HIGH PERFORMANCE ALGORITHMS TO SOLVE TOEPLITZ AND BLOCK TOEPLITZ MATRICES

BY

SRIKANTH THIRUMALAI

B.Tech., Indian Institute of Technology at Kharagpur, 1990
M.S., University of Illinois at Urbana-Champaign, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

HIGH PERFORMANCE ALGORITHMS TO SOLVE TOEPLITZ AND BLOCK TOEPLITZ MATRICES

Srikanth Thirumalai, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Kyle Gallivan, Advisor

Fast algorithms to factor Toeplitz matrices have existed since the beginning of this century. The two most notable algorithms to factor Toeplitz matrices are the Schur and the Levinson-Durbin. The former factors the Toeplitz matrix itself while the latter factors the inverse. In this thesis, we present several high performance variants of the classical Schur algorithm to factor various Toeplitz matrices. For positive definite block Toeplitz matrices, we show how hyperbolic Householder transformations may be blocked to yield a block Schur algorithm. This algorithm uses BLAS3 primitives and makes efficient use of a memory hierarchy. We present three algorithms for indefinite Toeplitz matrices. Two of these are based on look-ahead strategies and produce an exact factorization of the Toeplitz matrix. The third produces an inexact factorization via perturbations of singular principal minors. We also present an analysis of the numerical behavior of the third algorithm and derive a bound for the number of iterations to improve the accuracy of the solution. Recently, there have been several algorithms suggested to incorporate pivoting into the factorization of indefinite Toeplitz matrices by converting them to Cauchy-like matrices. We compare these algorithms from a computational standpoint and suggest a few algorithms that exploit properties such as realness and symmetry in the Toeplitz matrix while converting them to Cauchy-like matrices. In particular, we show how a Hermitian Toeplitz matrix may be converted to a real symmetric Cauchy-like matrix prior to factorization, yielding substantial savings in computation. For rank-deficient Toeplitz least-squares problems, we present a variant of the generalized Schur algorithm that avoids breakdown due to an exact rank deficiency. In the presence of a near rank deficiency, an approximate rank factorization of the Toeplitz matrix is produced. Algorithms to solve real Toeplitz least-squares problems and to obtain rank-revealing QR factorizations of real Toeplitz matrices are also presented. We demonstrate the use of the Schur algorithm in the construction of preconditioners to solve the problem of image deconvolution.

To my wife, Mona, and my parents, Vijaya and Thirumalai

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Professor Paul Van Dooren for introducing me to the problem of fast factorization of Toeplitz matrices. His patience during the long and frequent discussion sessions during the early phases of my research were instrumental in directing me into this avenue of research. In addition to his excellent mentoring, I was also deeply touched by his concern, understanding and support when I made the decision to work full-time at Cray Research, Inc., while continuing with my dissertation-related research. I am also grateful for the generous financial support extended to me through the ARPA Grant 60NANB2D1272.

I would also like to thank Professor Kyle Gallivan, my thesis advisor, who took up the responsibility of guiding me with my research after Professor Van Dooren accepted a professorship at the Université catholique de Louvain. His insights into the various algorithmic and implementation issues in this dissertation were invaluable. I would also like to thank him for making the seemingly difficult task of pursuing my research while working simultaneously at Cray Research, Inc. less burdensome than I had originally imagined. I also wish to thank Professors Paul Saylor, David Munson and W. Kenneth Jenkins for serving on my committee.

My sincere gratitude goes out to Dr. William Harrod, my manager at Cray Research, Inc., for his patience, support and encouragement during the last few months of my research. His easy accessibility and congenial nature have made it a pleasure to work for him. If it weren't for him, this thesis would have taken longer than it has.

My parents deserve appreciation for their unquestioning love as well as providing me with an academic attitude. Their innumerable sacrifices over the years have made an indelible impression on me. In particular, I will always remember and cherish those early hours of the morning in Bombay when they, despite all odds, made sure that their teenage son was awake and working on his assignments.

I would also like to thank my wife, Mona, for her patience during my extended hours at work and the support and encouragement she provided me to finish this dissertation. Her love and affection made working on my research and my job at Cray Research significantly less stressful.

Thanks are also due to my brother, Ashwath, for all the discussions we've had regarding each others work.

I wish to thank my friends in Minneapolis, Ananth and Vivek, for all the fun I have had in Minneapolis over the last couple of years and my co-workers, Ed and Sandra, for the several suggestions they made to improve my dissertation and my preliminary and final examination presentations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Several problems in science and engineering result in structured matrices such as Toeplitz, Hankel, Vandermonde, and Cauchy. These matrices are characterized by $O(n)$ elements, as opposed to unstructured matrices that have n^2 unrelated elements. As a result, it is possible to factor these matrices in $O(n^2)$ operations rather than the $O(n^3)$ operations required for unstructured matrices. In this thesis, we examine fast $(O(n^2))$ algorithms to solve Toeplitz systems and least-squares problems. To motivate the work presented in this thesis, we begin by describing some example problems in signal processing, control theory, and mathematics that give rise to Toeplitz systems. We later present a brief glimpse of the historical development of algorithms to factor Toeplitz matrices and give an outline of the rest of the thesis.

1.1 Examples of Toeplitz Problems

1.1.1 Constrained reconstruction in magnetic resonance imaging

The objective of magnetic resonance imaging is to obtain a *spectral function* that represents the spatial distribution of spectral information of the object image. Mathematically, the observed signal acquired from a phase-encoding spectroscopic experiment can be described by

$$s(\mathbf{k},t) = \int_{-\infty}^{\infty} \int_{\mathcal{D}} \rho(\mathbf{x},f) e^{-i2\pi(\mathbf{k}.\mathbf{x}+ft)}\, d\mathbf{x}\, df, \tag{1.1}$$

where $\rho(\mathbf{x},f)$ is the desired spectral function, $\mathcal{D}$ is the field-of-view, $\mathbf{k}$ is the phase-encoding wave vector, and $\mathbf{x}$ is the spatial coordinate vector. Oftentimes in spectroscopic imaging, it is not possible to obtain an accurate solution of $\rho(\mathbf{x},f)$ without *a priori* constraints. In [1], the authors use a generalized series model for the spectral function of the form

$$\rho(\mathbf{x},f) = \sum_n a_n(f)\psi_n(\mathbf{x},f). \tag{1.2}$$

This approach provides the flexibility to select the basis functions $\psi_n(\mathbf{x}, f)$, so that a variety of *a priori* constraints can be built into the model. Specifically, they use a spectral localization by imaging (SLIM) homogeneous compartment model constraints where

$$\psi_n(\mathbf{x}, f) = \rho_{\text{slim}}(\mathbf{x}, f) e^{i2\pi \mathbf{k_n}.\mathbf{x}}. \tag{1.3}$$

Knowing $\rho_{\text{slim}}(\mathbf{x}, f)$, the series coefficients $a_n(f)$ are determined under the following constraint:

$$s(\mathbf{k}, t) = \int_{-\infty}^{\infty} \int_{\mathcal{D}} \left\{ \rho_{\text{slim}}(\mathbf{x}, f) \sum_n a_n(f) e^{i2\pi \mathbf{k_n}.\mathbf{x}} \right\} e^{-i2\pi(\mathbf{k}.\mathbf{x}+ft)} \, d\mathbf{x} \, df. \tag{1.4}$$

Applying the inverse Fourier transform on both sides yields

$$\hat{s}(\mathbf{k}, f) = \int_{\mathcal{D}} \left\{ \rho_{\text{slim}}(\mathbf{x}, f) \sum_n a_n(f) e^{i2\pi \mathbf{k_n}.\mathbf{x}} \right\} e^{-i2\pi(\mathbf{k}.\mathbf{x})} \, d\mathbf{x}. \tag{1.5}$$

If we set

$$H(\mathbf{k}, f) = \int_{\mathcal{D}} \rho_{\text{slim}}(\mathbf{x}, f) e^{-i2\pi(\mathbf{k}.\mathbf{x})} d\mathbf{x}, \tag{1.6}$$

then (1.5) can be re-written as

$$\hat{s}(\mathbf{k}, f) = \sum_n a_n(f) H(\mathbf{k} - \mathbf{k_n}, f). \tag{1.7}$$

This is clearly a convolution operation. The coefficient matrix $H(f)$ is a Toeplitz matrix in the 1D case. In higher dimensions (2D and 3D), the matrix has nested levels of Toeplitzness. Solving the above equation for the coefficients $a_n(f)$ is a least-squares problem that requires regularization [2].

1.1.2 Restoration of blurred images

The problem of restoring an image that has been blurred by a linear space-invariant 2D impulse response can be expressed as

$$f(x, y) = \int_{\mathcal{D}} h(x - x', y - y') g(x', y') \, dx' \, dy'. \tag{1.8}$$

Here, $\mathcal{D}$ is the field-of-view, $h(x, y)$ is the space-invariant 2D impulse response, $g(x, y)$ is the original image and $f(x, y)$ is the blurred image. This problem can be expressed in matrix form as a least-squares problem with a coefficient matrix that is block Toeplitz with Toeplitz blocks. In the 1D case, the coefficient matrix simplifies to a point Toeplitz matrix. Again, as in the MRI problem, the solution (in this case the original image) is obtained by solving a least-squares problem using regularization.

1.1.3 Digital Wiener filtering

Wiener filtering, linear prediction, and predictive deconvolution of seismic traces all give rise to systems of equations that are Toeplitz in nature [3]. In this subsection, we briefly describe the mathematical formulation of designing a Wiener filter.

Consider the problem of approximating a desired signal $d(n)$ with an FIR filtered sequence $y(n)$. Let the input signal to the FIR filter be $x(n)$. We approximate the desired signal in the least-squares sense. This modeling assumes, of course, that an FIR filter can be used to obtain the desired response with the input signal $x(n)$.

Let the filter be defined as

$$W(z) = \sum_{k=0}^{p} a_k z^{-k}. \tag{1.9}$$

The optimal Wiener filter is obtained by solving the normal equations for least-squares problems. The error signal is written as

$$e(n) = y(n) - d(n) = \sum_{k=0}^{p} a_k x(n-k) - d(n). \tag{1.10}$$

Since the error signal $e(n)$ must be orthogonal to the basis signals $x(n-j), j = 0, \cdots, andp$, we have the following equations:

$$\begin{aligned}
E\{e(n)x(n-j)\} &= 0, \qquad j = 0, 1, \cdots, p && (1.11)\\
E\left\{\left[\sum_{k=0}^{p} a_k x(n-k) - d(n)\right] x(n-j)\right\} &= 0, \qquad j = 0, 1, \cdots, p && (1.12)\\
\sum_{k=0}^{p} a_k E\{x(n-k)x(n-j)\} &= E\{d(n)x(n-j)\}, \quad j = 0, 1, \cdots, p. && (1.13)
\end{aligned}$$

Defining $r_x(j) = E\{x(n)x(n-j)\}$ and $r_{dx}(j) = E\{d(n)x(n-j)\}$, we obtain

$$\sum_{k=0}^{p} a_k r_x(j-k) = r_{dx}(j), \qquad j = 0, 1, \cdots, p. \tag{1.14}$$

In matrix form, the coefficient matrix is a symmetric positive definite Toeplitz matrix, and the filter coefficients can be obtained by solving the Toeplitz system of equations.

1.1.4 System identification

In control theory, state space realizations are a popular means of describing a linear, time-invariant system. Consider a single-input-single-output system represented by the following state space equations:

$$\begin{aligned} x_{k+1} &= A\, x_k + b\, u_k \\ y_k &= c\, x_k + d\, u_k, \end{aligned} \tag{1.15}$$

where A is of size $n \times n$, b and c are of size $n \times 1$, and d is a scalar. The vector x_k of size $n \times 1$ is the state vector at time k. The input and output are denoted by u_k and y_k, respectively. The problem of determining the matrices A, b, c, and d from the impulse response sequence h_i, $i \geq 0$ gives rise to Hankel matrices [4]. Such a matrix can be permuted to a Toeplitz matrix. The transfer function of the system, $H(z)$, can be written as

$$H(z) = h_0 + z^{-1}h_1 + z^{-2}h_2 + \cdots = c(zI - A)^{-1}b + d. \tag{1.16}$$

Using the expansion of $(zI - A)^{-1}$ we see that

$$h_0 + z^{-1}h_1 + z^{-2}h_2 + z^{-3}h_3 + \cdots = d + cbz^{-1} + cAbz^{-2} + cA^2bz^{-3} + \cdots \tag{1.17}$$

This gives us the following identities:

$$\begin{aligned} h_0 &= d \\ h_i &= cA^{i-1}b, \qquad i \geq 1 \end{aligned} \tag{1.18}$$

These can now be arranged in an infinite Hankel matrix.

$$\mathcal{H} \doteq \begin{bmatrix} h_1 & h_2 & h_3 & h_4 & \dots \\ h_2 & h_3 & h_4 & \iddots & \dots \\ h_3 & h_4 & \iddots & \iddots & \dots \\ h_4 & \iddots & \iddots & \iddots & \\ \vdots & \vdots & \vdots & & \end{bmatrix} = \begin{bmatrix} c \\ cA \\ cA^2 \\ cA^3 \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} b & Ab & A^2b & A^3b & \dots \end{bmatrix}. \tag{1.19}$$

To obtain the state space matrices A, b, and c, the infinite Hankel matrix is factored via a rank factorization algorithm such as the SVD or a rank-revealing QR factorization. For a multiple-input-multiple-output system identification problem, the matrix H is a block Hankel matrix.

1.1.5 Other problems

Other problems that give rise to Toeplitz or Hankel matrices include the GCD algorithm to find the greatest common divisor of two polynomials [4], certain decoding algorithms for BCH codes [5, 6], and the classical Schur algorithm to test if the modulus of an analytic function does not exceed unity in the open unit disc [7].

In this subsection, we have shown several example problems in signal processing, control theory, and mathematics that give rise to Toeplitz matrices. Toeplitz matrices have been studied for close to a century, and the following section briefly outlines the progress in this area from a historical perspective.

1.2 A Brief History of Toeplitz Solvers

1.2.1 Factoring Toeplitz matrices

Fast algorithms to factor Toeplitz matrices can be broadly classified into two main categories: those that factor the inverse of the Toeplitz matrix, such as the Levinson algorithm; and those that factor the Toeplitz matrix itself, such as the Schur algorithm. Most of the early work in the area of Toeplitz algorithms was concerned with the Levinson algorithm.

In [8], Levinson proposed an algorithm to solve a system of Toeplitz equations with a general right-hand side. This algorithm essentially produces a factorization of the inverse of the Toeplitz matrix. The Levinson algorithm was later adapted by Durbin [9] to solve a set of Toeplitz equations with a specific right-hand side (the Yule-Walker equations). This algorithm has half the complexity of the original Levinson algorithm. In 1964, Trench proposed an algorithm [10] to explicitly compute the inverse of a positive definite Toeplitz matrix.

The first algorithm that factored the Toeplitz matrix itself rather than the inverse was proposed by E. Bareiss in 1969 [11]. Another early algorithm in this area was that of Rissanen [12], which factored block Toeplitz and Hankel matrices. It would later be shown that these algorithms are related to the classical Schur algorithm, which has its origins in an algorithm proposed by I. Schur [7] to test whether the modulus of an analytic function does not exceed unity in the open unit disc.

An analytic function is called a Schur function if its modulus does not exceed unity in the open unit disc. Schur's theorem yields a characterization of this function in terms of certain parameters (reflection coefficients of the Levinson algorithm) that are derived from the function itself. It was pointed out by Akhiezer [13] that the Schur parameters were exactly those occurring in the classical recurrences on Szegö's orthogonal polynomials. Positive definite Toeplitz matrices are known to be closely related to Szegö's polynomials. Matrix extensions of this relationship between the Schur-Szegö parameters and positive definite Toeplitz matrices can be found in [14]. Around the same time, motivated by the Szegö recursions and the Gohberg-Semencul formula for the inverse of a Toeplitz matrix [15], Kailath, Kung, and Morf discovered that the Toeplitz structure can be encompassed in a more general *displacement structure* [16]. The displacement structure in a Toeplitz matrix is defined as follows. Consider a Toeplitz matrix T of size n defined by its first row $[t_0\ t_1\ t_2\ \ldots\ t_{n-1}]$ and its first column $[t_0\ t_{-1}\ t_{-2}\ \ldots\ t{-}n+1]^T$. Let Z be a down-shift matrix defined by

$$Z = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & \ddots & & \vdots \\ 0 & 1 & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}. \tag{1.20}$$

The *displacement equation* of T with respect to (Z, Z^T) is defined by

$$T - Z\,T\,Z^T = \begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ t_{-1} & 0 & \cdots & \cdots & 0 \\ t_{-2} & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ t_{-n+1} & 0 & \cdots & \cdots & 0 \end{bmatrix} = G\,H^T, \tag{1.21}$$

where G and H are matrices of rank ≤ 2. The rank of G and H is called the *displacement rank* of T with respect to the displacement matrices (Z, Z^T). This concept was used to show that Schur's algorithm can be used to factor positive definite Toeplitz matrices. Schur's work was later studied and extended by Lev-Ari and Kailath through the use of generating functions and complex function theory [17, 18]. It was shown that the Schur complements of a Toeplitz matrix during factorization have the same displacement structure. This concept was also used by Chun, Kailath, and Lev-Ari [19, 20, 21] to derive algorithms to factor various quasi-Toeplitz or Toeplitz-like matrices (matrices with the similar displacement structure) such as T^{-1}, $T^T T$, and $T_1 T_2$.

An important element in the factorization of Toeplitz and quasi-Toeplitz matrices using the Schur algorithm is the use of hyperbolic Householder transformations in the elimination process. From a computational point of view, Cybenko and Berry [22] showed how the application of these transformations can be organized into matrix-vector multiplication primitives (BLAS2 primitives) instead of vector-scalar primitives (BLAS1 primitives), thereby improving performance. In this thesis, we take this idea one step further for block Toeplitz matrices and matrices with high displacement rank (this contribution was published in 1994 [23]). We show that a sequence of hyperbolic Householder transformations may be blocked to yield matrix-matrix multiplication primitives (BLAS3 primitives) and thereby efficiently exploit a memory hierarchy. Another contribution of [22] was the extension of the Schur algorithm to indefinite Toeplitz matrices (in the absence of breakdown). In this thesis, we extend our blocking schemes for positive definite matrices and show how hyperbolic Householder transformations may be blocked for indefinite Toeplitz matrices (see also [24]).

For indefinite Toeplitz and quasi-Toeplitz matrices, since fast factorization schemes such as the Levinson and the Schur algorithm do not permit pivoting (pivoting destroys the Toeplitz-like displacement structure), one may encounter a singular principal minor that causes the factorization algorithm to break down. Several algorithms were suggested to "look ahead" over these singular principal minors and obtain an exact factorization. Again, as in the case of positive definite Toeplitz algorithms, most of the early algorithmic work in this area was based on the Levinson algorithm. Notable among the earlier algorithms in this area are those developed by Heinig and Rost [25] for unsymmetric, indefinite Hankel and Toeplitz matrices, and by Delsarte, Genin, and Kamp [26] for indefinite Hermitian Toeplitz matrices. These algorithms are restrictive in that they can look ahead over only exactly singular principal minors. The errors in these algorithms for the case of near singular principal minors are similar to those caused by using a very small element as a pivot in Gaussian elimination. The early look-ahead Schur algorithms [27, 28, 29] were also limited to solving Toeplitz systems with exactly singular principal minors and were mainly based on polynomial recursions.

Very recently, algorithms have been proposed to solve Toeplitz and Hankel systems with near-singular principal minors. The first algorithm with this property for Toeplitz systems was developed by Chan and Hansen [30], and was based on the Levinson algorithm. For Hankel matrices, a look-ahead Padé algorithm that could handle near-singular principal minors was proposed by Cabay and Meleshko [31]. In [32], Freund and Zha proposed another look-ahead Levinson algorithm based on formally biorthogonal polynomials that could handle both exactly and nearly singular minors. This algorithm was slightly less expensive than that of Chan and Hansen. Later, Gutknecht and Hochbruck [33] developed look-ahead Levinson and Schur algorithms based on generalizing the Levinson and Schur recurrences to the case of a nonnormal Padé table.

Most of the look-ahead algorithms discussed thus far are based on polynomial recursions of one form or another. This polynomial notation does not easily extend to block Toeplitz matrices (one would have to deal with matrix polynomials presumably). This thesis discusses a method to overcome this problem with a block Toeplitz look-ahead Schur algorithm based on recovering the generators of the Schur-complement after a look-ahead step using the Bunch-Kaufman pivoting scheme. (This contribution was published in 1994 [34].) Later, two other block Toeplitz Schur algorithms based on the idea of completion of squares were proposed independently by Gallivan, Thirumalai, and Van Dooren [35], (which are also discussed in this thesis), and by Sayed and Kailath [36]. These block Toeplitz look-ahead Schur algorithms have the ability to look ahead over both exactly and nearly singular principal minors.

1.2.2 Transforming to Cauchy-like matrices

Look-ahead algorithms for indefinite Toeplitz and Hankel matrices are cost effective only when the look-ahead step size is small. For large look-ahead steps of size m, the complexity to invert the look-ahead block (which is $O(m^3)$) makes look-ahead algorithms quite expensive. A novel idea to overcome this problem was suggested by Heinig [37] and Gohberg, Kailath and Olshevsky [38]. They converted indefinite Toeplitz and Hankel matrices to Cauchy-like matrices using fast trigonometric transforms such as the discrete Fourier transform (DFT). The displacement structure of Cauchy-like matrices is invariant to permutation. This allows pivoting to be incorporated into the factorization algorithms. This idea was used to develop pivoted factorization algorithms for indefinite Toeplitz and Hankel matrices.

One drawback of the algorithms in [37, 38] was that Hermitian Toeplitz matrices were converted to non-Hermitian Cauchy-like matrices prior to factorization. We overcome this problem with an algorithm to update the generators of a Hermitian form of the displacement equation for Hermitian Cauchy-like matrices [24]. Another algorithm that preserves the Hermitian structure in the matrix was proposed in [39].

