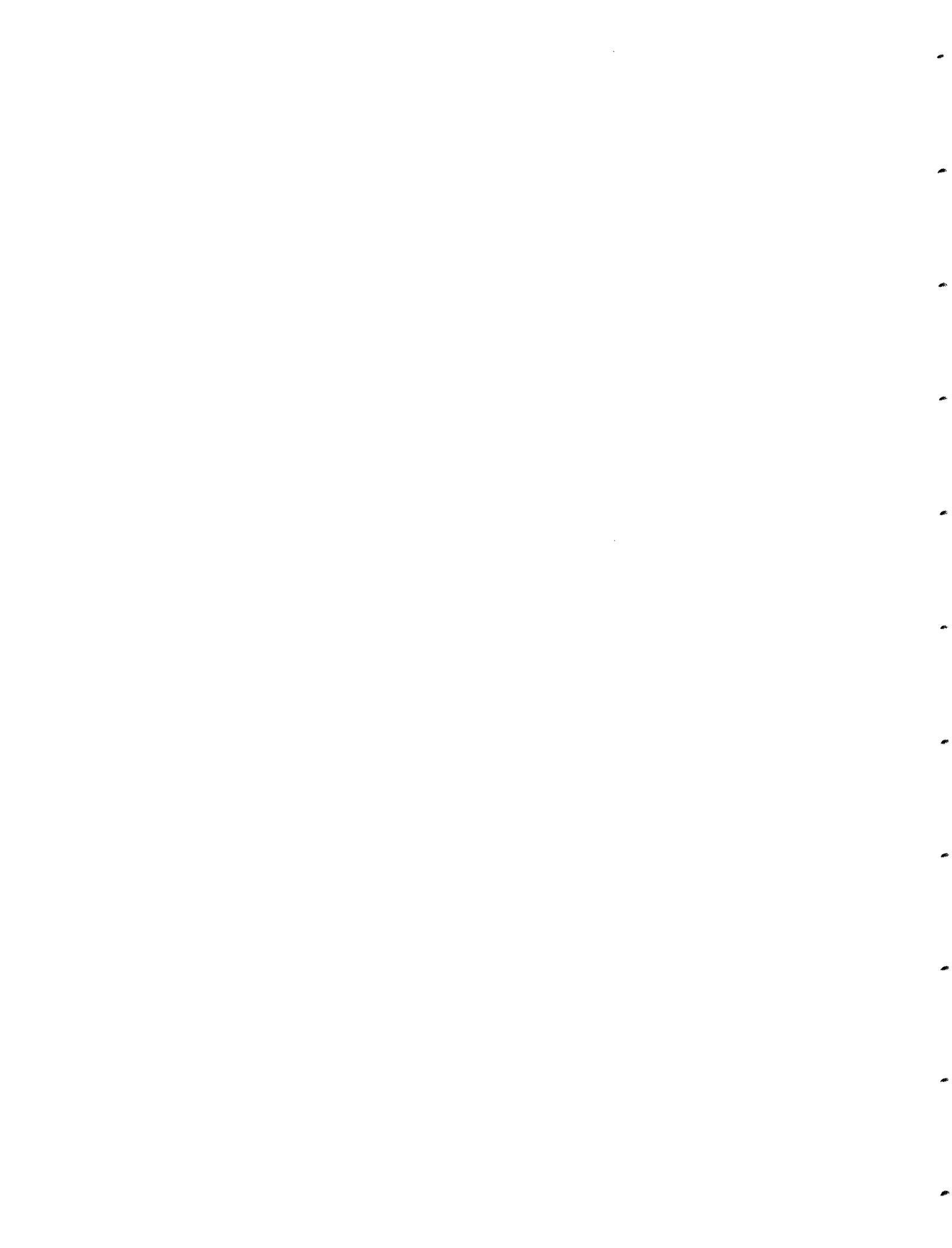
***Levinson and Fast Choleski Algorithms for
Toeplitz and Almost Toeplitz
Matrices***

RLE Technical Report No. 538

December 1988

Bruce R. Musicus

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA



Levinson and Fast Choleski Algorithms for Toeplitz and Almost Toeplitz Matrices

Bruce R. Musicus

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, Mass. 02139

ABSTRACT

In this paper, we review Levinson and fast Choleski algorithms for solving sets of linear equations involving Toeplitz or almost Toeplitz matrices. The Levinson-Trench-Zohar algorithm is first presented for solving problems involving exactly Toeplitz matrices. A fast Choleski algorithm is derived by a simple linear transformation. The almost Toeplitz problem is then considered and a Levinson-style algorithm is proposed for solving it. A set of linear transformations converts the algorithm into a fast Choleski method. Symmetric and band diagonal applications are considered. Formulas for the inverse of an almost Toeplitz matrix are derived. The relationship between the fast Choleski algorithms and a Euclidian algorithm is exploited in order to derive accelerated "doubling" algorithms for inverting the matrix. Finally, strategies for removing the strongly nonsingular constraint of Levinson recursion are considered. We conclude by applying the techniques to several applications, including covariance methods of linear prediction, rational Toeplitz matrices, and optimal finite interval ARMA smoothing filters.

November 3, 1981

Levinson and Fast Choleski Algorithms for Toeplitz and Almost Toeplitz Matrices

Bruce R. Musicus

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, Mass. 02139

1. Introduction

One of the most common problems in numerical calculation is to solve a set of linear equations:

$$R_N \underline{x}_N = \underline{y}_N \tag{1.1}$$

for the $(N+1)$ long vector \underline{x}_N where R_N is an $(N+1) \times (N+1)$ matrix. Standard methods for solving this problem, such as gaussian elimination or Choleski decomposition, generally require $O(N^3)$ operations. When R_N has additional structure, however, the computation can often be significantly reduced. In particular, if R_N is Toeplitz with $(i,j)^{th}$ element $R_{i,j} = r(i-j)$, then the Levinson-Trench-Zohar recursive algorithms^{1,2,3,4} can solve the linear equations using only $O(N^2)$ operations and $O(N)$ storage. Similar fast algorithms proposed originally by Friedlander, Morf, Kailath, and Ljung⁵ and others apply when R_N is almost Toeplitz in the sense of having "low displacement rank". These Levinson-style algorithms can be viewed as fast procedures for decomposing the inverse matrix R_N^{-1} into a product of Upper triangular, Diagonal, and Lower triangular (UDL) matrices. Applying a simple linear transformation to these algorithms yields a set of "fast Choleski" algorithms which compute an LDU decomposition of R_N itself using only $O(N^2)$ operations. These algorithms were first discussed by Bareiss⁶ Morf,⁷ and Rissanen.⁸ In general, the fast Choleski algorithms will either require much more storage or somewhat more computa-

tion than the Levinson-style algorithms in order to compute \underline{x}_N . When the matrix R_N is also band diagonal, however, variants of the fast Choleski algorithm can be derived which are significantly superior to the Levinson-style band diagonal algorithms suggested by Trench⁹ and Dickinson,¹⁰ and asymptotically superior to the $O(N \log N)$ matrix splitting and imbedding approaches of Jain¹¹ and Morf, Kailath.¹²

The fast Choleski algorithms bear a remarkable resemblance to a Euclidian polynomial algorithm. Since a "divide and conquer" strategy combined with Fast Fourier Transforms (FFT's) can be used to accelerate Euclidian algorithms,¹³ it can also be used to accelerate our fast Choleski algorithms. The result is an $O(N \log^2 N)$ doubling algorithm for computing R_N^{-1} , which is similar to those of Gustavson and Yun,¹⁴ Bitmead and Anderson,¹⁵ and Morf¹⁶ Unfortunately, the algorithm is relatively complex, so that even for exactly Toeplitz matrices it is only advantageous for matrices of size $N > 2000$.

One difficulty with all these algorithms, except that of Gustavson and Yun, is that they require that all the upper left principal minors of R_N must be non-singular. In the closing sections of this report, we show how this constraint can be removed.

This paper is intended as a coherent summary of Toeplitz matrix algorithms, as well as a presentation of several new results. The approach used for deriving the fast Choleski algorithm appears to be new. The displacement rank formalism of Kailath, Kung, Morf¹⁷ is used in deriving the Levinson-style almost-Toeplitz algorithms, instead of that of Friedlander *et al.*¹⁸ This simplifies the derivation and the inversion formulas. The band diagonal fast Choleski algorithms using forward and backward recursions to minimize storage appear to be new. Section 8 contains a new partial LDU decomposition formula for R_N , which suggests an interesting result for Schur complements. The derivation of

the doubling algorithms is completely new, and is considerably more concise and powerful than previous algorithms. The method for dealing with degeneracy is also new. Finally, in addition to several examples previously considered in the literature, we also present several new applications of the methods, including a very fast finite interval Wiener-Hopf smoothing filter for ARMA models, and a problem in which R_N is a rational matrix. This last example has also been considered by Dickinson;¹⁹ our approach is much faster, and appears to be numerically stable.

2. The Levinson-Trench-Zohar Algorithm for Exactly Toeplitz Matrices

The Levinson-Trench-Zohar (LTZ) algorithm⁴ is a recursive method for solving the simultaneous linear equations $R_N \underline{x}_N = \underline{y}_N$ when the matrix R_N is exactly Toeplitz. For simplicity we will consider the case when the entries of R_N are scalar, although the case of block Toeplitz matrices can be handled in much the same way.^{8,20,21} Let the entries of vectors \underline{x}_N and \underline{y}_N be $x_{i,N}$ and $y_{i,N}$ for $i=0, \dots, N$. Let R_n be the $(n+1) \times (n+1)$ upper left principal minor of R_N , and let \underline{y}_n be the vector containing the first $(n+1)$ components of \underline{y}_N . Note that because of the Toeplitz structure of R_N , we can partition each minor R_n so as to show its relationship to the lower order minor R_{n-1} :

$$R_n = \begin{bmatrix} R_{n-1} & r(-n) \\ & \vdots \\ r(n) & \cdots & r(0) \end{bmatrix} = \begin{bmatrix} r(0) & \cdots & r(-n) \\ \vdots & & \\ r(n) & & R_{n-1} \end{bmatrix} \quad (2.1)$$

This structure suggests an approach for solving the linear equations (1.1) in which we recursively calculate the solution to the following problems for $n=0, \dots, N$:

$$R_n \underline{a}_n = \begin{bmatrix} \varepsilon_n \\ \underline{0}_n \end{bmatrix} \quad ; \quad R_n \underline{b}_n = \begin{bmatrix} \underline{0}_n \\ \varepsilon_n \end{bmatrix} \quad ; \quad R_n \underline{x}_n = \underline{y}_n \quad (2.2)$$

where $\underline{0}_n$ is a vector of n zeroes, ε_n will be defined later, and where:

$$\begin{aligned}\underline{a}_n &= (1 \ a_{1,n} \ \cdots \ a_{n,n})^T \\ \underline{b}_n &= (b_{n,n} \ \cdots \ b_{1,n} \ 1)^T \\ \underline{x}_n &= (x_{0,n} \ \cdots \ x_{n,n})^T\end{aligned}\tag{2.3}$$

The solution for $n=0$ can be found by inspection:

$$\underline{a}_0 = \underline{b}_0 = \begin{bmatrix} 1 \end{bmatrix} \quad ; \quad \varepsilon_0 = r(0) \quad ; \quad \underline{x}_0 = \begin{bmatrix} y_0 \\ \varepsilon_0 \end{bmatrix}\tag{2.4}$$

The solutions at stage $n>0$ can now be calculated recursively using the solutions at stage $n-1$. To do this, assume that we know the values of \underline{a}_{n-1} , \underline{b}_{n-1} , ε_{n-1} , and \underline{x}_{n-1} . Then from (2.1) it is easy to show that:

$$R_n \begin{bmatrix} \underline{a}_{n-1} \\ 0 \end{bmatrix} = \begin{bmatrix} \varepsilon_{n-1} \\ \underline{0}_{n-1} \\ -\xi_n \varepsilon_{n-1} \end{bmatrix} \quad \text{and} \quad R_n \begin{bmatrix} 0 \\ \underline{b}_{n-1} \end{bmatrix} = \begin{bmatrix} -\nu_n \varepsilon_{n-1} \\ \underline{0}_{n-1} \\ \varepsilon_{n-1} \end{bmatrix}\tag{2.5}$$

where

$$\begin{aligned}\xi_n &= -\frac{1}{\varepsilon_{n-1}} \sum_{j=0}^{n-1} r(n-j) a_{j,n-1} \\ \nu_n &= -\frac{1}{\varepsilon_{n-1}} \sum_{j=1}^n r(-j) b_{n-j,n-1}\end{aligned}\tag{2.6}$$

Values of \underline{a}_n and \underline{b}_n which satisfy (2.2) can thus be computed as appropriate linear combinations of \underline{a}_{n-1} and \underline{b}_{n-1} :

$$\begin{aligned}\underline{a}_n &= \begin{bmatrix} \underline{a}_{n-1} \\ 0 \end{bmatrix} + \xi_n \begin{bmatrix} 0 \\ \underline{b}_{n-1} \end{bmatrix} \\ \underline{b}_n &= \begin{bmatrix} 0 \\ \underline{b}_{n-1} \end{bmatrix} + \nu_n \begin{bmatrix} \underline{a}_{n-1} \\ 0 \end{bmatrix}\end{aligned}\tag{2.7}$$

Direct substitution shows that:

$$\varepsilon_n = \varepsilon_{n-1}(1 - \xi_n \nu_n)\tag{2.8}$$

Finally, from (2.1) we see that:

$$R_n \begin{bmatrix} \underline{x}_{n-1} \\ 0 \end{bmatrix} = \underline{y}_n - \begin{bmatrix} 0 \\ \lambda_n \end{bmatrix}\tag{2.9}$$

$$\text{where } \lambda_n = y_n - \sum_{j=0}^{n-1} r(n-j) x_{j,n-1}$$

Thus the solution \underline{x}_n can be computed as:

$$\underline{x}_n = \begin{pmatrix} \underline{x}_{n-1} \\ 0 \end{pmatrix} + \frac{\lambda_n}{\varepsilon_n} \underline{b}_n \quad (2.10)$$

To summarize, we start with the initial solutions $\underline{a}_0, \underline{b}_0, \underline{x}_0$ in (2.4), then use the recursions in (2.7) and (2.10) to calculate $\underline{a}_n, \underline{b}_n, \underline{x}_n$ for $n=0, \dots, N$, at which point we will have found the desired solution \underline{x}_N . Total computation will be about $3N^2$ operations (1 operation \approx 1 add + 1 multiply). Because the computation can be done in place, total storage required will be about $5N$ locations for $\underline{a}_n, \underline{b}_n, \underline{y}_n$, and $\tau(-N), \dots, \tau(N)$. (The solution for \underline{x}_n can be stored in the same location used for \underline{y}_n .) The algorithm will work correctly provided that $\varepsilon_n \neq 0$ at each stage n . We will see later that this condition is equivalent to requiring that all the principal minors R_n of R_N must all be non-singular, a condition known as "strong non-singularity". In the terminology of linear prediction, the vectors \underline{a}_n and \underline{b}_n are known as the forward and backward predictors, ε_n is the prediction error, and ξ_n and ν_n are the forward and backward reflection coefficients.

Note that if R_N is symmetric, with $\tau(n)=\tau(-n)$, then the vector \underline{a}_n will be identical to \underline{b}_n except with the elements in reverse order, $a_{j,n}=b_{j,n}$. The forward and backward reflection coefficients will also be identical, $\xi_n=\lambda_n$. These relationships can be used to cut the computation required to only $2N^2$ operations, and cut the storage required to about $3N$ locations. If the matrix R_N is also positive definite, then by Sylvester's criterion all the principal minors R_n will be non-singular and thus ε_n will never be zero.

An interesting interpretation of the vectors \underline{a}_n and \underline{b}_n can be gained by forming the $(n+1) \times (n+1)$ matrices A_n and B_n whose j^{th} rows contain the coefficients of the vectors \underline{a}_j and \underline{b}_j .

$$A_n = \begin{pmatrix} 1 & & & 0 \\ a_{1,1} & 1 & & \\ \vdots & \cdot & \cdot & \\ a_{n,n} & \cdots & a_{1,n} & 1 \end{pmatrix} \quad B_n = \begin{pmatrix} 1 & & & 0 \\ b_{1,1} & 1 & & \\ \vdots & \cdot & \cdot & \\ b_{n,n} & \cdots & b_{1,n} & 1 \end{pmatrix} \quad (2.11)$$

From (2.2), and noting that R_n is a principal minor of R_N , it is easy to show that $A_n R_n$ and $R_n B_n^T$ are upper and lower triangular matrices respectively:

$$A_n R_n = \begin{pmatrix} \varepsilon_0 & * & \cdots & * \\ & \cdot & \cdot & \vdots \\ & & \cdot & * \\ 0 & & & \varepsilon_n \end{pmatrix} \quad \text{and} \quad R_n B_n^T = \begin{pmatrix} \varepsilon_0 & & & 0 \\ * & \cdot & & \\ \vdots & \cdot & \cdot & \\ * & \cdots & * & \varepsilon_N \end{pmatrix} \quad (2.12)$$

This in turn implies that

$$\Lambda_n = A_n R_n B_n^T = \begin{pmatrix} \varepsilon_0 & 0 \\ & \cdot \\ 0 & \varepsilon_n \end{pmatrix} \quad (2.13)$$

Rearranging gives:

$$R_n^{-1} = B_n^T \Lambda_n^{-1} A_n \quad (2.14)$$

The various predictor coefficients generated by the Levinson-Trench-Zohar thus form an Upper triangular, Diagonal, Lower triangular (UDL) decomposition of the inverse matrix R_n^{-1} . This interpretation suggests several interesting results.

For example, we could calculate the vector \underline{x}_n by exploiting the fact that:

$$\underline{x}_n = B_n^T \Lambda_n^{-1} A_n \underline{y}_N \quad (2.15)$$

We could thus compute \underline{x}_n recursively by:

$$\underline{x}_n = \begin{pmatrix} \underline{x}_{n-1} \\ 0 \end{pmatrix} + \frac{\lambda_n}{\varepsilon_n} \underline{b}_n \quad (2.16)$$

where $\lambda_n = \sum_{j=0}^n a_{n-j,n} y_j$

As noted by Zohar, however, this formula for λ_n appears to have no obvious advantages over the formula in (2.9).

The UDL decomposition in (2.14) also implies that:

$$\det(R_n) = \prod_{j=0}^n \varepsilon_j \quad (2.17)$$

This proves that the restriction that $\varepsilon_n \neq 0$ for all n is equivalent to requiring that all principal minors R_n of R_N must be non-singular.

Several interesting formulas for the inverse matrix R_n^{-1} were suggested by Gohberg and Semencul.²² If R_n and R_{n-1} are both invertible, then they can be expressed as sums of products of upper times lower triangular Toeplitz matrices:

$$R_n^{-1} = \frac{1}{\varepsilon_n} \left[\begin{pmatrix} 1 & b_{1,n} & \dots & b_{n,n} \\ & \cdot & \cdot & \cdot \\ & & \cdot & b_{1,n} \\ 0 & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & 0 \\ a_{1,n} & \cdot & & \\ \vdots & \cdot & \cdot & \\ a_{n,n} & \dots & a_{1,n} & 1 \end{pmatrix} \right] \quad (2.18)$$

$$- \begin{pmatrix} 0 & a_{n,n} & \dots & a_{1,n} \\ & \cdot & \cdot & \cdot \\ & & \cdot & a_{n,n} \\ 0 & & & 0 \end{pmatrix} \begin{pmatrix} 0 & & & 0 \\ b_{n,n} & \cdot & & \\ \vdots & \cdot & \cdot & \\ b_{1,n} & \dots & b_{n,n} & 0 \end{pmatrix}$$

$$R_{n-1}^{-1} = \frac{1}{\varepsilon_n} \left[\begin{pmatrix} 1 & b_{1,n} & \dots & b_{n-1,n} \\ & \cdot & \cdot & \cdot \\ & & \cdot & b_{1,n} \\ 0 & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & 0 \\ a_{1,n} & \cdot & & \\ \vdots & \cdot & \cdot & \\ a_{n-1,n} & \dots & a_{1,n} & 1 \end{pmatrix} \right] \quad (2.19)$$

$$- \begin{pmatrix} a_{n,n} & \dots & a_{1,n} \\ & \cdot & \cdot \\ 0 & & a_{n,n} \end{pmatrix} \begin{pmatrix} b_{n,n} & 0 \\ \vdots & \cdot \\ b_{1,n} & \dots & b_{n,n} \end{pmatrix}$$

We will derive similar formulas for the more general case of almost Toeplitz matrices in section 8. The important point is that the inverse matrix R_N^{-1} can be completely specified by the vectors \underline{a}_N , \underline{b}_N and ε_N . Moreover, we can compute $\underline{x}_N = R_N^{-1} \underline{y}_N$ solely from knowledge of \underline{a}_N , \underline{b}_N , ε_N and do not need to actually compute or store the elements of R_N^{-1} . In fact, forming the product $R_N^{-1} \underline{y}_N$ only

involves multiplying triangular Toeplitz matrices with vectors, an operation which is equivalent to convolution of the matrix elements with the vector elements. It is thus possible to calculate $\underline{x}_N = R_N^{-1} \underline{y}_N$ very efficiently by using $2N+1$ point FFT's in $O(N \log N)$ operations. In section 13 we will further exploit this idea to accelerate Levinson recursion by a doubling procedure employing Fast Fourier Transforms for speed.

Finally, another interesting interpretation of the Levinson recursion can be found if we define the Levinson-Szëgo polynomials $A_n(z)$ and $B_n(z)$ by:

$$\begin{aligned} A_n(z) &= 1 + a_{1,n} z^{-1} + \dots + a_{n,n} z^{-n} \\ B_n(z) &= 1 + b_{1,n} z^{-1} + \dots + b_{n,n} z^{-n} \end{aligned} \quad (2.20)$$

Because of the structure of Levinson recursion, the n^{th} order polynomials can be written as a function of the $(n-1)^{\text{th}}$ order polynomials as follows:

$$\begin{aligned} A_n(z) &= A_{n-1}(z) + \xi_n z^{-n} B_{n-1}(z^{-1}) \\ B_n(z) &= B_{n-1}(z) + \nu_n z^{-n} A_{n-1}(z^{-1}) \end{aligned} \quad (2.21)$$

Let us suppose that the entries $r(n)$ of R_N are samples of an infinite sequence

$$\left\{ r(n) \right\}_{n=-\infty}^{\infty} \text{ with a Z-transform } R(z) = \sum_{n=-\infty}^{\infty} r(n) z^{-n} \text{ whose region of convergence}$$

includes the unit circle. Let us define the function $\langle P(z), Q(z) \rangle_{F(z)}$ of the polynomials $P(z)$, $Q(z)$, $F(z)$ by:

$$\langle P(z), Q(z) \rangle_{F(z)} = \oint_{\mathbf{C}} P(z) Q(z^{-1}) F(z) \frac{dz}{z} \quad (2.22)$$

where the circular integral is evaluated on the unit circle \mathbf{C} in a clockwise direction. Then it is easy to show that equations (2.2) imply that the polynomials $z^n A_n(z)$ and $z^m B_m(z)$ are "biorthogonal" under the measure $R(z)$ on the unit circle in the sense that:

$$\langle z^n A_n(z), z^m B_m(z) \rangle_{R(z)} = \varepsilon_n \delta_{n,m} \quad (2.23)$$

where $\delta_{n,m}$ is the Kronecker delta function, $\delta_{n,m} = 1$ if $n = m$, $= 0$ else.

3. Fast Choleski Algorithm for Exactly Toeplitz Matrices

Bareiss⁶ has devised an alternative algorithm for solving the Toeplitz linear equations (1.1) which is based on a clever scheme for accelerating gaussian elimination on Toeplitz matrices. Though it is not obvious from his paper, his algorithm is actually quite similar to the "fast Choleski" algorithms of Morf⁷ and Rissanen.⁸ In this section we present a new derivation of these results, which presents the fast Choleski algorithm as a "mirror image" of the Levinson algorithm.

Start by defining new polynomials $\alpha_n(z)$ and $\beta_n(z)$ in terms of the Levinson-Szëgo polynomials of (2.20):

$$\begin{aligned}\alpha_n(z) &= A_n(z)R(z) = \sum_{j=-\infty}^{\infty} \alpha_{j,n} z^{-j} \\ \beta_n(z) &= B_n(z)R(z^{-1}) = \sum_{j=-\infty}^{\infty} \beta_{j,n} z^{-j}\end{aligned}\tag{3.1}$$

Unlike the Levinson-Szëgo polynomials, $\alpha_n(z)$ and $\beta_n(z)$ have an infinite number of non-zero coefficients. However, using (2.5) it is easy to show that:

$$\begin{aligned}\alpha_{j,n} &= \begin{cases} \varepsilon_n & \text{for } j=0 \\ 0 & \text{for } j=1, \dots, n \\ -\xi_n \varepsilon_n & \text{for } j=n+1 \end{cases} \\ \beta_{j,n} &= \begin{cases} \varepsilon_n & \text{for } j=0 \\ 0 & \text{for } j=1, \dots, n \\ -\nu_n \varepsilon_n & \text{for } j=n+1 \end{cases}\end{aligned}\tag{3.2}$$

The zeroth order coefficients are simply the prediction error ε_n , the next n coefficients are all zero, and the $(n+1)^{th}$ order coefficients are proportional to the reflection coefficients ξ_n and ν_n . Multiplying the Levinson Szëgo polynomial recursion formulas in (2.21) by $R(z)$ then leads to the following recursive algorithm for computing $\alpha_n(z)$ and $\beta_n(z)$:

$$\begin{aligned}\text{Initialization: } \alpha_0(z) &= R(z) \\ \beta_0(z) &= R(z^{-1})\end{aligned}$$

For $n=1, \dots, N$

$$\begin{aligned}
 \varepsilon_{n-1} &= \alpha_{0,n-1} = \beta_{0,n-1} \\
 \xi_n &= -\frac{\alpha_{n,n-1}}{\varepsilon_{n-1}} \\
 \nu_n &= -\frac{\beta_{n,n-1}}{\varepsilon_{n-1}} \\
 \alpha_n(z) &= \alpha_{n-1}(z) + \xi_n z^{-n} \beta_{n-1}(z^{-1}) \\
 \beta_n(z) &= \beta_{n-1}(z) + \nu_n z^{-n} \alpha_{n-1}(z^{-1})
 \end{aligned} \tag{3.3}$$

These recursions form the core of the fast Choleski algorithm.

To understand the role of these polynomials, note first that the polynomials $z^n \alpha_n(z)$ and $z^m \beta_m(z)$ are biorthogonal with respect to the measure $\frac{1}{R(z)}$ in the sense that:

$$\begin{aligned}
 \langle z^n \alpha_n(z), z^m \beta_m(z) \rangle_{\frac{1}{R(z)}} &= \langle z^n A_n(z), z^m B_m(z) \rangle_{R(z)} \\
 &= \varepsilon_n \delta_{n,m}
 \end{aligned} \tag{3.4}$$

Let us form the $(n+1) \times (n+1)$ lower triangular matrices α_n and β_n whose j^{th} columns are coefficients of the polynomials $\alpha_j(z)$ and $\beta_j(z)$:

$$\alpha_n = \begin{bmatrix} \alpha_{0,0} & & & 0 \\ \alpha_{-1,0} & \alpha_{0,1} & & \\ \vdots & & \ddots & \\ \alpha_{-n,0} & \alpha_{-n+1,1} & \cdots & \alpha_{0,n} \end{bmatrix} \quad \beta_n = \begin{bmatrix} \beta_{0,0} & & & 0 \\ \beta_{-1,0} & \beta_{0,1} & & \\ \vdots & \vdots & \ddots & \\ \beta_{-n,0} & \beta_{-n+1,1} & \cdots & \beta_{0,n} \end{bmatrix} \tag{3.5}$$

From (3.1) and (2.11) it is easy to see that:

$$\alpha_n = R_n A_n \quad \text{and} \quad \beta_n = R_n B_n \tag{3.6}$$

Substituting (3.6) into the UDL decomposition formula (2.14) yields:

$$R_n = \beta_n \Lambda_n^{-1} \alpha_n^T \tag{3.7}$$

Thus the fast Choleski algorithm (3.3) can be interpreted as calculating an LDU factorization of the matrix R_N by generating a series of polynomials $\alpha_n(z)$ and $\beta_n(z)$ which are biorthogonal with respect to the measure $\frac{1}{R(z)}$. Note the

symmetrical relationship of this algorithm to the Levinson-Trench-Zohar algorithm, which performs a UDL factorization of the inverse matrix R_N^{-1} by generating a series of polynomials which are biorthogonal with respect to the measure $R(z)$.