For real, symmetric Toeplitz matrices, we draw upon several enabling results from [40, 41, 42, 43] and propose an algorithm that preserves both realness and symmetry. This algorithm can be modified to convert a Hermitian Toeplitz matrix into a real, symmetric Cauchy-like matrix which results in a substantial savings in computation.

1.2.3 Toeplitz least-squares algorithms

Most of the early work in the area of Toeplitz least-squares problems was restricted to algorithms using the lattice or ladder recursions on special Toeplitz matrices (windowed Toeplitz matrices) and fast algorithms that produced Cholesky factorizations of the normal equations for regular Toeplitz matrices.

The first algorithm that produced a QR factorization of Toeplitz matrices was developed by Sweet [44]. This algorithm performed rank-1 updates of the Q and R factors while making use of the shift-invariance structure of Toeplitz matrices. This algorithm was later simplified and made computationally more efficient by Bojanczyk, Brent, and de Hoog [45]. The Schur algorithm to factor Toeplitz and quasi-Toeplitz matrices was generalized by Chun, Kailath, and Lev-Ari [20] to produce a QR factorization by applying it to an augmented matrix of the form $[T^TT \quad T^T]$. It was later shown that the Chun, Kailath, and Lev-Ari algorithm is essentially identical to the algorithm developed by Bojanczyk, Brent, and de Hoog. All three algorithms factored the Toeplitz matrix itself, and can be classified as Schur-like algorithms. The first Levinson-like algorithm was developed by Cybenko [46]. This algorithm essentially embeds the Toeplitz matrix into a windowed Toeplitz matrix and uses a modified version of the lattice algorithm by orthogonalizing this large windowed Toeplitz matrix with respect to a non-Euclidean inner product.

All of the QR factorization algorithms mentioned above break down if the Toeplitz matrix is not full column rank. In [47], Hansen and Gesmar present a modified version of Cybenko's algorithm that looks ahead over a block of columns that may be exactly or nearly independent of the previous columns. As in all look-ahead algorithms, they assume that the step size is not too large. In this thesis, we present a modification of the generalized Schur algorithm that skips over exactly linearly dependent columns of the Toeplitz matrix without incurring a performance penalty for large block sizes [24]. For nearly rank deficient Toeplitz matrices, selecting a threshold yields an approximate rank factorization. For an exact rank factorization, one can adapt the Chun, Kailath, and Lev-Ari algorithm to Cauchy-like matrices to obtain a rank-revealing QR-like algorithm for Toeplitz matrices. Such an algorithm is discussed in this thesis.

In this section, we have briefly outlined the historical development of fast algorithms for solving Toeplitz systems and least-square problems. A more detailed review of the displacement structure concept and its applications to signals, systems, and control can be found in [48].

1.3 Outline of the Thesis

We begin Chapter 2 by discussing the concept of displacement structure as applied to a variety of structured matrices such as Toeplitz, Hankel, Cauchy, and Vandermonde. This concept is the basis of all algorithms that are presented in this thesis. We then proceed to describe a block generalization of the classical Schur algorithm for block Toeplitz matrices.

Certain algorithmic and architecture independent implementation issues are also discussed in this chapter. In Chapter 3, we discuss the implementation of the block Schur algorithm on a few high performance architectures such as massively parallel processors (the Cray T3D) and parallel vector processors (the Cray J90). Chapter 4 deals with algorithms to solve indefinite Toeplitz and block Toeplitz matrices. We present three algorithms: two are look-ahead Schur algorithms that produce exact factorizations of the Toeplitz matrices, and the third produces an inexact factorization in the presence of singularities by introducing perturbations. The exact solution is then obtained using a few steps of iterative refinement. An analysis of this algorithm and an upper bound on the number of iterations are also presented. In Chapter 5, we discuss algorithms to factor indefinite Toeplitz matrices by transforming them to Cauchy-like matrices. There has been a large amount of work done in this area over the last few years. We compare all known algorithms in this area from a computational standpoint and point to versions that are suitable given the kind of Toeplitz matrix at hand. In Chapter 6, we present variants of the generalized Schur algorithm and algorithms based on converting to Cauchy-like matrices to solve Toeplitz least-squares and QR factorization problems. In Chapter 7, we discuss an application of the Schur algorithm in the construction of preconditioners to solve the problem of iterative image deconvolution. Chapter 8 concludes this thesis with a summary of the thesis and directions for future work.

CHAPTER 2

DISPLACEMENT STRUCTURE PRELIMINARIES AND A BLOCK SCHUR ALGORITHM

In this chapter, we briefly review the displacement structure concepts for various structured matrices, with specific emphasis on Toeplitz and block Toeplitz matrices. We then present a block generalization of the classical Schur algorithm for block Toeplitz matrices. Various algorithmic and architecture-independent implementation issues are also discussed.

2.1 Displacement Structure Preliminaries

In [16], Kailath, Kung, and Morf showed that the structure in Toeplitz matrices could, in fact, be generalized to include a broader class of matrices with a displacement structure. In this section, we discuss the concept of displacement structure as applied to various structured matrices such as Toeplitz, Hankel, Vandermonde, and Cauchy. This concept is fundamental to the existence of fast algorithms and is used in all the algorithms discussed in this thesis.

Consider an unsymmetric Toeplitz matrix T of size n and a down-shift matrix Z defined by

$$T = \begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ t_{-1} & t_0 & t_1 & \ddots & \vdots \\ t_{-2} & t_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & t_1 \\ t_{-n+1} & \cdots & \cdots & t_{-1} & t_0 \end{bmatrix} \qquad Z = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & \ddots & & \vdots \\ 0 & 1 & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}. \tag{2.1}$$

The displacement equation of T with respect to (Z, Z^T) is defined by

$$T - Z\,T\,Z^T = \begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ t_{-1} & 0 & \cdots & \cdots & 0 \\ t_{-2} & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ t_{-n+1} & 0 & \cdots & \cdots & 0 \end{bmatrix} = G\,H^T, \tag{2.2}$$

where

$$G = \begin{bmatrix} 1 & 0 \\ 0 & t_{-1} \\ 0 & t_{-2} \\ \vdots & \vdots \\ 0 & t_{-n+1} \end{bmatrix} \qquad H^T = \begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}. \tag{2.3}$$

G and H are matrices of rank ≤ 2. The rank of G and H is called the *displacement rank* of T with respect to the displacement matrices (Z, Z^T). The matrices G and H are referred to as the *generators* of T, since the matrix T can be generated completely from them :

$$T = G\,H^T + Z\,G\,H^T\,Z^T + Z^2 G\,H^T\,(Z^T)^2 + \cdots + Z^{n-1} G\,H^T\,(Z^T)^{n-1}. \tag{2.4}$$

If T were a block Toeplitz matrix with a block size of m, (i.e., if the entries of T were blocks rather than scalars), then we would have to replace the down-shift matrix Z by Z^m which shifts a vector down m positions. The displacement rank of T with respect to the matrices $(Z^m, (Z^m)^T)$ is $\leq 2m$.

If we now consider T to be a symmetric positive definite point Toeplitz matrix whose first row and column are defined by the vector $[t_0\ t_1\ t_2\ \cdots\ t_{n-1}]$, then the displacement equation is of the form

$$T - Z\,T\,Z^T = G\,\Sigma\,G^T, \tag{2.5}$$

where

$$G = \begin{bmatrix} \sqrt{t_0} & 0 \\ \frac{t_1}{\sqrt{t_0}} & \frac{t_1}{\sqrt{t_0}} \\ \frac{t_2}{\sqrt{t_0}} & \frac{t_2}{\sqrt{t_0}} \\ \vdots & \vdots \\ \frac{t_{n-1}}{\sqrt{t_0}} & \frac{t_{n-1}}{\sqrt{t_0}} \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{2.6}$$

The matrix Σ is usually referred to as the signature matrix. If the entry t_0 is either very small (close to machine precision) or zero (for some indefinite Toeplitz matrices), then the generator G can be defined by

$$G = \begin{bmatrix} \frac{1+t_0}{2} & \frac{1-t_0}{2} \\ t_1 & t_1 \\ t_2 & t_2 \\ \vdots & \vdots \\ t_{n-1} & t_{n-1} \end{bmatrix}. \tag{2.7}$$

From (2.6) and (2.7), it can be seen that the rank factorization of the displacement of T is not unique.

Any matrix that has a displacement structure similar to that of a Toeplitz matrix is called a Toeplitz-like or quasi-Toeplitz matrix. An example of a quasi-Toeplitz matrix is the normal equation matrix T^TT. This matrix has a displacement rank ≤ 4 with respect to the displacement matrices (Z, Z^T). The generator of such a matrix can be computed in $O(n \log(n))$ operations. An algorithm to compute the generator of T^TT is given in [20]. If T is a block Toeplitz matrix with a block size of m, then, correspondingly, the displacement rank of T^TT with respect to the matrices $(Z^m, (Z^m)^T)$ is $\leq 4m$. Henceforth, we use the notation Z to denote displacement matrices which shift one and m positions. The ambiguity in the notation is clarified by the context.

Other displacement matrices commonly used in fast algorithms to factor Toeplitz matrices are

$$Z_{\text{circ}} = \begin{bmatrix} 0 & & & 1 \\ 1 & \ddots & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{bmatrix} \qquad Z_{\epsilon\psi} = \begin{bmatrix} \epsilon & 1 & 0 & \cdots & 0 \\ 1 & 0 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & 1 \\ 0 & \cdots & 0 & 1 & \psi \end{bmatrix}, \tag{2.8}$$

where ϵ and ψ can take on values -1, 0, or 1. The displacement rank of a Toeplitz matrix T with respect to Z_{circ} is ≤ 2, whereas that with respect to $Z_{\epsilon\psi}$ is ≤ 4. The choice of the displacement matrix is dependent on the factorization algorithm.

A Hankel matrix H of size n is defined by its first row and last column as

$$H = \begin{bmatrix} h_0 & h_1 & h_2 & \cdots & h_{n-1} \\ h_1 & h_2 & \iddots & \iddots & h_n \\ h_2 & \iddots & \iddots & \iddots & h_{n+1} \\ \vdots & \iddots & \iddots & \iddots & \vdots \\ h_{n-1} & h_n & h_{n+1} & \cdots & h_{2n-2} \end{bmatrix}. \tag{2.9}$$

The displacement equation of H with respect to (Z, Z^T) can be written as

$$Z\,H - H\,Z^T = G\,W\,G^T, \tag{2.10}$$

where

$$G = \begin{bmatrix} 1 & 0 \\ 0 & h_0 \\ 0 & h_1 \\ \vdots & \vdots \\ 0 & h_{n-2} \end{bmatrix} \qquad \text{and} \qquad W = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}. \tag{2.11}$$

We now show the displacement structure in an ordinary Vandermonde matrix V defined as

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}. \tag{2.12}$$

Consider a diagonal matrix $\Omega = \text{diag}([\frac{1}{x_0}\ \frac{1}{x_1}\ \frac{1}{x_2}\ \cdots\ \frac{1}{x_{n-1}}])$. The displacement equation of V with respect to (Ω, Z^T) is given by

$$\Omega\, V - V\, Z^T = \begin{bmatrix} \frac{1}{x_0} & 0 & 0 & \cdots & 0 \\ \frac{1}{x_1} & 0 & 0 & \cdots & 0 \\ \frac{1}{x_2} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ \frac{1}{x_{n-1}} & 0 & 0 & \cdots & 0 \end{bmatrix} = G\, H^T, \tag{2.13}$$

where G and H are both matrices with a rank of 1. A commonly occurring Vandermonde matrix is the discrete Fourier transform matrix.

Another structured matrix that has assumed a great deal of importance in pivoted factorization algorithms for structured matrices is the Cauchy matrix. Consider two vectors: $\mathbf{x} = [x_0\ x_1\ x_2\ \cdots\ x_{n-1}]$, and $\mathbf{y} = [y_0\ y_1\ y_2\ \cdots\ y_{n-1}]$. Further assume that $x_i \neq 0, y_j \neq 0$ and $x_i - y_j \neq 0$ for $i, j = 0,\ 1,\ \cdots,\ and n - 1$. A Cauchy matrix C is then defined as

$$C(i,j) = \frac{1}{x_i - y_j} \qquad \text{for } i,j = 0,\ 1,\ \cdots,\ n-1. \tag{2.14}$$

Let $\Omega = \text{diag}(\mathbf{x})$ and $\Psi = \text{diag}(\mathbf{y})$. The displacement equation of C with respect to (Ω, Ψ) is given by

$$\Omega\, C - C\, \Psi = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \end{bmatrix}. \tag{2.15}$$

The displacement rank of C with respect to (Ω, Ψ) is 1. More generally, any matrix A that has a low displacement rank with respect to two diagonal matrices is called a Cauchy-like matrix.

An important property of Cauchy-like matrices is that the structure of the displacement equation is not destroyed by a permutation. This can be demonstrated easily for the Cauchy matrix of (2.14). Let P be a permutation matrix that interchanges any two rows when premultiplied with a matrix. Using the property that $P^T P = I$, we have

$$(P\,\Omega\,P^T)\,(P\,C) - (P\,C)\,\Psi = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \end{bmatrix}. \tag{2.16}$$

It can be seen that the displacement structure of PC with respect to $(P\Omega P^T, \Psi)$ is unchanged. It is this property of Cauchy and Cauchy-like matrices that allows pivoting within fast factorization algorithms.

From the above discussion, it is clear that the most general definition of a displacement equation for a matrix A can be given as

$$\Omega\,A\,\Lambda - \Delta\,A\,\Phi = GH^T, \tag{2.17}$$

where Ω, Λ, Δ, Φ are the displacement matrices, and the rank of G and H is the displacement rank of A with respect to these matrices.

2.2 Cholesky Factorization of Symmetric Positive Definite Block Toeplitz Matrices

This sections deals with the factorization of block Toeplitz matrices using a block generalization of the classical Schur algorithm. We first describe the block Schur algorithm and later study various algorithmic and architecture-independent implementation issues. For point Toeplitz and quasi-Toeplitz matrices, implementation of the classical Schur algorithm on high-performance architectures, such as systolic arrays, shared memory multiprocessors, and parallel vector processors, has been studied in [22, 49, 50]. This section attempts to extend some of the ideas in [22] to the factorization of block Toeplitz matrices.

Central to the classical Schur algorithm are certain skew transformations called hyperbolic Householder transformations. In [22], Cybenko and Berry show how these transformations may be implemented using matrix-vector (BLAS2) primitives rather than vector-scalar (BLAS1) primitives to improve performance. In this section, we show how this idea may be extended via the blocking of hyperbolic Householder transformations such that matrix-matrix primitives (BLAS3) may be used in the factorization of block Toeplitz matrices. On machines with a memory hierarchy, such as the Alliant FX/8 and present-day RISC microprocessor-based workstations, this provides a significant improvement in performance compared to BLAS2 primitives [51].

In this section, we deal only with the factorization of symmetric positive definite block Toeplitz matrices. If the matrix is indefinite, the Schur algorithm may break down if a principal minor is encountered. Factorization of indefinite block Toeplitz matrices is discussed in a later chapter.

2.2.1 A block Schur algorithm

Let T be an $mp \times mp$ symmetric positive definite block Toeplitz matrix with a block size of $m \times m$.

$$T = \begin{bmatrix} \hat{T}_1 & \hat{T}_2 & \ldots & \hat{T}_{p-1} & \hat{T}_p \\ \hat{T}_2^T & \hat{T}_1 & \hat{T}_2 & \ldots & \hat{T}_{p-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \hat{T}_{p-1}^T & \ddots & \ddots & \ddots & \vdots \\ \hat{T}_p^T & \hat{T}_{p-1}^T & \ldots & \ldots & \hat{T}_1 \end{bmatrix}. \qquad (2.18)$$

Let Z be a block down-shift matrix defined by

$$Z = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ I_m & 0 & \ddots & \ddots & 0 \\ 0 & I_m & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & I_m & 0 \end{bmatrix}. \qquad (2.19)$$

The Schur algorithm is based on the fact that the displacement of a block Toeplitz matrix T, defined as $T - ZTZ^T$, has a rank of at most $2m$ [16].

$$T - ZTZ^T = \begin{bmatrix} \hat{T}_1 & \hat{T}_2 & \dots & \hat{T}_{p-1} & \hat{T}_p \\ \hat{T}_2^T & 0 & 0 & \dots & 0 \\ \vdots & 0 & 0 & \ddots & \vdots \\ \hat{T}_{p-1}^T & \vdots & \ddots & \ddots & 0 \\ \hat{T}_p^T & 0 & \dots & 0 & 0 \end{bmatrix}. \tag{2.20}$$

The derivation of the Schur algorithm to compute the Cholesky factorization of a symmetric positive definite block Toeplitz matrix is outlined below.

Since $\hat{T}_1$ is a symmetric positive definite matrix, we can compute its Cholesky factorization $\hat{T}_1 = L_1 L_1^T$, where L_1 is an $m \times m$ lower triangular matrix. Let $T_j = L_1^{-1}\hat{T}_j$. It is easy to see that $T_1 = L_1^T$. We now define two matrices $G_1(T)$ and $G_2(T)$ as follows [16, 21] :

$$G_1(T) = \begin{bmatrix} T_1 & T_2 & T_3 & \dots & T_p \\ 0 & T_1 & T_2 & \dots & T_{p-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & T_2 \\ 0 & 0 & \cdots & 0 & T_1 \end{bmatrix} \qquad G_2(T) = \begin{bmatrix} 0 & T_2 & T_3 & \dots & T_p \\ 0 & 0 & T_2 & \dots & T_{p-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & \ddots & T_2 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}. \tag{2.21}$$

It follows that

$$T = \begin{bmatrix} G_1^T(T) & G_2^T(T) \end{bmatrix} \begin{bmatrix} I_{mp} & 0 \\ 0 & -I_{mp} \end{bmatrix} \begin{bmatrix} G_1(T) \\ G_2(T) \end{bmatrix} = G^T W_{mp} G. \tag{2.22}$$

Here $I_m p$ is an identity matrix of size $mp \times mp$,

$$G = \begin{bmatrix} G_1(T) \\ G_2(T) \end{bmatrix} \quad \text{and} \quad W_{mp} = \begin{bmatrix} I_{mp} & 0 \\ 0 & -I_{mp} \end{bmatrix}. \tag{2.23}$$

If we can obtain a transformation matrix U which satisfies the property $U^T W_{mp} U = W_{mp}$ such that $UG = R$, where R is upper triangular, then we have

$$\begin{aligned} T &= G^T W_{mp} G = G^T U^T W_{mp} U G \\ &= \begin{bmatrix} R^T & 0 \end{bmatrix} \begin{bmatrix} I_{mp} & 0 \\ 0 & -I_{mp} \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} \\ &= R^T R, \end{aligned} \tag{2.24}$$

which gives us the Cholesky factorization of T [22]. The transformation matrix U that satisfies the property $U^T W_{mp} U = W_{mp}$ is called a hyperbolic Householder transformation [52]. The basic properties of hyperbolic Householder reflectors are discussed in Section 2.2.2. Since the matrix G consists of two upper triangular block Toeplitz matrices, we show in Section 2.2.4 that considerable computational savings can be obtained by working with a generator matrix defined using the first block rows of G_1 and G_2 as

$$Gen = \begin{bmatrix} T_1 & T_2 & \cdots & T_{p-1} & T_p \\ 0 & T_2 & \cdots & T_{p-1} & T_p \end{bmatrix}. \tag{2.25}$$

It can also be seen that the above generator matrix Gen is obtained by a factorization of the displacement of the block Toeplitz matrix into

$$T - ZTZ^T = Gen^T \begin{bmatrix} I_m & 0 \\ 0 & -I_m \end{bmatrix} Gen. \tag{2.26}$$

Note that when $\hat{T}_1$ is not positive definite, we can consider the more general decomposition $T_1 = L_1 \Sigma L_1^T$, where Σ is some signature matrix with values of ± 1 on diagonal. This will exist provided $\hat{T}_1$ has nonsingular leading principal submatrices. The blocks T_j are obtained by $T_j = (L_1 \Sigma)^{-1} \hat{T}_j$, and the W_{mp} matrix becomes

$$W_{mp} = \begin{bmatrix} I_p \otimes \Sigma & 0 \\ 0 & -I_p \otimes \Sigma \end{bmatrix}. \tag{2.27}$$

We again use hyperbolic Householder transformations (now with respect to the new signature matrix W_{mp}) to reduce G to an upper triangular matrix. A detailed discussion of the Schur algorithm for indefinite Toeplitz matrices is presented in Chapter 4.

2.2.2 Hyperbolic Householder transformations

In [22], Cybenko and Berry use hyperbolic Householder transformations [52] to reduce the generator matrix G of a scalar Toeplitz matrix to an upper triangular matrix. We extend their idea to block hyperbolic Householder transformations (required in the block Schur algorithm), using representations very similar to those proposed in [53] and [54].

Let W be a diagonal matrix whose entries are either $+1$ or -1. It is easy to verify that the matrix W satisfies the equalities

$$W^2 = I \qquad \text{and} \qquad W^T = W. \tag{2.28}$$

Any matrix U that satisfies the equation $U^TWU = W$ is called a W-unitary matrix. Let x be a column vector such that $x^TWx \neq 0$. A hyperbolic Householder matrix is defined as

$$U_x = W - \frac{2xx^T}{x^TWx}. \tag{2.29}$$

It is easily verified that U_x is W-unitary (i.e., $U_x^TWU_x = W$). These transformations can be used to map one vector to another as long as they have the same hyperbolic norm (i.e., if $a^TWa = b^TWb$). In our algorithm, we reduce the generator matrix to an upper triangular matrix by successively zeroing elements below the diagonal of columns of the G matrix in (2.23). Given a column vector u, we would like to find a hyperbolic Householder matrix U_x such that

$$U_xu = -\sigma e_j, \tag{2.30}$$

where e_j is a column vector whose j^{th} element is 1; other elements are 0; and σ is a constant. We assume here that $e_j^TWe_j = 1$ (i.e., the j-th component corresponds to a $+1$ in W). Also all of the transformations constructed when the matrix T we decompose is positive definite will have an associated vector u with positive hyperbolic norm. Choosing

$$\sigma = \frac{u_j}{|u_j|}\sqrt{u^T W u}, \tag{2.31}$$

then u and σe_j have the same hyperbolic norm. If we take $x = Wu + \sigma e_j$, it can be shown that U_x is a hyperbolic Householder transformation that maps u to $-\sigma e_j$.

2.2.3 Block hyperbolic Householder representations

If we have to perform a sequence of hyperbolic Householder transformations, we could block these transformations together, and then apply this block to the appropriate matrices. This allows us to use BLAS3 primitives, rather than BLAS2 operations if we applied the transformations sequentially. Storage-efficient ways to block regular Householder transformations are derived in [53] and [54]. We extend these methods to hyperbolic Householder transforms.

Suppose $U^{(r)} = U_r U_{r-1} \ldots U_2 U_1$ is a product of r $n \times n$ hyperbolic Householder matrices. The matrix U can be written in two forms corresponding to the VY form and the YTY^T form derived in [53] and [54]. The two forms of the VY representation differ in the types of primitives they use.

Lemma 2.1 *Suppose $U^{(k)} = W^k + V_k Y_k^T$ is a product of k $n \times n$ hyperbolic Householder matrices, where V_k and Y_k are $n \times k$ matrices. If*

$$U_{k+1} = W - \frac{2x_{k+1}x_{k+1}^T}{x_{k+1}^T W x_{k+1}} \qquad \textit{and} \qquad z_{k+1} = \frac{-2x_{k+1}^T U^{(k)}}{x_{k+1}^T W x_{k+1}},$$

then

$$U^{(k+1)} = U_{k+1}U^{(k)} = W^{k+1} + V_{k+1}Y_{k+1}^T,$$

where $V_{k+1} = [WV_k \;\; x_{k+1}]$ and $Y_{k+1} = [Y_k \;\; z_{k+1}^T]$. We call this the first VY form.

Proof. If $r = 1$, then $U^{(1)} = U_1 = W - 2x_1x_1^T/(x_1^T W x_1)$, and we assign $V_1 = x_1$ and $Y_1 = -2x_1/x_1^T W x_1$, in order to have the desired form.

$$\begin{aligned} U_{k+1}\, U^{(k)} &= (W - \frac{2x_{k+1}x_{k+1}^T}{x_{k+1}^T W x_{k+1}})(W^k + V_k Y_k^T) \\ &= W^{k+1} + WV_kY_k^T - \frac{2x_{k+1}x_{k+1}^T U^{(k)}}{x_{k+1}^T W x_{k+1}} \end{aligned}$$

$$\begin{aligned} &= W^{k+1} + WV_kY_k^T + x_{k+1}z_{k+1} \\ &= W^{k+1} + [WV_k \;\; x_{k+1}] \begin{bmatrix} Y_k^T \\ z_{k+1} \end{bmatrix} \\ &= W^{k+1} + V_{k+1}Y_{k+1}^T. \end{aligned}$$

$\square$

Lemma 2.2 *Suppose $U^{(k)} = W^k + V_kY_k^T$ is a product of k $n \times n$ hyperbolic Householder matrices, where V_k and Y_k are $n \times k$ matrices. If*

$$U_{k+1} = W - \frac{2x_{k+1}x_{k+1}^T}{x_{k+1}^T W x_{k+1}} \qquad \text{and} \qquad z_{k+1} = \frac{-2x_{k+1}^T W^k}{x_{k+1}^T W x_{k+1}},$$

then

$$U^{(k+1)} = U_{k+1}U^{(k)} = W^{k+1} + V_{k+1}Y_{k+1}^T,$$

where $V_{k+1} = [U_{k+1}V_k \;\; x_{k+1}]$ and $Y_{k+1} = [Y_k \;\; z_{k+1}^T]$. We call this the second VY form.

Proof. If $r = 1$, then $U^{(1)} = U_1 = W - 2x_1x_1^T/(x_1^TWx_1)$, and we assign $V_1 = x_1$ and $Y_1 = -2x_1/x_1^TWx_1$ in order to have the desired form.

$$\begin{aligned} U_{k+1}U^{(k)} &= (W - \frac{2x_{k+1}x_{k+1}^T}{x_{k+1}^T W x_{k+1}})(W^k + V_kY_k^T) \\ &= W^{k+1} + U_{k+1}V_kY_k^T - \frac{2x_{k+1}x_{k+1}^T W^k}{x_{k+1}^T W x_{k+1}} \\ &= W^{k+1} + U_{k+1}V_kY_k^T + x_{k+1}z_{k+1} \\ &= W^{k+1} + [U_{k+1}V_k \;\; x_{k+1}] \begin{bmatrix} Y_k^T \\ z_{k+1} \end{bmatrix} \\ &= W^{k+1} + V_{k+1}Y_{k+1}^T. \end{aligned}$$

$\square$

Lemma 2.3 *Suppose $U^{(k)} = W^k + Y_kT_kY_k^TW^{k-1}$ is a product of k $n \times n$ hyperbolic Householder matrices, where Y_k is an $n \times k$ matrix and T_k is a $k \times k$ matrix. If*

$$U_{k+1} = W - \frac{2x_{k+1}x_{k+1}^T}{x_{k+1}^T W x_{k+1}}, \qquad a_{k+1} = -\frac{2}{x_{k+1}^T W x_{k+1}}(x_{k+1}^T Y_k T_k)$$

and

$$b_{k+1} = -\frac{2}{x_{k+1}^T W x_{k+1}},$$

then

$$U^{(k+1)} = U_{k+1} U^{(k)} = W^{k+1} + Y_{k+1} T_{k+1} Y_{k+1}^T W^k,$$

where $Y_{k+1} = \begin{bmatrix} WY_k & x_{k+1} \end{bmatrix}$ *and* $T_{k+1} = \begin{bmatrix} T_k & 0 \\ a_{k+1} & b_{k+1} \end{bmatrix}$.

Proof. For $k = 1$, it can be seen that $U_1 = W + Y_1 T_1 Y_1^T$, where $Y_1 = x_1$ and $T_1 = -2/x_1^T W x_1$.

$$\begin{aligned} U^{(k+1)} &= (W - \frac{2x_{k+1}x_{k+1}^T}{x_{k+1}^T W x_{k+1}})(W^k + Y_k T_k Y_k^T W^{k-1}) \\ &= W^{k+1} + x_{k+1}(-\frac{2}{x_{k+1}^T W x_{k+1}})(x_{k+1}^T W^k) + (WY_k)T_k(Y_k^T W^{k-1}) \\ &\quad + x_{k+1}(-\frac{2}{x_{k+1}^T W x_{k+1}} x_{k+1}^T Y_k T_k)(Y_k^T W^{k-1}) \\ &= W^{k+1} + x_{k+1} b_{k+1}(x_{k+1}^T W^k) + (WY_k)T_k(Y_k^T W^{k-1}) + x_{k+1} a_{k+1}(Y_k^T W^{k-1}) \\ &= W^{k+1} + \begin{bmatrix} WY_k & x_{k+1} \end{bmatrix} \begin{bmatrix} T_k & 0 \\ a_{k+1} & b_{k+1} \end{bmatrix} \begin{bmatrix} Y_k^T W^{k-1} \\ x_{k+1}^T W_k \end{bmatrix} \\ &= W^{k+1} + Y_{k+1}\, T_{k+1}\, Y_{k+1}^T\, W^k. \end{aligned}$$

□

The three blocking schemes discussed above differ in the computational primitives employed and the amount of storage required. A detailed performance analysis of the three blocking schemes is presented in Section 2.3.

2.2.4 The factorization algorithm

The following algorithm is used to reduce matrix G (2.23), described in Section 2.2.1, to an upper triangular matrix. It is essentially the same as the one described in [22], except that we are dealing with blocks instead of elements.

Let $T = G^T W_{mp} G$ where

$$G = \left[\begin{array}{c|ccccc} T_1 & T_2 & T_3 & T_4 & \cdots & T_p \\ 0 & T_1 & T_2 & T_3 & \ddots & T_{p-1} \\ 0 & 0 & T_1 & T_2 & \ddots & T_{p-2} \\ 0 & 0 & 0 & T_1 & \ddots & \vdots \\ \vdots & & & & \ddots & \vdots \\ \hline 0 & \boxed{T_2} & T_3 & T_4 & \cdots & T_p \\ 0 & 0 & \boxed{T_2} & T_3 & \ddots & T_{p-1} \\ 0 & 0 & 0 & \boxed{T_2} & \ddots & T_{p-2} \\ 0 & 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & \vdots \end{array}\right] \quad \text{and} \quad W_{mp} = \begin{bmatrix} I_{mp} & 0 \\ 0 & -I_{mp} \end{bmatrix}. \tag{2.32}$$

The goal of this algorithm is to reduce G into an upper triangular matrix using block hyperbolic Householder matrices. Since the first column of the generator is already in the right form, we use only the generator matrix from the second row down. The first row of the upper submatrix of the generator is the first block row of the triangular factor of the Toeplitz matrix. The first step in this algorithm therefore involves eliminating the first diagonal in the lower half of the generator matrix (the boxed T_2 blocks in (2.32)). If this is done while maintaining the Toeplitz structure of the remaining portion of the matrix (the submatrix from the third row downwards), we can repeat the process on the smaller generator until we triangularize G.

Consider the matrix formed by stacking the second block row of the upper submatrix and the first block row of the lower submatrix as

$$G' = \begin{bmatrix} 0 & T_1 & T_2 & T_3 & \cdots & T_{p-1} \\ 0 & T_2 & T_3 & T_4 & \cdots & T_p \end{bmatrix}. \tag{2.33}$$

Let U_1 be a block hyperbolic Householder transformation that eliminates T_2 using T_1. Applying this to G', we obtain

$$U_1 G' = \begin{bmatrix} 0 & \widetilde{T}_1 & \widetilde{T}_2 & \widetilde{T}_3 & \cdots & \widetilde{T}_{p-1} \\ 0 & 0 & \hat{T}_3 & \hat{T}_4 & \cdots & \hat{T}_p \end{bmatrix}. \tag{2.34}$$

The matrix formed by stacking the third row of the upper submatrix and the second row of the lower submatrix is just a shifted version of G'. Similarly, all matrices constructed by stacking the corresponding rows in the two halves of the generator matrix are shifted versions of the G' matrix in (2.33). Hence, all the work that was needed to zero out the diagonal row of T_2 in the lower submatrix was done in the first step. At this stage, the generator matrix G has a Toeplitz submatrix in its upper half (from the third row onwards), and another Toeplitz submatrix in its lower half as

$$G = \left[\begin{array}{cc|cccc}
T_1 & T_2 & T_3 & T_4 & \cdots & T_p \\
0 & \tilde{T}_1 & \tilde{T}_2 & \tilde{T}_3 & \ddots & \tilde{T}_{p-1} \\ \cline{3-6}
0 & 0 & \tilde{T}_1 & \tilde{T}_2 & \ddots & \tilde{T}_{p-2} \\
0 & 0 & 0 & \tilde{T}_1 & \ddots & \vdots \\
\vdots & & & & \ddots & \vdots \\ \cline{3-6}
0 & 0 & \boxed{\hat{T}_3} & \hat{T}_4 & \cdots & \hat{T}_p \\
0 & 0 & 0 & \boxed{\hat{T}_3} & \ddots & \hat{T}_{p-1} \\
0 & 0 & 0 & 0 & \ddots & \hat{T}_{p-2} \\
0 & 0 & 0 & \ddots & \ddots & \vdots \\
\vdots & & & & \ddots & \vdots
\end{array}\right]. \tag{2.35}$$

The second row of the upper submatrix of G is the second block row of the triangular factor of the Toeplitz matrix. The process is then repeated on the two lower right submatrices of the generator in (2.35). After $p-2$ steps, the generator is completely triangularized.

Note that in addition to being able to work with only two block rows, we can work with the same two block rows, because the reduced generator in the next step has the same lower block row, but the upper block row is shifted to the right by one block. Before this shift is made, the upper block row must be stored in the right place in the triangular factor of the original Toeplitz matrix. At the first step of the algorithm, this reduced matrix, which we refer to as the generator matrix, is

$$Gen = \left[\begin{array}{ccccc}
T_1 & T_2 & T_3 & \ldots & T_p \\
0 & T_2 & T_3 & \ldots & T_p
\end{array}\right]. \tag{2.36}$$

Also, we see that in the first step T_1 is upper triangular because, by construction, $T_1 = L_1^T$. The diagonal elements of T_1 are sequentially used to zero out all the elements in the corresponding column of the lower block (T_2). This implies that at each step of the algorithm, the block hyperbolic Householder matrices are computed using vectors that have one non-zero element in their upper half and a non-zero lower half. This means that the V, Y matrices in the first two forms and the Y matrix in the third form have more sparsity than usual. The sparsity patterns of the matrices V, Y and Y, T and their performance implications are discussed in the next section.

2.3 Implementation

2.3.1 Overview

A simple implementation of the algorithm has three phases.

(1) The first phase generates the hyperbolic Householder transformation U given the pivot block and the block below it to be eliminated. For example, consider the matrix

$$G' = \begin{bmatrix} 0 & T_1 & T_2 & T_3 & \cdots & T_{p-1} \\ 0 & T_2 & T_3 & T_4 & \cdots & T_p \end{bmatrix}.$$

T_1 is the pivot block and T_2 is the block to be eliminated. The matrix U is a block hyperbolic Householder transformation of size $2m \times 2m$, where m is the dimension of each block.

(2) The second phase applies the block transformation U to the portion of the matrix to the right of the pivot block column

$$U\,G' = U \begin{bmatrix} 0 & T_1 & T_2 & T_3 & \cdots & T_{p-1} \\ 0 & T_2 & T_3 & T_4 & \cdots & T_p \end{bmatrix} = \begin{bmatrix} 0 & \widetilde{T}_1 & \widetilde{T}_2 & \widetilde{T}_3 & \cdots & \widetilde{T}_{p-1} \\ 0 & 0 & \widetilde{T}_3 & \widetilde{T}_4 & \cdots & \widetilde{T}_p \end{bmatrix},$$

and copies the upper block row of the generator to the appropriate location in the triangular factor of the Toeplitz matrix. If R_2 is the second block row of the upper triangular factor of T, then

$$R_2 = \begin{bmatrix} 0 & \widetilde{T}_1 & \widetilde{T}_2 & \widetilde{T}_3 & \cdots & \widetilde{T}_{p-1} \end{bmatrix}. \qquad (2.37)$$

(3) The third phase shifts the first row of blocks one block to the right:

$$G'_{next} = \begin{bmatrix} 0 & 0 & \widetilde{T}_1 & \widetilde{T}_2 & \cdots & \widetilde{T}_{p-2} \\ 0 & 0 & \widetilde{T}_3 & \widetilde{T}_4 & \cdots & \widetilde{T}_p \end{bmatrix}.$$

Depending on the architecture of the machine and parameters such as the problem size and structure of the matrix (block size m), variations of this general implementation are chosen. The next three subsections discuss several implementation issues concerning the three phases.

2.3.2 Phase 1

In Phase 1, the transformation matrix U is constructed from a sequence of hyperbolic Householder reflectors using either the VY or the YTY^T representation. The sparsity pattern of the pivot block and the block below it to be eliminated are shown in Figure 2.1. The block hyperbolic Householder transformation U used to eliminate T_2 in Figure 2.1 con-

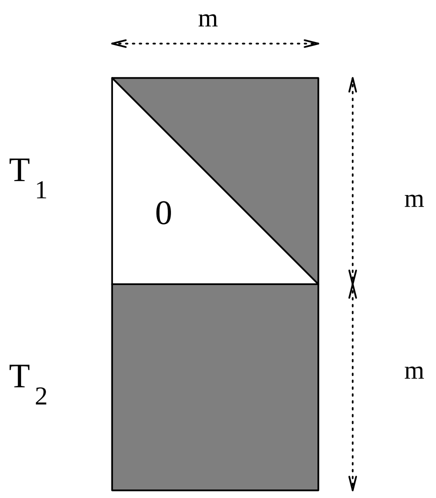


Figure 2.1 Sparsity pattern of the pivot block and the block below it.

sists of a series of hyperbolic Householder transformations $U_1, U_2, \ldots, U_m$ applied in that order. Each transformation U_k uses the (k, k) diagonal element of the upper block to zero out the elements of the k^{th} column below it. At the k^{th} step, the vector u_k has the form $(0, \ldots, 0, u_{k,k}, 0, \ldots, 0, u_{m+1,k}, \ldots, u_{2m,k})$, as discussed in Section 2.2.3.

The transformations $U_1, \ldots, U_k$ can be either applied sequentially to the rest of the generator or blocked using the blocking techniques described in Section 2.2.3. If the transformations are applied sequentially, the computation is carried out using BLAS2 primitives such as matrix-vector products and rank-1 updates. If a blocking scheme is chosen, then extra work has to be done to block the m hyperbolic Householder transformations, but the blocked transformations may be applied using BLAS3 primitives. In this section, we calculate the total number of floating-point operations for the computation of $U_i, i = 1, \ldots, m$ and their subsequent blocking. The benefit of blocking depends critically on the memory hierarchy of the machine. For parallel vector processors, such as the Cray YMP and C90 that have a high bandwidth path to memory and no cache, BLAS3 primitives do not perform much better than BLAS2 primitives. On such machines, the use of block transformations does not have a significant performance advantage. On the other hand, blocking schemes prove very useful on machines such as the Alliant FX/8 and present-day RISC-based microprocessors that have a slower main memory and a faster cache.

The transformations $U_1, \ldots, U_k$ can be blocked using either the two VY forms or the YTY^T form. The sparsity pattern of $U^{(k)}$ defined by

$$U^{(k)} = U_k U_{k-1} \ldots U_1 = W^k + V_k Y_k^T = W^k + Y_k T_k Y_k^T W^{k-1} \tag{2.38}$$

is shown in Figure 2.2. If the block hyperbolic Householder transformation is stored in factored form using the first VY form (requiring two matrix vector products), then the sparsity patterns of V and Y are shown in Figure 2.3. If the second form (requiring 1 matrix vector product and 1 rank-1 update) is used, then the sparsity pattern for V is the same as that of Y in Figure 2.3 and vice versa. If the YTY^T form is chosen, then the sparsity patterns of the corresponding matrices are shown in Figure 2.4.

Each representation of the block hyperbolic Householder transformation has different computational costs associated with producing the representation scheme and applying the transformation to the remainder of the generator. Every step in the generation of the V, Y or the Y, T matrices requires some BLAS1 routines, such as dotproducts and triads, and some BLAS2 routines, such as matrix-vector products and rank-1 updates. If the block size m is very large, on machines with hierarchical memory, such as the Alliant FX/8 or the Cedar multiprocessor, a two-level blocking scheme [51] can be used where the hyperbolic Householders are blocked every k steps, and the block transformations are applied to the remaining portion of the pivot

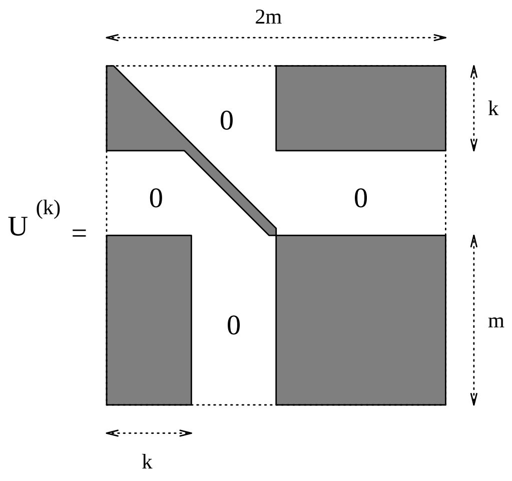


Figure 2.2 Sparsity pattern of $U^{(k)}$.

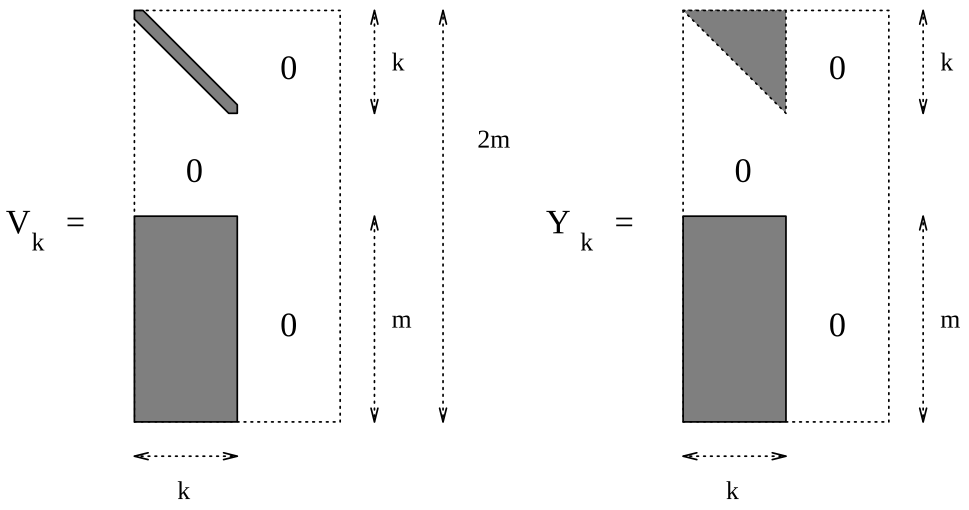


Figure 2.3 Sparsity pattern of V_k and Y_k where $U^{(k)} = W^k + V_k Y_k^T$.

block or the entire generator matrix. If the block size is small, then the generation of V, Y or Y, T can be carried through to the m^{th} step before applying it to the generator matrix.