This LDU factorization suggests a two stage method for solving our original set of linear equations $R_N \underline{x}_N = \underline{y}_N$. First we compute an intermediate solution $\underline{\lambda}_N$ by solving:

$$\beta_N \underline{\lambda}_N = \underline{y}_N \tag{3.8}$$

then we solve for \underline{x}_N :

$$\alpha_N^T \underline{x}_N = \Lambda_N \underline{\lambda}_N \tag{3.9}$$

Since α_N and β_N are triangular matrices and Λ_N is diagonal, solving these equations requires only about N^2 operations.

The chief difficulty is that since the polynomials $\alpha_n(z)$ and $\beta_n(z)$ have an infinite number of non-zero coefficients, the recursions in (3.3) would require an infinite amount of computation. Fortunately, to solve for \underline{x}_N we only need to compute a finite number of these coefficients. As noted by Bareiss⁶ and by Morf⁷ there are at least two different approaches for solving for the desired coefficients, depending on whether we generate the matrices α_n and β_n column by column or row by row.

3.1. Detailed Columnwise Fast Choleski Algorithm

The columnwise recursion starts with the values of $\alpha_{j,0}$ and $\beta_{j,0}$ for $j = -N, \dots, N$ and on the n^{th} step recursively generates the coefficients $\alpha_{j,n}$ and $\beta_{j,n}$ for $j = -(N-n), \dots, N$. The n^{th} recursion of the algorithm thus generates the coefficients of the n^{th} columns of α_N and β_N . The algorithm is as follows:

Columnwise Fast Choleski

$$\text{Initialization: } \left. \begin{array}{l} \alpha_{j,0} = r(j) \\ \beta_{j,0} = r(-j) \end{array} \right\} \begin{array}{l} \text{for } j = -N, \dots, -1 \\ \text{and } j = 1, \dots, N \end{array}$$

$$\varepsilon_0 = r(0)$$

For $n = 1, \dots, N$

$$\xi_n = -\frac{\alpha_{n,n-1}}{\varepsilon_{n-1}} \tag{3.10}$$

$$\nu_n = -\frac{\beta_{n,n-1}}{\varepsilon_{n-1}}$$

$$\left. \begin{array}{l} \alpha_{j,n} = \alpha_{j,n-1} + \xi_n \beta_{n-j,n-1} \\ \beta_{j,n} = \beta_{j,n-1} + \nu_n \alpha_{n-j,n-1} \end{array} \right\} \begin{array}{l} \text{for } j = -(N-n), \dots, -1 \\ \text{and } j = n+1, \dots, N \end{array}$$

$$\varepsilon_n = \varepsilon_{n-1}(1 - \xi_n \nu_n)$$

Note that on the n^{th} pass we can store the reflection coefficients ξ_n and ν_n in the locations previously used by $\alpha_{n,n-1}$ and $\beta_{n,n-1}$. It is convenient to organize storage so that after the n^{th} pass we will have saved:

$$\alpha_{-N,0} \cdots \alpha_{n-N,n} \alpha_{n-N+1,n} \cdots \alpha_{-1,n} \quad \left\| \quad \xi_1 \cdots \xi_n \alpha_{n+1,n} \cdots \alpha_{N,n} \tag{3.11}$$

$$\beta_{-N,0} \cdots \beta_{n-N,n} \beta_{n-N+1,n} \cdots \beta_{-1,n} \quad \left\| \quad \nu_1 \cdots \nu_n \beta_{n+1,n} \cdots \beta_{N,n}$$

Computation can thus be done in place. About $4N$ storage locations and about $2N^2$ operations will be needed to compute α_N and β_N , which is identical to the requirements of the Levinson-Trench algorithm for computing A_n and B_n .

The solutions $\underline{\lambda}_N$ and \underline{x}_N to the linear equations in (3.8) and (3.9) can also be computed by using the coefficients of α_n and β_n in columnwise order. Solving for $\underline{\lambda}_N$ requires a forward substitution step; solving next for \underline{x}_N requires a backward substitution step.

Forward Substitution:

$$\begin{aligned} &\text{Initialization: } y_{j,0} \leftarrow y_j \quad \text{for } j=0, \dots, N \\ &\text{For } n=0, \dots, N \end{aligned} \tag{3.12}$$

$$\lambda_n = \frac{y_{n,n}}{\varepsilon_n}$$

$$y_{j,n+1} = y_{j,n} - \lambda_n \beta_{n-j,n} \quad \text{for } j=n+1, \dots, N$$

Back Substitution

$$\text{For } n=N, \dots, 0$$

$$x_n = \lambda_n - \frac{1}{\varepsilon_n} \sum_{j=n+1}^N \alpha_{n-j,n} x_j \tag{3.13}$$

Since the computation can be done in place, putting λ_n and then x_n in the same location used for y_n , only $N+1$ extra storage locations are needed. Total computation for generating $\alpha_{j,n}$ and $\beta_{j,n}$ and then solving for \underline{x}_N is thus only about $3N^2$ operations, which is identical to the Levinson-Trench-Zohar algorithm.

The forward substitution phase for calculating the intermediate solution $\underline{\lambda}_N$ uses the coefficients $\beta_{j,n}$ in ascending order of n , which is the same order in which they are generated. It is thus easily integrated into our columnwise fast Choleski algorithm (3.10) for calculating $\alpha_{j,n}$ and $\beta_{j,n}$. Unfortunately, the backward substitution phase for calculating \underline{x}_N from $\underline{\lambda}_N$ requires the coefficients $\alpha_{j,n}$ in descending order of n , which is the reverse of the order in which they are generated. One approach would be to save the values of $\frac{\alpha_{j,n}}{\varepsilon_n}$ as they are generated. This, however, would require an extra $\frac{N^2}{2}$ storage locations, which is an order of magnitude more than Levinson recursion requires. A more storage efficient approach is to use a backward recursion to regenerate these polynomial coefficients in descending order of n for use in calculating x_n . To do this, we will need to save the reflection coefficients ξ_n and ν_n and the coefficients $\alpha_{n-N,n}$ calculated during the forward phase (note that these are

exactly the values saved in the scheme illustrated in (3.11)). Given these values, in the backward phase we can then reverse the polynomial recursion in (3.3):

$$\begin{aligned}\alpha_{n-1}(z) &= \frac{1}{1-\xi_n \nu_n} \left[\alpha_n(z) - \xi_n z^{-n} \beta_n(z^{-1}) \right] \\ \beta_{n-1}(z) &= \frac{1}{1-\xi_n \nu_n} \left[\beta_n(z) - \nu_n z^{-n} \alpha_n(z^{-1}) \right]\end{aligned}\quad (3.14)$$

Normalization by $(1-\xi_n \nu_n)^{-1}$ can be avoided by calculating the scaled polynomials $\tilde{\alpha}_n(z) = \frac{\varepsilon_N}{\varepsilon_n} \alpha_n(z)$ and $\tilde{\beta}_n(z) = \frac{\varepsilon_N}{\varepsilon_n} \beta_n(z)$ instead and compensating accordingly. Computational effort can also be reduced by exploiting the fact that we only need to calculate the matrix α_N . The complete backward phase will then be:

Backward Phase (Minimal Storage)

Saved from forward phase: $\alpha_{n-N,n}, \nu_n, \xi_n, \lambda_n$ for $n=0, \dots, N$

For $n=N, \dots, 0$

$$x_n = \lambda_n - \frac{1}{\varepsilon_N} \sum_{j=n+1}^N \tilde{\alpha}_{n-j,n} x_j \quad (3.15)$$

If $n=0$ END

$$\left. \begin{aligned}\tilde{\alpha}_{j,n-1} &= \tilde{\alpha}_{j,n} - \xi_n \tilde{\beta}_{n-j,n} \\ \tilde{\beta}_{n-j,n-1} &= \tilde{\beta}_{n-j,n} - \nu_n \tilde{\alpha}_{j,n}\end{aligned}\right\} \text{ for } j=-(N-n), \dots, -1$$

$$\tilde{\alpha}_{n-1-N,n-1} = \frac{\varepsilon_N}{\varepsilon_n} \alpha_{n-1-N,n-1}$$

$$\tilde{\beta}_{n,n-1} = -\nu_n \varepsilon_N$$

$$\varepsilon_{n-1} = \frac{\varepsilon_n}{(1-\xi_n \nu_n)}$$

Total storage required with this approach will now only be about $5N$ locations, which is the same as the Levinson-Trench-Zohar (LTZ) algorithm.

Unfortunately, about $4N^2$ operations are needed, compared to only $3N^2$ operations for the LTZ algorithm.

Note that if the matrix R_N were symmetric so that $R(z)=R(z^{-1})$, then $\xi_n=\nu_n$ and $\alpha_n(z)=\beta_n(z)$ for all n . This symmetry could then be used to reduce the requirements for calculating \underline{x}_N to $3N^2$ operations and $3N$ locations.

3.2. Detailed Rowwise Fast Choleski Algorithm

A somewhat different Choleski algorithm results when we solve for the coefficients of the α_n and β_n matrices in rowwise order. The n^{th} pass of this algorithm will generate the coefficients of the n^{th} row $\alpha_{j-n,j}$, $\alpha_{n,j}$, $\beta_{j-n,j}$, and $\beta_{n,j}$ for $j=0, \dots, n$.

Rowwise Fast Choleski

Initialization: $\varepsilon_0 = \tau(0)$

For $n=1, \dots, N$

$$\alpha_{n,0} = \beta_{-n,0} = \tau(n)$$

$$\alpha_{-n,0} = \beta_{n,0} = \tau(-n)$$

$$\left. \begin{aligned} \alpha_{j-n,j} &= \alpha_{j-n,j-1} + \xi_j \beta_{n,j-1} \\ \beta_{n,j} &= \beta_{n,j-1} + \nu_j \alpha_{j-n,j-1} \\ \alpha_{n,j} &= \alpha_{n,j-1} + \xi_j \beta_{j-n,j-1} \\ \beta_{j-n,j} &= \beta_{j-n,j-1} + \nu_j \alpha_{n,j-1} \end{aligned} \right\} \text{ for } j=1, \dots, n-1$$

$$\xi_n = - \frac{\alpha_{n,n-1}}{\varepsilon_{n-1}} \tag{3.16}$$

$$\nu_n = - \frac{\beta_{n,n-1}}{\varepsilon_{n-1}}$$

$$\varepsilon_n = \varepsilon_{n-1}(1 - \xi_n \nu_n)$$

The coefficients $\beta_{n,j}$ and $\alpha_{n,j}$ do not need to be stored since they are not used in the $(n+1)^{th}$ pass, nor are they needed for solving for \underline{x}_n . Total storage required for saving the coefficients $\alpha_{j-n,j}$, $\beta_{j-n,j}$ and the reflection coefficients ξ_j and ν_j is therefore about $4N$ locations, and all computation can be done in place. About $2N^2$ operations will be needed to compute the matrices α_N and β_N .

It is also possible to solve for the solution $\underline{\lambda}_n$ and \underline{x}_n to the linear equations in (1.1) by using the coefficients of α_N and β_N in rowwise order. Once again we have a forward substitution phase for generating λ_n and a backward substitution phase for generating x_n :

Forward Substitution

For $n=0, \dots, N$

$$\lambda_n = \frac{1}{\varepsilon_n} \left[y_n - \sum_{j=0}^{n-1} \beta_{j-n,j} \lambda_j \right] \quad (3.17)$$

Backward Substitution

Init: $\lambda_{j,N} = \lambda_j \varepsilon_j$ for $j=0, \dots, N$

For $n=N, \dots, 0$

$$x_n = \frac{\lambda_{n,n}}{\varepsilon_n} \quad (3.18)$$

$$\lambda_{j,n-1} = \lambda_{j,n} - x_n \alpha_{j-n,j} \quad \text{for } j=0, \dots, n-1$$

Computation can be done in place, so that the same $(N+1)$ long array can be used for storing y_n , λ_n , and x_n . Unfortunately, the rowwise algorithm has the same difficulties as the columnwise algorithm with the backward substitution step. The forward substitution phase for calculating λ_n uses the coefficients $\beta_{j-n,j}$ in ascending order of $n=0, \dots, N$, and is thus easily integrated into the rowwise fast Choleski algorithm (3.16). The backward substitution phase, however, requires the values of $\alpha_{j-n,j}$ in descending order of $n=N, \dots, 0$, which is the reverse of the order in which they are generated. These coefficients could be stored as they are computed, but this would require $\frac{N^2}{2}$ extra storage, which is an order of magnitude more than that used by the LTZ algorithm. Alternatively, we could recalculate the values of $\alpha_{j-n,j}$ in descending order of n by exploiting the backward polynomial recursion (3.14). We will start with the values of $\alpha_{n-N,n}$, ξ_n , ν_n , λ_n , and ε_N as calculated on the forward pass. Rescaling as in the columnwise algorithm to reduce the operations count, and eliminating all unnecessary computation, then yields the following backward substitution phase:

Backward Substitution (Minimal Storage)

Initialization: $\lambda_{n,N} \leftarrow \lambda_n$ for $n=0, \dots, N$

For $j=N-1, \dots, 0$

$$\varepsilon_j = \frac{\varepsilon_{j+1}}{(1 - \xi_{j+1} \nu_{j+1})}$$

$$\tilde{\alpha}_{j-N,j} \leftarrow \frac{\varepsilon_N}{\varepsilon_j} \alpha_{j-N,j}$$

For $n=N, \dots, 0$

(3.19)

$$x_n = \frac{\lambda_{n,n}}{\varepsilon_N}$$

If $n=0$ END

$$\lambda_{j,n-1} = \lambda_{j,n} - x_n \tilde{\alpha}_{j-n,j} \quad \text{for } j=0, \dots, n-1$$

$$\tilde{\beta}_{n,n-1} = -\nu_n \varepsilon_N$$

$$\left. \begin{aligned} \tilde{\alpha}_{j-n,j-1} &= \tilde{\alpha}_{j-n,j} - \xi_j \tilde{\beta}_{n,j} \\ \tilde{\beta}_{n,j-1} &= \tilde{\beta}_{n,j} - \nu_j \tilde{\alpha}_{j-n,j} \end{aligned} \right\} \quad \text{for } j=n-1, \dots, 1$$

Once again, computation can be done in place. Also note that, as in the forward phase, it is not necessary to save the values of $\tilde{\beta}_{n,j}$ since they are not used in computing x_N nor are they used in the $(n-1)^{th}$ pass. Total storage and computation requirements for the rowwise fast Choleski algorithm are thus $5N$ locations and $4N^2$ operations, which is the same as the columnwise algorithm. If R_N is symmetric, then $\xi_n = \nu_n$ and $\alpha_n(z) = \beta_n(z)$, and the storage and computation requirements reduce to $3N$ locations and $3N^2$ operations.

The rowwise algorithm thus has the same storage and computation requirements as the columnwise algorithm. Its chief advantage is in certain applications where the length of the available data N may increase as new data arrives. The rowwise algorithm easily adapts to this situation simply by resuming the forward iteration where it had left off. In general, both the fast Choleski algorithms are slower than the LTZ algorithm, unless $\frac{1}{2}N^2$ extra storage loca-

tions are used to save the values of $\alpha_{-j,n}$. As we will see in section 12, however, the fast Choleski algorithms are far superior to the LTZ algorithm when the Toeplitz matrix is band-diagonal.

4. Almost Toeplitz Matrices

Toeplitz matrices arise in applications in which the underlying system is characterized by some form of "shift invariance" (or "stationarity" or "homogeneity"). Many shift invariant systems, however, may lead to sets of linear equations which are closely related to Toeplitz matrices, but are not exactly Toeplitz. For example, we may have the (non-Toeplitz) inverse of a Toeplitz matrix, or the (non-Toeplitz) covariance matrix of a stationary process with an initial transient. Friedlander, Morf, Kailath, and Ljung⁵ and Kailath, Kung, and Morf¹⁷ have shown that in fact we can characterize such "almost-Toeplitz" matrices by a "distance from Toeplitz" κ , such that the amount of computation required to invert the matrix is $O(\kappa N^2)$. Kailath's idea was to consider the class of matrices R_N which could be represented as the sum of κ_+ products of lower \times upper triangular (block) Toeplitz matrices:

$$R_N = \sum_{i=1}^{\kappa_+} L(\underline{x}_i)U(\underline{y}_i^T) \quad (4.1)$$

where $L(\underline{x})$ (and $U(\underline{y}^T)$) are the (block) lower triangular (and upper triangular) Toeplitz matrices whose first column is \underline{x} (and whose first row is \underline{y}^T). in the following discussion, we will also use the notation $L(\underline{x}^T)$ (and $U(\underline{y})$) to refer to the (block) lower (and upper) triangular Toeplitz matrices whose last row is \underline{x}^T (last column is \underline{y} .) One reason for choosing a representation for R_N like (4.1) is that if we form the shifted difference $\lrcorner R_N$ defined by:

$$\lrcorner R_N = R_N - \begin{pmatrix} 0 & \cdots & \cdots & 0 \\ \vdots & R_{0,0} & \cdots & R_{0,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & R_{N-1,0} & \cdots & R_{N-1,N-1} \end{pmatrix} \quad (4.2)$$

then this matrix $\lfloor R_N$ can be factored in terms of the vector $\underline{x}_i, \underline{y}_i$:

Lemma 1 $R_N = \sum_{i=1}^{\kappa_+} L(\underline{x}_i)U(\underline{y}_i^T)$ if and only if $\lfloor R_N = \sum_{i=1}^{\kappa_+} \underline{x}_i \underline{y}_i^T$

This can be proven by direct calculation. In particular, this implies that κ_+ is the minimum number of terms in a lower \times upper representation for R_N like (4.1) if and only if $\kappa_+ = \text{rank}(\lfloor R_N)$. (If the entries of R_N are $p \times p$ blocks instead of scalars, then κ_+ will be the smallest integer greater than $\frac{1}{p} \text{rank}(\lfloor R_N)$.) Hence we call κ_+ the (+) displacement rank of R_N . In general, (block) Toeplitz matrices can be written as the sum of $\kappa_+ = 2$ products of lower \times upper triangular (block) Toeplitz matrices. For example, we could choose

$$\underline{x}_1 = \begin{bmatrix} \tau(0) \\ \vdots \\ \tau(N) \end{bmatrix} \quad \underline{x}_2 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.3}$$

$$\underline{y}_1^T = [1 \ 0 \ \dots \ 0] \quad \underline{y}_2^T = [0 \ \tau(-1) \ \dots \ \tau(-N)]$$

This representation is not unique; for example, we could also have chosen:

$$\underline{x}_1 = \begin{bmatrix} \tau(0) \\ \tau(1) \\ \vdots \\ \tau(N) \end{bmatrix} \tau(0)^{-\frac{1}{2}} \quad \underline{x}_2 = \begin{bmatrix} 0 \\ \tau(1) \\ \vdots \\ \tau(N) \end{bmatrix} \tau(0)^{-\frac{1}{2}} \tag{4.4}$$

$$\underline{y}_1 = \tau(0)^{-\frac{1}{2}} [\tau(0) \ \tau(-1) \ \dots \ \tau(-N)] \quad \underline{y}_2 = \tau(0)^{-\frac{1}{2}} [0 \ \tau(-1) \ \dots \ \tau(-N)]$$

This latter form is most convenient when R_N is symmetric, since then $\underline{x}_i = \underline{y}_i$.

Now the form of R_N in (4.1) is not the only one suitable for our needs. We might also consider matrices of the form of sums of products of upper \times lower triangular (block) Toeplitz matrices:

$$R_N = \sum_{i=1}^{\kappa_-} U(\underline{x}_i)L(\underline{y}_i^T) \tag{4.5}$$

The interesting feature of this representation is that if we form the shifted

difference ΓR_N defined by:

$$\Gamma R_N = R_N - \begin{pmatrix} R_{1,1} & \cdots & R_{1,N} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ R_{N,1} & \cdots & R_{N,N} & \vdots \\ 0 & \cdots & \cdots & 0 \end{pmatrix} \quad (4.6)$$

then this matrix ΓR_N can be factored in terms of the vectors $\underline{x}_i, \underline{y}_i$:

Lemma 2 $R_N = \sum_{i=1}^{\kappa_-} U(\underline{x}_i)L(\underline{y}_i^T)$ if and only if $\Gamma R_N = \sum_{i=1}^{\kappa_-} \underline{x}_i \underline{y}_i^T$

Once again, $\kappa_-(R_N)$ will be the minimum number of terms in this upper \times lower decomposition of R_N if and only if $\kappa_- = \text{rank}(\Gamma R_N)$. (This must be modified appropriately if the entries of R_N are themselves matrices.) We call $\kappa_-(R_N)$ the (-) displacement rank of R_N .

The lower \times upper and the upper \times lower representations are equivalent in the sense that if we can represent R_N in one form, we can also represent R_N in the other form with approximately the same number of terms. To do this, note that:

$$\begin{aligned} L(\underline{x})U(\underline{y}^T) &= \begin{pmatrix} x_0 & 0 \\ \vdots & \vdots \\ x_N & \cdots & x_0 \end{pmatrix} \begin{pmatrix} y_0^T & \cdots & y_N^T \\ \vdots & \ddots & \vdots \\ 0 & & y_0^T \end{pmatrix} \\ &= \begin{pmatrix} x_N & \cdots & x_0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & & x_N & \cdots & x_0 \end{pmatrix} \begin{pmatrix} y_N^T & 0 \\ \vdots & \vdots \\ y_0^T & y_N^T \\ \vdots & \vdots \\ 0 & y_0^T \end{pmatrix} - \begin{pmatrix} 0 & x_N & \cdots & x_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & \end{pmatrix} \begin{pmatrix} 0 & 0 \\ y_N^T & \vdots \\ \vdots & \vdots \\ y_1^T & y_N^T & 0 \end{pmatrix} \\ &= \mathbf{T} - \tilde{\mathbf{U}} \begin{pmatrix} x_1 \\ \vdots \\ x_N \\ 0 \end{pmatrix} \tilde{\mathbf{L}}(y_1^T \cdots y_N^T 0) \end{aligned} \quad (4.7)$$

where \mathbf{T} is a Toeplitz matrix with entries equal to the convolution of the x_i and

y_i^T sequences. We can thus convert the lower \times upper representation (4.1) into the form:

$$R_N = \sum_{i=1}^{\kappa_+} L(\underline{x}_i) U(\underline{y}_i^T) = T + \sum_{i=1}^{\kappa_+} U \begin{pmatrix} x_1^i \\ \vdots \\ x_N^i \\ 0 \end{pmatrix} L \begin{pmatrix} y_1^{iT} & \dots & y_N^{iT} & 0 \end{pmatrix} \quad (4.8)$$

Since a Toeplitz matrix T can be represented by a sum of no more than 2 products of upper \times lower triangular Toeplitz matrices, we have shown that we can construct an upper \times lower representation for R_N with $\kappa_- \leq \kappa_+ + 2$. Applying a similar argument in reverse proves that:

Lemma 3 $\left| \kappa_+(R_N) - \kappa_-(R_N) \right| \leq 2$

Thus the minimum number of terms in the lower \times upper and upper \times lower representations for R_N will differ by no more than 2.

The most interesting result for our purposes, however, is the following theorem proved by Kailath, Kung and Morf:¹⁷

Theorem 1: If R_N is invertible, then its (\pm) displacement rank is equal to the (\mp) displacement rank of its inverse:

$$\kappa_+(R_N) = \kappa_-(R_N^{-1}) \quad \kappa_-(R_N) = \kappa_+(R_N^{-1}) \quad (4.9)$$

Proof: The proof we give is due to Delosme and Morf.²³ Let Z be the "lower shift" matrix:

$$Z = \begin{pmatrix} 0 & & 0 \\ 1 & & \\ & \ddots & \\ 0 & 1 & 0 \end{pmatrix} \quad (4.10)$$

so that

$$\begin{aligned} \lfloor R &= R - ZRZ^T \\ \lceil R &= R - Z^T R Z \end{aligned} \quad (4.11)$$

Then:

$$\begin{aligned}
 \text{rank} \begin{pmatrix} R_N & Z \\ Z^T & R_N^{-1} \end{pmatrix} &= \text{rank} \left\{ \begin{pmatrix} I & 0 \\ Z^T R_N^{-1} & I \end{pmatrix} \begin{pmatrix} R_N & 0 \\ 0 & R_N^{-1} - Z^T R_N^{-1} Z \end{pmatrix} \begin{pmatrix} I & R_N^{-1} Z \\ 0 & I \end{pmatrix} \right\} \\
 &= \text{rank} \begin{pmatrix} R_N & 0 \\ 0 & \Gamma R_N^{-1} \end{pmatrix} \\
 &= \text{rank} \left(R_N \right) + \text{rank} \left(\Gamma R_N^{-1} \right) \tag{4.12}
 \end{aligned}$$

where we use the fact that the rank of a matrix is not changed by multiplication by a full rank matrix. Similarly, by performing a UDL rather than an LDU decomposition on the matrix on the left of (4.12), we can show that:

$$\text{rank} \begin{pmatrix} R_N & Z \\ Z^T & R_N^{-1} \end{pmatrix} = \text{rank} \left(R_N^{-1} \right) + \text{rank} \left(\lrcorner R_N \right) \tag{4.13}$$

Since $\text{rank}(R_N) = \text{rank}(R_N^{-1}) = N+1$, we are left with:

$$\kappa_+(R_N) = \text{rank}(\lrcorner R_N) = \text{rank}(\Gamma R_N^{-1}) = \kappa_-(R_N^{-1}) \tag{4.14}$$

The other equality can be proven in much the same way. ■

Note that the Gohberg-Semencul formula (2.18) expresses theorem 1 for exactly Toeplitz matrices. Starting with a Toeplitz matrix R_N with $\kappa_+=2$, this formula expresses R_N^{-1} as a sum of 2 products of upper \times lower triangular Toeplitz matrices. Moreover, the elements of this representation can be calculated by Levinson recursion. In the following sections we will develop an analogous constructive procedure for starting with any κ term (not necessarily minimal) lower \times upper representation of R_N , and calculating the corresponding upper \times lower representation of R_N^{-1} in only $O(\kappa N^2)$ operations. Both Levinson and fast Choleski algorithms will be developed. LDU decomposition formulas for R_N and R_N^{-1} will also be given, as well as techniques for calculating \underline{x}_N without explicitly

calculating R_N^{-1} . We will assume throughout that the lower \times upper factorization of R_N is given *a priori* by the structure of the applications. Finding such a representation otherwise would be a difficult numerical problem, especially if an upper bound on $\kappa_+(R_N)$ were not known.

5. Almost Toeplitz Matrices - Levinson-Style Algorithm

Let us assume that we are given a factorization of R_N as a sum of κ products of lower \times upper triangular Toeplitz matrices:

$$R_N = \sum_{i=1}^{\kappa} L(\underline{c}_N^i) U(\underline{d}_N^{iT}) \quad (5.1)$$

$$\text{where: } \underline{c}_n^i = \begin{bmatrix} c_0^i \\ \vdots \\ c_n^i \end{bmatrix} \quad \underline{d}_n^i = \begin{bmatrix} d_0^i \\ \vdots \\ d_n^i \end{bmatrix}$$

To simplify the presentation, we will assume that c_n^i and d_n^i are scalars (note that i is a superscript and not an exponent.) Extending the algorithm to the case where c_n^i and d_n^i are matrices is simple, and only requires somewhat greater care in the order in which we multiply and transpose the various quantities. We will implicitly assume that the (+) displacement rank κ is small relative to N . Let R_n be the $(n+1) \times (n+1)$ upper left principal minor of R_N ; we will assume that R_N is strongly non-singular so that all the minors R_n are non-singular. By definition, R_{n-1} is the upper left principal minor of R_n :

$$R_n = \begin{bmatrix} R_{n-1} & \star \\ \star & \dots \\ \star & \dots & \star \end{bmatrix} \quad (5.2)$$

Also, by lemma 1:

$$R_n = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & & \\ 0 & R_{n-1} & \end{bmatrix} + \sum_{i=1}^{\kappa} \underline{c}_n^i \underline{d}_n^{iT} \quad (5.3)$$

In the same way that (2.1) is the key partitioning relationship which defines Levinson recursion for exactly Toeplitz matrices, these two partitioning formu-

las are the key to the almost Toeplitz algorithms. As in Levinson recursion, we will use the solution to a set of auxiliary equations to help solve our given problem. In our case, the equations we will need to solve have the following form:

$$\begin{aligned} R_n \underline{b}_n &= \begin{pmatrix} 0_n \\ \varepsilon_n \end{pmatrix} \\ R_n \underline{f}_n^i &= \underline{c}_n^i \quad \text{for } i=1, \dots, \kappa \\ R_n \underline{x}_n &= \underline{y}_n \end{aligned} \tag{5.4}$$

where:

$$\underline{b}_n = \begin{pmatrix} b_{n,n} \\ \vdots \\ b_{1,n} \\ 1 \end{pmatrix} \quad \underline{f}_n^i = \begin{pmatrix} f_{0,n}^i \\ \vdots \\ f_{n,n}^i \end{pmatrix} \quad \underline{x}_n = \begin{pmatrix} x_{0,n} \\ \vdots \\ x_{n,n} \end{pmatrix} \tag{5.5}$$

The solution for $n=0$ can be found by inspection:

$$\begin{aligned} \underline{b}_0 &= \begin{pmatrix} 1 \end{pmatrix} \\ \varepsilon_0 &= R_{0,0} = \sum_{i=1}^{\kappa} c_{0,0}^i d_0^i \\ \underline{f}_0^i &= \begin{pmatrix} c_0^i \\ \varepsilon_0 \end{pmatrix} \quad \text{for } i=1, \dots, \kappa \\ \underline{x}_0 &= \begin{pmatrix} y_0 \\ \varepsilon_0 \end{pmatrix} \end{aligned} \tag{5.6}$$

where R_{ij} is the $(i,j)^{th}$ element of R_n . The solution at the n^{th} stage for $n > 0$ can now be recursively expressed in terms of the solutions at the $(n-1)^{th}$ stage.