Let us first consider the case in which the individual hyperbolic Householder transformations are not blocked but applied to the generator sequentially. At the k^{th} step, computing x_k and $-2/(x_k^T W x_k)$ requires $(3m+8)$ flops. In addition, the rest of the block has to be updated using the k^{th} hyperbolic Householder transform. The cost of this update is $(4m+3)(m-k)+(m+1)$

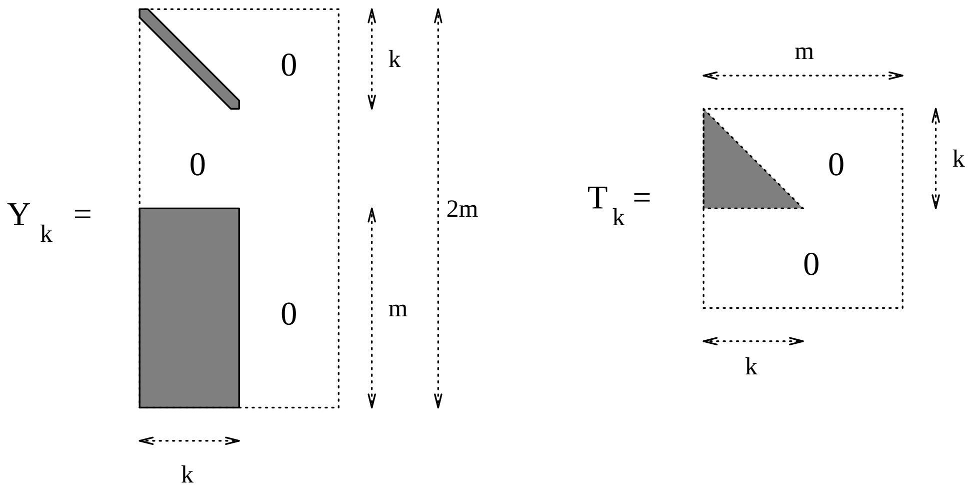


Figure 2.4 Sparsity pattern of Y_k and T_k where $U^{(k)} = W^k + Y_k T_k Y_k^T W^{k-1}$.

flops. The number of flops to compute all m hyperbolic Householder transforms is

$$\begin{aligned} \text{total flops} &= (3m+8)m + \sum_{k=1}^{m-1} (4m+3)(m-k) + (m+1) \\ &= 2m^3 + 3.5m^2 + 6.5m. \end{aligned} \tag{2.39}$$

If we choose to block the hyperbolic Householder transformations, then an extra cost is incurred.

If the VY representation of the block hyperbolic Householder reflector is chosen, then for the first form, the cost of computing V_j and Y_j from U_j and $(W^{j-1} + V_{j-1}Y_{j-1}^T)$ is $(4mj + j^2 + m + 9 - j + \lceil j/2 \rceil - 1)$ flops. For $j = 1$, the cost of computing V_1 and Y_1 is $(4m+9)$ flops. The total cost of computing V_k and Y_k using the first form is

$$\begin{aligned} \text{total flops} &\approx 4m + 9 + 0.5mk + \sum_{j=2}^{k}(4mj + j^2 + m + 9 - j/2) \\ &\approx 2mk^2 + 0.333k^3 + 3.5mk + 0.25k^2 - m + 9k \\ &\approx 2.333m^3 + 3.75m^2 + 8m \qquad \text{for } k = m. \end{aligned} \tag{2.40}$$

If the VY form computed using one matrix vector product and one rank-1 update is chosen, then the cost to compute V_1 and Y_1 is $(4m+9)$, and the cost to compute V_j and Y_j from U_j and $(W^{j-1} + V_{j-1}Y_{j-1}^T)$ is $(4mj + j + m + 8)$ flops. The total cost of computing V_k and Y_k using the first form is

$$\begin{aligned} \text{total flops} &= 4m + 9 + \sum_{j=2}^{k}(4mj + j + 0.5m + 8) \\ &= 2mk^2 + 2.5mk + 0.5k^2 - 0.5m + 8.5k \\ &= 2m^3 + 3m^2 + 8m \qquad \text{for } k = m. \end{aligned} \tag{2.41}$$

From (2.40) and (2.41), it can be seen that the first VY form is more expensive than the second VY form. However, on machines where a rank-1 update is slower than a matrix-vector product, the first VY form may be faster to compute.

If the YTY^T representation is chosen, the cost to compute Y_j and T_j from Y_{j-1} and T_{j-1} and x_j is $(2mj + 2m + 9 + j^2 - j + \lceil j/2 \rceil - 1)$ flops. The total cost of computing Y_k and T_k from $x_1, \ldots, x_k$ is

$$\begin{aligned} \text{total flops} &= 3m + 8 + \sum_{j=2}^{k}(2mj + 2m + 9 + j^2 - j + \lceil j/2 \rceil - 1) + 0.5mk \\ &\approx mk^2 + 0.333k^3 + 3.5mk + 0.25k^2 + 9k - m - 1 \\ &\approx 1.333m^3 + 3.75m^2 + 8m - 1. \end{aligned} \tag{2.42}$$

From the above calculations of total flops to compute a blocking representation, it can be seen that the YTY^T form is the least expensive. The two VY forms, albeit more expensive to compute, may be used because the cost of applying the transformation to the rest of the generator is less than the YTY^T form.

2.3.3 Phase 2

Having outlined the various schemes to block hyperbolic Householder reflectors, we discuss the cost of applying these block reflectors to the rest of the generator. As in the previous section, we begin by calculating the cost of applying the transformations $U_1, \ldots, U_k$ sequentially (without blocking) to the rest of the generator. Let the size of the generator matrix be $2m \times mp$. The cost of applying one transformation U_1 is $mp(4m + 3)$ flops. The total cost of applying k transformations sequentially is

$$\text{total flops} = 4km^2p + 3kmp$$

$$= \quad 4m^3p + 3m^2p. \qquad \text{(for } k = m\text{)} \tag{2.43}$$

If the first VY form is chosen, then the cost of applying the block reflector to a generator of size $2m \times mp$ is

$$\begin{aligned} \text{total flops} &= 4m^2pk + mpk^2 + m^2p + 3mpk \qquad \text{if } k \text{ is odd;} \\ &= 4m^2pk + mpk^2 + 3mpk \qquad \text{if } k \text{ is even;} \\ &= 5m^3p + 4m^2p \qquad \text{if } k = m \text{ and } m \text{ is odd; and} \\ &= 5m^3p + 3m^2p \qquad \text{if } k = m \text{ and } m \text{ is even.} \end{aligned} \tag{2.44}$$

In the second VY form, the Y matrix has the same sparsity pattern as the V matrix of the first form and vice versa. The cost of applying the block reflector in this form to the generator requires

$$\begin{aligned} \text{total flops} &= 4m^2pk + mpk^2 + m^2p + 2mpk \qquad \text{if } k \text{ is odd.} \\ &= 4m^2pk + mpk^2 + 2mpk \qquad \text{if } k \text{ is even.} \\ &= 5m^3p + 3m^2p \qquad \text{if } k = m \text{ and } m \text{ is odd.} \\ &= 5m^3p + 2m^2p \qquad \text{if } k = m \text{ and } m \text{ is even.} \end{aligned} \tag{2.45}$$

The cost of applying the block reflector in the YTY^T form to the rest of the generator is

$$\begin{aligned} \text{total flops} &= 4m^2pk + mpk^2 + m^2p + 4mpk \\ &= 5m^3p + 5m^2p. \end{aligned} \tag{2.46}$$

From the above calculations, it can be seen that applying the hyperbolic transformations sequentially (without blocking) is the least expensive. However, the computation uses BLAS2 primitives such as matrix-vector multiplication and rank-1 updates. On machines such as the Cray T3D on a single processor, the asymptotic computational rate for a matrix-vector multiplication is 50 Mflops, whereas that for a matrix-matrix multiplication is around 100 Mflops. This indicates that although the application of the blocked hyperbolic Householder transformations requires more raw floating-point operations, these operations are carried out much faster. Hence, using blocked hyperbolic Householder transformations may be beneficial.

Applying the hyperbolic Householder transformations that are blocked in the YTY^T form is slightly more expensive than for the two VY forms. However, on some distributed memory machines where the matrices Y and T have to be communicated to other processors, this form may prove to be very useful, because the YTY^T form is symmetric (as opposed to the two VY forms that are nonsymmetric) and requires smaller message lengths. If the cost of communicating messages critically depends on the message length, then the extra cost of applying this transform is more than offset by the shorter communication time.

2.3.4 Phase 3

Phase 3 of each step involves shifting the upper row of blocks in the generator matrix one block to the right. On shared-memory machines, this phase could be avoided if we apply the transformation matrices to the right portions of the matrix. This in-place implementation requires the V, Y matrices to be split into two $m \times m$ matrices. This not only avoids the shift of the upper block row of the generator matrix, but also allows the sparsity of the transformation matrices to the exploited.

For distributed memory machines on which portions of the generator are assigned to processors, the shift operation might include passing the local portions of the generator to a neighboring processor. In the next chapter, we suggest three different data distribution schemes for such machines. These three schemes have different amounts of data movement during the shift operation. The cost of communicating with a neighboring processor is an important parameter in deciding how to map the Schur algorithm to a linear array of processors.

2.4 LDL^T Factorization of an S.P.D. Block Toeplitz Matrix

In this section, we derive another form of the block hyperbolic Householder reflector that is used to obtain an LDL^T factorization of a symmetric positive definite block Toeplitz matrix, as opposed to a Cholesky factorization. This blocking scheme can be used if the matrix is symmetric indefinite, unless there is a breakdown. Modifications to the Schur algorithm in the presence of breakdowns are discussed in Chapter 4.

Consider a symmetric positive definite block Toeplitz matrix T having blocks $\widehat{T}_i, i = 1, \ldots, p$ of dimension $m \times m$. The generator for such a Toeplitz matrix can be written as

$$G = \begin{bmatrix} I & T_2 & T_3 & \ldots & T_p \\ 0 & T_2 & T_3 & \ldots & T_p \end{bmatrix}, \tag{2.47}$$

where $T_i = \widehat{T}_1^{-1}\widehat{T}_i, i = 1, \ldots, p$. The generator matrix shown above gives us a factorization of the displacement of the Toeplitz matrix T

$$\begin{aligned} T - ZTZ^T &= G^T \begin{bmatrix} \widehat{T}_1 & 0 \\ 0 & -\widehat{T}_1 \end{bmatrix} G \\ &= G^T W G, \end{aligned} \tag{2.48}$$

where Z is the block down shift matrix of size $mp \times mp$. The first step of the Schur algorithm for such a generator is trivial. After the shift at the end of the first step, the generator for the second step is

$$G^{(2)} = \begin{bmatrix} I & T_2 & T_3 & \ldots & T_{p-1} \\ T_2 & T_3 & T_4 & \ldots & T_p \end{bmatrix}. \tag{2.49}$$

If we choose a block hyperbolic Householder reflector U such that $U^T\widehat{W}U = W$ where $\widehat{W}$ is also block diagonal, then the factorization obtained is of the form LDL^T, where D is block diagonal. If T_{sc} is the Schur complement of T with respect to the first leading $m \times m$ block, and $\widehat{Z}$ is a block right shift matrix of size $m(p-1) \times m(p-1)$, then

$$\begin{aligned} T_{sc} - \widehat{Z}^T T_{sc} \widehat{Z} &= G^{(2)T} \begin{bmatrix} \widehat{T}_1 & 0 \\ 0 & -\widehat{T}_1 \end{bmatrix} G^{(2)} \\ &= G^{(2)T} U^T \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} U G^{(2)} \\ &= (\widehat{G}^{(2)})^T \widehat{W} \widehat{G}^{(2)}, \end{aligned}$$

where

$$\widehat{G}^{(2)} = \begin{bmatrix} I & \widetilde{T}_2 & \widetilde{T}_3 & \ldots & \widetilde{T}_{p-1} \\ 0 & \widetilde{T}_3 & \widetilde{T}_4 & \ldots & \widetilde{T}_p \end{bmatrix} \quad \text{and} \quad \widehat{W} = \begin{bmatrix} \widehat{\Sigma}_1 & 0 \\ 0 & \widehat{\Sigma}_2 \end{bmatrix}.$$

From the above equations we see that if $T = LDL^T$, then

$$\begin{aligned} L(m+1:2m, m+1:mp) &= \begin{bmatrix} I & \widetilde{T}_2 & \widetilde{T}_3 & \ldots & \widetilde{T}_{p-1} \end{bmatrix} \\ D(m+1:2m, m+1:2m) &= \Sigma_1. \end{aligned} \tag{2.50}$$

From this discussion it is obvious that we need to construct a block hyperbolic Householder reflector U such that

$$U^T \begin{bmatrix} \widehat{\Sigma}_1 & 0 \\ 0 & \widehat{\Sigma}_2 \end{bmatrix} U = \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \tag{2.51}$$

$$U \begin{bmatrix} I \\ X \end{bmatrix} = \begin{bmatrix} I \\ 0 \end{bmatrix}. \tag{2.52}$$

The steps to construct the block reflector U are shown below. From (2.51) and (2.52) it can be seen that

$$\widehat{\Sigma}_1 = \Sigma_1 + X^T \Sigma_2 X \tag{2.53}$$

and

$$U^{-1} \begin{bmatrix} I \\ 0 \end{bmatrix} = \begin{bmatrix} I \\ X \end{bmatrix} \Rightarrow U^{-1} = \begin{bmatrix} I & Y \\ X & Z \end{bmatrix}. \tag{2.54}$$

U^{-1} can be factored as

$$U^{-1} = \begin{bmatrix} I & 0 \\ X & I \end{bmatrix} \begin{bmatrix} I & Y \\ 0 & W \end{bmatrix}, \tag{2.55}$$

where $Z = XY + W$ and

$$U = \begin{bmatrix} I & -YW^{-1} \\ 0 & W^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -X & I \end{bmatrix}. \tag{2.56}$$

Substituting for U, Σ, and $\widehat{\Sigma}$ in (2.51) we obtain

$$\widehat{\Sigma}_1 = \Sigma_1 + X^T \Sigma_2 X \tag{2.57}$$

$$-\widehat{\Sigma}_1 Y W^{-1} = X^T \Sigma_2 \tag{2.58}$$

$$\Sigma_2 = W^{-T}(Y^T \widehat{\Sigma}_1 Y + \widehat{\Sigma}_2) W^{-1}. \tag{2.59}$$

If we choose $W = I$, then we have

$$\begin{aligned} \widehat{\Sigma}_1 &= \Sigma_1 + (X^T\Sigma_2)X && (2.60) \\ Y &= -\widehat{\Sigma}_1^{-1}(X^T\Sigma_2) && (2.61) \\ \widehat{\Sigma}_2 &= \Sigma_2 - Y^T\widehat{\Sigma}_1 Y = \Sigma_2 + (X^T\Sigma_2)^T Y. && (2.62) \end{aligned}$$

It can be seen from the above description that the primitives used in this blocking scheme are of the BLAS3 type. The cost of obtaining the block reflector in this form is $6.83m^3 + m^2$ flops. This is substantially higher than the cost of the previous blocking schemes, but the operations are performed at a higher rate (BLAS3 rate versus BLAS2 for the other schemes). The main advantage of this scheme over the others is that applying the block reflector to the rest of the generator of size $2m \times mp$ requires $4m^3p$ flops, which is significantly less than that of the other blocking schemes.

In this section we have described several techniques to block hyperbolic Householder transformations. These blocking schemes were motivated by the need to exploit a local memory hierarchy for performance reasons. Several machine-independent algorithmic and implementation issues were also discussed. In the next section, we discuss the performance of the block Schur algorithm on various high-performance architectures.

CHAPTER 3

PERFORMANCE RESULTS OF THE BLOCK SCHUR ALGORITHM

In this chapter, we discuss performance-related issues concerning the implementation of the block Schur algorithm on several high performance architectures. In [22], Cybenko and Berry discuss implementation and performance results of the Schur algorithm on several machines, such as the Alliant FX/80, Cray X-MP, and the Cray-2. They use BLAS2 primitives to implement hyperbolic Householder transformations and show the improvement in performance over BLAS1-based hyperbolic Givens rotations. In the previous chapter, we showed how this concept could be extended one step further by blocking the hyperbolic Householder transformations to use BLAS3 primitives.

We begin this chapter by discussing performance results of the various blocking schemes on a single processor of the Cray T3D and the IBM SP2. Each processor of these parallel machines has a memory hierarchy that includes a main memory, a secondary cache (on the SP2 only), and a primary data cache. On vector-pipeline machines such as the Cray J90, BLAS3 primitives do not yield much performance improvement over BLAS2 primitives. Implementations on the Cray J90 indicate that applying the hyperbolic Householder transformation sequentially, using BLAS2 primitives, yields a better performance than for block transformations. We also discuss the implementation of the block Schur algorithm on massively parallel machines such as the Cray T3D. The block Schur algorithm can be modified to compute the generators of the inverse of the Toeplitz matrix. This algorithm is better suited for implementation on distributed memory machines than the Cholesky factorization via the block Schur algorithm. We discuss various implementation issues concerning this algorithm and present performance results on the Cray T3D.

3.1 Performance Improvement Due to Blocking

In [51], Gallivan, Plemmons, and Sameh analyze the performance of matrix computations based on a classification into three main categories: vector-scalar or BLAS1 primitives, matrix-vector or BLAS2 primitives, and matrix-matrix or BLAS3 primitives. They show that on a machine with a memory hierarchy, organizing the computations so that BLAS3 primitives are used is highly desirable. This is true for the block Schur algorithm as well. In this section, we present corroborating evidence on a single processor of parallel machines such as the Cray T3D and the IBM SP2.

3.1.1 Overview of the Cray T3D, IBM SP2, and Alliant FX/80

The Cray T3D is a massively parallel computer in which processors are connected in the form of a 3D Torus. Each processor is, therefore, connected to six neighbors and has a peak data transfer rate of 300 MB/s to each neighbor. A single processor of the T3D is a DEC Alpha 21064 microprocessor, which is a dual-issue superscalar processor with a clock speed of 150 MHz. The peak performance of each processor is 150 Mflops. Every processor has a main memory of 64 MB, a 64 KB direct-mapped instruction cache, and a 64 KB direct-mapped, write-through data cache.

The IBM SP2 is composed of 1 to 16 frames, each containing 2 to 16 processors. A single processor of the SP2 is a POWER2© architecture RS/6000 microprocessor with a clock speed of 66.7 MHz. The peak performance of a single processor is 266 Mflops. Every processor has 64 to 512 MB of memory, a 64 KB data cache, a 32 KB instruction cache, and an optional 1 MB L2 (secondary) cache.

The Alliant FX/80 consists of up to eight register-based vector processors or computational elements (CEs), each capable of delivering a peak rate of 11.75 Mflops for calculations using 64-bit data, implying a total peak rate of approximately 94 Mflops. Each CE has eight 32-element vector registers and eight floating-point scalar registers, as well as other integer registers. The CEs are controlled by a concurrency control bus (used as a synchronization facility). The CEs share a physical memory as well as a write-back cache that allows eight simultaneous accesses per cycle. The size of the cache is 512 KB. The cache/main memory bandwidth on the machine is approximately 2.

3.1.2 Performance on a Single Processor of the Cray T3D and IBM SP2

Consider a block Toeplitz matrix of size n with a block size m. At every step of the block Schur algorithm, a sequence of m hyperbolic Householder transformations is applied to the rest of the generator. This can be done sequentially using BLAS2 primitives, or by blocking the transformations (at an extra cost of $O(m^3)$ flops) and using BLAS3 primitives. Table 3.1 shows the time, in seconds, to factor a 2048×2048 symmetric positive definite block Toeplitz matrix on a single processor of the Cray T3D. The second column indicates the time (in seconds) to factor the block Toeplitz matrix when the hyperbolic Householder transformations are applied sequentially (without blocking). The third, fourth, and fifth columns correspond to the two VY and the YTY^T blocking schemes discussed in the previous chapter. It can be seen that for

Table 3.1 Time, in seconds, to factor a 2048×2048 s.p.d. block Toeplitz matrix on one processor of the Cray T3D.

Block Size (m)	Sequential	VY1 Form	VY2 Form	YTY Form
2	3.642	4.707	4.686	4.674
4	3.270	2.776	2.796	2.818
8	3.864	2.658	2.642	2.754
16	5.113	3.432	3.403	3.404
32	8.556	5.711	5.712	5.652
64	15.765	11.650	11.949	11.281

small block sizes (in this case for $m = 2$), applying the hyperbolic Householder transformations sequentially yields a better performance than any of the blocking schemes. In addition, on the Cray T3D, there seems to be no clear winner among the blocking strategies. Note that for the three blocking schemes, though the complexity of the algorithm increases linearly with block size, the factorization times decrease when the block size is increased from 2 to 8. This is an end-case artifact of the implementation of the BLAS routine SGEMM on the machine. This indicates that on the Cray T3D, given the library routine SGEMM, it is beneficial to not exploit some Toeplitzness in the matrix and use a higher block size.

The performance results on a single processor of the IBM SP2 are shown in Table 3.2. The computation was done in double precision (64 bit floating-point number) using DGEMM from the Engineering and Scientific Subroutine Library (ESSL). From Table 3.2, we see that on the IBM SP2, as on the T3D, for a block size of 2, applying the hyperbolic Householder transforms

Table 3.2 Time, in seconds, to factor a 2048×2048 s.p.d. block Toeplitz matrix on one processor of the IBM SP2.

Block Size (m)	Sequential	VY1 Form
2	2.21	3.10
4	6.44	4.49
8	7.52	5.00
16	9.77	5.96
32	14.93	8.73
64	26.62	16.91

sequentially, is much faster than applying them after blocking. For larger block sizes, blocking a transformation before applying it to the rest of the generator yields a better performance.

3.1.3 Performance on the Alliant FX/80

Table 3.3 shows the performance of the block Schur algorithm on the Alliant FX/80. All codes are in Fortran and use either BLAS2 (sequential updates) or BLAS3 (blocked updates using the VY1 form). For a 2048×2048 block Toeplitz matrix with block sizes varying from 1 to 16, since the generator matrix fits completely in cache, all computation, whether BLAS2 or BLAS3, proceeds at the same rate. However, the blocking overhead, which increases linearly with block size, causes the BLAS3 version to be slower than the BLAS2 version. For block sizes from 32 to 128, the generator size is a multiple of cache size, hence, the BLAS2 version shows no further improvement in computational rate. The performance of the BLAS3 version continues to improve, but the blocking overhead (which is non-trivial, unlike the blocked LU factorization) causes the rate of improvement to be slow.

Having demonstrated the performance improvement due to blocking, we now present some performance results of parallel implementations of the block Schur algorithm on the Cray T3D.

3.2 Performance Results on Parallel Vector Processors

In this section, we present performance results of the block Schur algorithm on parallel vector processors. The Cray J90 series of computers are Cray's entry-level air-cooled parallel vector processor systems, with a maximum of 32 processors. We present performance results

Table 3.3 Time, in seconds, to factor a 2048×2048 s.p.d. block Toeplitz matrix on the Alliant FX/80.

Block Size(m)	hyp. Sequential	VY1 Form
1	25.46	26.20
2	22.96	22.76
4	22.57	23.18
8	25.01	32.21
16	32.04	41.92
32	62.33	53.23
64	123.59	78.26
128	229.19	129.63

on a Cray J916, which is a system in the J90 series with up to 16 CPUs and up to 4 GB of memory. The memory is organized into several modules that are connected to the CPUs via a back-plane switch.

Each CPU of a J90 consists of several CMOS ASICs, with a clock speed of 100 MHz. Every CPU has one vector unit with eight vector registers that are 64 elements long and one scalar unit. The peak vector performance of a CPU is 200 MFlops, because pipelined multiply and add units can be chained to compute two floating-point operations in a single clock cycle. Unlike traditional pipelined vector processors, the scalar unit has a 128 word (1 word = 8 bytes) two-way set-associative cache with a line size of one word. This cache is meant to improve scalar performance only (vector loads are not cached).

On vector-pipeline machines, since data can be pipelined to vector registers from main memory, the performance of BLAS2 primitives is about the same as that of BLAS3 primitives. This indicates that blocking the hyperbolic Householder transformations in the block Schur algorithm will not improve performance. Our results on the Cray J90 corroborate this fact. Table 3.4 shows the time, in seconds, to factor a 4096×4096 s.p.d. block Toeplitz matrix with block sizes $m = 32$ and $m = 64$. For each block size we consider two cases: applying the hyperbolic Householder transformations sequentially, and blocking them using the VY1 form. It can be seen that for both block sizes, the extra amount of work done in blocking the hyperbolic Householder transformations and applying the blocked transformations to the rest of the generator results in poorer performance. Blocking schemes are, therefore, unnecessary on vector-pipeline machines without a memory hierarchy.

Table 3.4 Time, in seconds, to factor a 4096×4096 s.p.d. block Toeplitz matrix with block sizes $m = 32$ and $m = 64$ on a Cray J916. The number of CPUs used varies from 1 to 12.

	$m = 32$, $p = 128$		$m = 64$, $p = 64$	
NCPUS	No Blocking	VY1	No Blocking	VY1
1	7.65	10.18	15.00	17.73
2	4.03	5.38	7.7	9.3
4	2.19	2.97	4.13	5.08
8	1.42	1.81	2.58	3.04
10	1.32	1.65	2.36	2.67
12	1.26	1.59	2.24	2.42

3.3 Implementation on Distributed Memory Multiprocessors

On parallel machines in which the memory is physically distributed across all the processors, distributing data across the processors such that there is minimal data movement across processors is crucial. This is referred to as the data distribution problem. While trying to reduce data movement, care should be taken not to severely reduce the parallelism in the implementation. On most machines, this results in a tradeoff between data communication and parallelism.

For the Schur algorithm, in addition to choosing the right blocking scheme to block the hyperbolic Householder transforms, it is important to lay out the generator across the processors in such a way that data movement during the algorithm is reduced without severely affecting the parallelism. In this section we present three data distribution schemes and discuss their usefulness, given various problem and block sizes.

Consider an s.p.d. block Toeplitz matrix T of size $mp \times mp$ with a block size of $m \times m$. The generator for this matrix at the start of the block Schur algorithm is of size $2m \times mp$. The generator has p block columns of size $2m \times m$ that correspond to the block structure of the block Toeplitz matrix. We refer to the size of the block Toeplitz matrix, $N = mp$, as the problem size. The data distribution problem in the implementation of the Schur algorithm deals with the way in which the generator is distributed across the processors. The size of the generator reduces by m columns at every step of the Schur algorithm. Consequently, the parallelism in the algorithm reduces with each step. In addition, at the end of every step, the upper m rows of the generator are to be shifted m columns to the right. Depending on the way the generator is distributed across the processors, this results in varying amounts of data being moved across processors. The trade-off between computation and communication specific to the Schur algorithm is between the shrinking parallelism in applying the block hyperbolic Householder transformation to the rest of the generator and shifting the upper half of the generator one block column to the right. Let us consider the distributed memory multiprocessor to be a linear array of P processors. In this section, we discuss three different ways to distribute the generator across this linear array of processors. Each technique results in different ratios between computation and communication. An optimal choice often has to be made after a thorough benchmarking of all computation and communication primitives used in the algorithm.

The three ways to distribute the generator matrix across a linear array of processors are

Version 1: To assign each block to a processor in a cyclic manner.

Version 2: To assign a group of b adjacent blocks to a processor.

Version 3: To divide each block among b adjacent processors.

Figure 3.1 shows the three data distribution schemes on a four-processor machine with processing elements (PEs) P_0, P_1, P_2 and P_3. In Versions 1 and 2, at every step of the Schur algorithm, the pivot block column of the generator resides wholly on one processor. The pivot block column of the generator is used to compute a block hyperbolic Householder transform, which is then communicated to the other processors through a broadcast operation. Depending on the cost of a broadcast operation and the preferred primitives on one processor of the machine, an optimal blocking scheme is chosen. In the following subsections, we examine the three data distribution schemes more closely with respect to the communication and computation trade-offs. In all three versions, we assume a compute/communicate paradigm with explicit barrier synchronization between each phase of the Schur algorithm.

3.3.1 Version 1

In this version, each block column of the generator resides completely within a processor. The block hyperbolic Householder transformation is computed by the processor that has the pivot block. This transformation is then broadcast to all the processors containing the rest of the generator. In this distribution, at the start of the Schur algorithm, the first $p - P\lfloor p/P \rfloor$ have $\lceil p/P \rceil$ block columns of the generator and the remaining processors have $\lfloor p/P \rfloor$ block columns.

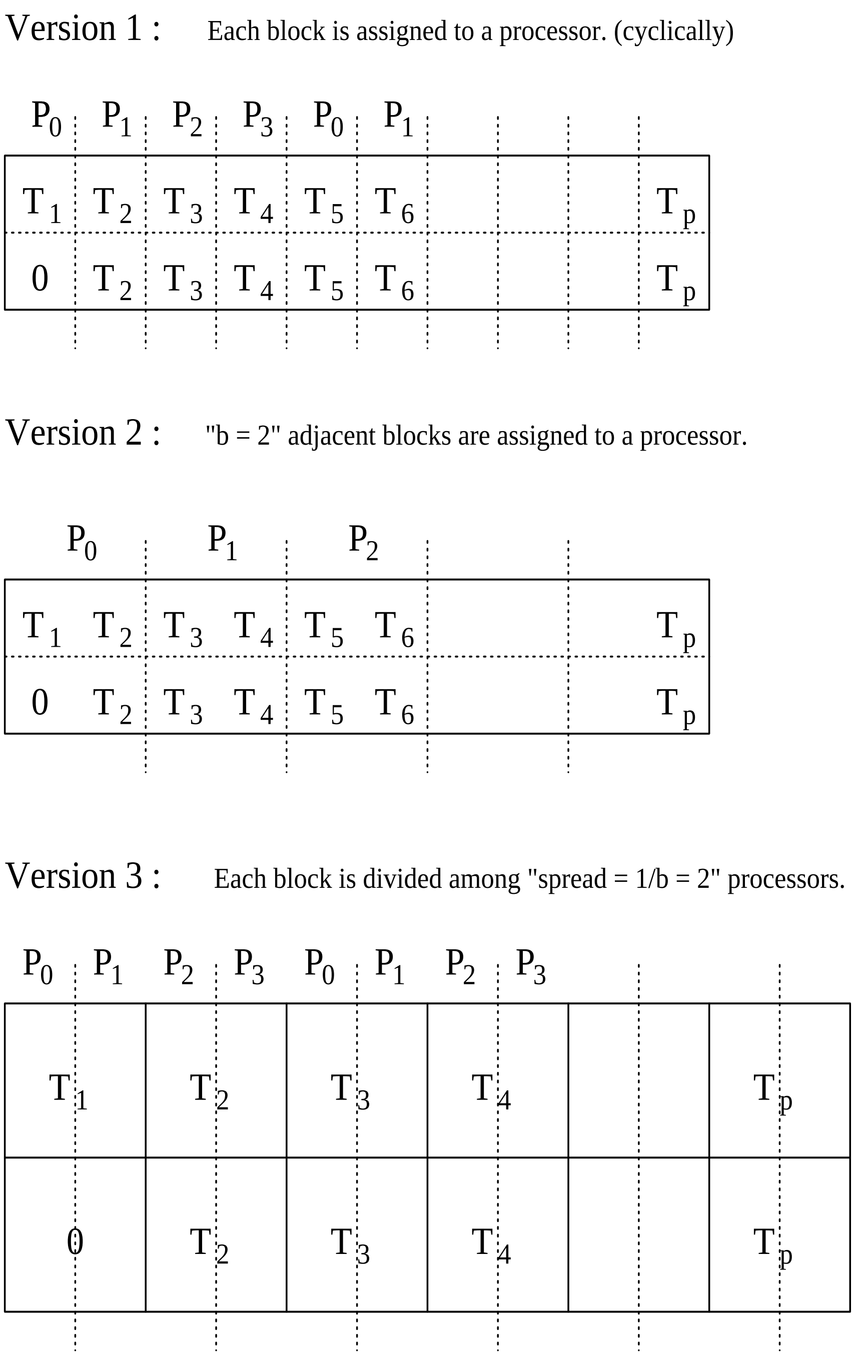

Version 1 : Each block is assigned to a processor. (cyclically)
P0 P1 P2 P3 P0 P1
T1 T2 T3 T4 T5 T6 Tp
0 T2 T3 T4 T5 T6 Tp
Version 2 : "b = 2" adjacent blocks are assigned to a processor.
P0 P1 P2
T1 T2 T3 T4 T5 T6 Tp
0 T2 T3 T4 T5 T6 Tp
Version 3 : Each block is divided among "spread = 1/b = 2" processors.
P0 P1 P2 P3 P0 P1 P2 P3
T1 T2 T3 T4 Tp
0 T2 T3 T4 Tp

Figure 3.1 Data distribution schemes for distributed memory machines.

At every step of the Schur algorithm, the number of block columns of the generator in the processor containing the pivot block reduces by one. After every P steps of the Schur algorithm, the number of block columns on all P processors reduces by one, and, after $p - P$ steps, processors start to become idle at the rate of one for every step. From this point on in the Schur algorithm, for this data distribution, the efficiency drops rapidly. If $p \gg P$, then this is not a serious problem.

As we mentioned earlier, at every step of the algorithm, the upper half of the generator is moved one block column to the right. Since we have a cyclic distribution of block columns across processors, at every step all the blocks residing on a processor will have to be shifted to the processor to the right. This results in a significant amount of data movement to remote processors (in this case the neighbor to the right) at every step of the algorithm. This problem can be mitigated if we assign multiple adjacent blocks to a single processor in a block cyclic manner. The amount of data moved across processors is decimated by approximately $1/b$, where b is the number of adjacent blocks assigned to a processor. As we shall see in the next section, this apparent benefit comes with a price.

3.3.2 Version 2

In this version, b adjacent blocks of the generator reside within a processor. Let us assume for simplicity that $p_b = p/b$ is an integer. At the start of the Schur algorithm, the first $p_b - P\lfloor P_b/P \rfloor$ processors have $b\lceil p_b/P \rceil$ block columns and the remaining processors have $b\lfloor p_b/P \rfloor$ block columns. At every step, the number of blocks in the processor containing the pivot block drops by 1. But, unlike Version 1, the onset of processors becoming idle is much earlier. After the first $p - b(P)$ steps of the Schur algorithm, processors start to become idle at the rate of one for every b steps. The load imbalance in Version 2 is, therefore, worse than that in version 1.

As far as communication is concerned, the shift of the upper half of the generator one block column to the right results in some blocks being shifted locally within a processor and some being shifted across processors. The number of blocks shifted to the processor to the right is reduced by a factor of $1/b$. This indicates that the communication cost at each step is less in Version 2 than in Version 1. In some cases this reduction in communication could offset the increased load imbalance.

For a problem size N, if m is very small and $p \gg P$, then the load imbalance occurs late in the computation, and Version 2 may be better than Version 1. For moderate block sizes, Version 1 would yield better performance because the load imbalance becomes severe in Version 2. For large block sizes, both Versions 1 and 2 would suffer from reduced parallelism because of the early onset of an unbalanced load. In such situations, one would have to split up a block and spread it across several PEs. This clearly increases the parallelism by delaying the time when processors start to become idle.

3.3.3 Version 3

In this version, a block is distributed across b adjacent processors in a cyclic manner. For simplicity let us assume that $m_b = m/b$ is an integer. At each step of the Schur algorithm, a set of b processors stores the pivot block. Each processor in this set has m_b columns of the pivot block. This distribution suggests the need for a two-level blocking strategy. The b processors that store the pivot block compute a block hyperbolic Householder transformation from the m_b sequential transforms corresponding to their share of the pivot block and broadcast this block transformation to the processors that store the rest of the generator.

At the start of the Schur algorithm there are bp second-level blocks that are distributed cyclically across P processors. Processors start to become idle after $bp - P$ second-level steps. This indicates that the load balancing problem arises much later in the algorithm than if Versions 1 or 2 were used resulting in a greater amount of parallelism.

In this distribution, the upper half of the generator is shifted m columns to the right after b second-level steps. Each processor would have to shift the upper half of all its block columns to a processor at a distance of b away. The communication cost in this scheme is higher than Version 1 because each processor communicates with a processor farther away (in Version 1, a processor communicates with its right-hand neighbor only) and the number of broadcasts increases by a factor of b. For large block sizes, when $p \approx P$, using this two level blocking strategy increases parallelism, which may offset the increased communication costs.

This suggests that for a problem size N, a block size m, and a machine size P, an optimal data distribution scheme exists that depends on the performance of the computational primitives, such as the BLAS2 and BLAS3, and the communication primitives, such as a broadcast and stores to a remote processors memory. Ideally, one would like to have a detailed model characterizing the performance of the block Schur algorithm on distributed memory machines based on the variables just described. An optimal data distribution scheme could then be calculated and suggested to the user. In the following section, we demonstrate the existence of such an optimal data distribution for some problem sizes on the Cray T3D.

3.4 Implementation Results on the Cray T3D

The three versions described above were implemented on the Cray T3D. It was seen that for small block sizes Version 2 was optimal. For medium block sizes Version 1 was found to be the preferred implementation, and, for large block sizes, Version 3 yielded the lowest factorization times. The VY1 blocking scheme was used to block the hyperbolic Householder transformations. The communication library used in the experiments was the *Shmem library*, which is based on low latency *puts* and *gets* from remote memory. The *shmem_put* routine uses no buffers. It writes directly to a remote processor's memory without any interference by the remote processor. The broadcast was done using the *shmem_broadcast* routine.

Consider a 4096×4096 point Toeplitz matrix ($m = 1$). Let $P = 32$. The time to factor the matrix (in seconds) using Versions 1 and 2 is shown in Figure 3.2. In this example $p \gg P$. Initially, increasing b such that $(p/b) \gg P$ does not affect parallelism. The communication cost, however, is reduced significantly. This results in a sharp fall in the time to factor the Toeplitz matrix. The best time is obtained when $b = 16$. When b is increased to 32 and 64, the reduction in parallelism outweighs the reduced communication and the execution time starts to increase. If the shift operation on the T3D were slower, then the optimal b is greater than 16, whereas if the shift operation were quicker, we would not have seen a significant reduction in execution times by increasing b.

Consider a 4096×4096 block Toeplitz matrix with $m = 8$. Let $P = 64$. The time to factor the matrix using all three data distribution schemes is shown in Figure 3.3. A value of b greater than 1 implies that Version 2 was used, whereas a value less than 1 implies that Version 3 was used. Version 1 was used when $b = 1$. It can be seen that, for moderate block sizes, if the parallelism is adequate (i.e., $p \gg P$), then Version 1 provides the fastest factorization scheme. On the T3D, experiments show that for block sizes around 8, Version 1 is the fastest

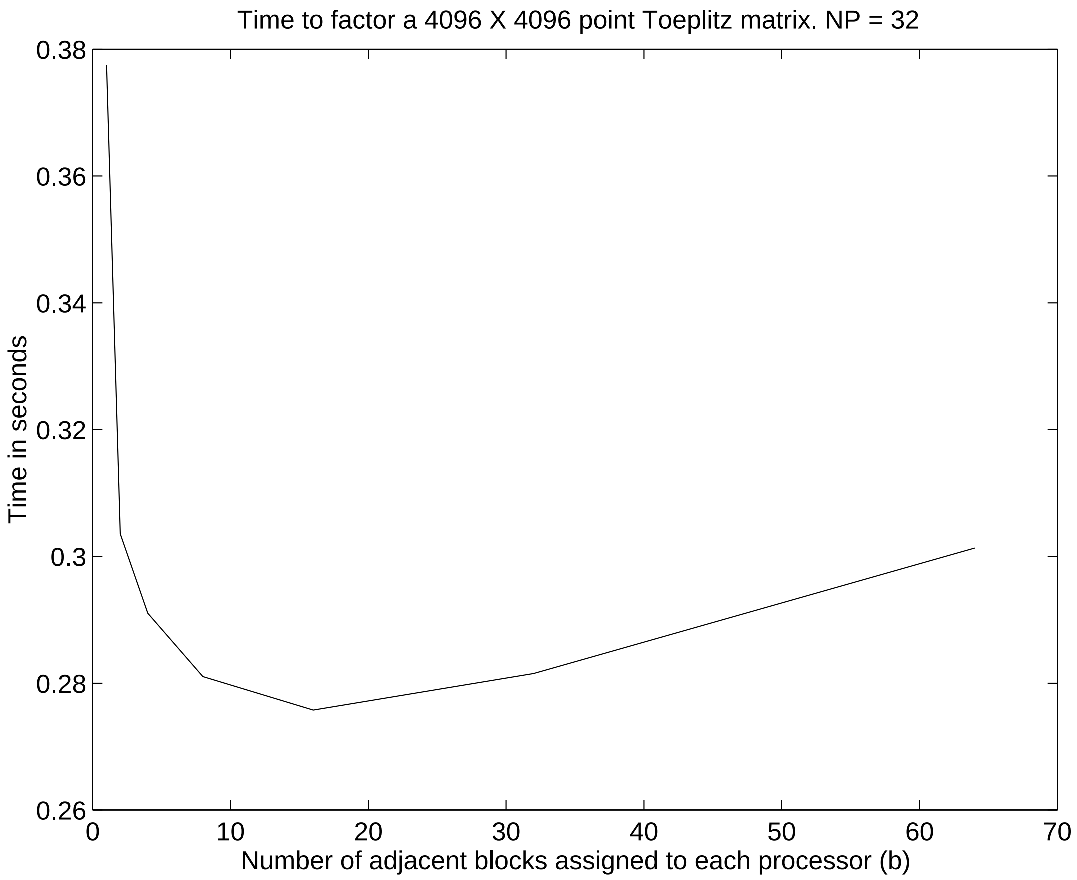


Figure 3.2 Time to factor a 4096×4096 point Toeplitz matrix on a 32-processor partition of the T3D. The parameter b, denoting number of adjacent blocks assigned to each processor, is varied from 1 to 64.

implementation. If the time for broadcasts and shifts increases, then the range of block sizes for which Version 1 provides the best factorization times increases.

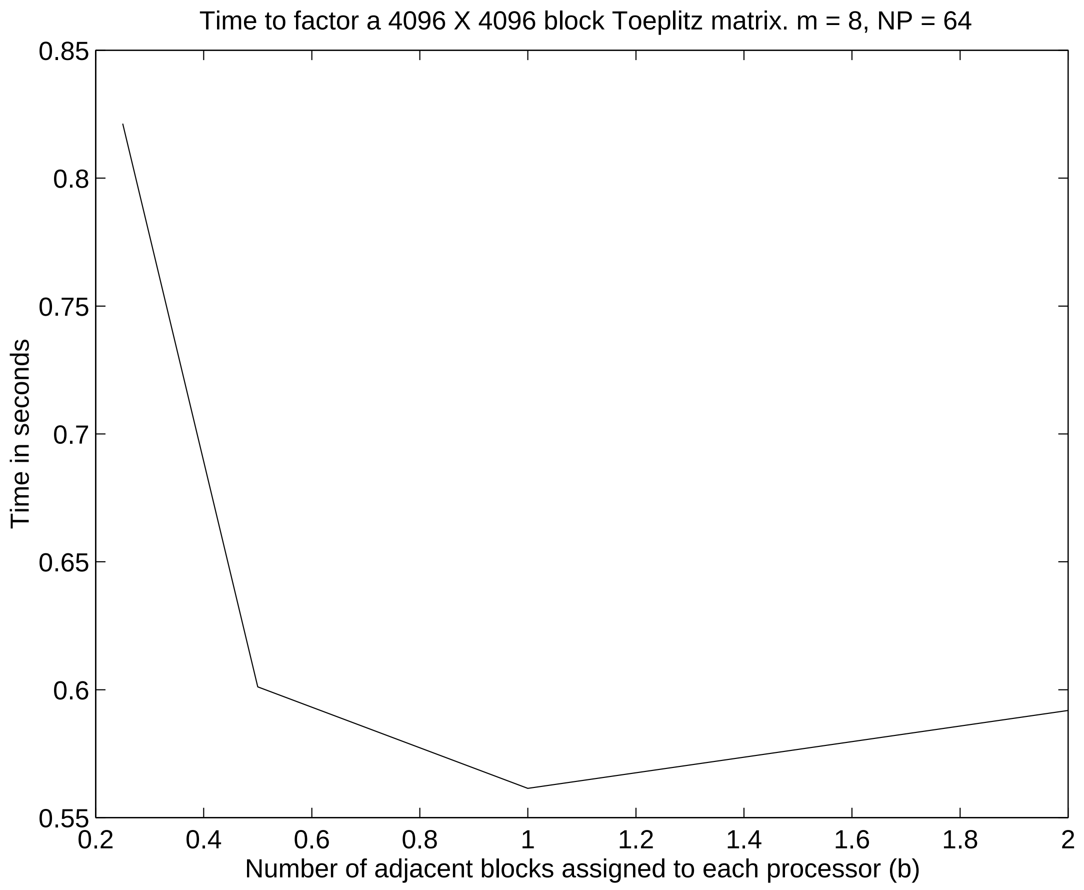


Figure 3.3 Time to factor a 4096×4096 block Toeplitz matrix with block size $m = 8$ on a 64-processor partition of the T3D. The parameter b, denoting number of adjacent blocks assigned to each processor, is varied from 0.25 to 2.