To see this, note that by using (5.3):

$$\begin{aligned} R_n \begin{pmatrix} 0 \\ \underline{b}_{n-1} \end{pmatrix} &= \begin{pmatrix} 0 \\ \varepsilon_{n-1} \end{pmatrix} - \sum_{i=1}^{\kappa} \nu_n^i \underline{c}_n^i \\ \text{where: } \nu_n^i &= -\underline{d}_n^{iT} \begin{pmatrix} 0 \\ \underline{b}_{n-1} \end{pmatrix} \end{aligned} \tag{5.7}$$

Also, by using (5.2):

$$\begin{aligned} R_n \begin{pmatrix} \underline{f}_{n-1}^i \\ 0 \end{pmatrix} &= \underline{c}_n^i - \begin{pmatrix} 0 \\ \xi_n^i \end{pmatrix} \quad \text{for } i=1, \dots, \kappa \\ \text{where: } \xi_n^i &= c_n^i - \sum_{j=0}^{n-1} R_{n,j} f_{j,n-1}^i \end{aligned} \tag{5.8}$$

It is thus possible to find a solution \underline{b}_n to (5.4) by combining appropriate multiples of \underline{f}_{n-1}^i with \underline{b}_{n-1} in order to cancel out the terms on the right hand side of (5.7):

$$\underline{b}_n = \begin{pmatrix} 0 \\ \underline{b}_{n-1} \end{pmatrix} + \sum_{i=1}^{\kappa} \nu_n^i \begin{pmatrix} \underline{f}_{n-1}^i \\ 0 \end{pmatrix} \quad (5.9)$$

Then it is easy to show that this \underline{b}_n satisfies (5.4) with:

$$\varepsilon_n = \varepsilon_{n-1} - \sum_{i=1}^{\kappa} \nu_n^i \xi_n^i \quad (5.10)$$

The solution for \underline{f}_n^i can then be constructed by combining appropriate multiples of \underline{b}_n and \underline{f}_{n-1}^i in order to cancel the extra term on the right hand side of (5.8):

$$\underline{f}_n^i = \begin{pmatrix} \underline{f}_{n-1}^i \\ 0 \end{pmatrix} + \frac{\xi_n^i}{\varepsilon_n} \underline{b}_n \quad \text{for } i=1, \dots, \kappa \quad (5.11)$$

Finally, to solve for \underline{x}_n , let us suppose we know the solution for \underline{x}_{n-1} . Then:

$$R_n \begin{pmatrix} \underline{x}_{n-1} \\ 0 \end{pmatrix} = \underline{y}_n - \begin{pmatrix} 0 \\ \lambda_n \end{pmatrix} \quad (5.12)$$

$$\text{where: } \lambda_n = y_n - \sum_{j=0}^{n-1} R_{n,j} x_{j,n-1}$$

The solution for \underline{x}_n can then be constructed in much the same manner used in constructing \underline{f}_n^i :

$$\underline{x}_n = \begin{pmatrix} \underline{x}_{n-1} \\ 0 \end{pmatrix} + \frac{\lambda_n}{\varepsilon_n} \underline{b}_n \quad (5.13)$$

By applying equations (5.9), (5.11) and (5.13) recursively for $n=1, \dots, N$, we will be able to construct \underline{x}_N . The coefficients ξ_n^i and ν_n^i can be viewed as generalized reflection coefficients, \underline{b}_n and \underline{f}_n^i can be viewed as generalized predictors, and ε_n is a prediction error. Note that the algorithm requires that $\varepsilon_n \neq 0$ at every step; as in the Levinson algorithm, we will see that this is equivalent to requiring that R_N be strongly non-singular.

The basic form of the algorithm above was given by Dickinson.¹⁹ It can be simplified somewhat further so that only $\kappa-1$ of the vectors \underline{f}_n^i need to be calculated. The key is to note that:

$$\sum_{i=1}^{\kappa} d_0^i \underline{f}_n^i = \begin{pmatrix} 1 \\ \underline{0}_n \end{pmatrix} \quad (5.14)$$

To prove this, it is only necessary to multiply both sides by R_n and note that the first column of R_n is $\sum_{i=1}^{\kappa} d_0^i \underline{c}_n^i$. We will assume that $c_0^1 d_0^1 \neq 0$ (if this is not true, simply renumber the vectors; at least one of these pairs must satisfy this, since by the assumption of strong non-singularity, $\varepsilon_0 = \sum_{i=1}^{\kappa} c_0^i d_0^i \neq 0$.) From (5.14):

$$\underline{f}_n^1 = \frac{1}{d_0^1} \left[\begin{pmatrix} 1 \\ \underline{0}_n \end{pmatrix} - \sum_{i=2}^{\kappa} d_0^i \underline{f}_n^i \right] \quad (5.15)$$

Substituting this into our update formula for \underline{b}_n gives:

$$\underline{b}_n = \begin{pmatrix} \nu_n^1 / d_0^1 \\ \underline{b}_{n-1} \end{pmatrix} + \sum_{i=2}^{\kappa} \gamma_n^i \begin{pmatrix} \underline{f}_{n-1}^i \\ 0 \end{pmatrix} \quad (5.16)$$

$$\text{where: } \gamma_n^i = \nu_n^i - \frac{d_0^i}{d_0^1} \nu_n^1 \quad \text{for } i=2, \dots, \kappa$$

and:

$$\varepsilon_n = \varepsilon_{n-1} - \sum_{i=2}^{\kappa} \gamma_n^i \xi_n^i \quad (5.17)$$

Since \underline{f}_n^1 and ξ_n^1 are no longer needed for calculating \underline{b}_n or ε_n , they need not be calculated at all. The complete Levinson-style algorithm thus takes the form:

Initialization: Get $\underline{b}_0, \underline{f}_0^2, \dots, \underline{f}_0^\kappa, \underline{x}_0, \varepsilon_0$ from (5.6)

For $n=1, \dots, N$

- a) Calculate reflection coefficients $\xi_n^2, \dots, \xi_n^\kappa$ and $\nu_n^1, \dots, \nu_n^\kappa$ from (5.8) and (5.7)
- b) Update \underline{b}_n and ε_n using (5.16) and (5.17)
- c) Update $\underline{f}_n^2, \dots, \underline{f}_n^\kappa$ using (5.11)
- d) Calculate λ_n from (5.12) and \underline{x}_n from (5.13)

Total computation is approximately $\frac{1}{2}(4\kappa-1)N^2$ operations and about $3N\kappa$ storage locations are needed for $\underline{c}_N^i, \underline{d}_N^i, \underline{b}_N, \underline{f}_N^i$ and \underline{y}_N . One difficulty with this algorithm is that it requires the values $R_{n,j}$ for $j < n$. If these are not stored, they they may have to be calculated recursively during the recursion above:

$$R_{n,j} = \begin{cases} \sum_{i=1}^{\kappa} c_n^i d_0^i & \text{for } j=0 \\ R_{n-1,j-1} + \sum_{i=1}^{\kappa} c_n^i d_j^i & \text{for } 0 < j < n \end{cases} \quad (5.18)$$

This would require an additional $\frac{1}{2}\kappa N^2$ operations and N storage locations. If R_N is symmetric, then the computation can be simplified slightly because it will be possible to calculate the \mathcal{V}_n^i coefficients directly from the ξ_n^i coefficients or vice versa. The details of this, however, we leave to a succeeding section. Finally, if c_n^i and d_n^i were $\tau \times \tau$ matrices instead of scalars, the block almost Toeplitz version of the algorithm above would require only τ^3 as much computation and τ^2 as much storage.

As a simple example of this technique, consider the case when R_n is an exactly Toeplitz matrix and we express R_N in the form suggested in (4.3). Then it is easy to see that \underline{b}_n is the backward predictor of the Levinson algorithm, $\varepsilon_n \underline{f}_n^2$ is the forward predictor, ε_n is the prediction error, $\nu_n^1=0$, and ξ_n^2 and $\frac{\mathcal{V}_n^2}{\varepsilon_{n-1}}$ are the forward and backward reflection coefficients respectively (the "2" is a superscript, not an exponent).

6. UDL Decomposition of R_N^{-1} by Extended Levinson-Style Algorithm

The Levinson-style algorithm we have presented does not calculate a full UDL decomposition of R_N^{-1} . To accomplish this, we will need to repeat our almost-Toeplitz algorithm except with R_N replaced by R_N^T . At each stage n we will need to solve for the $(n+1)$ long vectors \underline{a}_n and \underline{e}_n^i which satisfy:

$$\begin{aligned} R_n^T \underline{a}_n &= \begin{pmatrix} 0_n \\ \varepsilon_n \end{pmatrix} & \text{where: } \underline{a}_n &= (a_{n,n} \cdots a_{1,n} \ 1)^T \\ R_n^T \underline{e}_n^i &= \underline{d}_n^i & \text{where: } \underline{e}_n^i &= (e_{0,n}^i \cdots e_{n,n}^i)^T \end{aligned} \quad (6.1)$$

By considering the quadratic form $(\underline{a}_n^T R_n) \underline{b}_n = \underline{a}_n^T (R_n \underline{b}_n) = \varepsilon_n$ it is easy to show that the prediction errors ε_n of the original and the transposed almost-Toeplitz problems are identical. As in the previous section, we can show that the solution $\underline{a}_n, \underline{e}_n^i$ to the n^{th} order problem can be recursively constructed from the $(n-1)^{\text{th}}$ order solution as follows:

$$\begin{aligned} \underline{a}_n &= \begin{pmatrix} \mu_n^1 / c_n^1 \\ \underline{a}_{n-1} \end{pmatrix} + \sum_{i=2}^K \tilde{\mu}_n^i \begin{pmatrix} \underline{e}_{n-1}^i \\ 0 \end{pmatrix} \\ \text{where: } \mu_n^i &= -\underline{c}_n^{iT} \begin{pmatrix} 0 \\ \underline{a}_{n-1} \end{pmatrix} \\ \tilde{\mu}_n^i &= \mu_n^i - \frac{c_n^i}{c_n^1} \mu_n^1 \end{aligned} \quad (6.2)$$

and:

$$\begin{aligned} \underline{e}_n^i &= \begin{pmatrix} \underline{e}_{n-1}^i \\ 0 \end{pmatrix} + \frac{\rho_n^i}{\varepsilon_n} \underline{a}_n \\ \text{where: } \rho_n^i &= d_n^i - \sum_{j=0}^{n-1} R_{j,n} e_{j,n-1}^i \end{aligned} \quad (6.3)$$

Note that because the prediction errors of the original and the transposed problems will be identical, we will have:

$$\sum_{i=1}^K \xi_n^i \nu_n^i = \sum_{i=2}^K \xi_n^i \mathcal{V}_n^i = \sum_{i=1}^K \rho_n^i \mu_n^i = \sum_{i=2}^K \rho_n^i \tilde{\mu}_n^i \quad (6.4)$$

Now that we have calculated $\underline{b}_n, \underline{f}_n^i$ and $\underline{a}_n, \underline{e}_n^i$, let us define the $(N+1)$ long vec-

tors $\underline{\alpha}_n, \underline{\beta}_n$ by:

$$\underline{\alpha}_n = R_N^T \begin{pmatrix} \underline{a}_n \\ \underline{0}_{N-n} \end{pmatrix} = \begin{pmatrix} \underline{0}_n \\ \alpha_{n,n} \\ \vdots \\ \alpha_{N,n} \end{pmatrix} \quad (6.5)$$

$$\underline{\beta}_n = R_N \begin{pmatrix} \underline{b}_n \\ \underline{0}_{N-n} \end{pmatrix} = \begin{pmatrix} \underline{0}_n \\ \beta_{n,n} \\ \vdots \\ \beta_{N,n} \end{pmatrix}$$

where $\alpha_{n,n} = \beta_{n,n} = \varepsilon_n$. As in the Levinson algorithm, let us form the $(n+1) \times (n+1)$ lower triangular matrices A_n and B_n whose j^{th} rows contain the coefficients of the vectors \underline{a}_n and \underline{b}_n :

$$A_n = \begin{pmatrix} 1 & & & 0 \\ a_{1,1} & 1 & & \\ \vdots & & \ddots & \\ a_{n,n} & \dots & a_{1,n} & 1 \end{pmatrix} \quad B_n = \begin{pmatrix} 1 & & & 0 \\ b_{1,1} & 1 & & \\ \vdots & & \ddots & \\ b_{n,n} & \dots & b_{1,n} & 1 \end{pmatrix} \quad (6.6)$$

Let us also form the $(n+1) \times (n+1)$ lower triangular matrices α_n and β_n whose j^{th} column contains the first $(n+1)$ entries of the vectors \underline{a}_n and $\underline{\beta}_n$:

$$\alpha_n = \begin{pmatrix} \alpha_{0,0} & & & 0 \\ \alpha_{1,0} & \alpha_{1,1} & & \\ \vdots & & \ddots & \\ \alpha_{n,0} & \alpha_{n,1} & \dots & \alpha_{n,n} \end{pmatrix} \quad \beta_n = \begin{pmatrix} \beta_{0,0} & & & 0 \\ \beta_{1,0} & \beta_{1,1} & & \\ \vdots & & \ddots & \\ \beta_{n,0} & \beta_{n,1} & \dots & \beta_{n,n} \end{pmatrix} \quad (6.7)$$

Then from the definitions of $\underline{\alpha}_n$ and $\underline{\beta}_n$ in (6.5) it is easy to see that:

$$R_n B_n^T = \beta_n \quad (6.8)$$

$$A_n R_n = \alpha_n^T$$

From this we can conclude, as in the Levinson case (2.14), that:

$$R_n^{-1} = B_n^T \Lambda_n^{-1} A_n \quad \text{where: } \Lambda_n = \text{diag}(\varepsilon_0 \dots \varepsilon_n) \quad (6.9)$$

Thus the extended Levinson-style algorithm which calculates $\underline{a}_n, \underline{b}_n, \underline{f}_n^i$ and \underline{e}_n^i effectively performs a UDL decomposition of the inverse matrix R_n^{-1} . From (6.8)

we can also conclude that

$$R_n = \beta_n \Lambda_n^{-1} \alpha_n^T \quad (6.10)$$

This formula suggest that a fast Choleski algorithm for computing the LDU factorization of R_n will need to recursively compute the vectors $\underline{\alpha}_n$ and $\underline{\beta}_n$ in (6.5).

7. Fast Choleski Algorithms for Almost-Toeplitz Matrices

Fast Choleski algorithms for solving almost Toeplitz linear equations can be developed in much the same way as for exactly Toeplitz equations. We start with the extended Levinson-style algorithm above which calculates the full UDL decomposition of the inverse matrix R_N^{-1} . A linear transformation similar to the one used in section 3 then converts the algorithm into a fast Choleski method for computing the LDU factorization of R_N . Exploiting several relationships among the various predictors and reflection coefficients reduces the computational effort substantially, and gives us our final algorithm.

The vectors $\underline{\alpha}_n$ and $\underline{\beta}_n$ in (6.5) are linear transformations of the vectors \underline{a}_n and \underline{b}_n . In fact, their definition is quite similar to the linear transformation used in section 3 for the exactly Toeplitz problem. To complete the transformation, let us define the $(N+1)$ long vectors $\underline{\varphi}_n^i$ and $\underline{\psi}_n^i$ as transformations of \underline{f}_n^i and \underline{e}_n^i :

$$\begin{aligned} \underline{\varphi}_n^i &= R_N \begin{bmatrix} \underline{f}_n^i \\ \underline{0}_{N-n} \end{bmatrix} - \underline{c}_N^i = \begin{bmatrix} \underline{0}_{n+1} \\ \varphi_{n+1,n}^i \\ \vdots \\ \varphi_{N,n}^i \end{bmatrix} \\ \underline{\psi}_n^i &= R_N^T \begin{bmatrix} \underline{e}_n^i \\ \underline{0}_{N-n} \end{bmatrix} - \underline{d}_N^i = \begin{bmatrix} \underline{0}_{n+1} \\ \psi_{n+1,n}^i \\ \vdots \\ \psi_{N,n}^i \end{bmatrix} \end{aligned} \quad (7.1)$$

Note that the $(n+1)^{th}$ elements of these vectors are the reflection coefficients:

$$\begin{aligned}\varphi_{n+1,n}^i &= -\xi_n^i \\ \psi_{n+1,n}^i &= -\rho_n^i\end{aligned}\tag{7.2}$$

Because \underline{a}_n , \underline{b}_n , $\underline{\varphi}_n^i$ and $\underline{\psi}_n^i$ have been defined as linear transformations of the vectors \underline{a}_n , \underline{b}_n , \underline{f}_n^i and \underline{e}_n^i , they will obey the same type of recursive relationships:

$$\begin{aligned}\underline{b}_n &= \begin{pmatrix} 0 \\ \beta_{0,n-1} \\ \vdots \\ \beta_{N-1,n-1} \end{pmatrix} + \sum_{i=2}^{\kappa} \nu_n^i \underline{\varphi}_{n-1}^i \\ \underline{\varphi}_n^i &= \underline{\varphi}_{n-1}^i + \frac{\xi_n^i}{\varepsilon_n} \underline{b}_n \quad \text{for } i=2, \dots, \kappa\end{aligned}\tag{7.3}$$

and:

$$\begin{aligned}\underline{a}_n &= \begin{pmatrix} 0 \\ \alpha_{0,n-1} \\ \vdots \\ \alpha_{N-1,n-1} \end{pmatrix} + \sum_{i=2}^{\kappa} \mu_n^i \underline{\psi}_{n-1}^i \\ \underline{\psi}_n^i &= \underline{\psi}_{n-1}^i + \frac{\rho_n^i}{\varepsilon_n} \underline{a}_n \quad \text{for } i=2, \dots, \kappa\end{aligned}\tag{7.4}$$

These recursions form the heart of the fast Choleski algorithm for almost Toeplitz matrices. The chief remaining difficulty is to find the values of the reflection coefficients ν_n^i , ξ_n^i , μ_n^i and ρ_n^i without having to calculate the original vectors \underline{a}_n , \underline{b}_n , \underline{f}_n^i or \underline{e}_n^i . The reflection coefficients ν_n^1 and μ_n^1 do not appear explicitly in the fast Choleski recursion, and so do not have to be computed. As noted in (7.2), the coefficients ξ_n^i and ρ_n^i can be read off directly from the vectors $\underline{\varphi}_n^i$ and $\underline{\psi}_n^i$. This leaves the more difficult problem of finding the values of ν_n^i and μ_n^i . To do this, note that:

$$\begin{aligned}\underline{a}_n^T (\mathbf{R}_n \underline{f}_n^j) &= \underline{a}_n^T \underline{c}_n^j \\ &= \left[\begin{pmatrix} \mu_n^1 / c_n^1 \\ \underline{a}_{n-1} \end{pmatrix} + \sum_{i=2}^{\kappa} \mu_n^i \begin{pmatrix} \underline{e}_{n-1}^i \\ 0 \end{pmatrix} \right]^T \underline{c}_n^j \\ &= - \left[\mu_n^j - \frac{c_0^j}{c_0^1} \mu_n^1 \right] + \sum_{i=2}^{\kappa} \mu_n^i \underline{e}_{n-1}^{iT} \underline{c}_{n-1}^j\end{aligned}$$

$$= -\tilde{\mu}_n^j + \sum_{i=2}^{\kappa} \tilde{\mu}_n^i \underline{e}_{n-1}^{i\top} R_{n-1} \underline{f}_{n-1}^j \quad \text{for } j=2, \dots, \kappa \quad (7.5)$$

But also:

$$\begin{aligned} \left(\underline{a}_n^\top R_n \right) \underline{f}_n^j &= (0 \ \dots \ 0 \ \varepsilon_n)^\top \left[\begin{pmatrix} \underline{f}_{n-1}^j \\ 0 \end{pmatrix} + \frac{\xi_n^j}{\varepsilon_n} \underline{b}_n \right] \\ &= \xi_n^j \end{aligned} \quad (7.6)$$

Combining (7.5) and (7.6) yields:

$$\begin{pmatrix} \xi_n^2 \\ \vdots \\ \xi_n^\kappa \end{pmatrix} = M_{n-1}^\top \begin{pmatrix} \tilde{\mu}_n^2 \\ \vdots \\ \tilde{\mu}_n^\kappa \end{pmatrix} \quad (7.7)$$

where M_{n-1} is a $(\kappa-1) \times (\kappa-1)$ matrix with entries $[M_{n-1}]_{i,j} = \underline{e}_{n-1}^{i\top} R_{n-1} \underline{f}_{n-1}^j - \delta_{ij}$.

Similarly, by considering the quadratic product $\underline{e}_n^{j\top} R_n \underline{b}_n$ we can show that:

$$\begin{pmatrix} \rho_n^2 \\ \vdots \\ \rho_n^\kappa \end{pmatrix} = M_{n-1} \begin{pmatrix} \tilde{\nu}_n^2 \\ \vdots \\ \tilde{\nu}_n^\kappa \end{pmatrix} \quad (7.8)$$

Equations (7.7) and (7.8) together suggest that we can calculate the reflection coefficients $\tilde{\mu}_n^i$ and $\tilde{\nu}_n^i$ from ξ_n^i and ρ_n^i provided we know M_{n-1} and provided that this matrix is invertible. To calculate this matrix efficiently, note that:

$$\begin{aligned} \underline{e}_n^{i\top} R_n \underline{f}_n^j &= \left[\begin{pmatrix} \underline{e}_{n-1}^i \\ 0 \end{pmatrix} + \frac{\rho_n^i}{\varepsilon_n} \underline{a}_n \right]^\top R_n \left[\begin{pmatrix} \underline{f}_{n-1}^j \\ 0 \end{pmatrix} + \frac{\xi_n^j}{\varepsilon_n} \underline{b}_n \right] \\ &= \underline{e}_{n-1}^{i\top} R_n \underline{f}_{n-1}^j + \frac{\rho_n^i \xi_n^j}{\varepsilon_n} \end{aligned} \quad (7.9)$$

Thus:

$$M_n = M_{n-1} + \frac{1}{\varepsilon_n} \begin{pmatrix} \rho_n^2 \\ \vdots \\ \rho_n^\kappa \end{pmatrix} \begin{pmatrix} \xi_n^2 & \dots & \xi_n^\kappa \end{pmatrix} \quad (7.10)$$

Assuming that M_{n-1} is invertible, then the inverse of M_n can be calculated recursively by using the Woodbury formula:

$$(A-BD^{-1}C)^{-1} = A^{-1} - A^{-1}B(CA^{-1}B+D)^{-1}CA^{-1} \quad (7.11)$$

Applying equations (7.7) and (7.8), and noting that:

$$\varepsilon_n = \varepsilon_{n-1} - \sum_{i=2}^{\kappa} \mathcal{V}_n^i \xi_n^i = \varepsilon_{n-1} - \begin{pmatrix} \xi_n^2 & \cdots & \xi_n^{\kappa} \end{pmatrix} M_{n-1}^{-1} \begin{pmatrix} \rho_n^2 \\ \vdots \\ \rho_n^{\kappa} \end{pmatrix} \quad (7.12)$$

leads to:

$$M_n^{-1} = M_{n-1}^{-1} - \frac{1}{\varepsilon_{n-1}} \begin{pmatrix} \mathcal{V}_n^2 \\ \vdots \\ \mathcal{V}_n^{\kappa} \end{pmatrix} \begin{pmatrix} \tilde{\mu}_n^2 & \cdots & \tilde{\mu}_n^{\kappa} \end{pmatrix} \quad (7.13)$$

Thus M_n will be invertible if and only if M_{n-1} is invertible and $\varepsilon_{n-1} \neq 0$. Applying this argument recursively leads to the conclusion that M_n is invertible if and only if R_N is strongly non-singular.

Let us summarize our progress so far. Calculating a full LDU decomposition of R_N requires calculating the vectors $\underline{\alpha}_n$ and $\underline{\beta}_n$. Equations (7.3) and (7.4) define a recursion for the vectors $\underline{\alpha}_n$, $\underline{\beta}_n$, $\underline{\varphi}_n^i$ and $\underline{\psi}_n^i$. The reflection coefficients ξ_n^i and ρ_n^i for this recursion can be found directly from $\underline{\varphi}_n^i$ and $\underline{\psi}_n^i$ by using (7.2). The coefficients \mathcal{V}_n^i and $\tilde{\mu}_n^i$ can be calculated as a linear transformation of ξ_n^i and ρ_n^i from (7.7) and (7.8). This linear transformation matrix M_{n-1}^{-1} can be calculated recursively from (7.13).

Given the LDU factorization of R_N , the solution \underline{x}_N to our linear equations can be found by the usual two stage forward/backward substitution algorithm:

$$\begin{aligned} \text{Solve: } \beta_N \underline{\lambda}_N &= \underline{y}_N & (7.14) \\ \text{Solve: } \alpha_N^T \underline{x}_N &= \Lambda_N \underline{\lambda}_N \end{aligned}$$

As in the fast Choleski algorithm for exactly Toeplitz matrices, the forward substitution phase for calculating $\underline{\lambda}_N$ is easily integrated into the fast Choleski

algorithm. The backward substitution phase, however, requires the coefficients $\alpha_{j,n}$ in the reverse order in which they are generated. These coefficients must therefore either be stored or regenerated in a backward phase.

The iteration can be simplified slightly if we define our initial conditions for $n=-1$:

$$\begin{aligned}
 \text{Initialization: } \varepsilon_{-1} &= c_0^1 d_0^1 \\
 \beta_{j,-1,-1} &= d_0^1 c_j^1 \quad \text{for } j=0, \dots, N \\
 \alpha_{j,-1,-1} &= c_0^1 d_j^1 \\
 \varphi_{j,-1}^i &= -c_j^i \\
 \psi_{j,-1}^i &= -d_j^i \\
 M_{-1}^{-1} &= -I
 \end{aligned} \tag{7.15}$$

The forward phase of our fast Choleski algorithm will then be as follows:

Forward Phase:

$$\text{For } n=0, \dots, N \tag{7.16}$$

$$\xi_n^i = -\varphi_{n,n-1}^i \quad \text{for } i=2, \dots, \kappa$$

$$\rho_n^i = -\psi_{n,n-1}^i$$

$$\begin{pmatrix} \mathcal{V}_n^2 \\ \vdots \\ \mathcal{V}_n^\kappa \end{pmatrix} = M_{n-1}^{-1} \begin{pmatrix} \rho_n^2 \\ \vdots \\ \rho_n^\kappa \end{pmatrix}$$

$$\begin{pmatrix} \tilde{\mu}_n^2 \\ \vdots \\ \tilde{\mu}_n^\kappa \end{pmatrix} = M_{n-1}^{-T} \begin{pmatrix} \xi_n^2 \\ \vdots \\ \xi_n^\kappa \end{pmatrix}$$

$$M_n^{-1} = M_{n-1}^{-1} - \frac{1}{\varepsilon_{n-1}} \begin{pmatrix} \mathcal{V}_n^2 \\ \vdots \\ \mathcal{V}_n^\kappa \end{pmatrix} \begin{pmatrix} \tilde{\mu}_n^2 & \dots & \tilde{\mu}_n^\kappa \end{pmatrix}$$

$$\varepsilon_n = \varepsilon_{n-1} - \sum_{i=2}^{\kappa} \mathcal{V}_n^i \xi_n^i$$

$$\left. \begin{aligned}
 \beta_{j,n} &= \beta_{j-1,n-1} + \sum_{i=2}^{\kappa} \mathcal{V}_n^i \varphi_{j,n}^i \\
 \alpha_{j,n} &= \alpha_{j-1,n-1} + \sum_{i=2}^{\kappa} \tilde{\mu}_n^i \psi_{j,n}^i
 \end{aligned} \right\} \quad \text{for } j=n+1, \dots, N$$

$$\left. \begin{aligned} \varphi_{j,n}^i &= \varphi_{j,n-1}^i + \frac{\xi_n^i}{\varepsilon_n} \beta_{j,n} \\ \psi_{j,n}^i &= \psi_{j,n-1}^i + \frac{\rho_n^i}{\varepsilon_n} \alpha_{j,n} \end{aligned} \right\} \begin{array}{l} \text{for } j=n+1, \dots, N \\ i=2, \dots, \kappa \end{array}$$

$$\lambda_n = \frac{y_n}{\varepsilon_n}$$

$$y_j \leftarrow y_j - \lambda_n \beta_{j,n} \quad \text{for } j=n+1, \dots, N$$

Total storage required in this forward phase is about $(2\kappa+1)N$ locations for the $\underline{\alpha}_n$, $\underline{\beta}_n$, $\underline{\varphi}_n^i$, $\underline{\psi}_n^i$ and \underline{y}_N vectors. Computation can be done in place, and it will be convenient on the n^{th} pass to save the values of the reflection coefficients $\tilde{\mu}_n^i$ and ρ_n^i in the locations previously used by the entries $\varphi_{n,n-1}^i$ and $\psi_{n,n-1}^i$. It is also useful to retain the values of $\alpha_{N,j}$. Storage after the n^{th} pass should thus be organized as follows:

$$\begin{array}{ll} \underline{\alpha}: & \alpha_{N,0} \cdots \alpha_{N,n} \quad \alpha_{N-1,n} \cdots \alpha_{n+1,n} \\ \underline{\beta}: & \beta_{N,0} \cdots \beta_{N,n} \quad \beta_{N-1,n} \cdots \beta_{n+1,n} \\ \underline{\varphi}^i: & \varphi_{N,n}^i \cdots \varphi_{n+1,n}^i \quad \tilde{\mu}_n^i \cdots \tilde{\mu}_1^i \\ \underline{\psi}^i: & \psi_{N,n}^i \cdots \psi_{n+1,n}^i \quad \rho_n^i \cdots \rho_1^i \\ \underline{y}: & y_N \cdots y_{n+1} \quad \lambda_n \cdots \lambda_1 \end{array} \quad (7.17)$$

Total computation on the forward phase will be about $\frac{1}{2}(4\kappa-3)N^2$ operations.