For large block sizes, it was argued that distributing a block across a few adjacent processors is most desirable. We present an example to illustrate this point. Consider a 4096×4096 block Toeplitz matrix with $m = 32$. Let $P = 64$. The time to factor the matrix using Versions 1 and 3 is shown in Figure 3.4. A value of b less than 1 indicates that a block was distributed across $1/b$ adjacent processors. In this example $p = 128$. If Version 1 is used, then after the 64-th step processors start to become idle and the load imbalance becomes acute. Delaying the onset of load imbalance by increasing the number of processors over which each block is distributed results in improved performance. The optimal number of processors over which to distribute each block in this example is 8. Further increases in the number of processors over which to

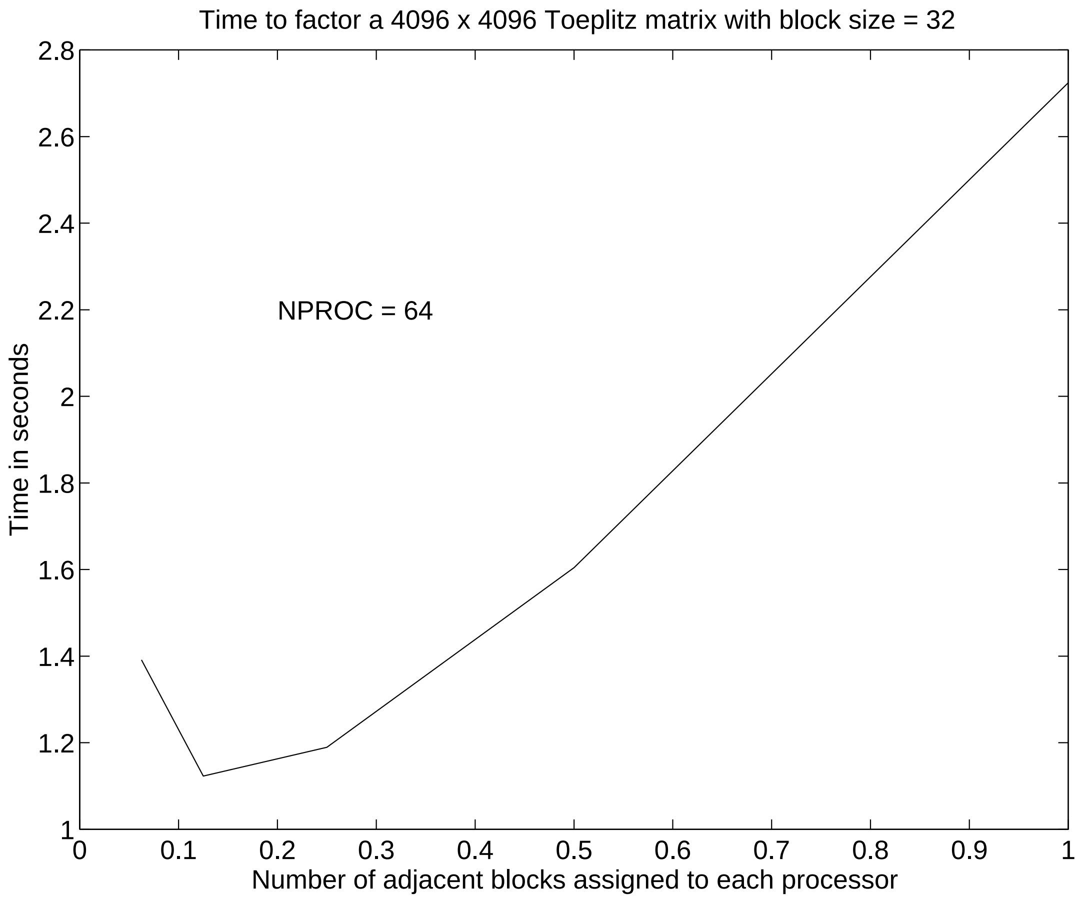


Figure 3.4 Time to factor a 4096×4096 block Toeplitz matrix with block size $m = 32$ on a 64-processor partition of the T3D. The parameter b, denoting number of adjacent blocks assigned to each processor, is varied from 0.0625 to 1.

distribute each block results in higher broadcast costs and offsets the increased parallelism. If the cost of broadcast on the T3D were to reduce, then the optimal number of processors over which to distribute a block to increase parallelism would increase.

The above examples serve to demonstrate the ranges over which the three data distribution schemes are useful. In this section, we have used either a cyclic or a block cyclic distribution to distribute the generator across a linear array of processors. This was done to mitigate the problem of load imbalance that degrades parallel performance. Cholesky factorization of block Toeplitz matrices can also be done using the SCALAPACK routine PSPOTRF, commonly available in vendor-provided scientific libraries, without taking advantage of the block Toeplitz structure. To compare the performance of the block Schur algorithm with this alternate scheme, we present performance results of PSPOTRF on the Cray T3D. Table 3.5 lists the time, in seconds, to factor a general matrix using the Cholesky factorization algorithm on a 256 processor Cray T3D. A block cyclic data distribution scheme with a block size of 32 is used along both matrix dimensions and the processors are configured as a 16×16 virtual grid. From Figure

Table 3.5 Time, in seconds, to factor a general s.p.d. matrix on a 256-processor Cray T3D.

Matrix Size	Time in seconds
3000	1.41
4000	2.56
6000	6.32
8000	12.57
10000	21.87

3.4 and Table 3.5 we see that the block Schur algorithm factors a 4096×4096 block Toeplitz matrix with a block size of 32 in about 1.1 seconds on a 64-processor T3D, whereas the Cholesky factorization routine from SCALAPACK factors a 4000×4000 general s.p.d. matrix in 2.56 seconds on a 256-processor T3D. This shows the tremendous savings in factorization time obtained from using the block Schur algorithm if the matrix has a block Toeplitz structure.

In the next section, we discuss a modification of the Schur algorithm proposed by Chun [19] to compute the generators of the inverse of a Toeplitz matrix. This modified algorithm is better suited to implementation on parallel machines but has twice the complexity as that of the block Schur algorithm used to obtain the Cholesky factors of a Toeplitz matrix. It will be shown that this algorithm is particularly useful when multiple right-hand sides are to be solved for, or when solving a Toeplitz system of equations is part of an iterative method such as iterative refinement.

3.5 Computing Generators of the Inverse of a Toeplitz Matrix

In [19], Chun showed how generators of the inverse of a Toeplitz matrix (the Gohberg-Semencul formula) may be computed using the Schur algorithm. This algorithm has several benefits from a computational standpoint. In this section, we discuss this algorithm and its implementation on parallel machines such as the Cray T3D.

Consider a symmetric positive definite block Toeplitz matrix T of size $mp \times mp$ with a block size of $m \times m$. Let the first block row of this matrix be defined by $[\widehat{T}_1 \; \widehat{T}_2 \; \cdots, \; \widehat{T}_p]$. Further, let the Cholesky factorization of $\widehat{T}_1$ be defined by $\widehat{T}_1 = LL^T$. The augmented matrix A defined by

$$A = \begin{bmatrix} T & I \\ I & 0 \end{bmatrix}$$

has a displacement rank ≤ 2 with respect to the displacement matrix

$$Z_{\text{aug}} = \begin{bmatrix} Z & 0 \\ 0 & Z \end{bmatrix},$$

where Z is the block down shift matrix with a block size of m. The displacement of A with respect to $(Z_{\text{aug}}, Z_{\text{aug}}^T)$ can be written in factored form as

$$A - Z_{\text{aug}} A Z_{\text{aug}}^T = G \Sigma G^T, \tag{3.1}$$

where

$$G^T = \left[\begin{array}{cccc|cccc} T_1 & T_2 & \cdots & T_p & T_1^{-T} & 0 & \cdots & 0 \\ 0 & T_2 & \cdots & T_p & T_1^{-T} & 0 & \cdots & 0 \end{array} \right] \quad \text{and} \quad \Sigma = \begin{bmatrix} I_m & 0 \\ 0 & -I_m \end{bmatrix}. \tag{3.2}$$

Here, $T_1 = L^T$ and $T_i = L^{-1}\widehat{T}_i$ for $i = 2, \cdots, p$. The block Schur algorithm can be adapted to the matrix A and its displacement Equation (3.1). The Schur complement of A with respect to T is $-T^{-1}$; hence, p steps of the block Schur algorithm would yield the generators of $-T^{-1}$. Let us denote the generator at the end of p steps by

$$G_p = \begin{bmatrix} G_{p_1} & G_{p_2} \end{bmatrix}, \tag{3.3}$$

where G_{p_1} and G_{p_2} are matrices of size $mp \times m$. From the relation between $-T^{-1}$ and its displacement with respect to (Z, Z^T), we have

$$T^{-1} = L(G_{p_2})L^T(G_{p_2}) - L(G_{p_1})L^T(G_{p_1}), \tag{3.4}$$

where $L(G_{p_1})$ and $L(G_{p_2})$ are lower block triangular Toeplitz matrices whose first block columns are specified by the matrices G_{p_1} and G_{p_2}, respectively. For a block size of 1, the multiplication of a lower triangular Toeplitz matrix with a vector is done by embedding the Toeplitz matrix in a circulant convolution matrix and using the 1D FFT to compute the circulant convolution. On a distributed memory machine, a 1D FFT is computed using the distributed four-step [55] algorithm. If the block size m is greater than 1, then the lower triangular block Toeplitz matrix is permuted to a Toeplitz block matrix where the blocks are lower triangular Toeplitz matrices. This is done by using a mod-m sort [55] permutation. For a lower triangular block Toeplitz matrix of size $mp \times mp$ with a block size of m, the mod-m sort permutation of rows and columns results in a Toeplitz block matrix with lower triangular Toeplitz blocks of size p. Multiplying a vector with this matrix is done by first applying the mod-m permutation matrix to the vector and carrying out the block matrix-vector products using the circulant convolution algorithm. In all, there will be m^2 such matrix-vector products of size p. The total complexity of this algorithm will, therefore, be $O(m^2 p \log(p))$. If m^2 is greater than or equal to the number of processors, then the parallelism in the algorithm is trivial. If m^2 is less than the total number of processors available, then the individual matrix-vector products may be computed using the parallel 1D FFT algorithm over subsets of processors. If the number of right-hand sides is greater than one, then the parallelism in the problem increases and an optimal partitioning of the computation across the processors will critically depend on the number of right-hand sides, the block size, and the number of processors.

At the start of the modified Schur algorithm, the generator G^T (3.2) has $(p+1)$ non-zero blocks. At every step of the algorithm, one extra block on the right side of the generator fills up while one block on the left is zeroed out due to a shift. This leaves the generator with $(p+1)$ blocks throughout the Schur algorithm. Since the amount of computation remains the same at each step of the Schur algorithm, the problem of load imbalance due to reduction in the size of the generator does not arise. This allows us to distribute the generator among a linear array of processors in a block distribution rather than a cyclic or block cyclic distribution that was used in the Cholesky factorization.

Besides being more efficient due to the absence of any load imbalance, the block distribution allows us to incorporate certain other algorithm-specific optimizations to overlap computation and communication. This can be done as follows. Let the number of blocks in the generator be p. Assuming a linear array of P processors and a block distribution, each processor receives $b = \lceil p/P \rceil$ blocks. In this distribution, some processors at the end may have no blocks. This can be remedied by using a nonstandard definition of a block distribution that aims to reduce load imbalance. In this discussion we use the High Performance Fortran (HPF) definition of a block distribution. For simplicity, let us assume that $b = p/P$ is an integer. At every step of the Schur algorithm, the lower half of the generator is shifted one block to the left. During this shift phase, the first block in the lower half of the generator on every processor is shifted to the processor to the left while the other $b-1$ blocks are shifted locally within a processor's memory. If we are to overlap computation and communication, we must apply the block hyperbolic Householder transformation to the first block and send it out to the processor to the left immediately after the update. Meanwhile, the block hyperbolic Householder transformation may be applied to the rest of the generator stored locally. This requires the use of a non-blocking send type of message-passing primitive. Note that this kind of overlap between computation and communication is not possible in a cyclic distribution and is not very effective in a block cyclic distribution.

This optimization and the absence of any load imbalance due to the idling of processors make this algorithm more suited for implementation on distributed memory machines than the classical Cholesky factorization algorithm. On the Cray T3D, such an algorithm was implemented with the use of the *shmem_put* message-passing routine as the communication primitive. Figure 3.5 plots the time, in seconds, to compute the generators of the inverse of a 4096×4096 block Toeplitz matrix with block sizes $m = 1$, $m = 4$, and $m = 8$.

We now compare the Cholesky factorization version of the block Schur algorithm with the version that computes the generator of the inverse of the Toeplitz matrix. The Cholesky factorization version of the block Schur algorithm has a complexity of approximately $2mn^2$, where m is the block size and n is the size of the Toeplitz matrix. The version that computes the generator of the inverse has a complexity of approximately $4mn^2$ (twice that of the Cholesky factorization version) because the non-zero portion of the generator is of constant size at every step of the algorithm instead of decreasing by one block with each step as in the Cholesky factorization version. Although it is twice as expensive to compute the generator of the inverse, there are several factors that make this version more attractive for some problem sizes than the Cholesky factorization version.

An obvious advantage is that the memory requirement for the Cholesky factorization routine ($O(n^2)$ is much larger than that for the version that computes the generator of the inverse ($O(mn)$); hence, a larger problem may be solved on a machine using the version that computes

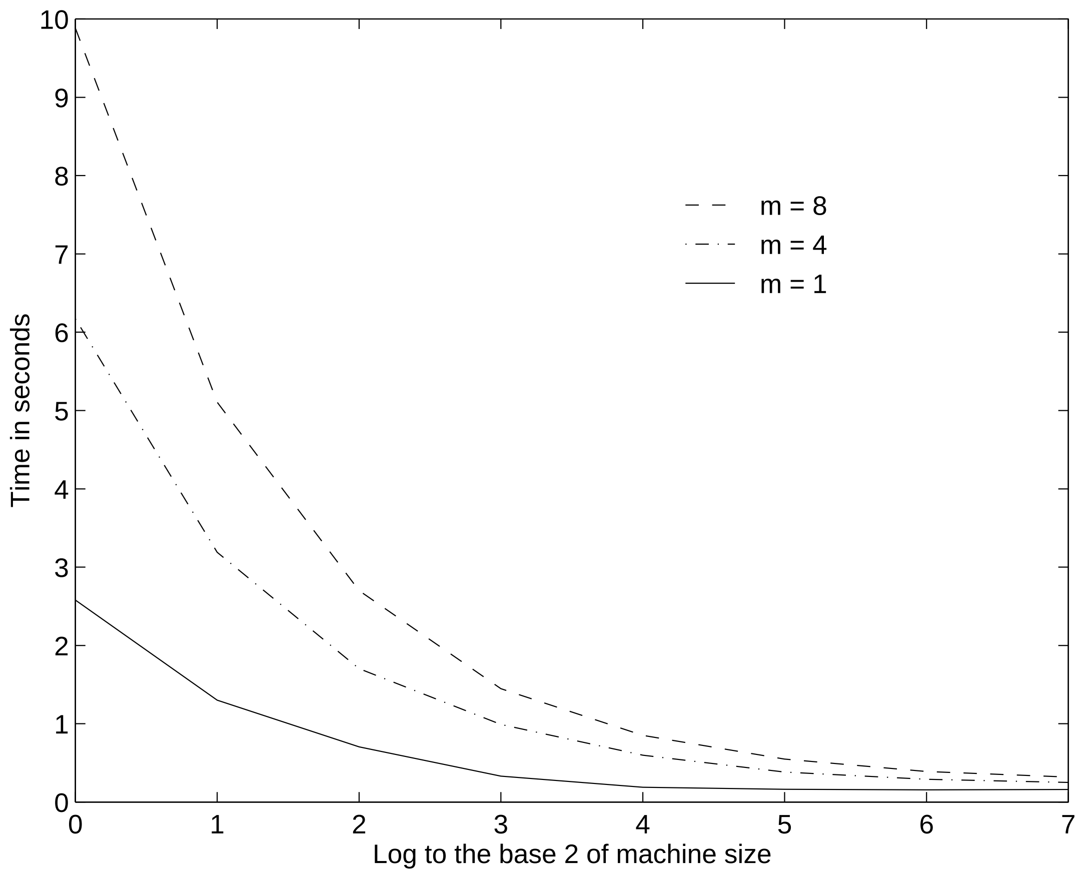


Figure 3.5 Time to compute the generators of the inverse of a 4096×4096 s.p.d. block Toeplitz matrix with block sizes $m = 1, 4$ and 8. The number of processors is varied from 1 to 128.

the generator of the inverse. The $O(n^2)$ extra writes to memory (for storing the Cholesky factor) may in fact cause the performance to suffer. In addition, since the size of the generator decreases with every step, the amount of parallelism in the Cholesky factorization version decreases with each step. Towards the end of the algorithm, processors start becoming idle at a rate determined by the data distribution. This problem does not exist in the algorithm that computes the generator of the inverse. To delay the onset of load imbalance, a cyclic or block cyclic data distribution is used in the Cholesky factorization version. The amount of data that is communicated to a neighboring processor during the shift operation is more for these data distributions than for the block distribution that is used in the version that computes the generator of the inverse. As a result, the overlap of computation and communication is better in the algorithm to compute the generator of the inverse than in the Cholesky factorization version. A combination of all the reasons mentioned above makes the algorithm that computes the generator of the inverse very competitive with the Cholesky factorization version. As an example we compare the performance of the two algorithms on the Cray T3D.

Consider a 4096×4096 block Toeplitz matrix with a block size of 4. At the start of the Schur algorithm, the generator has 1024 blocks. If we consider a machine size of eight processors, it can be seen that load imbalance in the computation of the Cholesky factor is not a serious issue. We can, therefore, use Version 2 of the data distribution schemes described in Figure 3.1. Assigning eight adjacent blocks to a processor in Version 2 yielded the best Cholesky factorization time for this machine and problem size. The total time to factor the block Toeplitz matrix on an eight-processor T3D was 0.85 second. It must be mentioned that the implementation of Version 2 on the T3D did not exploit the little overlap of computation and communication that is possible for this data distribution. Had this overlap been exploited, the factorization time would have been less. The time to compute the generator of the inverse of the block Toeplitz matrix, in comparison, was approximately 1.0 second. This shows that for large problem sizes on small machine partitions, the Cholesky factorization version may in fact be preferred to the version that computes the generator of the inverse. When the number of processors is increased to 128, the time to compute the Cholesky factorization using Version 1 described in Figure 3.1 is 0.25 second, whereas that to compute the generator of the inverse is also 0.25 second, This indicates that due to severe load imbalance and increased communication, the Cholesky factorization version performs worse than the version that computes the generator of the inverse.

Hence, for large machine partitions, if the complexity of the forward and backward solves is greater than the complexity to multiply the right-hand side with T^{-1}, it may be beneficial from a computational standpoint to use the algorithm that computes the generator of the inverse rather than the Cholesky factorization version. The limits, if any, of the numerical reliability of this method is the subject of future work.

In this chapter, we have only introduced the various issues involved in making implementation choices for the block Schur algorithm on distributed memory machines such as the Cray T3D. A more detailed study of the performance is a topic for future work.

CHAPTER 4

FACTORING SYMMETRIC INDEFINITE BLOCK TOEPLITZ MATRICES

In Chapter 2, we described a block Schur algorithm to obtain the Cholesky (LL^T) factorization and an LDL^T factorization of block Toeplitz matrices. At each step of the factorization, the block Schur algorithm computes a row of the upper triangular factor of the block Toeplitz matrix and the generator of the Schur complement. For s.p.d. block Toeplitz matrices, since pivoting is unnecessary, the block Schur algorithm is backward stable provided the hyperbolic transformations are applied in a certain manner [56]. For indefinite block Toeplitz matrices, the Schur algorithm is prone to numerical instability or breakdown in the presence of ill-conditioned or singular pivot blocks. In this chapter, we discuss modifications to the block Schur algorithm to obtain an LDL^T factorization of symmetric indefinite block Toeplitz matrices. Specifically, we present three algorithms: two of these algorithms look ahead over ill-conditioned pivot blocks until a well-conditioned pivot block is obtained, and the third perturbs singular pivot blocks away from singularity and produces an approximate factorization. The numerical accuracy of the solution is then restored through a few steps of iterative refinement.