We still need to calculate \underline{x}_N from $\underline{\lambda}_N$. If an extra $\frac{1}{2}N^2$ storage locations are available for saving the values of $\frac{\alpha_{j,n}}{\varepsilon_n}$, then we can solve for \underline{x}_N using simple back substitution:

Back Substitution (Extra $\frac{1}{2}N^2$ storage)

For $n=N, \dots, 0$

$$x_n = \lambda_n - \frac{1}{\varepsilon_n} \sum_{j=n+1}^N \alpha_{j,n} x_j \quad (7.18)$$

Total computation will then be about $(2\kappa-1)N^2 + O(\kappa^2 N)$ operations, which is slightly faster than the Levinson-style algorithm in section 5.

If no extra storage is available, it will be necessary to recalculate the values of $\alpha_{j,n}$ in descending order of n in order to solve for \underline{x}_N . If the forward phase has saved the values of $\tilde{\mu}_n^i$, ρ_n^i and $\alpha_{N,n}$ as suggested above, then the backward substitution phase would be as follows:

Backward Phase: (no extra storage)

$$\underline{x}_N = \lambda_N \tag{7.19}$$

For $n=N, \dots, 1$

$$\psi_{j,n-1}^i = \begin{cases} -\rho_n^i & \text{for } j=n \\ \psi_{j,n}^i - \frac{\rho_n^i}{\varepsilon_n} \alpha_{j,n} & \text{for } j=n+1, \dots, N \end{cases} \quad \text{for } i=2, \dots, \kappa$$

$$\alpha_{j-1,n-1} = \alpha_{j,n} - \sum_{i=2}^{\kappa} \tilde{\mu}_n^i \psi_{j,n-1}^i \quad \text{for } j=n+1, \dots, N$$

(value $\alpha_{N,n-1}$ saved from forward phase)

$$\varepsilon_{n-1} = \varepsilon_n + \sum_{i=2}^{\kappa} \tilde{\mu}_n^i \rho_n^i$$

$$\underline{x}_{n-1} = \lambda_{n-1} - \frac{1}{\varepsilon_{n-1}} \sum_{j=n}^N \underline{x}_j \alpha_{j,n-1}$$

Total computation time is now about $\frac{1}{2}(5\kappa-2)N^2 + O(\kappa^2 N)$ operations, and total storage required is about $(2\kappa+1)N$ locations. This fast Choleski algorithm thus uses 50% less storage and about $\frac{1}{2}\kappa N^2$ more computation than the Levinson-style algorithm.

The above algorithms are similar to the columnwise fast Choleski algorithms for exactly Toeplitz matrices. As in section 3, it is possible to rearrange the computation into a rowwise form, which computes the LDU decomposition of R_N row by row instead of column by column. Although we will not present the details, this variation has certain advantages in problems such as adaptive filtering where the data length N may not be fixed in advance.

8. Formulas for R_N and R_N^{-1}

Before continuing, given a vector \underline{x}_N , let us define:

$$\underline{x}_N^v = \begin{pmatrix} 0 \\ x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} \quad \underline{\hat{x}}_N = \begin{pmatrix} x_1 \\ \vdots \\ x_N \\ 0 \end{pmatrix} \quad \underline{\bar{x}}_N = \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} \quad (8.1)$$

Also let us define:

$$\begin{aligned} C_n &= \begin{pmatrix} c_n^2 & \cdots & c_n^\kappa \end{pmatrix} & D_n &= \begin{pmatrix} d_n^2 & \cdots & d_n^\kappa \end{pmatrix} \\ F_n &= \begin{pmatrix} f_n^2 & \cdots & f_n^\kappa \end{pmatrix} & E_n &= \begin{pmatrix} e_n^2 & \cdots & e_n^\kappa \end{pmatrix} \\ \Phi_n &= \begin{pmatrix} \varphi_n^2 & \cdots & \varphi_n^\kappa \end{pmatrix} & \Psi_n &= \begin{pmatrix} \psi_n^2 & \cdots & \psi_n^\kappa \end{pmatrix} \end{aligned} \quad (8.2)$$

As we have emphasized in earlier sections, the fast Choleski algorithm can be interpreted as an N step algorithm for performing an LDU decomposition (6.10) of the matrix R_N . After n steps of the columnwise fast Choleski algorithm, we will have computed only the first $(n+1)$ columns and rows of this LDU decomposition. It is of considerable interest to note that the matrix R_N can be written as the product of these first $(n+1)$ columns and rows of α_N and β_N^T plus a matrix \tilde{R}_n which is zero everywhere except in the lower right $(N-n-1) \times (N-n-1)$ corner, and which can be expressed as a sum of κ products of lower, diagonal, and upper triangular block Toeplitz matrices:

$$R_N = \begin{pmatrix} \beta_{0,0} & & 0 \\ \vdots & & \vdots \\ \vdots & & \beta_{n,n} \\ \vdots & & \vdots \\ \beta_{N,0} & \cdots & \beta_{N,n} \end{pmatrix} \begin{pmatrix} \varepsilon_0 & & 0 \\ & \cdot & \\ 0 & & \varepsilon_n \end{pmatrix}^{-1} \begin{pmatrix} \alpha_{0,0} & \cdots & \cdots & \cdots & \alpha_{N,0} \\ & \cdot & & & \vdots \\ 0 & & \alpha_{n,n} & \cdots & \alpha_{N,n} \end{pmatrix} + \tilde{R}_n \quad (8.3)$$

where:

$$\tilde{R}_n = L(\underline{\beta}_n^v) D(\varepsilon_n^{-1}) U(\underline{\alpha}_n^T) - L(\Phi_n) D(M_n^{-1}) U(\Psi_n^T) \quad (8.4)$$

where $L(\underline{x})$, $U(\underline{y}^T)$ should be interpreted as block lower (upper) triangular Toe-

plitz matrices with first column of \underline{x} (first row of \underline{y}^T) and $D(S)$ is a block diagonal Toeplitz matrix with diagonal elements S . The proof of this formula uses induction, and may be found in Appendix A. This formula has a very interesting interpretation in terms of Schur complements. Partition off the first $(n+1)$ rows and columns of R_N :

$$R_N = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad (8.5)$$

where A is $(n+1) \times (n+1)$, D is $(N-n) \times (N-n)$, and B and C are sized accordingly.

Factor $A = L_\alpha U_\alpha$ into lower and upper triangular matrices. Then:

$$R_N = \begin{pmatrix} L_\alpha \\ U_\alpha^{-1} \end{pmatrix} \left(U_\alpha L_\alpha^{-1} B \right) + \begin{pmatrix} 0 & 0 \\ 0 & D - CA^{-1}B \end{pmatrix} \quad (8.6)$$

Comparing (8.3) and (8.6), it is clear that:

$$\begin{pmatrix} 0 & 0 \\ 0 & D - CA^{-1}B \end{pmatrix} = \tilde{R}_n \quad (8.7)$$

Thus the Schur complement $D - CA^{-1}B$ can be represented as a sum of lower \times upper triangular Toeplitz matrices composed of the n^{th} order fast Choleski predictors $\underline{\beta}_n, \underline{\alpha}_n, \underline{\varrho}_n, \underline{\psi}_n$. This fact could have been used, for example, to simplify the doubling algorithms of Bitmead and Anderson¹⁵ and Morf.¹⁶

The important Gohberg-Semencul formulas (2.18) and (2.19) for exactly Toeplitz matrices can also be generalized to almost-Toeplitz matrices. Appendix A proves that if R_N and R_{N-1} are both invertible, then we can write R_N^{-1} and R_{N-1}^{-1} as sums of upper times diagonal times lower triangular Toeplitz matrix products:

$$\begin{aligned} R_N^{-1} &= U(\underline{b}_N)D(\varepsilon_N^{-1})L(\underline{a}_N^T) - U(\hat{F}_n)D(M_N^{-1})L(\hat{E}_N^T) \\ R_{N-1}^{-1} &= U(\underline{b}_{N-1})D(\varepsilon_{N-1}^{-1})L(\underline{a}_{N-1}^T) - U(\bar{F}_N)D(M_N^{-1})L(\bar{E}_N^T) \end{aligned} \quad (8.8)$$

where $U(\underline{x})$ (and $L(\underline{y}^T)$) are block upper (lower) triangular Toeplitz matrices with last column \underline{x} (last row \underline{y}^T). These formulas are the upper \times lower

representation of R_N^{-1} whose existence was guaranteed in Theorem 1 of section 4. The advantage of these formulas is that they completely specify R_N^{-1} and R_{N-1}^{-1} in terms of the Levinson-style algorithm predictors \underline{a}_N , \underline{b}_N , \underline{f}_N^i , \underline{e}_N^i and ε_N^{-1} , M_N^{-1} . We can thus calculate $\underline{x}_N = R_N^{-1} \underline{y}_N$ directly from the output of the Levinson-style algorithm without needing to actually compute or store R_N^{-1} . Furthermore, computing $R_N^{-1} \underline{y}_N$ only involves multiplying (block) triangular or diagonal matrices times vectors, which is equivalent to convolving the matrix and vector coefficients. Thus we can use $2N+1$ point FFT's to solve for \underline{x}_N in $O(N \log N)$ operations. It is these formulas for R_N^{-1} which allow the use of the doubling algorithm described in sections 13 and 14 for computing \underline{x}_N in $O(\kappa^2 N \log^2 N)$ operations.

9. Alternative Algorithms Exploiting Other Almost-Toeplitz Forms

The Levinson-style and fast Choleski algorithms developed above are not the only available algorithm for dealing with almost-Toeplitz matrices. Other algorithms can be easily devised which normalize the various predictors in different ways, or define the \underline{f}_n^i and \underline{e}_n^i vectors differently (see, for example, the square root algorithms of Morf²⁴ or the Friedlander algorithm⁵ .)

Other almost-Toeplitz algorithms can be developed by using different formulations of R_N . In the preceding algorithms, we have assumed that R_N can be written as the sum of κ_+ products of lower and upper triangular Toeplitz matrices.

$$R_N = \sum_{i=1}^{\kappa_+} L(\underline{c}_N^i) U(\underline{d}_N^i) \quad (9.1)$$

Similar algorithms could be developed for the case when R_N can be written as the sum of κ_- products of lower and upper triangular Toeplitz matrices:

$$R_N = \sum_{i=1}^{\kappa_-} U(\underline{c}_N^i) L(\underline{d}_N^i) \quad (9.2)$$

The Levinson and fast Choleski algorithms appropriate to this form of R_N will look similar to those developed above, except that they work with the lower right principal minors of R_N . Mixed representations of the form:

$$R_N = \sum_{i=1}^{\kappa_1} L(\underline{c}_N^i) U(\underline{d}_N^{iT}) + \sum_{i=1}^{\kappa_2} U(\underline{d}_N^i) L(\underline{c}_N^{iT}) \quad (9.3)$$

could also be considered. Again, the corresponding algorithms will be similar to those above, except that the recursions on the predictors will be considerably more complicated.

Of course, as described in section 5, all these representations are equivalent in the sense that each form can be converted into any of the others. Nevertheless, certain representations may have slightly fewer terms than the others, and may lead to a faster algorithm. Moreover, certain forms may allow exploiting any additional structure in R_N . Marple,²⁵ for example, considers the modified covariance method of linear prediction in which a set of linear equations must be solved for which $\text{rank}(\underline{\square} R_N) = \text{rank}(\overline{\square} R_N) = 6$. By choosing a mixed representation of the form (9.3), however, he is able to exploit the symmetry of R_N about both the main and secondary diagonals in order to reduce the number of different predictors to only 3.

Finally, we should note that the key concept exploited by all these algorithms is the type of partitioning of R_N in (5.3), where we express R_N in terms of a matrix R_{N-1} , for which we know how to solve the problem, plus a correction term of low rank. This same idea can be exploited in many other contexts as well. Morf, for example, has developed numerous "time update" recursive algorithms for linear prediction problems in which the size of the matrix does not change from iteration to iteration, but low rank correction terms are added as new data points arrive. Jain^{11,26} has also exploited this idea in order to express a band diagonal almost-Toeplitz matrix R_N as the sum of a circulant matrix (or

some other convenient matrix) plus a correction term. The Woodbury formula (7.11) is then used to calculate R_N^{-1} using FFT's (or similar transform.)

10. Quadratic Forms

Many applications require evaluating the quadratic form:

$$\tau = \underline{y}_1^T R^{-1} \underline{y}_2 \quad (10.1)$$

where R is almost-Toeplitz. If we use a fast Choleski method to factor $R = \beta \Lambda^{-1} \alpha^T$, then:

$$\tau = \left(\alpha^{-1} \underline{y}_1 \right)^T \Lambda \left(\beta^{-1} \underline{y}_2 \right) \quad (10.2)$$

Evaluating $\alpha^{-1} \underline{y}_1$ and $\beta^{-1} \underline{y}_2$ will only require forward substitution, and is easily integrated into the fast Choleski recursion. The backward recursion will be unnecessary. A different approach would be to use the Levinson-style algorithm to factor $R^{-1} = B^T \Lambda^{-1} A$, in which case

$$\tau = \left(B \underline{y}_1 \right)^T \Lambda^{-1} \left(A \underline{y}_2 \right) \quad (10.3)$$

Evaluation of the terms $B \underline{y}_1$ and $A \underline{y}_2$ is easily integrated into the recursion. Still another approach would be to use the Levinson-style algorithm to calculate the upper \times lower representation of R_N^{-1} in (8.8), in which case τ can be evaluated using Fast Fourier Transforms in $O(N \log N)$ operations.

11. Symmetry

Both the fast Choleski and Levinson-style algorithms simplify when R_N is symmetric, $R_{ij} = R_{ji}$. The first step is to start with a symmetric representation of R_N in the form:

$$R_N = \sum_{i=1}^{\kappa_1} L(\underline{c}_N^i) U(\underline{c}_N^{i^T}) - \sum_{i=\kappa_1+1}^{\kappa} L(\underline{c}_N^i) U(\underline{c}_N^{i^T}) \quad (11.1)$$

For example, if R_N were exactly Toeplitz, we could choose $\kappa_1 = 1$, $\kappa = 2$ and:

$$\underline{c}_N^1 = \frac{1}{\sqrt{r(0)}} \begin{bmatrix} r(0) \\ r(1) \\ \vdots \\ r(N) \end{bmatrix} \quad \underline{c}_N^2 = \frac{1}{\sqrt{r(0)}} \begin{bmatrix} 0 \\ r(1) \\ \vdots \\ r(N) \end{bmatrix} \quad (11.2)$$

With this representation, symmetry guarantees that:

$$\begin{aligned} \underline{a}_n &= \underline{b}_n & \underline{\alpha}_n &= \underline{\beta}_n \\ \underline{L}_n^i &= \pm \underline{e}_n^i & \underline{\varrho}_n^i &= \pm \underline{\psi}_n^i \\ \underline{\xi}_n^i &= \pm \underline{\rho}_n^i & \underline{\nu}_n^i &= \pm \underline{\mu}_n^i \end{aligned} \quad (11.3)$$

where the + sign is valid if $i \leq \kappa_1$ and the - sign is valid if $\kappa_1 < i < \kappa$. These relationships immediately cut the computational effort and storage requirements of the fast Choleski algorithm in half. Solving for \underline{x}_N using our storage efficient forward/backward recursion will require only about $2\kappa N^2$ operations and about $(\kappa+1)N$ storage locations.

The Levinson-style algorithm does not simplify quite so drastically. The chief savings is that now it is not necessary to calculate the reflection coefficients ξ_n^i using the formula in (5.8), since they can be calculated exactly from \mathcal{V}_n^i and M_{n-1} :

$$\begin{pmatrix} \xi_n^2 \\ \vdots \\ \xi_n^\kappa \end{pmatrix} = M_{n-1}^T \begin{pmatrix} \tilde{\mu}_n^2 \\ \vdots \\ \tilde{\mu}_n^\kappa \end{pmatrix} \quad (11.4)$$

where $\tilde{\mu}_n^i = \pm \mathcal{V}_n^i$ as explained above. This removes the need for storing and/or calculating the values of $R_{i,j}$, and it reduces the computational effort for calculating \underline{x}_N to about $\frac{3}{2}\kappa N^2$ operations and about $(2\kappa+1)N$ storage locations for \underline{a}_N , \underline{b}_N , \underline{L}_N^i , and \underline{u}_N . The symmetric Levinson-style algorithm thus is still faster than the symmetric fast Choleski algorithm for calculating \underline{x}_N , but it uses more storage.

12. Band Diagonal Toeplitz Matrices

When R_N is band diagonal with $R_{i,j} = 0$ except for $-q \leq i - j \leq p$, then the fast Choleski algorithm simplifies so that the computational effort is only $O(N(p+q))$ operations. Band Diagonal Levinson-style algorithms can also be devised,^{9,10} but they are usually numerically unstable. It is difficult to treat this case in

general because of the wide variation in the structure of the problem for various applications. We therefore treat the exactly Toeplitz band diagonal case in some depth, and then indicate how the ideas can be extended to the almost-Toeplitz case. Several examples involving band diagonal almost-Toeplitz matrices are presented in sections 18,19.

Trench⁹ and Dickinson¹⁰ have noted that when the matrix R_N is Toeplitz and band diagonal, with $r(n)=0$ for $n < -q$ and $n > p$, then the Levinson-Trench-Zohar algorithm can be simplified somewhat. They pointed out that if we knew the last p coefficients x_{N-p+1}, \dots, x_N of \underline{x}_N , then the remaining coefficients could be recursively generated by exploiting the band diagonal structure of R_N :

$$x_n = \frac{1}{r(p)} \left[y_n - r(p-1)x_{n+1} - \dots - r(-q)x_{n+p+q} \right] \quad (12.1)$$

for $n=N-p, \dots, 0$. Dickinson and Trench further pointed out that calculating the last p points of \underline{x}_N only required knowledge of the coefficients $a_{j,n}$ and $b_{n-j,n}$ for $j=0, \dots, q-1$ and $j=n-p+1, \dots, n$. Total computation required to calculate \underline{x}_N using the band diagonal LTZ algorithm is thus only about $(6p+4q)N$ operations. Total storage required for saving \underline{y}_N , $a_{j,n}$, $b_{n-j,n}$ and $r(n)$ is only about $N+3(p+q+1)$ locations.

The crucial flaw in this algorithm, which was not remarked on by Trench or Dickinson, is that the recursion in (12.1) for x_n is often numerically unstable for many problems of interest. Suppose, for instance, that R_N is symmetric and positive definite, with $R(z) = \sum_{n=-p}^p r(n)z^{-n}$. Let us factor $R(z) = P(z)P(z^{-1})$ where $P(z)$ is a causal and minimum phase polynomial of order p . The recursion in (12.1) is thus equivalent to filtering y_n with an infinite impulse response filter with transfer function

$$\frac{1}{z^p R(z)} = \frac{1}{z^p P(z^{-1})P(z)} \quad (12.2)$$

Unfortunately, all the roots of $P(z^{-1})$ will be outside the unit circle, and thus this recursion for calculating x_n will be numerically unstable.

The fast Choleski algorithm, on the other hand, is ideally suited for solving linear equations involving band diagonal Toeplitz matrices. Because only $(p+q+1)$ coefficients of $R(z)$ are non-zero, the polynomials $\alpha_n(z)$ and $\beta_n(z)$ will also have only $(p+q+1)$ non-zero coefficients. Thus the matrices α_N and β_N will be band diagonal. This enables us to significantly reduce the storage and computation requirements of both the columnwise and rowwise fast Choleski algorithms. Several variations are possible which use differing amounts of storage and computation. The most storage efficient method (and the slowest) uses less storage than the Trench-Dickinson algorithm and about the same number of operations. Unfortunately, this particular variation is not numerically stable in general. The less storage efficient (and faster) fast Choleski algorithms, however, appear to be numerically stable.

We will assume that $r(p), r(-q) \neq 0$. Only the following coefficients of $\alpha_n(z)$ and $\beta_n(z)$ will be non-zero:

$$\begin{aligned} \alpha_n(z): & \alpha_{-q,n} \cdots \alpha_{0,n} \quad \alpha_{n+1,n} \cdots \alpha_{n+p,n} \\ \beta_n(z): & \beta_{-p,n} \cdots \beta_{0,n} \quad \beta_{n+1,n} \cdots \beta_{n+q,n} \end{aligned} \quad (12.3)$$

It is thus quite feasible to compute the coefficients in ascending order of n using the polynomial recursion in (3.3). About $2(p+q)N$ operations will be required. If we need to solve $R_N \underline{x}_N = \underline{y}_N$, then we will use the usual forward/backward substitution algorithm of section 3. The forward substitution phase for calculating $\underline{\lambda}_N$ uses the values $\beta_{-j,n}$ in ascending order of n , and is easily integrated into the fast Choleski algorithm. If Nq extra storage is available for saving the values of $\frac{\alpha_{-j,n}}{\epsilon_n}$ for $n=0, \dots, N$ and $j=-q, \dots, -1$, then the backward phase for computing \underline{x}_N would take the form:

Backward Phase (Nq extra storage for $\alpha_{-j,n}$)

For $n=N, \dots, 0$

$$x_n = \lambda_n - \frac{1}{\varepsilon_n} \sum_{j=1}^{\max(q, N-n)} x_{n+j} \alpha_{-j,n} \quad (12.4)$$

Total computation would then be about $(3(p+q)+4)N$ operations, and total storage required would be $N(q+1)+2(p+q)$ locations. This is $(3p+q)N$ fewer operations but Nq more storage than the Dickinson-Trench algorithm.

If only $2N$ extra storage locations were available, then we could save the values of the reflection coefficients ξ_n and ν_n as they were calculated during the forward phase of the fast Choleski algorithm. The backward phase for calculating x_N from λ_N could then start with the values of $\alpha_N(z)$ and $\beta_N(z)$ and exploit the backward polynomial recursion in (3.14). Renormalizing to decrease the operation count, and noting that $\beta_{n+q,n-1}=0$ so that $\beta_{n+q,n}=\nu_n \alpha_{-q,n}$, then gives the following algorithm for the backward phase for calculating x_N : (we delete the "~" to simplify the notation):

Backward Phase ($2N$ extra storage for ξ_n, ν_n)

$$x_N = \lambda_N \quad (12.5)$$

For $n=N, \dots, 1$

$$\left. \begin{aligned} \alpha_{j,n-1} &= \alpha_{j,n} - \xi_n \beta_{n-j,n} \\ \beta_{n-j,n-1} &= \beta_{n-j,n} - \nu_n \alpha_{j,n} \end{aligned} \right\} \text{ for } j=-1, \dots, -q+1$$

$$\beta_{n,n-1} = -\nu_n \varepsilon_N$$

$$\alpha_{-q,n-1} = \alpha_{-q,n} (1 - \xi_n \nu_n)$$

$$x_{n-1} = \lambda_{n-1} - \frac{1}{\varepsilon_N} \sum_{j=1}^{\min(q, N-n+1)} x_{n+j-1} \alpha_{-j,n-1}$$

Total computation required is $(3p+5q+5)N$ operations and $3N+2(p+q)$ storage locations are needed. This is still faster than the Dickinson or Trench algorithms, but it requires $2N$ more storage locations.

Even if no extra storage locations are available beyond those needed for the forward recursion, it is still possible to regenerate the $\alpha_{j,n}$ coefficients in descending order of n given only the values of ε_N , $\alpha_N(z)$, and $\beta_N(z)$ calculated on the forward phase. The trick is to note that in the band diagonal case we can calculate the reflection coefficients ξ_n and ν_n from the outermost coefficients of $\alpha_n(z)$ and $\beta_n(z)$:

$$\xi_n = \frac{\alpha_{n+p,n}}{\beta_{-p,n}} \quad \text{and} \quad \nu_n = \frac{\beta_{n+q,n}}{\alpha_{-q,n}} \quad (12.6)$$

It is easy to show that:

$$\left. \begin{array}{l} \alpha_{-q,n} = \alpha_{-q,0} = r(-q) \neq 0 \\ \beta_{-p,n} = \beta_{-p,0} = r(p) \neq 0 \end{array} \right\} \text{ for all } n \quad (12.7)$$

and thus the denominators in (12.6) will always be non-zero provided that $r(p), r(-q) \neq 0$. Given $\alpha_n(z)$ and $\beta_n(z)$ on the n^{th} pass of the backward phase, we can thus use (12.6) to calculate ξ_n and ν_n , and then exploit the backward recursion in (3.14) to generate $\alpha_{n-1}(z)$ and $\beta_{n-1}(z)$. Renormalizing to minimize the operation count then yields the following backward phase for calculating \underline{x}_N :

Backward Phase (no extra storage)

$$\begin{array}{l} x_N = \underline{\lambda}_N \\ \text{For } n = N, \dots, 1 \end{array} \quad (12.8)$$

$$\xi_n = \frac{\alpha_{n+p,n}}{\beta_{-p,n}}$$

$$\nu_n = \frac{\beta_{n+q,n}}{\alpha_{-q,n}}$$

$$\left. \begin{array}{l} \alpha_{j,n-1} = \alpha_{j,n} - \xi_n \beta_{n-j,n} \\ \beta_{n-j,n-1} = \beta_{n-j,n} - \nu_n \alpha_{j,n} \end{array} \right\} \begin{array}{l} \text{for } j = -q+1, \dots, -1 \\ \text{and } j = n+1, \dots, n+p-1 \end{array}$$

$$\alpha_{-q,n-1} = \alpha_{-q,n} (1 - \xi_n \nu_n)$$

$$\alpha_{n,n-1} = -\xi_n \varepsilon_N$$

$$\beta_{-p,n-1} = \beta_{-p,n}(1 - \xi_n \nu_n)$$

$$\beta_{n,n-1} = -\nu_n \varepsilon_N$$

$$x_{n-1} = \lambda_{n-1} - \frac{1}{\varepsilon_N} \sum_{j=1}^{\max(q, N-n+1)} x_{n+j-1} \alpha_{-j,n-1}$$

Total computation is $(5(p+q)+7)N$ operations, and total storage required is only $N+2(p+q)$ locations. If the matrix R_N is symmetric, then $p=q$, $\xi_n = \nu_n$, and $\alpha_{j,n} = \beta_{j,n}$. This symmetry can be exploited to reduce the computation to $(3(p+q)+6)N$ operations and $N+(p+q)$ storage locations. These figures are quite similar to those of the Trench-Dickinson algorithm.