4.1 Modifications to the Schur Algorithm for the Indefinite Case

Consider a symmetric indefinite block Toeplitz matrix T of size $mp \times mp$ and a block size of $m \times m$ whose first block row is defined by $[\widehat{T}_1,\ \widehat{T}_2,\ \cdots,\ \widehat{T}_p]$. We consider the block Schur algorithm described in Section 2.4 to compute an LDL^T factorization.

If the matrix $\widehat{T}_1$ is singular, then the generator G (2.47), the signature matrix W (2.48), the first block row of L, and the first block of D (2.50) cannot be computed. Let us assume that the matrix $\widehat{T}_1$ is well-conditioned. At every step of the algorithm, the matrix $\widehat{\Sigma}_1$ is the pivot block of the Schur complement. From (2.56) and (2.61), we see that the block hyperbolic transformation U cannot be computed if $\widehat{\Sigma}_1$ is singular. If $\widehat{\Sigma}_1$ is ill-conditioned, the Schur algorithm produces an inaccurate factorization. If $\widehat{\Sigma}_1$ is well-conditioned, then one can proceed with the Schur algorithm exactly as described in Section 2.4 to the next step.

There are two ways in which one can, in the event of degeneracy, avoid the problem of near or total breakdown of the Schur algorithm. The first method involves perturbing the pivot element of the generator such that the matrix $\widehat{\Sigma}_1$ in (2.61) is invertible. This method provides an inexact factorization of the block Toeplitz matrix. Iterative refinement may be used to correct the solution of such a system. The other method of avoiding degeneracy is to look ahead a few steps of the Schur algorithm until a well-conditioned principal minor is obtained. These two techniques are discussed in Sections 4.2 and 4.3.

4.2 Approximate Factorization of Indefinite Toeplitz Matrices Using Perturbations

In this section, we outline a modification to the Schur algorithm to factor symmetric indefinite block Toeplitz matrices with singular principal minors. If the block Toeplitz matrix has a singular principal minor, then at the corresponding step of the Schur algorithm, the pivot block is singular and a hyperbolic Householder transformation cannot be constructed. If the pivot block is perturbed so that it becomes nonsingular, then the Schur algorithm can proceed to compute an approximate factorization of the block Toeplitz matrix.

4.2.1 Block hyperbolic Householder transformations

The blocking scheme described in this section is a modification of the techniques discussed in Chapter 2. Consider a symmetric indefinite block Toeplitz matrix T of size $mp \times mp$ with a block size $m \times m$ whose first block row is given as $[\widehat{T}_1, \widehat{T}_2, \cdots, \widehat{T}_p]$. If $\widehat{T}_1$ is nonsingular and can be factored as $\widehat{T}_1 = PL_1\Sigma_1 L_1^T P^T$ (P is a permutation matrix), then the generator for the Toeplitz matrix is given as

$$G = \begin{pmatrix} T_1 & T_2 & \dots & T_p \\ 0 & T_2 & \dots & T_p \end{pmatrix} \quad \text{and} \quad W = \begin{pmatrix} \Sigma_1 & 0 \\ 0 & -\Sigma_1 \end{pmatrix}, \tag{4.1}$$

where $T_i = (L\Sigma_1)^{-1}P^T\widehat{T}_i$, $i = 1, \ldots, p$ and Σ_1 is a diagonal signature matrix. If the leading block $\widehat{T}_1$ is singular, then the generator is given as

$$G = \begin{pmatrix} 0.5(\hat{T}_1 + I_m) & \hat{T}_2 & \ldots & \hat{T}_p \\ 0.5(\hat{T}_1 - I_m) & \hat{T}_2 & \ldots & \hat{T}_p \end{pmatrix} \quad \text{and} \quad W = \begin{pmatrix} I_m & 0 \\ 0 & -I_m \end{pmatrix},$$

where I_m is the identity matrix of size m.

At each step of the Schur algorithm, a block hyperbolic Householder matrix is constructed using the first block column of the generator at that step. Let us consider the blocking schemes discussed in Chapter 2. A sequence of hyperbolic Householder transformations is constructed such that the diagonal element of the upper block is used to zero out all the elements of the column below it. At the j-th step of the process of zeroing out the lower block, the vector u has the form $[0, \ldots, 0, u_j, \ldots, u_{2m}]$. The first block column of the generator and the signature matrix are shown in Figure 4.1. The hyperbolic norm of u is given by u^TWu. A hyperbolic

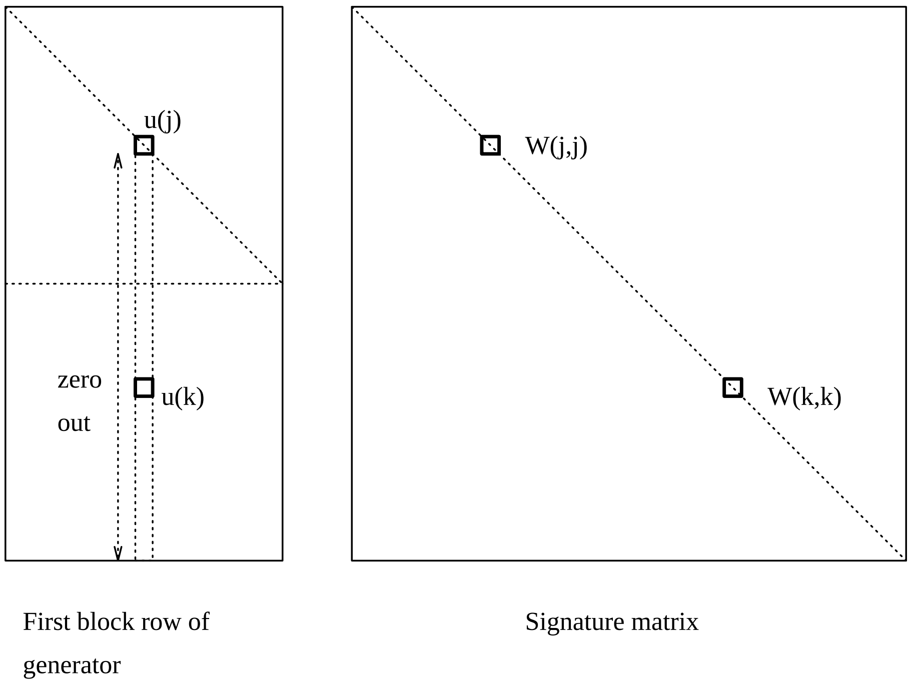


Figure 4.1 The first block column of the generator and the signature matrix.

Householder transformation, by definition, transforms a vector u to another vector b such that $u^TWu = b^TWb$. If we choose b to be $-\sigma e_j$ (using u_j to zero out the column), then $b^TWb = W(j,j)\sigma^2$. If $\text{sign}(W(j,j)) \neq \text{sign}(u^TWu)$, then one cannot obtain a hyperbolic

Householder transformation U such that $Uu = -\sigma e_j$. We must look for an alternate nonzero pivot element in the column u that has the same signature as $\text{sign}(u^T W u)$. Let this be u_k ($\text{sign}(W(k,k)) = \text{sign}(u^T W u)$). The element u_k can be permuted to the j-th position and used as a pivot element to zero out the column below it.

Let us first assume that the hyperbolic norms of all of the u vectors during the block transformation computation process are nonzero. The case of a zero hyperbolic norm is discussed later. The blocking schemes discussed in Chapter 2 can be easily extended to the indefinite case in the presence of permutations of the kind described above. In this section, we describe a VY blocking scheme. A derivation of a YTY^T form can be obtained similarly.

Consider a particular step in the Schur algorithm where the Schur complement of the symmetric indefinite block Toeplitz matrix is given by $\tilde{T}$. The generator G and signature matrix W satisfy the displacement equation

$$\tilde{T} - \tilde{Z}\tilde{T}\tilde{Z}^T = G^T W G. \tag{4.2}$$

Consider the first step of the blocking process. Let P_1 be the permutation matrix to get the correct pivot element in place. The hyperbolic transformation U_1 is given as

$$U_1 = \widetilde{W}_1 - \frac{2\tilde{x}_1\tilde{x}_1^T}{\tilde{x}_1^T \widetilde{W}_1 \tilde{x}_1}, \tag{4.3}$$

where $\widetilde{W}_1 = P_1 W_1 P_1^T$ ($W_1 = W$) and $\tilde{x}_1 = P_1 x$. Denote the first block column of the generator G that is used to produce the block transformation as A. The transformation U_1 is applied to a permuted version of A.

$$\begin{aligned}
U_1 P_1 A &= \left(\widetilde{W}_1 - \frac{2\tilde{x}_1\tilde{x}_1^T}{\tilde{x}_1^T \widetilde{W}_1 \tilde{x}_1}\right) P_1 A \\
U^{(1)} A &= \left\{ P_1 W_1 + (P_1 x_1)\left(-\frac{2x_1^T}{x_1^T W x_1}\right) \right\} A \\
U^{(1)} A &= (P_1 W_1 + v_1 y_1^T) A
\end{aligned} \tag{4.4}$$

The transformation $U^{(1)}$ shown above is W-unitary in the following sense:

$$U^{(1)^T} \widetilde{W}_1 U^{(1)} = P_1^T (U_1^T \widetilde{W}_1 U_1) P_1 = P_1^T \widetilde{W}_1 P_1 = W_1. \tag{4.5}$$

Let $C^{(1)} = P_1 W_1$, $V^{(1)} = v_1$ and $Y^{(1)} = y_1$. We show, by induction, that at the $(i+1)^{th}$ step, the block transformation $U^{(i+1)}$ has the form $U^{(i+1)} = C^{(i+1)} + V^{(i+1)} Y^{(i+1)^T}$, where $C^{(i+1)} = P^{(i+1)} W^{(i+1)}$. At the first step $P^{(1)} = P_1$, $W^{(1)} = W_1$ and $U^{(1)} = U_1 P_1$. Assume that $U^{(i)}$ has been obtained in the correct form. We show that $U^{(i+1)}$ can be obtained in the correct form.

Let the signature matrix at the start of the (i+1)-th step be W_{i+1}. It follows that $W_{i+1}(i+1 : 2m, i+1 : 2m) = \widetilde{W}_i(i+1 : 2m, i+1 : 2m)$. At the (i+1)-th step, since the first i rows of the first block of the generator are not affected, we choose the first i diagonal elements of W_{i+1} to be 1. This is done only for blocking purposes. The signature at the beginning of the (i+1)-th step is, therefore, given by

$$W_{i+1} = \left(\begin{array}{c|c} I_i & 0 \\ \hline 0 & \widetilde{W}_i(i+1 : 2m, i+1 : 2m) \end{array} \right). \tag{4.6}$$

Let P_{i+1} be the permutation matrix that is applied to place the element with the appropriate signature in the pivot position. The permuted signature matrix for this step is given by $\widetilde{W}_{i+1} = P_{i+1}^T W_{i+1} P_{i+1}$. The blocked transformation $U^{(i+1)}$ is then computed as

$$\begin{aligned} U^{(i+1)} &= U_{i+1} \;\; P_{i+1} \;\;\; U^{(i)} \\ &= (P_{i+1} W_{i+1} + P_{i+1} x_{i+1} y_{i+1}^T)(C^{(i)} + V^{(i)} Y^{(i)^T}) \\ &= P_{i+1} W_{i+1} C^{(i)} \\ &\quad + \left(\begin{array}{c|c} P_{i+1} W_{i+1} V^{(i)} & P_{i+1} x_{i+1} \end{array} \right) \left(\begin{array}{c} Y^{(i)^T} \\ \hline y_{i+1}^T (C^{(i)} + V^{(i)} Y^{(i)^T}) \end{array} \right) \\ &= C^{(i+1)} + V^{(i+1)} Y^{(i+1)^T}, \end{aligned}$$

where

$$\begin{aligned} C^{(i+1)} &= P_{i+1} W_{i+1} C^{(i)} \\ &= P_{i+1} W_{i+1} P^{(i)} W^{(i)} \\ &= (P_{i+1} P^{(i)})(P^{(i)^T} W_{i+1} P^{(i)} W^{(i)}) \\ &= P^{(i+1)} W^{(i+1)}. \end{aligned} \tag{4.7}$$

The block hyperbolic Householder transformation at the end of m steps has the form $U^{(m)} = C^{(m)} + V^{(m)}Y^{(m)^T}$. In addition, the signature matrix for the next step of the block Schur algorithm, W_{next}, is given by

$$W_{\text{next}} = P^{(m)} W P^{(m)^T}. \tag{4.8}$$

If, during the blocking scheme described above, we encounter a column that has a zero hyperbolic norm, then the Schur algorithm breaks down. A small perturbation of the column such that its hyperbolic norm is made nonzero allows the Schur algorithm to run to completion. This produces an approximate factorization of the block Toeplitz matrix. To obtain an exact solution an iterative scheme such as iterative refinement is needed. We now present an algorithm to perturb the column of the generator by an amount $|\delta|$. Later we present a derivation for an optimal value of $|\delta|$ and derive bounds for the number of steps of iterative refinement. Let us assume that at the j-th step of the blocking scheme, the hyperbolic norm of the column u ($u^T \widetilde{W}_{j-1} u$) is either zero or close to machine precision ϵ. An algorithm for the perturbation of this column of the generator is shown below.

Algorithm 4.1

```
if ($u^T \widetilde{W}_{j-1} u = 0$) then
   $u_j \rightarrow$ pivot element
   $a = (0, \ldots, 0, u_{j+1}, \ldots, u_{2m})$
   if ($\widetilde{W}_{j-1}(j,j) = 1$) then
      $u_j = \sqrt{\widetilde{W}_{j-1}(j,j)(|\delta| - a^T \widetilde{W}_{j-1} a)}$
   else
      $u_j = \sqrt{\widetilde{W}_{j-1}(j,j)(-|\delta| - a^T \widetilde{W}_{j-1} a)}$
   end
else if ($u^T \widetilde{W}_{j-1} u > 0$) then      ($+|\epsilon|$ say)
   if ($\widetilde{W}_{j-1}(j,j) = 1$) then
      $u_j \rightarrow$ pivot element
      $a = (0, \ldots, 0, u_{j+1}, \ldots, u_{2m})$
      $u_j = \sqrt{\widetilde{W}_{j-1}(j,j)(|\delta| + |\epsilon| - a^T \widetilde{W}_{j-1} a)}$
   else
      $u_k \rightarrow$ pivot element      ($\widetilde{W}_{j-1}(k,k) = 1$ say)
```

```
            $a = (0, \ldots, 0, u_j, \ldots, u_{k-1}, 0, u_{k+1}, \ldots, u_{2m})$
            $u_k = \sqrt{\widetilde{W}_{j-1}(k,k)(|\delta| + |\epsilon| - a^T \widetilde{W}_{j-1} a)}$
        end
    else        $(u^T \widetilde{W}_{j-1} u = -|\epsilon|$  say$)$
        if $(\widetilde{W}_{j-1}(j,j) = 1)$ then
            $u_k \rightarrow$ pivot element       $(\widetilde{W}_{j-1}(k,k) = -1$  say$)$
            $a = (0, \ldots, 0, u_j, \ldots, u_{k-1}, 0, u_{k+1}, \ldots, u_{2m})$
            $u_k = \sqrt{\widetilde{W}_{j-1}(k,k)(-|\delta| - |\epsilon| - a^T \widetilde{W}_{j-1} a)}$
        else
            $u_j \rightarrow$ pivot element
            $a = (0, \ldots, 0, u_{j+1}, \ldots, u_{2m})$
            $u_j = \sqrt{\widetilde{W}_{j-1}(j,j)(-|\delta| - |\epsilon| - a^T \widetilde{W}_{j-1} a)}$
        end
    end
```

4.2.2 Iterative refinement

The perturbation of a column of the pivot block column of the generator with zero hyperbolic norm allows us to continue the factorization process but introduces numerical instability into the algorithm. One way to circumvent the possible numerical instability of the Schur algorithm is to use iterative refinement on the system of equations. A similar perturbation technique has been used in [57] for the Levinson algorithm. They use the approximate factorization as a preconditioner in the conjugate-gradient algorithm. The iterative refinement technique we propose requires less work than the preconditioned conjugate-gradient algorithm per iteration.

Let us consider the system of equations $Tx = b$, where T is a symmetric indefinite block Toeplitz with singular principal submatrices. Using the perturbation technique described above, we obtain an approximate factorization

$$T + \delta T = LDL^T. \tag{4.9}$$

We solve the system of equations

$$LDL^T x_1 = b \tag{4.10}$$

to obtain x_1 and then compute the residual r_1

$$r_1 = -Tx_1 + b. \tag{4.11}$$

Using the correction term Δx_1 obtained from

$$LDL^T \Delta x_1 = r_1, \tag{4.12}$$

we improve the estimated solution by

$$x_2 = x_1 + \Delta x_1. \tag{4.13}$$

The iterative refinement algorithm is described below.

Algorithm 4.2

```
Construct $LDL^T = T + \delta T$ using the Schur algorithm
Solve $LDL^T x_1 = b$, and set $r_1 = -Tx_1 + b$
for $i = 1, \ldots$
    Solve $LDL^T \Delta x_i = r_i$
    if $\|\Delta x_i\| <$ tol $\|x_i\|$ then stop
    else
        $x_{i+1} = x_i + \Delta x_i$
        $r_{i+1} = -Tx_{i+1} + b$
    end
end
```

From the error analysis of [58] we know that the computed quantities $\overline{x}_i$, $\Delta\overline{x}_i$ and $\overline{r}_i$, satisfy the identities

$$\overline{r}_i = -T\overline{x}_i + b + \delta\overline{r}_i = r_i + \delta\overline{r}_i, \qquad \|\delta\overline{r}_i\| \leq \epsilon_i \|T\| \, \|\overline{x}_i\| \tag{4.14}$$

$$(LDL^T + \delta T_i)\Delta\overline{x}_i = \overline{r}_i, \qquad \|\delta T_i\| \leq \eta_i \|L\|^2 \, \|D\|, \tag{4.15}$$

where ϵ_i, η_i are of the order of the machine precision of the computer. From these equations we obtain

$$(T + \delta T + \delta T_i)\Delta\overline{x}_i = b - T\overline{x}_i + \delta\overline{r}_i, \tag{4.16}$$

and after some rewriting

$$\begin{aligned} r_{i+1} &= b - T(\overline{x}_i + \Delta\overline{x}_i) \\ &= (\delta T + \delta T_i)\Delta\overline{x}_i - \delta\overline{r}_i \\ &= (\delta T + \delta T_i)(T + \delta T + \delta T_i)^{-1}(r_i + \delta\overline{r}_i) - \delta\overline{r}_i \\ &= \Delta T_i(T + \Delta T_i)^{-1} r_i - T(T + \Delta T_i)^{-1}\delta\overline{r}_i, \end{aligned}$$

where the terms δT and δT_i, which are typically of the same order, have been grouped together in ΔT_i. Defining $M_i = \Delta T_i\, T^{-1}$ we have

$$r_{i+1} = M_i(I + M_i)^{-1} r_i - (I + M_i)^{-1}\delta\overline{r}_i. \tag{4.17}$$

If we can now obtain that $\max_i \|\Delta T_i\, T^{-1}\| = \gamma \ll 1$, then the above equation is a difference equation that will converge linearly, with a factor $\beta = \gamma(1-\gamma)$, to a steady state value of

$$\|r_\infty\| \approx \frac{1}{1-\beta}\frac{1}{1-\gamma}\|\delta r_{\max}\| = \frac{1}{1-2\gamma}\|\delta r_{\max}\| \leq \frac{\epsilon_{\max}}{1-2\gamma}\|T\|\|x\|. \tag{4.18}$$

Since we assume that γ is small, this final residual is about what one can expect from a stable algorithm. If we obtain that $\gamma = \sqrt[k]{\epsilon}$, then the number of iterations to converge to this result is k.

As shown above it is important to bound $\|\delta T\, T^{-1}\|$ in the construction of the factorization. Since LDL^T is only an approximate decomposition of T (but an exact decomposition of $T + \delta T$), we have the freedom to perturb T so as to obtain a better bound for $\delta T\, T^{-1}$. In this section we show how to obtain this using selective perturbations introduced in the Schur algorithm.

At the i-th step of the Schur algorithm we apply a block hyperbolic Householder transformation U_i to the generator $G'(i)$ to obtain $G'(i+1)$ (i.e., $U_i G'(i) = G'(i+1)$). The corresponding decomposition for the Toeplitz matrix is

$$\begin{aligned} T &= \begin{bmatrix} G_1^T(i) & G_2^T(i) \end{bmatrix} \hat{U}_i W \hat{U}_i \begin{bmatrix} G_1(i) \\ G_2(i) \end{bmatrix} \\ &= \left[G_1^T(i+1) G_2^T(i+1) \right] W \begin{bmatrix} G_1(i+1) \\ G_2(i+1) \end{bmatrix}, \end{aligned}$$

where $G_1(1)$ and $G_2(1)$ are given by (2.21), and $\hat{U}_i$ is essentially a block arrangement of identity matrices and U_i blocks. Hence,

$$\|\hat{U}_i\|_2 = \|U_i\|_2 \quad \text{and} \quad \|\hat{U}_i^{-1}\|_2 = \|U_i^{-1}\|_2. \tag{4.19}$$

If we now perturb the generator matrix $G'(i)$ by a perturbation of norm $\delta\|G'(1)\|_2$, then the equivalent perturbation $\|\Delta G'(1)\|$ of $G'(1)$ is bounded by

$$\|\Delta G'(1)\| \le \delta \, \|U_1^{-1}\|_2 \, \cdots \, \|U_{i-1}^{-1}\|_2 \, \|G'(1)\|$$

and that of T is proportional to $\delta \, \|U_1^{-1}\|_2 \, \cdots \, \|U_{i-1}^{-1}\|_2 \, \|T\|$. In other words, the norms of the inverses of the block transformations performed thus far act as a growth factor in the back-transformations of the perturbation to the original matrix. Another factor that we have to be concerned about is that the transformation U_i for which the δ perturbation was done will have a norm of approximately $1/\delta$, and the norm of the next generator $G'(i+1)$ will be increased by that amount. Numerical errors in subsequent steps will thus be proportional to this value, and when transforming these back to the original matrix T, we find again that we have to keep

$$\epsilon \, \|U_1\| \, \cdots \, \|U_{n-1}\|$$

bounded. Experience has shown that for each perturbation δ performed at a certain step i, there will be two block transformations of norm approximately $1/\delta$. For hyperbolic Householder transformations, $\|U\| = \|U^{-1}\|$ and the total error due to one perturbation is

$$\frac{\|\Delta T\|}{\|T\|} = \delta + \frac{\epsilon}{\delta^2}. \tag{4.20}$$

The value of δ that minimizes the above expression is $\sqrt[3]{2\epsilon}$ or $\delta \approx \sqrt[3]{\epsilon}$. This gives us

$$\begin{aligned}
\gamma &= \|\Delta T\, T^{-1}\| \\
&\leq \|\Delta T\|\|T^{-1}\| \\
&\leq \frac{\|\Delta T\|}{\|T\|}\mathrm{cond}(T) \\
&\approx \delta + \frac{\epsilon}{\delta^2} \;\; \text{(If } T \text{ is well conditioned)} \\
&\approx \sqrt[3]{\epsilon} \;\; \text{(If we set } \delta = \sqrt[3]{\epsilon}\text{)}. \qquad (4.21)
\end{aligned}$$

The subsequent number of steps of iterative refinement is 3. The above analysis holds true if we perturb the generator matrix just once.