Unfortunately, while our first two backward phase algorithms are numerically stable, this storage efficient algorithm is not. If R_N is symmetric, Toeplitz and positive definite, then it can be shown²⁷ that as $n \rightarrow \infty$ the polynomials $\alpha_n(z)$ and $\beta_n(z)$ converge to anticausal, maximum phase polynomials $\alpha_\infty(z)$ and $\beta_\infty(z)$ with $\alpha_{j,\infty} = \beta_{j,\infty} = 0$ for $j > 0$ and $R(z) = \alpha_\infty(z)\beta_\infty(z^{-1})$. The minimum storage backward phase algorithm above attempts to reverse this stable recursion, starting from $\alpha_N(z)$, $\beta_N(z)$ and generating $\alpha_0(z)$ and $\beta_0(z)$ after N steps. The difference between $\alpha_N(z)$, $\beta_N(z)$ and $\alpha_\infty(z)$, $\beta_\infty(z)$, however, can be quite insignificant for large N , and small errors in the values of $\alpha_N(z)$, $\beta_N(z)$ will inevitably lead to large errors in the final values of $\alpha_0(z)$ and $\beta_0(z)$. (This problem can be partially cured by saving "snapshot" values of $\alpha_n(z)$, $\beta_n(z)$ after every m steps of the forward recursion, and using these to periodically "reset" the backward recursion on $\alpha_n(z)$, $\beta_n(z)$.)

To demonstrate the relative accuracy of our various methods, we have run the Levinson-Trench-Zohar (LTZ) algorithm, the Trench-Dickinson (TD) algorithm, and the last two variants of the fast Choleski algorithm (the "extra $2N$ " storage and the "minimal" algorithms) on the same band diagonal Toeplitz

matrix. The coefficients of \underline{y}_N were independent gaussian random variables with unit variance. The following table compares the mean square error

$$\sum_{n=0}^N (x_n - \hat{x}_n)^2$$

between the calculated value of \underline{x}_N and the known solution \hat{x}_N for

various values of N . (Double precision floating point (64 bits) was used.)

$$R(z) = (1 - .8z^{-1})(1 - .7z^{-1})(1 - .9e^{-j\pi/4}z)(1 - .9e^{+j\pi/4}z)$$

| | LTZ | TD | Fast Choleski | |
|---------|------------------|------------------|------------------|------------------|
| | | | extra $2N$ | minimal |
| $N=20$ | $5.0 * 10^{-27}$ | $3.5 * 10^{-25}$ | $1.3 * 10^{-27}$ | $1.2 * 10^{-25}$ |
| $N=50$ | $6.1 * 10^{-27}$ | $3.0 * 10^{-13}$ | $2.8 * 10^{-27}$ | $4.9 * 10^{-22}$ |
| $N=100$ | $2.9 * 10^{-28}$ | $1.8 * 10^{+4}$ | $7.2 * 10^{-27}$ | $6.2 * 10^{-16}$ |
| $N=250$ | $6.7 * 10^{-28}$ | $2.3 * 10^{+37}$ | $6.6 * 10^{-26}$ | $9.6 * 10^{-1}$ |

Trench-Dickinson has the worst behavior due to the instability of the recursion in (12.1) for $x(n)$. The fast Choleski algorithm using $2N$ extra storage locations to save the reflection coefficient values ξ_n and ν_n from the forward phase is most accurate. The minimal storage fast Choleski algorithm is also clearly unstable, though it is not as bad as Trench-Dickinson.

Exactly the same type of reasoning can be applied to almost-Toeplitz problems involving band diagonal matrices. Only a few coefficients of the vectors $\underline{\alpha}_n$, $\underline{\beta}_n$, $\underline{\varphi}_n^i$, $\underline{\psi}_n^i$ will be non-zero, and this drastically simplifies the computational effort and storage requirements. In solving for \underline{x}_N , a numerically stable backward phase will require saving the $2(\kappa-1)N$ values ρ_n^i and $\tilde{\mu}_n^i$. Several examples are given in section 18,19.

13. Doubling Algorithms - Exactly Toeplitz Matrices

The Levinson-Trench and fast Choleski algorithms are not the fastest available algorithms for solving Toeplitz or almost-Toeplitz problems. Gustavson and Yun¹⁴ have shown that the exactly Toeplitz problem can be solved using a Euclidian algorithm involving repeated division of polynomials. Applying the divide and conquer strategy of Aho, Ullman and Hopcroft (chap 8)¹³ and using Fast

Fourier Transforms for fast multiplication of polynomials, they arrived at an algorithm for calculating $A_N(z)$ and $B_N(z)$ in only $O(N \log^2 N)$ operations. Using the Gohberg-Semencul-like formulas (8.8) for R_N^{-1} , they then calculate $\underline{x}_N = R_N^{-1} \underline{y}_N$ in only $O(N \log N)$ additional operations. Their algorithm does not require the strongly non-singular constraint of Levinson recursion. Bitmead and Anderson¹⁵ and Morf¹⁶ applied a similar divide and conquer strategy to the almost-Toeplitz problem, although their algorithms are rather complex and are not easily automated. Bitmead and Anderson, in particular, rely on a "generic" method for factoring a low rank displacement matrix, which will fail in many applications.

We take a somewhat different approach towards deriving these doubling algorithms, in which we exploit the resemblance between the fast Choleski recursion and a Euclidian algorithm. Euclidian polynomial algorithms calculate the greatest common divisor of two polynomials by recursively dividing polynomials. Thus if $F_n(z)$ and $G_m(z)$ are n^{th} and m^{th} degree polynomials respectively with $n > m$, we can divide $G_m(z)$ into $F_n(z)$, giving a quotient $Q_{n-m}(z)$ of degree $(n-m)$ and a remainder $H_r(z)$ with degree $r < m$:

$$F_n(z) = Q_{n-m}(z)G_m(z) + H_r(z) \quad (13.1)$$

The key idea is to subtract shifted and scaled versions of the polynomial $G_m(z)$ from $F_n(z)$ in order to drive the $(n-m)$ highest order coefficients of $F_n(z)$ to zero. Now the polynomials $\alpha_n(z)$ and $\beta_n(z)$ potentially have infinite degree, and thus applying a Euclidian algorithm to these would be infeasible. However, the fast Choleski algorithm can be loosely viewed as an "inside out" Euclidian algorithm, in which we subtract shifted and scaled multiples of $z^{-n} \beta_n(z^{-1})$ from $\alpha_n(z)$ and vice versa in order to drive the positive coefficients to zero, starting at the zeroth coefficient and working outward. It is thus reasonable to expect that the HGCD divide and conquer strategy of Aho, Hopcroft and Ullman

for accelerating Euclidian algorithms can be modified for our "inside out" Euclidian algorithm. The resulting scheme retains the strongly non-singular constraint of Levinson recursion, but its structure is more intuitive than that of Gustavson and Yun. Furthermore, it directly generalizes to almost-Toeplitz matrices. For exactly Toeplitz matrices, our method requires approximately $16N \log^2 N$ operations and about $10N$ storage, and is thus faster than Levinson recursion for $N > 2500$.

In this section we will treat the case of exactly Toeplitz matrices. Let us first put the Levinson recursion on the predictor polynomials $A_n(z)$, $B_n(z)$ into matrix polynomial form as follows:

$$\begin{pmatrix} A_n(z) \\ z^{-n} B_n(z^{-1}) \end{pmatrix} = \vartheta_{n,n-1} \begin{pmatrix} A_{n-1}(z) \\ z^{-(n-1)} B_{n-1}(z^{-1}) \end{pmatrix} \quad (13.2)$$

where $\vartheta_{n,n-1} = \begin{pmatrix} 1 & \xi_n z^{-1} \\ \nu_n & z^{-1} \end{pmatrix}$

The matrix $\vartheta_{n,n-1}$ transforms the $(n-1)^{th}$ order polynomials into the n^{th} order polynomials. Let us define $\vartheta_{n,m}$ for $n > m$ by:

$$\vartheta_{n,m} = \vartheta_{n,n-1} \vartheta_{n-1,n-2} \cdots \vartheta_{m+1,m} \quad (13.3)$$

Multiplying this matrix $\vartheta_{n,m}$ by the m^{th} order polynomials $A_m(z)$, $z^{-m} B_m(z^{-1})$ yields the n^{th} order polynomials $A_n(z)$, $z^{-n} B_n(z^{-1})$. In particular, for $m=0$:

$$\begin{pmatrix} A_n(z) \\ z^{-n} B_n(z^{-1}) \end{pmatrix} = \vartheta_{n,0} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad (13.4)$$

The goal of our doubling procedure will be to calculate the matrix polynomial $\vartheta_{N,0}$ and the error ε_N . Formula (13.4) will then be used to give the desired polynomials $A_N(z)$, $z^{-N} B_N(z^{-1})$, and R_N^{-1} can be constructed using the Gohberg-Semencul formula (2.18).

Multiply both sides of (13.2) by $R(z)$ to get a matrix polynomial form of the

fast Choleski recursion:

$$\begin{bmatrix} \alpha_n(z) \\ z^{-n} \beta_n(z^{-1}) \end{bmatrix} = \vartheta_{n,m} \begin{bmatrix} \alpha_m(z) \\ z^{-m} \beta_m(z^{-1}) \end{bmatrix} \quad (13.5)$$

Note that the fast Choleski algorithm uses exactly the same recursion as the Levinson algorithm. The important point is that it is easy to derive the matrices $\vartheta_{n,m}$ in terms of the coefficients of the polynomials $\alpha_m(z)$, $z^{-m} \beta_m(z^{-1})$. To calculate $\vartheta_{n,n-1}$ and ε_n , all we need are the n^{th} order reflection coefficients ξ_n , ν_n . Given $\alpha_{n-1}(z)$ and $\beta_{n-1}(z)$, these can be calculated as follows:

$$\begin{aligned} \varepsilon_{n-1} &= \alpha_{0,n-1} = \beta_{0,n-1} \\ \xi_n &= -\frac{\alpha_{n,n-1}}{\varepsilon_{n-1}} \\ \nu_n &= -\frac{\beta_{n,n-1}}{\varepsilon_{n-1}} \\ \vartheta_{n,n-1} &= \begin{bmatrix} 1 & \xi_n z^{-1} \\ \nu_n & z^{-1} \end{bmatrix} \\ \varepsilon_n &= \varepsilon_{n-1}(1 - \xi_n \nu_n) \end{aligned} \quad (13.6)$$

All we need to calculate $\vartheta_{n,n-1}$ and ε_n , therefore, are the 4 coefficients $\alpha_{0,n-1}$, $\alpha_{n,n-1}$, $\beta_{0,n-1}$, $\beta_{n,n-1}$. In a similar manner, to calculate $\vartheta_{n,m}$ for $n-m > 1$, we will need the values of the $(m+1)^{\text{th}}$ through n^{th} order reflection coefficients. These can be calculated solely from knowledge of the $4(n-m)$ coefficients $\alpha_{j,m}$, $\beta_{j,m}$ for $j = -(n-m)+1, \dots, 0$ and $j = m+1, \dots, n$.

To calculate $\vartheta_{n,m}$ for $n-m > 1$, we will use a divide and conquer strategy. We start with our $4(n-m)$ coefficients of $\alpha_m(z)$, $\beta_m(z)$. Let $l = \lceil \frac{1}{2}(n+m) \rceil$ be the smallest integer greater than $\frac{1}{2}(n+m)$. The divide and conquer strategy will be to compute $\vartheta_{l,m}$ and $\vartheta_{n,l}$ separately, and then form $\vartheta_{n,m} = \vartheta_{n,l} \vartheta_{l,m}$.

Step 1: Calculate $\vartheta_{l,m}$ by calling our doubling procedure recursively with the coefficients:

$$\alpha_{-(l-m)+1,m} \cdots \alpha_{0,m} \quad \alpha_{m+1,m} \cdots \alpha_{l,m} \quad (13.7)$$

$$\beta_{-(l-m)+1,m} \cdots \beta_{0,m} \quad \beta_{m+1,m} \cdots \beta_{l,m}$$

Now in order to calculate $\vartheta_{n,l}$ we will need the appropriate coefficients of $\alpha_l(z)$ and $\beta_l(z)$.

Step 2: Calculate the coefficients:

$$\begin{array}{l} \alpha_{-(n-l)+1,l} \cdots \alpha_{0,l} \quad \alpha_{l+1,l} \cdots \alpha_{n,l} \\ \beta_{-(n-l)+1,l} \cdots \beta_{0,l} \quad \beta_{l+1,l} \cdots \beta_{n,l} \end{array} \quad (13.8)$$

by computing:

$$\begin{pmatrix} \alpha_l(z) \\ z^{-l} \beta_l(z^{-1}) \end{pmatrix} = \vartheta_{l,m} \begin{pmatrix} \alpha_m(z) \\ z^{-m} \beta_m(z^{-1}) \end{pmatrix} \quad (13.9)$$

The coefficients of $\vartheta_{l,m}$ are polynomials of degree no larger than $l-m$. Thus to calculate the values of $\alpha_{l+1,l}, \dots, \alpha_{n,l}$, for example, we will only need the coefficients $\alpha_{m+1,m}, \dots, \alpha_{n,m}$ and $\beta_{-(n-m)+1,m}, \dots, \beta_{0,m}$. Moreover, since multiplying two polynomials is equivalent to convolving their coefficients, it will be possible to compute $\alpha_{l+1,l}, \dots, \alpha_{n,l}$ by taking an $(n-m+1)$ point FFT of $\vartheta_{l,m}$, multiplying by $(n-m+1)$ point FFT's of $\alpha_{m+1,m}, \dots, \alpha_{n,m}$ and of $\beta_{-(n-m)+1,m}, \dots, \beta_{0,m}$, and then calculating an inverse $(n-m+1)$ point FFT. The other needed coefficients of $\alpha_l(z)$ and $\beta_l(z)$ can be computed similarly.

Step 3: Calculate $\vartheta_{n,l}$ and ε_n by calling our doubling procedure recursively with the coefficients of $\alpha_l(z)$ and $\beta_l(z)$ computed on step 2.

Step 4: Compute $\vartheta_{n,m} = \vartheta_{n,l} \vartheta_{l,m}$

Multiplying these two matrices simply involves multiplying $(n-l)^{th}$ degree polynomials times $(l-m)^{th}$ degree polynomials to give $(n-m)^{th}$ degree polynomials. Once again, since polynomial multiplication is equivalent to convolution of the polynomial coefficients, these polynomials can be computed using $(n-m+1)$ point FFT's.

Now that we have defined how to recursively compute $\vartheta_{n,m}$ for $n-m=1$ and

for $n-m > 1$, we can state the entire doubling algorithm for calculating R_N^{-1} :

- a) Initialization: $\alpha_0(z) = \beta_0(z^{-1}) = R(z)$
- b) Calculate $\vartheta_{N,0}$ and ε_N by our 4 step doubling algorithm
- c) Compute $A_N(z)$ and $B_N(z)$ from $\vartheta_{N,0}$ using (13.4)
- d) Construct R_N^{-1} from the Gohberg-Semencul formula (2.18)

Note that to solve $\underline{x}_N = R_N^{-1} \underline{y}_N$ it is not necessary to multiply out the matrices in the Gohberg-Semencul formula for R_N^{-1} . Multiplying a triangular Toeplitz matrix by a vector is equivalent to convolving the $(N+1)$ elements of the matrix with the $(N+1)$ elements of the vector. With some care, we can thus compute \underline{x}_N using eight $(2N+1)$ point FFT's in about $16N \log N$ operations.

Total computation time is dominated by the time required to compute $\vartheta_{N,0}$. Let $C(n-m)$ be the computational cost for generating $\vartheta_{n,m}$. In step 1 of our doubling procedure, we will compute $\vartheta_{\frac{N}{2},0}$ in $C(\frac{N}{2})$ operations. Step 2 can be solved using four $(N+1)$ point FFT's of the entries of $\vartheta_{\frac{N}{2},0}$, four $(N+1)$ point FFT's of N positive and N negative coefficients of $\alpha_0(z)$, $\beta_0(z)$, four $(N+1)/2$ point complex vector multiplies and adds, and four inverse FFT's to compute the needed $\frac{N}{2}$ positive and $\frac{N}{2}$ negative coefficients of $\alpha_{\frac{N}{2}}(z)$ and $\beta_{\frac{N}{2}}(z)$. Step 3 computes $\vartheta_{\frac{N}{2},\frac{N}{2}}$ recursively in $C(\frac{N}{2})$ operations. If we have saved the FFT's of $\vartheta_{\frac{N}{2},0}$ from step 2, then step 4 only needs four $(N+1)$ point FFT's of the entries of $\vartheta_{\frac{N}{2},\frac{N}{2}}$, eight $(N+1)/2$ point complex vector multiplies and adds, and four inverse FFT's.

With some cleverness, this computation time can be further reduced. For

example, the last step of the calculation of $\vartheta_{\frac{N}{2},0}$ (and $\vartheta_{N,\frac{N}{2}}$) computes the approximately $\frac{N}{2}$ point transforms of $\vartheta_{\frac{N}{2},0}$ (and $\vartheta_{N,\frac{N}{2}}$), and then inverse transforms them. The very next step is to take the approximately N point transforms of $\vartheta_{\frac{N}{2},0}$ and $\vartheta_{N,\frac{N}{2}}$ in preparation for further computation. Since the $\frac{N}{2}$ point transform contains all the even samples of the N point transforms, it is actually only necessary to compute the odd samples of the N point transforms. This reduces the computational effort for computing the eight $\approx N$ point FFT's of $\vartheta_{\frac{N}{2},0}$ and $\vartheta_{N,\frac{N}{2}}$ by about half. Thus:

$$\begin{aligned} C(N) &\approx 2C\left(\frac{N}{2}\right) + 16N\log N \\ &= 16N\log N + 2\left[16\frac{N}{2}\log\frac{N}{2}\right] + 4\left[16\frac{N}{4}\log\frac{N}{4}\right] + \dots \\ &\approx 16N\log^2 N \end{aligned} \tag{13.10}$$

This algorithm will thus be faster than Levinson recursion for $N > 2000$.

We will need $2(2N+1)$ storage locations for saving $\alpha_0(z)$ and $\beta_0(z)$ until $\alpha_{\frac{N}{2},0}$ and $\beta_{\frac{N}{2},0}$ are calculated. Also a workspace of about $4(N+1)$ locations will be needed for computing $\vartheta_{\frac{N}{2},0}$, plus $4(N+1)$ more locations for computing $\vartheta_{N,\frac{N}{2}}$. However, if the only purpose for computing $\vartheta_{N,0}$ is as a means for getting the Levinson-Szegö polynomials $A_N(z)$ and $B_N(z)$, then as soon as $\vartheta_{\frac{N}{2},0}$ is computed, we could calculate $A_{\frac{N}{2}}(z)$ and $B_{\frac{N}{2}}(z)$ and discard $\vartheta_{\frac{N}{2},0}$. Then when $\vartheta_{N,\frac{N}{2}}$ has been found, we simply multiply by $A_{\frac{N}{2}}(z)$ and $B_{\frac{N}{2}}(z)$ to get $A_N(z)$, $B_N(z)$. This will cut the necessary storage to approximately $10N$ locations.

Further optimization will be useful for choosing reasonable sizes for the

FFT's. For example, it may be worthwhile to split the computation of $\mathfrak{V}_{N,0}$ into three or more phases in order to efficiently use available storage or FFT sizes. Also, beware also that choosing $N+1$ to be a power of 2 is a poor idea, since the transforms on the next lower level will need to be length $\frac{N}{2}+1$, which will be slightly too large for the next smaller size transform.

14. Doubling Algorithms - Almost-Toeplitz Matrices

Except for one minor difficulty, the doubling procedure presented in the previous section can be applied directly to the almost-Toeplitz matrix algorithms. The problem is that the doubling algorithm for exactly Toeplitz matrices relies heavily on the exact symmetry between the fast Choleski and Levinson algorithms. By accelerating the fast Choleski algorithm we simultaneously accelerate the Levinson algorithm. The almost-Toeplitz problem has a similar symmetry between the fast Choleski and the extended Levinson-style algorithm, except that the reflection coefficients ν_n^1 and μ_n^1 needed by the Levinson-style algorithm do not appear in the fast Choleski algorithm. To get around this problem we will have to restrict our attention to almost-Toeplitz matrices R_N for which the values of ν_n^1 and μ_n^1 can be deduced from the other reflection coefficients. For example, suppose R_N could be written as the sum of a Toeplitz matrix T plus products of lower and upper triangular Toeplitz matrices:

$$R_N = T + \sum_{i=3}^K L(\underline{c}_N^i) U(\underline{d}_N^{iT}) \quad (14.1)$$

where $T_{ii} = t(0) \neq 0$. To put this into our usual lower \times upper representation, choose:

$$\underline{c}_N^1 = \frac{1}{\sqrt{t(0)}} \begin{bmatrix} t(0) \\ \vdots \\ t(N) \end{bmatrix} \quad \underline{d}_N^1 = \frac{1}{\sqrt{t(0)}} \begin{bmatrix} t(0) \\ \vdots \\ t(-N) \end{bmatrix} \quad (14.2)$$

$$\underline{c}_N^2 = \frac{1}{\sqrt{t(0)}} \begin{pmatrix} 0 \\ t(1) \\ \vdots \\ t(N) \end{pmatrix} \quad \underline{d}_N^2 = \frac{1}{\sqrt{t(0)}} \begin{pmatrix} 0 \\ t(-1) \\ \vdots \\ t(-N) \end{pmatrix}$$

Note that if R_N can be represented in this way, then ν_n^1 and μ_n^1 do not have to be computed independently, but can be found from:

$$\begin{aligned} \nu_n^1 &= \nu_n^2 = \tilde{\nu}_n^2 \\ \mu_n^1 &= \mu_n^2 = \tilde{\mu}_n^2 \end{aligned} \tag{14.3}$$

In some cases R_N does not have the form (14.1), and cannot be put into any other form in which ν_n^1 and μ_n^1 can be deduced from the other reflection coefficients. Then as a last resort, we can consider using a non-minimal representation for R_N so that \underline{c}_N^1 is linearly dependent on the other \underline{c}_N^i vectors, and \underline{d}_N^1 is linearly dependent on the other \underline{d}_N^i vectors. When this is done, ν_n^1 and μ_n^1 can be computed as appropriate linear combinations of the other ν_n^i and μ_n^i coefficients.

With this caveat, we now develop a doubling procedure for the case of almost-Toeplitz matrices. Define the polynomials $\alpha_n(z)$, $\beta_n(z)$, $\varphi_n^i(z)$, $\psi_n^i(z)$ and $A_n(z)$, $B_n(z)$, $f_n^i(z)$, $e_n^i(z)$ and $C^i(z)$, $D^i(z)$ in the obvious ways. Then the fast Choleski recursion can be stated in matrix polynomial form as follows:

$$\begin{pmatrix} \beta_n(z) \\ \varphi_n^2(z) \\ \vdots \\ \varphi_n^\kappa(z) \end{pmatrix} = \tilde{\mathcal{V}}_{n,n-1} \begin{pmatrix} \beta_{n-1}(z) \\ \varphi_{n-1}^2(z) \\ \vdots \\ \varphi_{n-1}^\kappa(z) \end{pmatrix} \tag{14.4}$$

$$\text{where } \tilde{\mathcal{V}}_{n,n-1} = \begin{bmatrix} \begin{matrix} \varepsilon_n \\ \xi_n^2 \\ \vdots \\ \xi_n^\kappa \end{matrix} \\ \frac{1}{\varepsilon_n} \end{bmatrix} \left(z^{-1} \mathcal{V}_n^2 \cdots \mathcal{V}_n^\kappa \right) + \begin{bmatrix} 0 & 0 \\ 0 & I \end{bmatrix}$$

$$\begin{pmatrix} \alpha_n(z) \\ \psi_n^2(z) \\ \vdots \\ \psi_n^\kappa(z) \end{pmatrix} = \mathcal{V}_{n,n-1} \begin{pmatrix} \alpha_{n-1}(z) \\ \psi_{n-1}^2(z) \\ \vdots \\ \psi_{n-1}^\kappa(z) \end{pmatrix} \tag{14.5}$$

$$\text{where } \tilde{\mathfrak{D}}_{n,n-1} = \left[\frac{1}{\varepsilon_n} \begin{pmatrix} \varepsilon_n \\ \rho_n^2 \\ \vdots \\ \rho_n^\kappa \end{pmatrix} \begin{pmatrix} z^{-1} \tilde{\mu}_n^2 & \dots & \tilde{\mu}_n^\kappa \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & I \end{pmatrix} \right]$$

The matrices $\tilde{\mathfrak{V}}_{n,n-1}$, $\tilde{\mathfrak{D}}_{n,n-1}$ transform the $(n-1)^{th}$ order polynomials into the n^{th} order polynomials. Define $\tilde{\mathfrak{V}}_{n,m}$ and $\tilde{\mathfrak{D}}_{n,m}$ for $n > m$ by:

$$\begin{aligned} \tilde{\mathfrak{V}}_{n,m} &= \tilde{\mathfrak{V}}_{n,n-1} \tilde{\mathfrak{V}}_{n-1,n-2} \cdots \tilde{\mathfrak{V}}_{m+1,m} \\ \tilde{\mathfrak{D}}_{n,m} &= \tilde{\mathfrak{D}}_{n,n-1} \tilde{\mathfrak{D}}_{n-1,n-2} \cdots \tilde{\mathfrak{D}}_{m+1,m} \end{aligned} \quad (14.6)$$

These polynomial matrices $\tilde{\mathfrak{V}}_{n,m}$ and $\tilde{\mathfrak{D}}_{n,m}$ have degree $(n-m)$ and they transform the m^{th} degree polynomials into the n^{th} degree polynomials.

The Levinson-style recursion can also be stated in a similar form:

$$\begin{pmatrix} 1 \\ z^{-n} B_n(z^{-1}) \\ f_n^2(z) \\ \vdots \\ f_n^\kappa(z) \end{pmatrix} = \mathfrak{V}_{n,n-1} \begin{pmatrix} 1 \\ z^{-(n-1)} B_{n-1}(z^{-1}) \\ f_{n-1}^2(z) \\ \vdots \\ f_{n-1}^\kappa(z) \end{pmatrix} \quad (14.7)$$

$$\text{where } \mathfrak{V}_{n,n-1} = \begin{pmatrix} 1 & 0 \\ \nu_n^1 / d_0^1 & \tilde{\mathfrak{V}}_{n,n-1} \\ 0 & \tilde{\mathfrak{V}}_{n,n-1} \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ z^{-n} A_n(z^{-1}) \\ e_n^2(z) \\ \vdots \\ e_n^\kappa(z) \end{pmatrix} = \omega_{n,n-1} \begin{pmatrix} 1 \\ z^{-(n-1)} A_{n-1}(z^{-1}) \\ e_{n-1}^2(z) \\ \vdots \\ e_{n-1}^\kappa(z) \end{pmatrix} \quad (14.8)$$

$$\text{where } \omega_{n,n-1} = \begin{pmatrix} 1 & 0 \\ \mu_n^1 / c_0^1 & \tilde{\mathfrak{D}}_{n,n-1} \\ 0 & \tilde{\mathfrak{D}}_{n,n-1} \end{pmatrix}$$

Define $\mathfrak{V}_{n,m}$ and $\omega_{n,m}$ for $n > m$ by:

$$\begin{aligned} \mathfrak{V}_{n,m} &= \mathfrak{V}_{n,n-1} \mathfrak{V}_{n-1,n-2} \cdots \mathfrak{V}_{m+1,m} \\ \omega_{n,m} &= \omega_{n,n-1} \omega_{n-1,n-2} \cdots \omega_{m+1,m} \end{aligned} \quad (14.9)$$

Because $\mathfrak{V}_{n,n-1}$ and $\omega_{n,n-1}$ are block lower triangular, $\mathfrak{V}_{n,m}$ and $\omega_{n,m}$ will have

the form:

$$\begin{aligned} \vartheta_{n,m} &= \begin{pmatrix} 1 & 0 \\ * & \tilde{\vartheta}_{n,m} \end{pmatrix} \\ \omega_{n,m} &= \begin{pmatrix} 1 & 0 \\ * & \tilde{\omega}_{n,m} \end{pmatrix} \end{aligned} \quad (14.10)$$

Our strategy for calculating R_N^{-1} or \underline{x}_N is the same as in the exactly Toeplitz case. Our goal is to compute $\underline{a}_N, \underline{b}_N, \underline{f}_N^i, \underline{e}_N, \varepsilon_N, M_N^{-1}$ so that we can compute R_N^{-1} as a sum of upper times lower triangular Toeplitz matrices as in (8.8). We do this by computing $\vartheta_{N,0}$ and $\omega_{N,0}$ using an accelerated fast Choleski algorithm. The desired vectors can then be found from:

$$\begin{pmatrix} 1 \\ z^{-N} B_N(z^{-1}) \\ f_N^2(z) \\ \vdots \\ f_N^K(z) \end{pmatrix} = \vartheta_{N,0} \begin{pmatrix} 1 \\ 1 \\ c_0^2 / \varepsilon_0 \\ \vdots \\ c_0^K / \varepsilon_0 \end{pmatrix} \quad (14.11)$$

$$\begin{pmatrix} 1 \\ z^{-N} A_N(z^{-1}) \\ e_N^2(z) \\ \vdots \\ e_N^K(z) \end{pmatrix} = \omega_{N,0} \begin{pmatrix} 1 \\ 1 \\ d_0^2 / \varepsilon_0 \\ \vdots \\ d_0^K / \varepsilon_0 \end{pmatrix} \quad (14.12)$$

We again use a divide and conquer strategy to compute $\vartheta_{n,m}$ and $\omega_{n,m}$. Using our assumption about the form of R_N in (14.1), we can compute $\vartheta_{n,n-1}$ and $\omega_{n,n-1}$ from $\underline{a}_{n-1}, \underline{b}_{n-1}, \underline{\varphi}_{n-1}^i, \underline{\psi}_{n-1}^i, M_{n-1}^{-1}$:

$$\begin{aligned} \xi_n^i &= -\varphi_{n,n-1}^i \\ \rho_n^i &= -\psi_{n,n-1}^i \\ \begin{pmatrix} \mathcal{V}_n^2 \\ \vdots \\ \mathcal{V}_n^K \end{pmatrix} &= M_{n-1}^{-1} \begin{pmatrix} \rho_n^2 \\ \vdots \\ \rho_n^K \end{pmatrix} \\ \begin{pmatrix} \tilde{\mu}_n^2 \\ \vdots \\ \tilde{\mu}_n^K \end{pmatrix} &= M_{n-1}^{-T} \begin{pmatrix} \xi_n^2 \\ \vdots \\ \xi_n^K \end{pmatrix} \end{aligned} \quad (14.13)$$

$$\varepsilon_{n-1} = \alpha_{n-1,n-1} = \beta_{n-1,n-1}$$

$$M_n^{-1} = M_{n-1}^{-1} - \frac{1}{\varepsilon_{n-1}} \begin{pmatrix} \mathcal{V}_n^2 \\ \vdots \\ \mathcal{V}_n^\kappa \end{pmatrix} \begin{pmatrix} \tilde{\mu}_n^2 & \dots & \tilde{\mu}_n^\kappa \end{pmatrix}$$

$$\varepsilon_n = \varepsilon_{n-1} - \sum_{i=2}^{\kappa} \xi_n^i \mathcal{V}_n^i$$

Calculate ν_n^1 and μ_n^1 from the other reflection coefficients as in (14.3)

Compute $\vartheta_{n,n-1}$ and $\omega_{n,n-1}$ from (14.4,5) and (14.7,8)

Note that computing $\vartheta_{n,n-1}$ and $\omega_{n,n-1}$ only requires knowledge of the 2κ values $\alpha_{n-1,n-1}, \beta_{n-1,n-1}, \varphi_{n,n-1}^i, \psi_{n,n-1}^i$.

Now to compute $\vartheta_{n,m}$ and $\omega_{n,m}$ for $n-m > 1$ we will need to start with ε_m , M_m^{-1} and the $2\kappa(n-m)$ coefficients $\alpha_{j-1,m}, \beta_{j-1,m}, \varphi_{j,m}^i, \psi_{j,m}^i$ for $j=m+1, \dots, n$. Let $l = \lfloor \frac{1}{2}(n+m) \rfloor$. Then:

Step 1: Compute $\vartheta_{l,m}, \omega_{l,m}$ and M_l^{-1}, ε_l by calling the procedure recursively with coefficients

$$\begin{aligned} & \alpha_{j-1,m} \cdot \beta_{j,m} \cdot \varphi_{j,m}^i \cdot \psi_{j,m}^i && \text{for } j=m+1, \dots, n \\ & M_m^{-1} \cdot \varepsilon_m && \text{for } i=2, \dots, \kappa \end{aligned} \quad (14.14)$$

Step 2: Compute the coefficients $\alpha_{j-1,l}, \beta_{j-1,l}, \varphi_{j,l}^i, \psi_{j,l}^i$ for $j=l+1, \dots, n$ by exploiting the relationships:

$$\begin{pmatrix} \beta_l(z) \\ \varphi_l^2(z) \\ \vdots \\ \varphi_l^\kappa(z) \end{pmatrix} = \tilde{\vartheta}_{l,m} \begin{pmatrix} \beta_m(z) \\ \varphi_m^2(z) \\ \vdots \\ \varphi_m^\kappa(z) \end{pmatrix} \quad (14.15)$$

$$\begin{pmatrix} \alpha_l(z) \\ \psi_l^2(z) \\ \vdots \\ \psi_l^\kappa(z) \end{pmatrix} = \tilde{\omega}_{l,m} \begin{pmatrix} \alpha_m(z) \\ \psi_m^2(z) \\ \vdots \\ \psi_m^\kappa(z) \end{pmatrix}$$

Step 3: Compute $\vartheta_{n,l}, \omega_{n,l}$, and M_n^{-1}, ε_n by calling the procedure recursively with the coefficients calculated in step 2.

Step 4: Compute $\vartheta_{n,m} = \vartheta_{n,l} \vartheta_{l,m}$ and $\omega_{n,m} = \omega_{n,l} \omega_{l,m}$

Step 2 can be performed using $(n-m+1)$ point FFT's to convolve the $(l-m)$ degree polynomial entries of $\vartheta_{l,m}$ and $\omega_{l,m}$ with the appropriate elements of the right hand sides of (14.5). Step 4 can also be performed using $(n-m+1)$ point FFT's to convolve the $(n-l)$ and $(l-m)$ degree polynomial entries of $\vartheta_{n,l}$, $\omega_{n,l}$ and $\vartheta_{l,m}$, $\omega_{l,m}$. In all, steps 2 and 4 will require about $(6\kappa^2+8\kappa)$ FFT's of length $(n-m+1)$. Computing $\vartheta_{N,0}$ and $\omega_{N,0}$ will thus require about $(6\kappa^2+8\kappa)N \log^2 N$ operations. The vectors \underline{a}_N , \underline{b}_N , \underline{f}_N^i , \underline{g}_N^i can then be found from (14.11), (14.12) in $O(N)$ operations, and the solution \underline{x}_N can be computed in $O(\kappa N \log N)$ operations using formula (8.8). Total storage required will be about $(4\kappa^2+8\kappa)N$ locations. Careful optimization can probably reduce all these requirements somewhat.

15. Degeneracy

The Levinson-style, the fast Choleski, and the doubling algorithms will all fail unless the matrix R_N is strongly non-singular so that the prediction error ε_n is always non-zero. One practical solution to this difficulty, if the prediction error ε_n should hit zero at some stage, would be to simply perturb the matrix R_N slightly in order to drive ε_n slightly away from zero. More elegant methods for dealing with this difficulty, however, can be devised. The strongly non-singular constraint arises from Levinson recursion's inflexible strategy of solving a series of problems $R_n \underline{x}_n = \underline{y}_n$ in which the matrix R_n is always the n^{th} principal minor of R_N . A very desirable solution to the problem of zero prediction error, therefore, would be to incorporate some form of partial pivoting into the Levinson algorithm so that the sequence of nested minors R_n do not necessarily lie along the main diagonal. Musicus has in fact presented a Euclidian algorithm replacement for *inverse* Levinson recursion which has exactly this struc-

ture. Unfortunately, this solution does not appear to be easily applicable to the forward Levinson recursion algorithms that we have discussed in this paper. A less desirable solution to the problem has been suggested by Gustavson and Yun,¹⁴ whose algorithm effectively uses the lower left minors of R_N to recursively construct $\underline{a}_N, \underline{b}_N$. Bareiss also suggested a fix for one type of degeneracy in the Choleski algorithms. We present yet another approach, a "shifting" procedure which resumes the Levinson recursion when the prediction error ε_n at some stage is exactly zero. Our approach will not cure the problems of numerical ill-conditioning which occur when ε_n is very tiny but non-zero, but it should be regarded as a first attempt toward a more general procedure. The Levinson-style almost-Toeplitz algorithm can also be patched up to handle the case when $\varepsilon_n=0$, but the method requires adding an extra predictor, and is sufficiently inelegant that we do not present the details

Assume that R_N is exactly Toeplitz, and is non-singular but not necessarily strongly non-singular. We will then modify Levinson recursion so that at the n^{th} stage we calculate $(n+\tau_n)$ long vectors \underline{a}_n and \underline{b}_n satisfying:

$$R_N \begin{pmatrix} \underline{a}_n \\ \underline{0} \end{pmatrix} = \begin{pmatrix} \varepsilon_n \\ \underline{0}_n \\ * \\ \vdots \end{pmatrix} \quad ; \quad R_N \begin{pmatrix} \underline{b}_n \\ \underline{0} \end{pmatrix} = \begin{pmatrix} \underline{0}_n \\ 1 \\ * \\ \vdots \end{pmatrix} \quad (15.1)$$

where the first τ_n coefficients of \underline{a}_n and \underline{b}_n are zero:

$$\begin{aligned} \underline{a}_n &= (\underline{0}_{\tau_n}^T \ 1 \ a_{1,n} \ \cdots \ a_{n,n})^T \\ \underline{b}_n &= (\underline{0}_{\tau_n}^T \ b_{n,n} \ \cdots \ b_{0,n})^T \end{aligned} \quad (15.2)$$

We start at $n=0$. Let τ_0 be the smallest positive integer $0 \leq \tau_0 \leq N$ such that $\tau(-\tau_0) \neq 0$. Then choose:

$$\underline{a}_0 = \begin{pmatrix} \underline{0}_{\tau_0} \\ 1 \end{pmatrix} \quad \underline{b}_0 = \begin{pmatrix} \underline{0}_{\tau_0} \\ 1 \\ \varepsilon_0 \end{pmatrix} \quad \varepsilon_0 = \tau(-\tau_0) \quad (15.3)$$

Clearly there must be such an integer τ_0 or else the entire first row of R_N will be zero and R_N would be singular. (In most cases, $\tau_0=0$ and the initial choices above are similar to the initial choices of the usual Levinson algorithm.) Now we can calculate the n^{th} order solutions $\underline{a}_n, \underline{b}_n$ in terms of the $(n-1)^{th}$ order solutions in the following way:

$$\begin{aligned} \underline{a}_n &= \begin{pmatrix} \underline{a}_{n-1} \\ 0 \end{pmatrix} + \xi_n \begin{pmatrix} 0 \\ \underline{b}_{n-1} \end{pmatrix} \\ \underline{b}_n &= \begin{pmatrix} 0 \\ \underline{b}_{n-1} \end{pmatrix} + \frac{\nu_n}{\varepsilon_n} \underline{a}_n \end{aligned} \quad (15.4)$$

where:

$$\begin{aligned} \xi_n &= - \sum_{j=0}^{n-1} r(n-j) a_{j,n-1} \\ \nu_n &= - \sum_{j=1}^n r(-j-\tau_{n-1}) b_{n-j,n-1} \\ \varepsilon_n &= \varepsilon_{n-1} - \xi_n \nu_n \end{aligned} \quad (15.5)$$

Trouble arises in this algorithm at some stage m if when we compute \underline{a}_m from $\underline{a}_{m-1}, \underline{b}_{m-1}$ in (15.4) we find that $\varepsilon_m=0$. This makes it impossible to calculate \underline{b}_m . One possible solution to this deadlock is to try shifting the coefficients in the vector \underline{a}_m down σ steps until we find a $(m+\tau_{n-1}+\sigma)$ long vector $\begin{pmatrix} 0_\sigma \\ \underline{a}_m \end{pmatrix}$ with $\tau_m = \tau_{m-1} + \sigma$ leading zeroes such that:

$$R_N \begin{pmatrix} 0_\sigma \\ \underline{a}_m \\ \underline{0} \end{pmatrix} = \begin{pmatrix} \varepsilon_{m+\sigma} \\ 0_{m+\sigma} \\ * \\ \vdots \end{pmatrix} \quad (15.6)$$

$$\text{where } \varepsilon_{m+\sigma} = \left[r(0) \cdots r(-n) \right] \begin{pmatrix} 0_\sigma \\ \underline{a}_m \end{pmatrix} \neq 0$$

Clearly there must be such a $\sigma \leq N-m-\tau_{m-1}$ for which we would find $\varepsilon_{m+\sigma} \neq 0$, or else we would find that

$$R_N \begin{pmatrix} 0_{N-m-\tau_{m-1}} \\ \underline{a}_m \end{pmatrix} = \underline{0} \quad (15.7)$$

with $\underline{a}_m \neq 0$, which would contradict our assumption that R_N is non-singular. Let us define the errors $\varepsilon_n = \varepsilon_{m+\sigma}$ for $n = m, \dots, m+\sigma$, and let us define the $(n + \tau_m)$ long vectors \underline{a}_n by extending \underline{a}_m with zeroes:

$$\underline{a}_n = \begin{pmatrix} 0_\sigma \\ \underline{a}_m \\ 0_{n-m} \end{pmatrix} \quad (15.8)$$

Extend \underline{b}_{m-1} with σ zeroes, $\underline{b}_{m-1} \leftarrow \begin{pmatrix} \underline{b}_{m-1} \\ 0_\sigma \end{pmatrix}$ and now recursively generate the $(n + \tau_m)$ long vectors $\underline{b}_m, \dots, \underline{b}_{m+\sigma}$ satisfying (15.1) by the following recursion:

For $n = m, \dots, m + \sigma$

$$\underline{b}_n = \begin{pmatrix} 0 \\ \underline{b}_{n-1} \end{pmatrix} + \frac{\nu_n}{\varepsilon_n} \underline{a}_n \quad (15.9)$$

where $\nu_n = - \left[\tau(-1) \cdots \tau(-n-\sigma) \right] \underline{b}_{n-1}$

At this point we will have $(m + \sigma + \tau_m)$ long vectors $\underline{a}_{m+\sigma}$ and $\underline{b}_{m+\sigma}$ which satisfy (15.1) whose first $\tau_{m+\sigma} = \tau_m = \tau_{m-1} + \sigma$ coefficients are zero. We now resume our normal Levinson recursion step (15.4) at $n = m + \sigma + 1$. Should $\varepsilon_n = 0$ again at some stage, we simply repeat our shifting step.

Eventually we will reach a stage $M = N - \tau_M$ at which the vectors $\underline{a}_M, \underline{b}_M$ will have length N so that they can no longer shift down. For the last τ_M steps we will recursively create vectors \underline{a}_n by shifting \underline{a}_{n-1} up one notch and adding an appropriate multiple of \underline{b}_M to ensure that only the last $N - M$ coefficients of $R_N \underline{a}_n$ will be non-zero. (Shifting \underline{a}_{n-1} upwards is feasible since its first $\tau_M - (n - M)$ coefficients will be zero.)

For $n = M + 1, \dots, N$

$$\underline{a}_n = \begin{pmatrix} a_{1,n-1} \\ \vdots \\ a_{N,n-1} \\ 0 \end{pmatrix} + \xi_n \underline{b}_M \quad (15.10)$$

where $\xi_n = - \sum_{j=N-n}^{N-1} \tau(M-j) a_{j+1,n-1}$

It is easy to verify that only the last $N-M$ coefficients of $R_N \underline{a}_n$ will be non-zero; call these coefficients $\alpha_{M+1,n}, \dots, \alpha_{N,n}$:

$$R_N \underline{a}_n = \begin{pmatrix} 0_{M+1} \\ \alpha_{M+1,n} \\ \vdots \\ \alpha_{N,n} \end{pmatrix} \quad \text{for } n=M+1, \dots, N \quad (15.11)$$

Now to solve for the vector \underline{x}_N , we can start by using our usual algorithm for the first M steps:

$$\begin{aligned} \underline{x}_{-1} &= \underline{0}_N \\ \text{For } n=0, \dots, M & \\ \underline{x}_n &= \underline{x}_{n-1} + \lambda_n \underline{b}_n \\ \text{where } \lambda_n &= y_n - \sum_{j=0}^n r(n-j)x_{j,n-1} \end{aligned} \quad (15.12)$$

At this point the vector \underline{x}_M will satisfy:

$$R_N \underline{x}_M = \underline{y}_N - \begin{pmatrix} 0_{M+1} \\ \lambda_{M+1} \\ \vdots \\ \lambda_N \end{pmatrix} \quad (15.13)$$

Computing the final solution \underline{x}_N will now require adding appropriate multiples of $\underline{a}_M, \dots, \underline{a}_N$ to \underline{x}_M .

$$\underline{x}_N = \underline{x}_M + \sum_{n=M+1}^N \eta_n \underline{a}_n \quad (15.14)$$

where the coefficients η_n solve:

$$\begin{bmatrix} \alpha_{M+1,M+1} & \cdots & \alpha_{M+1,N} \\ \vdots & \ddots & \vdots \\ \alpha_{N,M+1} & \cdots & \alpha_{N,N} \end{bmatrix} \begin{bmatrix} \eta_{M+1} \\ \vdots \\ \eta_N \end{bmatrix} = \begin{bmatrix} \lambda_{M+1} \\ \vdots \\ \lambda_N \end{bmatrix} \quad (15.15)$$

This matrix on the left of (15.15) can be shown to have full rank because the vectors $\underline{a}_{M+1}, \dots, \underline{a}_N$ will be linearly independent.

This particular algorithm can be viewed as an extension of an idea by Bareiss⁶ for dealing with the case when $r(0) \neq 0$. It has an interesting

interpretation in terms of non-principal minors of R_N . Suppose that step n of the algorithm is a "normal" step using the recursion in (15.4). Then because the first τ_n coefficients of $\underline{a}_n, \underline{b}_n$ are zero, the last n coefficients of these vectors must satisfy:

$$\begin{aligned} R_n^{\tau_n} \begin{pmatrix} a_{\tau_n, n} \\ \vdots \\ a_{n, n} \end{pmatrix} &= \begin{pmatrix} \varepsilon_n \\ \underline{0}_n \end{pmatrix} \\ R_n^{\tau_n} \begin{pmatrix} b_{n, n} \\ \vdots \\ b_{0, n} \end{pmatrix} &= \begin{pmatrix} \underline{0}_n \\ 1 \end{pmatrix} \end{aligned} \quad (15.16)$$

where:

$$R_n^{\tau_n} = \begin{pmatrix} r(-\tau_n) & \cdots & r(-n-\tau_n) \\ \vdots & \ddots & \vdots \\ r(n-\tau_n) & \cdots & r(-\tau_n) \end{pmatrix} \quad (15.17)$$

In other words, the last n coefficients of the vectors $\underline{a}_n, \underline{b}_n$ satisfy a set of linear equations involving a non-principal minor $R_n^{\tau_n}$ of R_N . This modified Levinson recursion therefore starts by solving linear equations involving principal minors R_m of R_N . If one of the minors R_m happens to be singular, however, the algorithm effectively shifts off the main diagonal to a non-principal minor $R_{m+\sigma}^\sigma$ and resumes the iteration.

16. Example - Covariance Method of Linear Prediction

We first consider a problem originally treated by Morf, Dickinson, Kailath and Vieira.²⁸ The covariance method of linear prediction fits an optimal p^{th} order linear predictor to a given N point data sequence $x(0), \dots, x(N-1)$ by minimizing:

$$E \leftarrow \min_{\alpha_j} \frac{1}{N-p} \sum_{k=p}^{N-1} \left[x(k) + \alpha_1 x(k-1) + \cdots + \alpha_p x(k-p) \right]^2 \quad (16.1)$$

Since this is quadratic in the unknowns α_j , minimizing (16.1) is equivalent to

solving:

$$R_p \begin{pmatrix} \underline{a}_p \\ \vdots \\ \underline{a}_1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ E \end{pmatrix} \quad (16.2)$$

$$\text{where: } [R_p]_{kj} = \frac{1}{N-p} \sum_{k=0}^{N-p-1} w(k+i)w(k+j)$$

This matrix R_p can be written in the form of a symmetric almost-Toeplitz matrix with displacement rank $\kappa=4$:

$$R_p = \begin{pmatrix} 0 & 0 \\ 0 & R_{p-1} \end{pmatrix} + \underline{c}_N^1 \underline{c}_N^{1T} - \underline{c}_N^2 \underline{c}_N^{2T} + \underline{c}_N^3 \underline{c}_N^{3T} - \underline{c}_N^4 \underline{c}_N^{4T} \quad (16.3)$$

with

$$\underline{c}_p^1 = \frac{1}{\sqrt{R_{0,0}}} \begin{pmatrix} R_{0,0} \\ R_{1,1} \\ \vdots \\ R_{p,0} \end{pmatrix} \quad \underline{c}_p^2 = \frac{1}{\sqrt{R_{0,0}}} \begin{pmatrix} 0 \\ R_{1,0} \\ \vdots \\ R_{p,0} \end{pmatrix} \quad (16.4)$$

$$\underline{c}_p^3 = \begin{pmatrix} 0 \\ x(N-p) \\ \vdots \\ x(N-1) \end{pmatrix} \quad \underline{c}_p^4 = \begin{pmatrix} 0 \\ x(0) \\ \vdots \\ x(p-1) \end{pmatrix}$$

We now simply apply our symmetric almost-Toeplitz algorithm to calculate \underline{a}_n , \underline{e}_p^i . After p steps, the vector \underline{a}_p will solve (16.2), and will thus be the desired predictor, with $E=\varepsilon_p$ the prediction error. Moreover, the intermediate solutions \underline{a}_n , ε_n for $n=0, \dots, p$ represent optimal n^{th} order forward predictors and prediction errors given the data $x(p-n), \dots, x(N-1)$.

In some cases, it would be convenient for the intermediate solutions \underline{a}_n to represent the optimal n^{th} order forward predictor given *all* the data $x(0), \dots, x(N-1)$. This becomes possible by choosing a mixed representation for R_p like that in (9.3). Let R_n be the covariance matrix formed as in (16.2) with p replaced by n . The matrices R_n are no longer the principal minors of

R_p , but instead satisfy:

$$R_n = \begin{bmatrix} R_{n-1} & * \\ * & * \end{bmatrix} - \begin{bmatrix} x(N-n) \\ \vdots \\ x(N-1) \\ 0 \end{bmatrix} \begin{bmatrix} x(N-n) \cdots x(N-1) & 0 \end{bmatrix} \quad (16.5)$$

$$R_n = \begin{bmatrix} * & * \\ * & R_{n-1} \end{bmatrix} - \begin{bmatrix} 0 \\ x(0) \\ \vdots \\ x(n-1) \end{bmatrix} \begin{bmatrix} 0 & x(0) \cdots x(n-1) \end{bmatrix}$$

We will need 4 predictor vectors in order to solve this problem recursively. Several selections are possible; we choose to calculate $\underline{a}_n, \underline{b}_n, \underline{e}_n, \underline{f}_n$ defined by:

$$R_n \underline{a}_n = \begin{bmatrix} 1 \\ \underline{0}_n \end{bmatrix} \quad R_n \underline{b}_n = \begin{bmatrix} \underline{0}_n \\ 1 \end{bmatrix} \quad (16.6)$$

$$R_n \underline{e}_n = \begin{bmatrix} x(N-n-1) \\ \vdots \\ x(N-1) \end{bmatrix} \quad R_n \underline{f}_n = \begin{bmatrix} x(0) \\ \vdots \\ x(n) \end{bmatrix}$$

The n^{th} order predictors can be recursively computed from the $(n-1)^{\text{th}}$ order predictors. The computation is somewhat involved, however, and so we will omit the details.

A similar approach could be used for other pre-windowing and post-windowing covariance methods, where we pad the data sequence on the left or on the right with zeroes. Padding on the left with zeroes ensures that $\underline{c}_p^4 = \underline{0}$ in (16.4), and then we need only retain 3 vectors in the recursion. Similarly, padding on the right makes $\underline{c}_p^3 = 0$. Padding on both the right and the left, as in the Yule-Walker equations, sets $\underline{c}_p^3 = \underline{c}_p^4 = 0$, and the matrix R_p becomes exactly Toeplitz with displacement rank 2.

17. Example - Modified Covariance Method

The modified covariance method of linear prediction is quite similar to the covariance method above, except that it selects an optimal predictor by averaging the forward and backward prediction errors:

$$E \leftarrow \min_{\alpha_j} \frac{1}{2(N-p)} \left[\sum_{k=p}^{N-1} \left(x(k) + \alpha_1 x(k-1) + \dots + \alpha_p x(k-p) \right)^2 + \sum_{k=0}^{N-p-1} \left(x(k) + \alpha_1 x(k+1) + \dots + \alpha_p x(k+p) \right)^2 \right] \quad (17.1)$$

This is again quadratic in the parameters α_j , and so minimizing (17.1) is equivalent to solving:

$$R_p \begin{bmatrix} \alpha_p \\ \vdots \\ \alpha_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ E \end{bmatrix} \quad (17.2)$$

$$\text{where } [R_p]_{ij} = \frac{1}{2(N-p)} \left[\sum_{k=0}^{N-p-1} x(k+i)x(k+j) + \sum_{k=p}^{N-1} x(k-i)x(k-j) \right]$$

Marple²⁵ treated this problem originally. It is possible to represent R_p as a sum of lower \times upper triangular Toeplitz matrices with displacement rank $\kappa=6$. However, Marple pointed out that if we let R_n be the modified covariance matrix for an n^{th} order model, rather than simply the n^{th} principal minor of R_p , then:

$$R_n = \begin{bmatrix} * & * \\ * & R_{n-1} \end{bmatrix} - \begin{bmatrix} 0 \\ x(0) \\ \vdots \\ x(n-1) \end{bmatrix} \begin{bmatrix} 0 & x(0) & \dots & x(n-1) \end{bmatrix} - \begin{bmatrix} 0 \\ x(N-1) \\ \vdots \\ x(N-n) \end{bmatrix} \begin{bmatrix} 0 & x(N-1) & \dots & x(N-n) \end{bmatrix} \quad (17.3)$$

$$R_n = \begin{bmatrix} R_{n-1} & * \\ * & * \end{bmatrix} - \begin{bmatrix} x(n-1) \\ \vdots \\ x(0) \\ 0 \end{bmatrix} \begin{bmatrix} x(n-1) & \dots & x(0) & 0 \end{bmatrix} - \begin{bmatrix} x(N-n) \\ \vdots \\ x(N-1) \\ 0 \end{bmatrix} \begin{bmatrix} x(N-n) & \dots & x(N-1) & 0 \end{bmatrix}$$

This partitioning can be exploited to produce a fast recursive algorithm for computing the modified covariance predictor. We will need 6 different predictors; let us use:

$$\begin{aligned}
 R_n \underline{a}_n &= \begin{pmatrix} 1 \\ \underline{0}_n \end{pmatrix} & R_n \underline{b}_n &= \begin{pmatrix} \underline{0}_n \\ 1 \end{pmatrix} \\
 R_n \underline{e}_n^1 &= \begin{pmatrix} x(0) \\ \vdots \\ x(n) \end{pmatrix} & R_n \underline{f}_n^1 &= \begin{pmatrix} x(n) \\ \vdots \\ x(0) \end{pmatrix} \\
 R_n \underline{e}_n^2 &= \begin{pmatrix} x(N-n-1) \\ \vdots \\ x(N-n) \end{pmatrix} & R_n \underline{f}_n^2 &= \begin{pmatrix} x(N-1) \\ \vdots \\ x(N-n-1) \end{pmatrix}
 \end{aligned} \tag{17.4}$$

These can be computed recursively by an updating procedure which is conceptually similar to those we have used before. The important point noted by Marple, however, is that the matrices R_n are not only symmetric about the main diagonal but are also symmetric about the secondary diagonal (i.e. they are persymmetric.) Thus $\underline{a}_n, \underline{e}_n^i$ are simply the vectors $\underline{b}_n, \underline{f}_i$ upside down. Only 3 different predictors will need to be calculated; in fact, the final algorithm he derives is almost as fast as Burg's linear prediction algorithm.

18. Example - Rational Toeplitz Matrices

A common problem in many filtering and deconvolution applications is to solve a set of linear equations:

$$S_N \underline{x}_N = \underline{y}_N \tag{18.1}$$

where the $(N+1) \times (N+1)$ matrix S_N is exactly Toeplitz and its elements $S_{i,j} = s(i-j)$ form the inverse Z-transform of a rational polynomial:

$$S(z) = \sum_{n=-\infty}^{\infty} s(n)z^{-n} = \frac{F(z)}{G(z)H(z)} \tag{18.2}$$

We will assume that $G(z)$ and $H(z)$ are causal monic polynomials with all their roots inside the unit circle:

$$\begin{aligned}
 G(z) &= g_0 + g_1 z^{-1} + \dots + g_p z^{-p} & ; g_0 &= 1 \\
 H(z) &= h_0 + h_1 z^{-1} + \dots + h_q z^{-q} & ; h_0 &= 1 \\
 F(z) &= f_{-\sigma} z^{\sigma} + \dots + f_{\tau} z^{-\tau} & ; \tau, \sigma &\geq 0
 \end{aligned} \tag{18.3}$$

Dickinson¹⁹ has treated this problem and shown that the rational polynomial structure can be exploited to linearly transform the set of linear equations into a form involving an almost Toeplitz band diagonal matrix with displacement rank $\kappa=2$. He then showed that a fast Levinson-style almost Toeplitz algorithm could be used to solve for \underline{x}_N with a number of operations proportional to the length of the data N times the degree of the rational polynomial. His approach, however, had two faults - it used a slow version of the Levinson-style algorithm, and the final step involved a recursion similar to (12.1) which is usually numerically unstable. We will present a much superior approach in which we apply our fast Choleski algorithm to this problem, and thereby derive an algorithm which is at least 33% faster and uses less storage than Dickinson's.

Our presentation of the problem initially follows Dickinson quite closely. First find a partial fraction expansion of the rational polynomial $S(z)$ as follows:

$$S(z) = \frac{V(z)}{G(z)} + \frac{W(z^{-1})}{H(z^{-1})} \quad (18.4)$$

where $V(z)$ and $W(z)$ are causal polynomials with degrees $\bar{\tau}=\max(\tau,p)$ and $\bar{\sigma}=\max(\sigma,q)$ respectively. (A very fast method for calculating $V(z)$ and $W(z)$ is the Euclidean algorithm developed by Musicus.) The first term in (18.4), $S^+(z)=\frac{V(z)}{G(z)}$, has an inverse Z-transform $s^+(n)$ which is causal, while the second term, $S^-(z)=\frac{W(z^{-1})}{H(z^{-1})}$, has an inverse Z-transform $s^-(n)$ which is anti-causal. Then

$$s(n) = s^+(n) + s^-(n) \quad (18.5)$$

$$\text{where } s^+(n)=0 \text{ and } s^-(-n)=0 \text{ for } n < 0$$

In an analogous fashion, we can decompose the matrix S_N into a sum of a lower triangular Toeplitz matrix S_N^+ and an upper triangular Toeplitz matrix S_N^- with entries $S_{i,j}^+ = s^+(i-j)$ and $S_{i,j}^- = s^-(i-j)$:

$$S_N = S_N^+ + S_N^- \quad (18.6)$$

Define:

$$\underline{g}_n = \begin{pmatrix} g_0 \\ \vdots \\ g_p \\ \underline{0} \end{pmatrix} \quad \underline{h}_n = \begin{pmatrix} h_0 \\ \vdots \\ h_q \\ \underline{0} \end{pmatrix} \quad \underline{v}_n = \begin{pmatrix} v_0 \\ \vdots \\ f_r \\ \underline{0} \end{pmatrix} \quad \underline{w}_n = \begin{pmatrix} w_0 \\ \vdots \\ w_s \\ \underline{0} \end{pmatrix} \quad (18.7)$$

where the $(n+1)$ long vectors are suitably truncated if n is smaller than the degree of the polynomial. Because of the isomorphism between multiplication of lower (upper) triangular Toeplitz matrices and causal (anti-causal) polynomials, it is easy to see that:

$$\begin{aligned} L(\underline{g}_N)S_N^+ &= L(\underline{v}_N) \\ S_N^-U(\underline{h}_N^T) &= U(\underline{w}_N^T) \end{aligned} \quad (18.8)$$

Combining (18.8) and (18.6), the matrix R_N defined by:

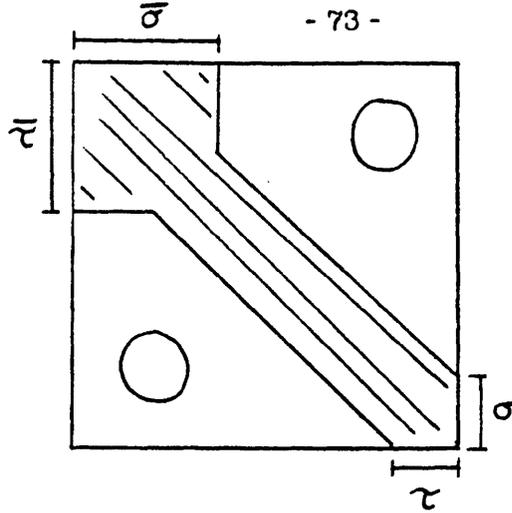
$$\begin{aligned} R_N &= L(\underline{g}_N)S_N U(\underline{h}_N^T) \\ &= L(\underline{v}_N)U(\underline{h}_N^T) + L(\underline{g}_N)U(\underline{w}_N^T) \end{aligned} \quad (18.9)$$

will be band diagonal and almost Toeplitz with displacement rank $\kappa=2$. This suggests the following three phase procedure for calculating \underline{x}_N :

- a) Calculate $\underline{y}_N = G_N \underline{y}_N$
- b) Solve $R_N \underline{x}_N = \underline{y}_N$
- c) Calculate $\underline{x}_N = H_N^T \underline{x}_N$

Steps a) and c) only involve multiplying a band diagonal Toeplitz matrix and a vector; they thus consume only Np and Nq operations respectively. Step b) can be most efficiently solved using a band diagonal fast Choleski algorithm.

The matrix R_N will have the shape shown below:



Except for the upper left $\bar{\tau} \times \bar{\sigma}$ corner, R_N will actually be a banded exactly Toeplitz matrix with entries equal to the numerator polynomial coefficients:

$$R_{i,j} = \begin{cases} f_{i-j} & -\sigma \leq i-j \leq \tau \\ 0 & \text{else} \end{cases} \quad \text{for } i > \bar{\tau}, j > \bar{\sigma} \quad (18.10)$$

(Beware that the width of the band of non-zero diagonals, $\tau + \sigma + 1$, may be smaller than the width of the upper left $\bar{\tau} \times \bar{\sigma}$ corner if $\tau < p$ or $\sigma < q$.) To apply our fast Choleski method to this problem, let us assume that $v_0 h_0 \neq 1$, and take:

$$\underline{c}_n^1 = \underline{v}_n \quad \underline{d}_n^1 = \underline{h}_n \quad \underline{c}_n^2 = \underline{g}_n \quad \underline{d}_n^2 = \underline{w}_n \quad (18.11)$$

(If $v_0 h_0 \neq 0$, we can try swapping the roles of $\underline{c}_n^1, \underline{d}_n^1$ and $\underline{c}_n^2, \underline{d}_n^2$.) Because R_N is only band diagonal, only the following coefficients of the vectors $\underline{\alpha}_n, \underline{\beta}_n, \underline{\varphi}_n^2$ and $\underline{\psi}_n^2$ will be non-zero:

$$\begin{aligned} & \beta_{n,n} \cdots \beta_{\max(\bar{\tau}, n+\bar{\tau}), n} \\ & \alpha_{n,n} \cdots \alpha_{\max(\bar{\sigma}, n+\bar{\sigma}), n} \\ & \varphi_{n+1,n}^2 \cdots \varphi_{\max(\bar{\tau}, n+\bar{\tau}), n}^2 \\ & \psi_{n+1,n}^2 \cdots \psi_{\max(\bar{\sigma}, n+\bar{\sigma}), n}^2 \end{aligned} \quad (18.12)$$

This fact can be used to significantly reduce the necessary computational effort. To simplify the notation, we will drop the "~" from \mathcal{V}_n^i and \mathcal{W}_n^i , omit the superscript "2", and take $g_n = 0$ for $n > p$ and $h_n = 0$ for $n > q$. The forward phase of the fast Choleski algorithm is then:

Forward Phase:

$$\begin{aligned}
 \text{Initialization: } \bar{\tau} &= \max(\tau, p) \\
 \bar{\sigma} &= \max(\sigma, q) \\
 \varepsilon_{-1} &= v_0 h_0 \\
 \beta_{j-1,-1} &= h_0 v_j \quad \text{for } j=1, \dots, \bar{\tau} \\
 \varphi_{j,-1} &= -g_j \\
 \alpha_{j-1,-1} &= v_0 h_j \quad \text{for } j=1, \dots, \bar{\sigma} \\
 \psi_{j,-1} &= -w_j \\
 M_{-1}^{-1} &= -1
 \end{aligned}$$

$$\text{For } n=0, \dots, N \quad (18.13)$$

$$\begin{aligned}
 \xi_n &= -\varphi_{n,n-1} \\
 \rho_n &= -\psi_{n,n-1} \\
 \nu_n &= M_{n-1}^{-1} \rho_n \\
 \mu_n &= M_{n-1}^{-1} \xi_n \\
 M_n^{-1} &= M_{n-1}^{-1} - \frac{\xi_n \rho_n}{\varepsilon_{n-1}}
 \end{aligned}$$

$$\varepsilon_n = \varepsilon_{n-1} - \nu_n \xi_n$$

If $n \leq \bar{\tau} - \tau$

$$\begin{aligned}
 \beta_{n+j,n} &= \beta_{n+j-1,n-1} + \nu_n \varphi_{n+j,n-1} \quad \text{for } j=1, \dots, \bar{\tau}-n \\
 \varphi_{n+j,n} &= \varphi_{n+j,n-1} + \frac{\xi_n}{\varepsilon_n} \beta_{n+j,n} \quad \text{for } j=1, \dots, \bar{\tau}-n
 \end{aligned}$$

Else:

$$\begin{aligned}
 \beta_{n+j,n} &= \begin{cases} \beta_{n+j-1,n-1} + \nu_n \varphi_{n+j,n-1} & \text{for } j=1, \dots, \tau-1 \\ \beta_{n+\tau-1,n-1} & \text{for } j=\tau \end{cases} \\
 \varphi_{n+j,n} &= \begin{cases} \varphi_{n+j,n-1} + \frac{\xi_n}{\varepsilon_n} \beta_{n+j,n} & \text{for } j=1, \dots, \tau-1 \\ \frac{\xi_n}{\varepsilon_n} \beta_{n+\tau,n} & \text{for } j=\tau \end{cases}
 \end{aligned}$$

If $n \leq \bar{\sigma} - \sigma$

$$\begin{aligned}
 \alpha_{n+j,n} &= \alpha_{n+j-1,n-1} + \mu_n \psi_{n+j,n-1} \quad \text{for } j=1, \dots, \bar{\sigma}-n \\
 \psi_{n+j,n} &= \psi_{n+j,n-1} + \frac{\rho_n}{\varepsilon_n} \alpha_{n+j,n} \quad \text{for } j=1, \dots, \bar{\sigma}-n
 \end{aligned}$$

Else:

$$\alpha_{n+j,n} = \begin{cases} \alpha_{n+j-1,n-1} + \mu_n \psi_{n+j,n-1} & \text{for } j=1, \dots, \sigma-1 \\ \alpha_{n+\sigma-1,n-1} & \text{for } j=\sigma \end{cases}$$

$$\psi_{n+j,n} = \begin{cases} \psi_{n+j,n-1} + \frac{\rho_n}{\varepsilon_n} \alpha_{n+j,n} & \text{for } j=1, \dots, \sigma-1 \\ \frac{\rho_n}{\varepsilon_n} \alpha_{n+\sigma-1,n-1} & \text{for } j=\sigma \end{cases}$$

$$\lambda_n = \frac{\vartheta_n}{\varepsilon_n}$$

$$\vartheta_{n+j} \leftarrow \vartheta_{n+j} - \lambda_n \varphi_{n+j,n} \quad \text{for } j=1, \dots, \max(\bar{\tau}-n, \tau)$$

As before, the structure of the backward phase depends on how much extra storage is available to save values from the forward phase. If we had stored all the values $\frac{\alpha_{n+j,n}}{\varepsilon_n}$ for $j=0, \dots, \max(\bar{\sigma}-n, \sigma)$ and $n=0, \dots, N$ then we could immediately solve for \underline{X}_N by back substitution:

Backward Phase: (extra $N\sigma$ storage)

$$\text{For } n=N, \dots, 0 \tag{18.14}$$

$$\underline{X}_n = \lambda_n - \frac{1}{\varepsilon_n} \sum_{j=n+1}^{\min(\max(\bar{\sigma}, n+\sigma), N)} \alpha_{j,n} \underline{X}_j$$

Total computation will be about $[3(\tau+\sigma)+6]N$ operations.

If only $2N$ extra storage locations were available, then we could save the values of ρ_n and μ_n from the forward phase and recalculate the values of $\alpha_{n+j,n}$ in descending order of n for use in calculating \underline{X}_N :

Backward Phase:

$$\underline{X}_N = \lambda_N \tag{18.15}$$

For $n=N, \dots, 1$

$$\psi_{n+j,n-1} = \begin{cases} -\rho_n & \text{for } j=0 \\ \psi_{n+j,n} - \frac{\rho_n}{\varepsilon_n} \alpha_{n+j,n} & \text{for } j=1, \dots, \max(\bar{\sigma}-n, \sigma-1) \end{cases}$$

$$\alpha_{n+j-1,n-1} = \begin{cases} \alpha_{n+j,n} - \mu_n \psi_{n+j,n-1} & \text{for } j=1, \dots, \max(\bar{\sigma}-n, \sigma-1) \\ \alpha_{n+\sigma,n} & \text{for } j=\sigma \text{ if } \sigma > \bar{\sigma}-n \\ 0 & \text{for } j=\bar{\sigma}-n+1 \text{ if } \sigma \leq \bar{\sigma}-n \end{cases}$$

$$\varepsilon_{n-1} = \varepsilon_n + \mu_n \rho_n$$

$$\tilde{x}_{n-1} = \lambda_{n-1} - \frac{1}{\varepsilon_{n-1}} \sum_{j=n}^{\min(\max(\bar{\sigma}-n+1, \sigma), N-n+1)} \alpha_{j, n-1} \tilde{x}_j$$

Total computation is now about $[3\tau+5\sigma+6]N$ operations.

Finally, if no extra storage is available, so that we only want to use about $2(\bar{\tau}+\bar{\sigma})$ locations for the recursions plus about N locations for \tilde{y}_N , then it is still possible to recalculate the values of $\alpha_{n+j, n}$ in descending order of n starting only from the knowledge of the various vectors at stage N . We do this by exploiting the fact that:

$$\begin{aligned} \varphi_{n+\tau, n} &= \frac{\xi_n}{\varepsilon_n} \beta_{n+\tau, n} & \text{for } n > \bar{\tau} - \tau \\ \beta_{n+\tau, n} &= f_{\tau} \neq 0 \\ \psi_{n+\sigma, n} &= \frac{\rho_n}{\varepsilon_n} \alpha_{n+\sigma, n} & \text{for } n > \bar{\sigma} - \sigma \\ \alpha_{n+\sigma, n} &= f_{-\sigma} \neq 0 \end{aligned} \tag{18.16}$$

Given $\underline{\beta}_n, \underline{\alpha}_n, \underline{\varphi}_n, \underline{\psi}_n$ we can thus derive ξ_n, ρ_n . These can then be used to generate ν_n and μ_n , and then the $(n-1)^{th}$ order vectors $\underline{\alpha}_{n-1}, \underline{\beta}_{n-1}, \underline{\varphi}_{n-1}$ and $\underline{\psi}_{n-1}$. The only difficulty will be for values of $n \leq \bar{\tau} - \tau$ and $n \leq \bar{\sigma} - \sigma$. To cover these cases, the values of ξ_n and ρ_n will have to be saved for $n=1, \dots, \bar{\tau} - \tau$ and $n=1, \dots, \bar{\sigma} - \sigma$ respectively. This can actually be done on the forward phase without using up any extra storage. After the n^{th} step in the forward phase for $n > \bar{\tau} - \tau$ and $n > \bar{\sigma} - \sigma$ we should organize storage as follows:

$$\begin{aligned} \underline{\beta}_n: & 0 \cdots 0 & \beta_{n+\tau, n} \cdots \beta_{n+1, n} & \text{(length } \bar{\tau} \text{)} \\ \underline{\varphi}_n: & \xi_1 \cdots \xi_{\bar{\tau}-\tau} & \varphi_{n+\tau, n} \cdots \varphi_{n+1, n} & \text{(length } \bar{\tau} \text{)} \\ \underline{\alpha}_n: & 0 \cdots 0 & \alpha_{n+\sigma, n} \cdots \alpha_{n+1, n} & \text{(length } \bar{\sigma} \text{)} \\ \underline{\psi}_n: & \rho_1 \cdots \rho_{\bar{\sigma}-\sigma} & \psi_{n+\sigma, n} \cdots \psi_{n+1, n} & \text{(length } \bar{\sigma} \text{)} \\ \underline{y}_N: & \lambda_0 \cdots \lambda_n & \tilde{y}_{n+1} \cdots \tilde{y}_N & \text{(length } N \text{)} \end{aligned} \tag{18.17}$$

The backward phase will then take the form:

Backward Phase: (no extra storage)

$$E_N = \nu_N$$

For $n=N, \dots, 1$

(18.18)

$$\xi_n = \begin{cases} \frac{\varphi_{n+\tau,n}}{\beta_{n+\tau,n}} \varepsilon_n & \text{for } n > \bar{\tau} - \tau \\ \text{(value saved for } n \leq \bar{\tau} - \tau) \end{cases}$$

$$\rho_n = \begin{cases} \frac{\psi_{n+\sigma,n}}{\alpha_{n+\sigma,n}} \varepsilon_n & \text{for } n > \bar{\sigma} - \sigma \\ \text{(value saved for } n \leq \bar{\sigma} - \sigma) \end{cases}$$

$$M_{n-1} = M_n - \frac{\xi_n \rho_n}{\varepsilon_n}$$

$$\nu_n = \frac{\rho_n}{M_{n-1}}$$

$$\mu_n = \frac{\xi_n}{M_{n-1}}$$

$$\varepsilon_{n-1} = \varepsilon_n + \nu_n \xi_n$$

$$\varphi_{n+j,n-1} = \begin{cases} -\xi_n & \text{for } j=0 \\ \varphi_{n+j,n} - \frac{\xi_n}{\varepsilon_n} \beta_{n+j,n} & \text{for } j=1, \dots, \max(\bar{\tau}-n, \tau-1) \end{cases}$$

$$\psi_{n+j,n-1} = \begin{cases} -\rho_n & \text{for } j=0 \\ \psi_{n+j,n} - \frac{\rho_n}{\varepsilon_n} \alpha_{n+j,n} & \text{for } j=1, \dots, \max(\bar{\sigma}-n, \sigma-1) \end{cases}$$

$$\beta_{n+j-1,n-1} = \begin{cases} \beta_{n+j,n} - \nu_n \varphi_{n+j,n-1} & \text{for } j=1, \dots, \max(\bar{\tau}-n, \tau-1) \\ \beta_{n+j,n} & \text{for } j=\tau \text{ if } n > \bar{\tau} - \tau \\ 0 & \text{for } j=\bar{\tau}-n+1 \text{ if } n \leq \bar{\tau} - \tau \end{cases}$$

$$\alpha_{n+j-1,n-1} = \begin{cases} \alpha_{n+j,n} - \mu_n \psi_{n+j,n-1} & \text{for } j=1, \dots, \max(\bar{\sigma}-n, \sigma-1) \\ \alpha_{n+\sigma,n} & \text{for } j=\sigma \text{ if } n > \bar{\sigma} - \sigma \\ 0 & \text{for } j=\bar{\sigma}-n+1 \text{ if } n \leq \bar{\sigma} - \sigma \end{cases}$$

$$\tilde{x}_{n-1} = \lambda_{n-1} - \frac{1}{\varepsilon_{n-1}} \sum_{j=n}^{\min(\max(\bar{\sigma}-n+1, \sigma), N-n+1)} \alpha_{j, n-1} \tilde{x}_j$$

Total computation time will be about $[5(\tau+\sigma)+10]N$ operations (at least 33% faster than Dickinson's algorithm) and about $2(\bar{\tau}+\bar{\sigma})+N$ storage locations are needed (less than Dickinson's algorithm.) Unfortunately, as in the case of exactly Toeplitz matrices, this minimal storage algorithm is not always stable. One partial solution to this difficulty is to save the values of $\alpha_n(z)$ and $\beta_n(z)$ after every m^{th} step of the forward recursion, then periodically reset the backward recursion to the correct values of $\alpha_n(z)$ and $\beta_n(z)$. A simpler solution, of course, would be to add $2N$ extra storage locations, and use our previous backward phase algorithm since it appears to be numerically stable.

If the rational polynomial $S(z)$ is also symmetric with $\tau=\sigma$, $p=q$, $G(z)=H(z)$ and $F(z)=F(z^{-1})$ then the computation and storage requirements can be reduced considerably. We will be able to choose a symmetric partial fraction decomposition (18.4) with $V(z)=W(z)$. Also R_N will be symmetric, $R_N=R_N^T$. We can choose a symmetric lower \times upper decomposition of R_N by setting:

$$\begin{aligned} \underline{c}_n^1 &= \frac{1}{2}(\underline{v}_n + \underline{g}_n) \\ \underline{c}_n^2 &= \frac{1}{2}(\underline{v}_n - \underline{g}_n) \end{aligned} \tag{18.19}$$

so that:

$$R_N = L(\underline{c}_N^1)U(\underline{c}_N^{1T}) - L(\underline{c}_N^2)U(\underline{c}_N^{2T}) \tag{18.20}$$

Then by symmetry:

$$\underline{\alpha}_n = \underline{\beta}_n \quad \xi_n = -\rho_n \quad \nu_n = -\mu_n \quad \underline{\varphi}_n = -\underline{\psi}_n \tag{18.21}$$

This cuts the computation and storage requirements of the minimal storage algorithm to about $(6\tau+11)N$ operations and about $N+(\bar{\tau}+\bar{\sigma})$ storage locations. (If an extra N locations are available for saving the values ξ_n , then the algorithm would be stable and $(6\tau+9)N$ operations would be needed.)

19. Example - Smoothing Filter for Noisy ARMA System

It is possible to apply our almost Toeplitz algorithms even to certain problems which don't quite fit the model we have assumed for R_N . A good example of this is Maximum Likelihood estimation of a noisy Autoregressive Moving Average (ARMA) signal like that in figure 19.1

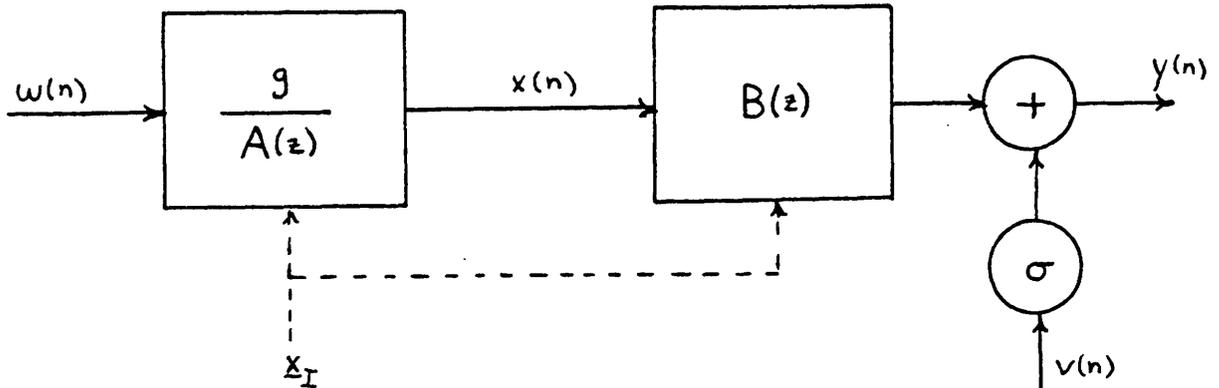


Figure 19.1 - Noisy ARMA Model

Unit variance white gaussian noise $w(n)$ drives a known p^{th} order autoregressive filter $\frac{g}{H(z)}$ to generate an autoregressive signal $x(n)$. This signal is further processed by a known q^{th} order all-zero filter $F(z)$ to form $s(n)$. (We will assume that $q < p$ to simplify the following.) Independent unit variance white gaussian noise $v(n)$ is added with gain σ to form the observations $y(n)$:

$$\begin{aligned}
 x(n) &= h_1 x(n-1) + \dots + h_p x(n-p) + g w(n) \\
 y(n) &= f_0 x(n) + \dots + f_q x(n-q) + \sigma v(n) \qquad (19.1) \\
 \text{where: } p(v(n)) &= p(w(n)) = N(0,1) \\
 \text{for } n &= 0, \dots, N
 \end{aligned}$$

Values of h_n for $n > p$ and f_n for $n > q$ will be taken to be 0. We will assume that the *a priori* distribution of the initial condition $\underline{x}_I = (x(-1) \dots x(-p))^T$ is also gaussian:

$$p(\underline{x}_I) = N(\underline{\bar{x}}_I, \Sigma_I) \qquad (19.2)$$

Let:

$$\underline{x}_{N+p} = \begin{pmatrix} x(N) \\ \vdots \\ x(-p) \end{pmatrix} \quad \underline{y}_N = \begin{pmatrix} y(N) \\ \vdots \\ y(0) \end{pmatrix} \quad (19.3)$$

(Note that the order of the coefficients in these vectors has been reversed from our previous examples.) It is easy to show that the joint probability density of \underline{x}_{N+p} and \underline{y}_N is:

$$\log p(\underline{x}, \underline{y}) = -\frac{1}{2} \left\{ \underline{x}^T \left[\frac{1}{g^2} H^T H + \begin{pmatrix} 0 & 0 \\ 0 & \Sigma_I^{-1} \end{pmatrix} \right] \underline{x} + \frac{1}{\sigma^2} (\underline{y} - F\underline{x})^T (\underline{y} - F\underline{x}) + \text{constant} \right\}$$

$$\text{where: } F = \begin{pmatrix} f_0 & \cdots & f_p & 0 \\ 0 & f_0 & \cdots & f_p \end{pmatrix} \quad \text{and } H = \begin{pmatrix} h_0 & \cdots & h_p & 0 \\ 0 & h_0 & \cdots & h_p \end{pmatrix} \quad (19.4)$$

and F and H are $N \times (N+p)$ band diagonal Toeplitz matrices. Given the observations $y(n)$ for $n=0, \dots, N$, the Maximum Likelihood (ML) estimate of \underline{x}_{N+p} is found by maximizing this log likelihood function over all \underline{x} . Since the log likelihood function is quadratic in \underline{x} , the ML estimate $\hat{\underline{x}}_{N+p}$ can be calculated by solving the linear equations:

$$\left[\frac{1}{g^2} H^T H + \frac{1}{\sigma^2} F^T F + \begin{pmatrix} 0 & 0 \\ 0 & \Sigma_I^{-1} \end{pmatrix} \right] \hat{\underline{x}}_{N+p} = \begin{pmatrix} 0 \\ \Sigma_I^{-1} \underline{x}_I \end{pmatrix} + \frac{1}{\sigma^2} F^T \underline{y}_N \quad (19.5)$$

The matrix on the left, which we will call R_{N+p} , is close to being almost Toeplitz with displacement rank $\kappa=2$. Unfortunately, because of the addition of the initial covariance matrix Σ_I^{-1} and because H and F are not square, the lower right $p \times p$ corner does not fit this pattern. The first N minors R_n of R_{N+p} , however, are not only almost Toeplitz with displacement rank $\kappa=2$, but are also symmetric and band diagonal. One approach for solving (19.5), therefore, would be to use our fast Choleski algorithm to calculate the LDU decomposition of the first N rows and columns of R_{N+p} . The remaining p rows and columns of the decomposition can be computed by standard gaussian elimination. Let us set:

$$\underline{c}_n^1 = \underline{d}_n^1 = \frac{1}{g} \begin{pmatrix} h_0 \\ \vdots \\ h_p \\ 0 \end{pmatrix} \quad \underline{c}_n^2 = \underline{d}_n^2 = \frac{1}{\sigma} \begin{pmatrix} f_0 \\ \vdots \\ f_p \\ 0 \end{pmatrix} \quad (19.6)$$

smoothing filter problem: (we delete the "~" and the superscripts "2" to simplify the notation)

Forward Phase:

$$\begin{aligned}
 \text{Initialization: } \varepsilon_{-1} &= \frac{1}{g^2} h_0^2 \\
 \beta_{j-1,-1} &= \frac{1}{g^2} h_0 h_n \quad \text{for } j=1, \dots, p \\
 \varphi_{j,-1} &= -\frac{1}{\sigma} f_j \\
 M_{-1}^{-1} &= -1 \\
 \underline{y}_{N+p} &= \begin{pmatrix} 0 \\ \Sigma_I^{-1} \underline{x}_I \end{pmatrix} + \frac{1}{\sigma^2} F^T \underline{y}_N
 \end{aligned} \tag{19.9}$$

For $n=0, \dots, N$

$$\begin{aligned}
 \xi_n &= -\varphi_{n,n-1} \\
 \nu_n &= M_{n-1}^{-1} \xi_n \\
 M_n^{-1} &= M_{n-1}^{-1} - \frac{1}{\varepsilon_{n-1}} \nu_n^2 \\
 \varepsilon_n &= \varepsilon_{n-1} - \nu_n \xi_n \\
 \beta_{n+j,n} &= \begin{cases} \beta_{n+j-1,n-1} + \nu_n \varphi_{n+j,n-1} & \text{for } j=1, \dots, p-1 \\ \beta_{n+p-1,n-1} & \text{for } j=p \end{cases} \\
 \varphi_{n+j,n} &= \begin{cases} \varphi_{n+j,n-1} + \frac{\xi_n}{\varepsilon_n} \beta_{n+j,n} & \text{for } j=1, \dots, p-1 \\ \frac{\xi_n}{\varepsilon_n} \beta_{n+p,n} & \text{for } j=p \end{cases} \\
 \lambda_n &= \frac{\underline{y}_n}{\varepsilon_n}
 \end{aligned}$$

$$\underline{y}_{n+j} \leftarrow \underline{y}_{n+j} - \lambda_n \beta_{n+j,n} \quad \text{for } j=1, \dots, p$$

Middle Phase:

Compute \tilde{R}_p from (19.8) and (19.5)

$$\text{Solve } \tilde{R}_p \begin{pmatrix} x_{N+1} \\ \vdots \\ x_{N+p} \end{pmatrix} = \begin{pmatrix} \lambda_{N+1} \\ \vdots \\ \lambda_{N+p} \end{pmatrix}$$

Backward Phase: (Minimal Storage)

For $n=N, \dots, 1$

$$x_n = \lambda_n - \frac{1}{\varepsilon_n} \sum_{j=1}^p \beta_{n+j,n} x_{n+j}$$

$$\xi_n = \frac{\varphi_{n+p,n}}{\beta_{n+p,n}} \varepsilon_n$$

$$M_{n-1} = M_n - \frac{1}{\varepsilon_n} \xi_n^2$$

$$\nu_n = \frac{\xi_n}{M_{n-1}}$$

$$\varepsilon_{n-1} = \varepsilon_n + \nu_n \xi_n$$

$$\varphi_{n+j,n-1} = \begin{cases} -\xi_n & \text{for } j=0 \\ \varphi_{n+j,n} - \frac{\xi_n}{\varepsilon_n} \beta_{n+j,n} & \text{for } j=1, \dots, p-1 \end{cases}$$

$$\beta_{n+j-1,n-1} = \begin{cases} \beta_{n+j,n} - \nu_n \varphi_{n+j,n-1} & \text{for } j=1, \dots, p-1 \\ \beta_{n+p,n} & \text{for } j=p \end{cases}$$

Total computation required will be about $(6p+11)N+O(p^3)$ operations and about $N+3p+\frac{p^2}{2}$ storage locations will be needed. If an extra N storage locations are available, we will be able to save the values of ξ_n generated on the forward phase, and thus avoid recalculating these on the backward recursion and also avoid a potentially unstable recursion. Note that, unlike the Chandrasekhar filter,¹⁸ the complexity of this algorithm is independent of the initial covariance Σ_1 . In fact, our method is faster than the Chandrasekhar algorithm. Its disadvantage is that it is strictly a "batch" processing method, and a completely new calculation is needed if more data becomes available.

A somewhat different approach would be to recognize that R_{N+p} can be expressed exactly in the form of a sum of products of lower \times upper triangular Toeplitz matrices, provided we allow up to $p+2$ terms in this sum. As in the algorithm above, the first two terms will involve the pole and zero coefficients h_j and f_j . The last p terms will only be non-zero in the lower right $p \times p$ corner. Since these terms will not contribute to the first N minors R_n , they will only be involved in the recursion on the final p steps. The first N steps will thus be identical to the algorithm above. Adding these p extra terms thus only avoids the need in the previous algorithm for a middle phase to solve the p simultaneous equations involving \tilde{R}_p .

A special case in which this purely lower \times upper approach is quite successful is when the signal process $x(n)$ is assumed to be stationary. The initial signal mean \underline{x}_1 will be zero, and the initial signal covariance Σ_1 will be Toeplitz with entries equal to the ideal correlations of the autoregressive power spectrum $\frac{g^2}{H(z)H(z^{-1})}$. Applying a variant of the Gohberg-Semencul formula (2.18):

$$\Sigma_1^{-1} = \frac{1}{g^2} \begin{bmatrix} h_0 & & 0 \\ & \ddots & \\ & & h_0 \end{bmatrix} \begin{bmatrix} h_0 & \cdots & h_{p-1} \\ & \ddots & \\ 0 & & h_0 \end{bmatrix} + \frac{1}{g^2} \begin{bmatrix} h_p & & 0 \\ & \ddots & \\ h_1 & \cdots & h_p \end{bmatrix} \begin{bmatrix} h_p & \cdots & h_1 \\ & \ddots & \\ 0 & & h_0 \end{bmatrix}$$

Substituting this into our formula for R_{N+p} , it is straightforward to show that R_{N+p} will be an almost Toeplitz matrix with displacement rank $\kappa=4$. Vectors \underline{c}_n^1 , \underline{d}_n^1 , \underline{c}_n^2 , and \underline{d}_n^2 will be the same as in (19.6), and:

$$\underline{c}_n^3 = -\underline{d}_n^3 = \frac{1}{g} \begin{bmatrix} 0_{N+1} \\ h_p \\ \vdots \\ h_1 \end{bmatrix} \quad \underline{c}_n^4 = -\underline{d}_n^4 = \frac{1}{\sigma} \begin{bmatrix} 0_{N+1} \\ f_0 \\ \vdots \\ f_{p-1} \end{bmatrix} \quad (19.11)$$

The first $N+1$ elements of these last two vectors are zero. Thus $\xi_n^i = \nu_n^i = \tilde{\rho}_n^i = \rho_n^i = 0$ for $i=3,4$ and $n=0, \dots, N$. This implies that these last two vec-

tors need only be considered in the last p steps; the first N steps will be identical to our fast Choleski algorithm above. Because of symmetry, $\varphi_n^i = -\psi_n^i$, $\mathcal{V}_n^i = -\mu_n^i$, and $\xi_n^i = -\rho_n^i$ for $i=3,4$. Also, because R_{N+p} is band diagonal, at most the last p coefficients of φ_n^3 and φ_n^4 will be non-zero. Putting all this together, the complete algorithm starts with the forward phase described above for $n=0, \dots, N$. Steps $n=N+1, \dots, N+p$ have the following form:

Forward Phase (last p steps)

Init: $\varphi_{N+j,N}^2 = \varphi_{N+j,N}$ (from first N steps) for $j=1, \dots, p$

$$\varphi_{N+j,N}^3 = -\frac{1}{g} h_{p-j+1} \quad \text{for } j=1, \dots, p$$

$$\varphi_{N+j,N}^4 = -\frac{1}{\sigma} f_{j-1} \quad \text{for } j=0, \dots, p$$

Extend M_N^{-1} into a 3×3 matrix

$$M_N^{-1} = \begin{bmatrix} M_N^{-1} & 0 \\ & -1 \\ 0 & -1 \end{bmatrix}$$

For $n=N+1, \dots, N+p$

(19.12)

$$\xi_n^i = -\varphi_{n,n-1}^i \quad \text{for } i=2,3,4$$

$$\begin{bmatrix} \mathcal{V}_n^2 \\ \mathcal{V}_n^3 \\ \mathcal{V}_n^4 \end{bmatrix} = M_{n-1}^{-T} \begin{bmatrix} \xi_n^2 \\ -\xi_n^3 \\ -\xi_n^4 \end{bmatrix}$$

$$M_n^{-1} = M_{n-1}^{-1} - \frac{1}{\varepsilon_{n-1}} \begin{bmatrix} \mathcal{V}_n^2 \\ \mathcal{V}_n^3 \\ \mathcal{V}_n^4 \end{bmatrix} (\mathcal{V}_n^2 - \mathcal{V}_n^3 - \mathcal{V}_n^4)$$

$$\varepsilon_n = \varepsilon_{n-1} - \sum_{i=2}^4 \mathcal{V}_n^i \xi_n^i$$

$$\beta_{n+j,n} = \begin{cases} \beta_{n+j-1,n-1} + \sum_{i=2}^4 \mathcal{V}_n^i \varphi_{n+j,n-1}^i & \text{for } j=1, \dots, p-1 \\ \beta_{n+p-1,n-1} & \text{for } j=p \end{cases}$$

$$\varphi_{n+j,n}^i = \begin{cases} \varphi_{n+j,n-1}^i + \frac{\xi_n^i}{\varepsilon_n} \beta_{n+j,n} & \text{for } 0, \dots, p-1 \\ \frac{\xi_n^i}{\varepsilon_n} \beta_{n+p,n} & \text{for } p \end{cases} \quad \text{for } i=2,3,4$$

$$\lambda_n = \frac{\mathcal{Y}_n}{\varepsilon_n}$$

$$\mathcal{Y}_{n+j} \leftarrow \mathcal{Y}_{n+j} - \lambda_n \beta_{n+j,n} \quad \text{for } j=1, \dots, N+p-n$$

It is convenient to save not only the usual vectors $\underline{\alpha}_{N+p}$, $\underline{\varphi}_{N+p}$, $\underline{\lambda}_{N+p}$ and ε_{N+p} generated on the forward phase, but also the values of the reflection coefficients \mathcal{V}_n^i for $i=N+1, \dots, N+p$ as well as the value of M_N^{-1} . The first p steps of the storage efficient back substitution phase for regenerating the values of $\beta_{j,n}$ in descending order of n and calculating \underline{x}_{N+p} then has the form:

Backward Phase: (first p steps)

For $n=N+p, \dots, N+1$

$$x_n = \lambda_n - \frac{1}{\varepsilon_n} \sum_{j=n+1}^{N+p} \beta_{j,n} x_j \quad (19.13)$$

$$\xi_n^i = \frac{\varphi_{n+p,n}^i}{\beta_{n+p,n}} \varepsilon_n \quad \text{for } i=2,3,4$$

\mathcal{V}_n^i saved from forward phase

$$\varepsilon_{n-1} = \varepsilon_n + \sum_{i=2}^4 \mathcal{V}_n^i \xi_n^i$$

$$\varphi_{n+j,n-1}^i = \begin{cases} -\xi_n^i & \text{for } j=0 \\ \varphi_{n+j,n}^i - \frac{\xi_n^i}{\varepsilon_n} \beta_{n+j,n} & \text{for } j=1, \dots, p-1 \end{cases}$$

$$\beta_{n+j-1,n-1} = \begin{cases} \beta_{n+j,n} - \sum_{i=2}^4 \mathcal{V}_n^i \varphi_{n+j,n-1}^i & \text{for } j=1, \dots, p-1 \\ \beta_{n+p,n} & \text{for } j=p \end{cases}$$

The remaining N steps of the backward substitution phase are then identical to the previous fast Choleski algorithm. Note that the chief advantage of this stationary filtering algorithm is that it avoids the need for solving a set of p linear

equations involving the matrix \tilde{R}_p . It thus replaces a symmetric gaussian elimination problem ($O(p^3)$ operations and $\frac{1}{2}p^2$ extra storage) by p steps of a fast Choleski algorithm with $\kappa=4$ (about $(7\frac{1}{2}p+22)p$ operations and $(5p+7)$ extra storage.)

20. Conclusion

In this massive tome, we have attempted to present a concise (unsuccessfully) and coherent development of the Levinson-style, fast Choleski and doubling algorithms for solving exactly Toeplitz and almost Toeplitz linear equations. Levinson-style algorithms result when we recursively solve an almost-Toeplitz problem by recursively solving the linear equations associated with the upper left principal minors of the matrix. These algorithms can be viewed as performing a UDL decomposition of R_N^{-1} . Applying a simple linear transformation to the Levinson recursions results in the fast Choleski algorithms, which effectively perform an LDU decomposition of R_N itself. Several variants of this fast Choleski algorithm can be derived; the predictor values can be computed in columnwise or rowwise order, and various backward recursions can be employed to minimize the required storage. In general the fast Choleski algorithms are somewhat slower than the Levinson-style algorithm for computing $\underline{x}_N = R_N^{-1} \underline{y}_N$. When R_N is band diagonal, however, the fast Choleski algorithms simplify dramatically and are quite attractive. Other algorithms for band diagonal matrices which should also be considered, however, are the matrix splitting and imbedding techniques of Jain,¹¹ Morf and Kailath,¹² and Fisher, Golub, Hald, Leiva, Widlund.²⁹ It is also possible to view the Choleski algorithm as an "inside out" Euclidian algorithm. This leads to $O(N \log^2 N)$ doubling algorithms for inverting R_N which use a divide and conquer strategy combined with FFT's to achieve their speed. Unfortunately, these algorithms are rather complex, and are competitive only for $N > 2500$ or more. Finally, the range of applications of

these algorithms is enormous, since almost-Toeplitz equations often arise whenever stationary data is processed. We have discussed covariance methods of linear prediction, rational Toeplitz matrices, and an ARMA filtering problem. In addition, the basic concept behind these algorithms, splitting the matrix into a sum of a more convenient matrix plus a correction term, is applicable to an extremely wide range of applications and matrix structures.

Appendix A - Proofs for Section 8

Proof of formula (8.3),(8.4)

The proof of equation (8.3) proceeds inductively. We start by proving it for $n=-1$. We are given:

$$R_N = \sum_{i=1}^{\kappa} L(\underline{c}_N^i) U(\underline{d}_N^{i\top}) \quad (A.1)$$

Applying lemma 1 in section 4 gives

$$\lrcorner R_N = \underline{c}_N^1 \underline{d}_N^{1\top} - \left(\underline{c}_N^2 \cdots \underline{c}_N^{\kappa} \right) \begin{pmatrix} -I \\ \vdots \\ -I \end{pmatrix} \begin{pmatrix} \underline{d}_N^{2\top} \\ \vdots \\ \underline{d}_N^{\kappa\top} \end{pmatrix} \quad (A.2)$$

Using the definitions in (7.15):

$$\begin{aligned} \beta_{j-1,-1} &= c_j^i d_0^i \\ \alpha_{j-1,-1} &= d_j^i c_0^i \\ \varphi_{-1}^i &= -\underline{c}_N^i \\ \psi_{-1}^i &= -\underline{d}_N^i \\ M_{-1} &= -I \\ \varepsilon_{-1} &= \frac{1}{c_0^1 d_0^1} \end{aligned} \quad (A.3)$$

we immediately have:

$$\lrcorner R_N = \begin{pmatrix} \beta_{0,-1} \\ \vdots \\ \beta_{N,-1} \end{pmatrix} \left(\varepsilon_{-1} \right)^{-1} \left(\alpha_{0,-1} \cdots \alpha_{N,-1} \right) - \Phi_{-1} M_{-1}^{-1} \Psi_{-1}^{\top} \quad (A.4)$$

Applying lemma 1 gives the formula in (8.3) for $n=-1$.

Now suppose the formula is correct for $n=-1, \dots, m-1$. We now prove it for $n=m$. From lemma 1, we know that:

$$\begin{aligned} \lrcorner \tilde{R}_{m-1} &= \underline{\beta}_{m-1}^v (\varepsilon_{m-1}^{-1}) \underline{\alpha}_{m-1}^{v\top} - \Phi_{m-1} M_{m-1}^{-1} \Psi_{m-1}^{\top} \\ &= \begin{pmatrix} 0 \\ \beta_{0,m-1} \\ \vdots \\ \beta_{N-1,m-1} \end{pmatrix} \begin{pmatrix} \varphi_{m-1}^2 \cdots \varphi_{m-1}^{\kappa} \end{pmatrix} \begin{pmatrix} \varepsilon_{m-1} & 0 \\ 0 & -M_{m-1} \end{pmatrix}^{-1} \begin{pmatrix} 0 \ \alpha_{0,m-1} \cdots \alpha_{N-1,m-1} \\ \psi_{m-1}^{2\top} \\ \vdots \\ \psi_{m-1}^{\kappa\top} \end{pmatrix} \end{aligned} \quad (A.5)$$

The recursion on the vectors $\underline{\beta}_n, \underline{\varphi}_n^i$ can be stated in the form:

$$\begin{pmatrix} \underline{\beta}_m & \underline{\varphi}_m^2 & \dots & \underline{\varphi}_m^\kappa \end{pmatrix} = \begin{pmatrix} 0 & & & \\ \beta_{0,m-1} & & & \\ \vdots & \underline{\varphi}_{m-1}^2 & \dots & \underline{\varphi}_{m-1}^\kappa \\ \beta_{N-1,m-1} & & & \end{pmatrix} \left[\frac{1}{\varepsilon_m} \begin{pmatrix} 1 \\ \underline{\varphi}_m^2 \\ \vdots \\ \underline{\varphi}_m^\kappa \end{pmatrix} \begin{pmatrix} \varepsilon_m & \xi_m^2 & \dots & \xi_m^\kappa \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & I \end{pmatrix} \right]$$

The inverse of the matrix on the right can be computed from the general partitioning formula:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} -A^{-1}B \\ I \end{pmatrix} [D - CA^{-1}B]^{-1} \begin{pmatrix} -CA^{-1}I \\ \end{pmatrix} \quad (\text{A.7})$$

Multiplying equation (A.6) on the right by this inverse gives:

$$\begin{pmatrix} \underline{\beta}_m & \underline{\varphi}_m^2 & \dots & \underline{\varphi}_m^\kappa \end{pmatrix} \begin{pmatrix} \frac{\varepsilon_{m-1}}{\varepsilon_m} & \frac{\xi_m^2}{\varepsilon_m} & \dots & \frac{\xi_m^\kappa}{\varepsilon_m} \\ \underline{\varphi}_m^2 & & & \\ \vdots & & I & \\ \underline{\varphi}_m^\kappa & & & \end{pmatrix} = \begin{pmatrix} 0 & & & \\ \beta_{0,m-1} & & & \\ \vdots & \underline{\varphi}_{m-1}^2 & \dots & \underline{\varphi}_{m-1}^\kappa \\ \beta_{N-1,m-1} & & & \end{pmatrix} \quad (\text{A.8})$$

Similarly:

$$\begin{pmatrix} \underline{\alpha}_m & \underline{\psi}_m^2 & \dots & \underline{\psi}_m^\kappa \end{pmatrix} \begin{pmatrix} \frac{\varepsilon_{m-1}}{\varepsilon_m} & \frac{\rho_m^2}{\varepsilon_m} & \dots & \frac{\rho_m^\kappa}{\varepsilon_m} \\ \underline{\psi}_m^2 & & & \\ \vdots & & I & \\ \underline{\psi}_m^\kappa & & & \end{pmatrix} = \begin{pmatrix} 0 & & & \\ \alpha_{0,m-1} & & & \\ \vdots & \underline{\psi}_{m-1}^2 & \dots & \underline{\psi}_{m-1}^\kappa \\ \alpha_{N-1,m-1} & & & \end{pmatrix} \quad (\text{A.9})$$

Substituting (A.8) and (A.9) into (A.5) and using the recursion formulas involving M_m^{-1} in (7.7), (7.8) and (7.13):

$$\underline{\mathbb{R}}_{m-1} = \begin{pmatrix} \underline{\beta}_m & \underline{\varphi}_m^2 & \dots & \underline{\varphi}_m^\kappa \end{pmatrix} \begin{pmatrix} \varepsilon_m & 0 \\ 0 & -M_m \end{pmatrix}^{-1} \begin{pmatrix} \underline{\alpha}_m^T \\ \underline{\psi}_m^{2T} \\ \vdots \\ \underline{\psi}_m^{\kappa T} \end{pmatrix} \quad (\text{A.10})$$

Applying lemma 1:

$$\underline{\mathbb{R}}_{m-1} = L(\underline{\beta}_m) D \left(\frac{1}{\varepsilon_m} \right) U(\underline{\alpha}_m^T) - L(\underline{\Phi}_m) D(M_m^{-1}) U(\underline{\Psi}_m^T)$$

$$\begin{aligned}
 &= \underline{\beta}_m \left[\frac{1}{\varepsilon_m} \right] \underline{\alpha}_m^T + L \left[\underline{\beta}_m \right] D \left[\frac{1}{\varepsilon_m} \right] U \left[\underline{\alpha}_m^T \right] - L \left[\underline{\phi}_m \right] D (M_m^{-1}) U \left[\underline{\psi}_m^T \right] \\
 &= \underline{\beta}_m \left[\frac{1}{\varepsilon_m} \right] \underline{\alpha}_m^T + \tilde{R}_m
 \end{aligned} \tag{A.11}$$

Substituting back into equation (8.4) proves the formula for $n=m$. Continuing inductively for $n=0, \dots, N$ then proves equations (8.3),(8.4) for all n .

Proof of Formula (8.8)

The UDL decomposition $R_N^{-1} = B_N^T \Lambda_N^{-1} A_N$ derived in section 6 implies that:

$$R_N^{-1} = \begin{pmatrix} R_{N-1}^{-1} & 0 \\ 0 & 0 \end{pmatrix} + \underline{b}_N \left[\frac{1}{\varepsilon_N} \right] \underline{a}_N^T \tag{A.12}$$

Define the matrix \bar{R}_N by:

$$\bar{R}_N = \begin{pmatrix} 0 & 0 \\ 0 & R_{N-1} \end{pmatrix} + \underline{c}_N \underline{d}_N^T = R_N - C_N D_N^T \tag{A.13}$$

Applying the Woodbury inversion formula (7.11) gives:

$$\bar{R}_N^{-1} = R_N^{-1} - R_N^{-1} C_N \left[D_N^T R_N^{-1} C_N - I \right]^{-1} D_N^T R_N^{-1} \tag{A.14}$$

or since:

$$\begin{aligned}
 R_N E_N &= C_N \\
 R_N F_N &= D_N \\
 M_N &= E_N^T R_N F_N - I
 \end{aligned} \tag{A.15}$$

then:

$$\bar{R}_N^{-1} = R_N^{-1} - F_N M_N^{-1} E_N^T \tag{A.16}$$

Applying the partitioning formula (A.7) to \bar{R}_N^{-1} then gives:

$$\bar{R}_N^{-1} = \begin{pmatrix} * & * \\ * & R_{N-1}^{-1} \end{pmatrix} \tag{A.17}$$

and thus

$$R_N^{-1} = \begin{pmatrix} * & * \\ * & R_{N-1}^{-1} \end{pmatrix} + F_N M_N^{-1} E_N^T \tag{A.18}$$

Combining equations (A.12) and (A.18) for R_N^{-1} , we get:

$$\Gamma R_N^{-1} = \underline{b}_N \left(\frac{1}{\varepsilon_N} \right) \underline{a}_N^T - \hat{F}_N M_N^{-1} \hat{E}_N^T \quad (\text{A.19})$$

Equating (A.18) with ΓR_N^{-1} as calculated from (A.12) gives:

$$\Gamma R_{N-1}^{-1} = \underline{b}_N \left(\frac{1}{\varepsilon_N} \right) \underline{a}_N^T - \bar{F}_N M_N^{-1} \bar{E}_N^T \quad (\text{A.20})$$

Applying lemma 2 immediately gives our formulas (8.8) for R_{N-1}^{-1} and R_N^{-1} .

References

1. N. Levinson, "The Weiner RMS Error Criterion in Filter Design and Prediction," *J. Math. Phys* **25**, pp.261-278 (Jan 1947).
2. William F. Trench, "An Algorithm for the Inversion of Finite Toeplitz Matrices," *J. Soc. Indust. Appl. Math* **12**, pp.515-522 (Sept 1964).
3. S. Zohar, "Toeplitz Matrix Inversion: The Algorithm of W. F. Trench," *J. Ass. Comput. Mach.* **16**, pp.529-601 (Oct 1969).
4. S. Zohar, "The Solution of a Toeplitz Set of Linear Equations," *J. Ass. Comput. Mach.* **21**(2), pp.272-276 (April 1974).
5. B. Friedlander, M. Morf, T. Kailath, and L. Ljung, "New Inversion Formulas for Matrices Classified In Terms of Their Distance from Toeplitz Matrices," *Lin. Alg. and Appl.* **27**, pp.31-60 (1979).
6. Erwin H. Bareiss, "Numerical Solution of Linear Equations with Toeplitz and Vector Toeplitz Matrices," *Numer. Math.* **13**, pp.404-424 (Oct 1969).
7. M. Morf, *Fast Algorithms for Multivariable Systems*, Ph.D. Dissertation, Dept. of Elec. Eng., Stanford Univ., Stanford, CA (1974).
8. J. Rissanen, "Algorithms for Triangular Decomposition of Block Hankel and Toeplitz Matrices with Application to Factoring Positive Matrix Polynomials," *Math. Comput.* **27**(121), pp.147-154 (Jan 1973).
9. William F. Trench, "Weighting Coefficients for the Prediction of Stationary Time Series from the Finite Past," *SIAM J. Appl. Math* **15**(6), pp.1502-1510 (Nov 1967).
10. Bradley W. Dickinson, "Efficient Solution of Linear Equations with Banded Toeplitz Matrices," *IEEE Trans. Acoust., Speech, Sig. Proc.* ASSP-27(4), pp.421-423 (Aug 1979).
11. Anil K. Jain, "Fast Inversion of Banded Toeplitz Matrices by Circular Decomposition," *IEEE Trans. on ASSP* ASSP-26(2), pp.121-126 (April 1978).
12. M. Morf and T. Kailath, "Recent Results in Least Squares Theory," *Annals of Econ. and Soc. Meas.* **6**(3), pp.261-274 (Summer 1977).
13. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, Mass. (1974).
14. Fred G. Gustavson and David Y.Y. Yun, "Fast Algorithms for Rational Hermite Approximation and Solution of Toeplitz Systems," *IEEE Trans. Circ. and Sys.* CAS-26(9) (Sept 1979).
15. Robert R. Bitmead and Brian D.O. Anderson, "Asymptotically Fast Solution of Toeplitz and Related Systems of Linear Equations," *Lin. Alg. and Appl.* **34**, pp.103-116 (1980).
16. M. Morf, "Doubling Algorithms for Toeplitz and Related Equations," *Proc. 1980 Int. Conf. on Acoustics, Speech, Sig. Proc.*, pp.954-959 (April 1980).
17. Thomas Kailath, Sun-Yuan Kung, and Martin Morf, "Displacement Ranks of Matrices and Linear Equations," *J. of Math Anal. and Appl.* **68**, pp.395-407 (April 1979).
18. B. Friedlander, T. Kailath, M. Morf, and L. Ljung, "Extended Levinson and Chandrasekhar Equations for General Discrete-Time Linear Estimation Problems," *IEEE Auto Control* AC-23, pp.653-659 (1978).
19. Bradley W. Dickinson, "Solution of Linear Equations with Rational Toeplitz Matrices," *Math. Comput.* **34**(149), pp.227-233 (Jan 1980).

20. Ralph A. Wiggins and Enders A. Robinson, "Recursive Solution to the Multichannel Filtering Problem," *J. of Geophys. Res.* **70**(8), pp.1885-1891 (April 15, 1965).
21. Hirotugu Akaike, "Block Toeplitz Matrix Inversion," *SIAM J. Appl. Math* **24**(2), pp.234-241 (March 1973).
22. I.C. Gohberg and A.A. Semencul, "On the Inversion of Finite Toeplitz Matrices and Their Continuous Analogs," *Mat. Issled* **2**, pp.201-233, (In Russian) (1972).
23. J.-M. Delosme and M. Morf, *Mixed and Minimal Representations for Toeplitz and Related Systems*, Information System Laboratory, Stanford Univ..
24. M. Morf, A. C. G. Vieira, D. T. L. Lee, and T. Kailath, "Recursive Multichannel Maximum Entropy Spectral Estimation," *IEEE Trans. on Geo. El.* **GE-16**(2), pp.85-94 (April 1978).
25. Larry Marple, "A New Autoregressive Spectrum Analysis Algorithm," *IEEE Trans. Acoust., Speech, Sig. Proc.* **ASSP-28**(4), pp.441-454 (Aug 1980).
26. Anil K. Jain, "An Operator Factorization Method for Restoration of Blurred Images," *IEEE Trans. Comp.* **C-26**(11), pp.1061-1071 (Nov 1977).
27. J. Rissanen and L. Barbosa, "Properties of Infinite Covariance Matrices and Stability of Optimum Predictors," *Information Sciences* **1**, pp.221-236 (1969).
28. Martin Morf, Bradley Dickinson, Thomas Kailath, and Augusto Vieira, "Efficient Solution of Covariance Equations for Linear Prediction," *IEEE Trans. Acoust., Speech, Sig. Proc.* **ASSP-25**, pp.429-433, reprinted in *Modern Spectrum Analysis*, ed. Donald G. Childers, IEEE Press (Oct 1977).
29. D. Fisher, G. Golub, O. Hald, C. Leiva, and O. Widlund, "On Fourier-Toeplitz Methods for Separable Elliptic Problems," *Math. Comput.* **28**(126), pp.349-368 (April 1974).