Let us consider the case when we have to perturb twice. Let δ_1 and δ_2 be the two perturbations at steps i and j, respectively. The total perturbation to the original Toeplitz matrix can be expected to be of the order

$$\begin{aligned}
\|\delta T\| &= (\, \delta_1 \, \|U_1^{-1}\| \ldots \|U_{i-1}^{-1}\| + \delta_2 \, \|U_1^{-1}\| \ldots \|U_{j-1}^{-1}\| \,) \;\; \|T\| \\
&\approx (\delta_1 + \frac{\delta_2}{\delta_1^2})\|T\|. \qquad (4.22)
\end{aligned}$$

The numerical error due to the block transformations of norms approximately equal to $1/\delta_1^2$ and $1/\delta_2^2$ is

$$\begin{aligned}
\text{Numerical error} &= \epsilon\|U_1\| \ldots \|U_{n-1}\| \; \|T\| \\
&= \frac{\epsilon}{\delta_1^2\delta_2^2}. \qquad (4.23)
\end{aligned}$$

The total error due to the two factors is

$$\frac{\|\Delta T\|}{\|T\|} = \delta_1 + \frac{\delta_2}{\delta_1^2} + \frac{\epsilon}{\delta_1^2\delta_2^2}. \qquad (4.24)$$

The above expression is minimized by choosing $\delta_1 = \sqrt[9]{\epsilon}$ and $\delta_2 = \sqrt[3]{\epsilon}$. This means that we require nine iterations to achieve machine precision. It is impossible to know ahead of time how many perturbations one requires to carry on with the Schur algorithm. If upon performing one perturbation of $\sqrt[3]{\epsilon}$ we see during the Schur algorithm that another perturbation is required, we have to backtrack to the first perturbation and change the value of δ_1 from $\sqrt[3]{\epsilon}$ to $\sqrt[9]{\epsilon}$ and recompute the factorization from that step. This can increase the number of operations significantly. Also, if the number of times the generator has to be perturbed increases, the accuracy is lost very quickly and we must to look for other ways to handle such cases. Fortunately, in our experiments with Toeplitz matrices, we have observed that even for Toeplitz matrices with several singular minors one perturbation is sufficient, because the first perturbation affects the eigenvalues of the Toeplitz matrix and all its future minors and Schur complements. Of course, it is always possible to perturb multiple times without using the restrictions described above and to use more aggressive iterative methods. This technique has been used successfully to develop hybrid direct/iterative solvers for sparse systems.

We now present an example of a symmetric Toeplitz matrix with a singular principal minor. Consider the following block Toeplitz matrix T with a block size of 2.

$$\begin{aligned}
T(1:2,1:2) &= \begin{pmatrix} 0.04324379151529 & 0.29158091418984 \\ 0.29158091418984 & 0.67982106506507 \end{pmatrix} \\
T(1:2,3:4) &= \begin{pmatrix} 0.00769818621115 & 0.06684223751856 \\ 0.38341565075489 & 0.41748597445781 \end{pmatrix} \\
T(1:2,5:6) &= \begin{pmatrix} 0.68677271236050 & 0.93043649472782 \\ 0.58897664285683 & 0.84616689050857 \end{pmatrix} \\
T(1:2,7:8) &= \begin{pmatrix} 0.52692877758617 & 0.65391896229885 \\ 0.09196489075756 & 0.41599935685098 \end{pmatrix}
\end{aligned}$$

This matrix has a singular principal minor ($T(1:4,1:4)$ is singular). At the second step of the Schur algorithm, while blocking the two hyperbolic Householder transformations, the second column of the pivot block column of the generator has zero hyperbolic norm. We introduce a perturbation of $\sqrt{10^{-16}} \approx 10^{-5}$. The norm of the block hyperbolic Householder after perturbation is 2.2172×10^7 and the norm of U_4 is 2.821×10^7. This indicates that a single perturbation of δ produces two block hyperbolic Householder transformations of norm approximately equal to $1/\delta$. The norm of $\delta T.T^{-1}$ is 5.5761×10^{-4}. If we consider $x = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]^T$, then $b = [3.2074\ 3.7154\ 2.4177\ 3.6918\ 2.0762\ 4.0332\ 2.6206\ 4.3022]^T$. We see that $\|x - x_1\| = 3.1699 \times 10^{-4}$. Using iterative refinement, we find that after one step

$\|x - x_2\| = 9.7515 \times 10^{-8}$; after the second step, $\|x - x_3\| = 3.2389 \times 10^{-11}$; and, after the third step, $\|x - x_4\| = 3.5231 \times 10^{-15}$, which is approximately equal to the machine precision. Note that this is consistent with the above analysis.

From the analysis presented in this section, it can be seen that if the generator is perturbed once during the Schur algorithm, then three steps of iterative refinement are required. At each step of the iterative refinement, one has to do a forward and backward solve using the factorization computed by the Schur algorithm. The complexity for each step of iterative refinement is, therefore, $2n^2$ (n^2 for a forward solve and another n^2 for a backward solve). The complexity of the Schur algorithm for a block Toeplitz matrix of size $n \times n$ with a block size of $m \times m$ is $2mn^2$. Assuming three steps of iterative refinement to obtain an exact solution, the total complexity for k right-hand sides is $4mn^2 + 8kn^2$. In Section 3.5, we discussed an algorithm to compute the generators of the inverse of a block Toeplitz matrix. This algorithm has twice the complexity of the Schur algorithm that computes the $\mathrm{LDL}^{\mathrm{T}}$ factorization. However, multiplying the inverse of the Toeplitz matrix with a vector requires only $O(m^2 p \log(p))$ operations ($p = n/m$). For small block sizes, this is substantially less than the complexity of a forward and backward solve. We illustrate this for a real, symmetric point Toeplitz matrix. Let us assume that n is a power of 2. We make this assumption because the complexity of a power-of-2 length FFT is easy to compute. Let us also assume that the number of right-hand sides is 1 ($k = 1$). The complexity of the iterative refinement algorithm when the $\mathrm{LDL}^{\mathrm{T}}$ factorization is computed using the Schur algorithm is $2n^2 + 8n^2 = 10n^2$. The complexity of multiplying the inverse of a Toeplitz matrix with a vector using a factorization of the form shown in (3.4) is $45n\log_2(2n)$. This was arrived at assuming that the complexity of a length $2n$ real FFT is $5n\log_2(2n)$ flops. The total complexity of using the modified Schur algorithm is $4n^2 + 4.45.n\log_2(2n)$. Comparing the complexities of the two schemes, we see that for $n > 15\log_2(2n)$ (i.e., for $n > 128$), using the modified Schur algorithm is advantageous.

On distributed memory machines, the Schur algorithm to compute the Cholesky factors and forward and backward solves does not scale very well. The modified Schur algorithm does not have this problem because the amount of parallel work at each step of the algorithm remains constant. The FFTs used in the multiplication of the inverse of a Toeplitz matrix with a vector can be done using a parallel implementation of the four-step method for FFTs of large vectors [55].

4.3 Look-ahead Schur Algorithms

Perturbing the generator when singularities are encountered during the Schur algorithm produces an approximate factorization of indefinite block Toeplitz matrices. Iterative refinement is needed to improve the accuracy of the solution. If an exact factorization of a symmetric indefinite block Toeplitz matrix is desired, then we must deal with the singular principal minors of the Toeplitz matrix in a different way.

One important way to continue with the Schur algorithm when encountering singular principal minors is to look ahead over the singularities. This technique may also be used when the principal minors are badly conditioned. The first few look-ahead algorithms proposed for structured matrices [25, 26] were based on the Levinson algorithm and were restrictive in that they could look ahead only over exactly singular principal minors. For the case of near-singular principal minors, the errors in these algorithms are similar to those caused by using a very small pivot element in Gaussian elimination. The early look-ahead Schur algorithms also had this restriction. The first look-ahead algorithm for Toeplitz matrices that could look ahead over both exact and near singular principal minors was proposed by Chan and Hansen [30] and was an extension of the Levinson algorithm. Since then there have been other look-ahead Levinson [32] and look-ahead Schur algorithms [33] based on orthogonal polynomials that can look ahead over both exact and near singularities.

Most of the look-ahead algorithms in the literature are based on polynomial recursions of one form or another and, hence, cannot be easily extended to block Toeplitz matrices. In Sections 4.3.1 and 4.3.2, we discuss two algorithms that can look ahead over both exact and near singular principal minors. Both these algorithms are based solely on matrix operations and, hence, easily extend to block Toeplitz matrices. The first algorithm is based on computing a rank factorization of the displacement of the Schur complement after a look-ahead step using the Bunch-Kaufman pivoting scheme. The second algorithm is based on computing the generator using a completion of squares strategy. A similar look-ahead algorithm was proposed independently by Kailath and Sayed [36]. We also compare the two algorithms from a computational standpoint.

4.3.1 Look-ahead Algorithm 1

In this subsection, we discuss a block Toeplitz look-ahead algorithm that is based on computing the rank factorization (generator) of the displacement of the Schur complement. Consider a symmetric indefinite block Toeplitz T of size $mp \times mp$ with a block size of $m \times m$ whose first block row is given by $[T_1,\ T_2,\ \cdots,\ T_p]$. The displacement equation of this matrix is given by

$$T - ZTZ^T = G^T W G, \tag{4.25}$$

where Z is a block down shift matrix with block size m and

$$G = \begin{bmatrix} 0.5(I_m + T_1) & T_2 & \cdots & T_p \\ 0.5(I_m - T_1) & T_2 & \cdots & T_p \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} I_m & 0 \\ 0 & -I_m \end{bmatrix}. \tag{4.26}$$

The block Schur algorithm proceeds by applying a W-unitary transformation U $(U^T \widetilde{W} U = W)$ to G such that the block $0.5(I_m - T_1)$ is zeroed out using the block $0.5(I_m + T_1)$ to yield

$$\widetilde{G} = UG = \begin{bmatrix} \widetilde{T}_{11} & \widetilde{T}_{12} & \widetilde{T}_{13} & \cdots & \widetilde{T}_{1p} \\ 0 & \widetilde{T}_{22} & \widetilde{T}_{23} & \cdots & \widetilde{T}_{2p} \end{bmatrix} \quad \text{and} \quad \widetilde{W} = \begin{bmatrix} \widetilde{\Sigma}_1 & 0 \\ 0 & \widetilde{\Sigma}_2 \end{bmatrix}.$$

The first block row of the generator $\widetilde{G}$ is then shifted one block to the right to give the generators of the displacement of the Schur complement of T with respect to T_1. Essentially, the block Schur algorithm proceeds from step to step by producing a block row of the factorization and computing the generator of the next Schur complement from the generator of the previous Schur complement. If, during the Schur algorithm, the leading $m \times m$ block (principal minor) of the Schur complement is singular or ill-conditioned (for example, if at the start of the Schur algorithm, T_1 is singular or ill-conditioned), then the W-unitary transformation is ill-conditioned and the Schur algorithm either breaks down or introduces significant numerical errors in the factorization. In this section, we present an algorithm that looks ahead over this ill-conditioned principal minor. For simplicity in the description of the algorithm, we assume that the exact or near singularity is encountered at the start of the Schur algorithm. Note that this assumption is made only to ensure that the description of the look-ahead algorithm is easy to follow. This look-ahead scheme can be used at any stage of the Schur algorithm.

We have assumed that the matrix T_1 is ill-conditioned. Further, let us assume that the first $(k-1)m$ principal minors are ill-conditioned and that the (km)-th principal minor is well-conditioned. Note that we are forced to step in increments of the block size of the Toeplitz matrix in order to comply with the displacement structure. To preserve the numerical accuracy of the factorization, we have to look ahead over the first $(k-1)m$ principal minors.

Let the symmetric block Toeplitz matrix T be partitioned as

$$T = \left[\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{12}^T & T_{22} \end{array}\right], \tag{4.27}$$

where T_{11} is the $km \times km$ principal minor of T that is well-conditioned. If we are to jump over $(k-1)m$ steps of the Schur algorithm, the off-diagonal entries of $T_{11}^{-1}T_{12}$ must not be too large. A detailed discussion on the determination of the look-ahead step size (denoted here by k) can be found in [30] and [32]. We restrict our discussion to the look-ahead scheme after the determination of the step size k. Let the generator G be partitioned conformally as

$$G = \left[\begin{array}{c|c} G_{11} & G_{12} \\ \hline G_{21} & G_{22} \end{array}\right], \tag{4.28}$$

where G_{11} and G_{21} are of size $m \times km$, and G_{12} and G_{22} are of size $m \times n - km$. In addition, let the displacement matrix Z be partitioned as

$$Z = \left[\begin{array}{cc} Z_{11} & \\ Z_{21} & Z_{22} \end{array}\right], \tag{4.29}$$

where Z_{11} and Z_{22} are block down shift matrices of size km and $n - km$ respectively.

The first step in this look-ahead scheme is the computation of the first km rows of the Toeplitz or quasi-Toeplitz (Schur complement) matrix given by $[T_{11} \mid T_{12}\ ]$. If a look-ahead step is performed at the start of the Schur algorithm, then $[T_{11} \mid T_{12}\ ]$ is known completely from the first block row of the Toeplitz matrix T. If the look-ahead step is performed at any intermediate step of the Schur algorithm, then the first km rows of the quasi-Toeplitz Schur complement have to be calculated from the generator matrix. We now show how this may be done. From the displacement equation for the Toeplitz matrix T (4.25) we see that the displacement equation for $[T_{11} \mid T_{12}]$ is given by

$$\left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] - Z_{11}\left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] Z^T = \left[\begin{array}{c|c} G_{11}^T & G_{21}^T \end{array}\right] WG. \tag{4.30}$$

The matrix $[T_{11} \mid T_{12}]$ can be computed from its generator as

$$\begin{aligned}\left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] &= \left[\begin{array}{c|c} G_{11}^T & G_{21}^T \end{array}\right] WG + Z_{11}\left[\begin{array}{c|c} G_{11}^T & G_{21}^T \end{array}\right] WGZ^T + \cdots \\ &\quad + Z_{11}^{k-1}\left[\begin{array}{c|c} G_{11}^T & G_{21}^T \end{array}\right] WG(Z^T)^{k-1}. \end{aligned} \tag{4.31}$$

The complexity of computing $[T_{11} \mid T_{12}]$ from its generator using (4.31) is $O(km^2n)$, where n is the size of the Toeplitz matrix (if look-ahead is performed at the start) or the quasi-Toeplitz Schur complement.

The matrix $[T_{11} \mid T_{12}]$ can now be used to compute the corresponding diagonal block and km rows of the upper triangular factor of the LDL^T factorization of T. This can be done using a slow $O((km)^3)$ factorization algorithm such as Bunch-Kaufman or Gaussian elimination with partial pivoting. Let the matrix $[T_{11} \mid T_{12}]$ be factored as

$$\left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] = T_{11}\left[\begin{array}{c|c} I_{km} & T_{11}^{-1}T_{12} \end{array}\right] = D_k\left[\begin{array}{c|c} I_{km} & L_k \end{array}\right]. \tag{4.32}$$

The computational complexity of this step is $O((km)^2n)$.

The Schur complement of the Toeplitz matrix T or quasi-Toeplitz matrix before the look-ahead step with respect to its (km)-th principal minor T_{11} is

$$T_{sc}^{(k)} = T_{22} - L_kD_kL_k^T. \tag{4.33}$$

Since the Schur complement $T_{sc}^{(k)}$ has the same displacement structure as T, the Schur algorithm can be continued if we obtain the generator of $T_{sc}^{(k)}$.

Note that the displacement of the Schur complement $T_{sc}^{(k)}$, denoted by $\Delta T_{sc}^{(k)}$, is given by

$$\Delta T_{sc}^{(k)} = T_{sc}^{(k)} - Z_{22}T_{sc}^{(k)}Z_{22}T = T_{22} - Z_{22}T_{22}Z_{22}^T - L_kD_kL_k^T + Z_{22}L_kD_kL_k^TZ_{22}^T. \tag{4.34}$$

Let us denote the matrix $[G_{12}^T \mid G_{22}^T\ ]$ by G_2^T. The displacement of T_{22} with respect to the displacement matrix Z_{22} is given by

$$T_{22} - Z_{22}T_{22}Z_{22}^T = G_2^T W G_2. \tag{4.35}$$

From (4.35) and (4.34) we get

$$\Delta T_{sc}^{(k)} = G_2^T W G_2 - L_k D_k L_k^T + Z_{22} L_k D_k L_k^T Z_{22}^T. \tag{4.36}$$

The above equation can be rewritten in a factored form as

$$\begin{aligned} \Delta T_{sc}^{(k)} &= \begin{bmatrix} G_2^T & L_k & Z_{22}L_k \end{bmatrix} \begin{bmatrix} W & 0 & 0 \\ 0 & -D_k & 0 \\ 0 & 0 & D_k \end{bmatrix} \begin{bmatrix} G_2 \\ L_k^T \\ L_k^T Z_{22}^T \end{bmatrix} \\ &= \widehat{G}^T \widehat{W} \widehat{G}. \end{aligned} \tag{4.37}$$

This indicates that we can obtain a generator for the Schur complement. The problem with (4.37) is that the generator $\widehat{G}$ has a rank of at most $2km + 2m$. We know that a minimal generator of a block Toeplitz matrix has rank $\leq 2m$. We, therefore, have to reduce the generator shown above so that a minimal generator is obtained. $\Delta T_{sc}^{(k)}$ is a symmetric indefinite matrix of rank $\leq 2m$. The minimal generator of $T_{sc}^{(k)}$ can, therefore, be obtained by computing a rank factorization of $\Delta T_{sc}^{(k)}$. We suggest the use of the Bunch-Kaufman pivoting strategy to compute an LDL^T factorization of $\Delta T_{sc}^{(k)}$. The rank factorization of $\Delta T_{sc}^{(k)}$ computed using the Bunch-Kaufman pivoting strategy has to be done without using $O(n^2)$ storage because the generator matrix in the Schur algorithm uses only $O(n)$ storage. This can be achieved by using a delayed update version of the factorization algorithm.

We first describe the Bunch-Kaufman algorithm [59] for the sake of completeness. Consider a symmetric indefinite matrix A of size n. Let the (i,j)-th element of A be denoted by a_{ij}. The Bunch-Kaufman algorithm defines a pivot search strategy that maintains the symmetric structure of A while having a stability similar to that of Gaussian elimination with partial pivoting. In this algorithm, a maximum of two rows of the matrix are searched for either a

1×1 or a 2×2 pivot block. The permutation matrix that brings the pivot block to the correct position is denoted by P, and the pivot block size is denoted by s.

Algorithm 4.3

```
    $alpha = (1+\sqrt{17})/8$; $\lambda = |a_{1r}| = \max\{|a_{12}|, \ldots, |a_{1n}|\}$
    if $(\lambda > 0)$ then
      if $(|a_{11}| \geq \alpha\lambda)$ then
        $s = 1;\ P = I$
      else
        $\sigma = |a_{rp}| = \max\{|a_{r1}|, \ldots, |a_{r,r-1}|, |a_{r,r+1}|, \ldots, |a_{rn}|\}$
        if $\sigma|a_{11}| \geq \alpha\lambda^2$ then
          $s = 1;\ P = I$
        else if $|a_{rr}| \geq \alpha\sigma$ then
          $s = 1$; and choose $P$ so $(P^T AP)_{11} = a_{rr}$
        else
          $s = 2$; and choose $P$ so $(P^T AP)_{22} = a_{rr}$
        end
      end
    end
```

If this pivoting strategy is incorporated into an LDL^T factorization of $\Delta T_{sc}^{(k)}$, then the generator can be obtained by stopping after $2m$ rows of the factor are computed. The cost of computing a row of $\Delta T_{sc}^{(k)}$ from (4.37) is $O(kmn)$. As mentioned earlier, we choose to use a delayed update version of the LDL^T factorization algorithm because it allows us to use only $O(kmn)$ storage for the generator as opposed to an immediate update scheme which requires the storage of the entire matrix $\Delta T_{sc}^{(k)}$ in $(n-km)^2$ elements. A brief description of a delayed update version of the algorithm follows.

Consider an intermediate step of the factorization. Let the partial factorization of the matrix $\Delta T_{sc}^{(k)}$ be

$$\Delta T_{sc}^{(k)} = \left[\begin{array}{c|c} d & u \\ \hline u^T & X \end{array}\right], \tag{4.38}$$

where d is a block diagonal matrix with 1×1 or 2×2 pivot blocks computed using the Bunch-Kaufman algorithm, u is the upper triangular factor, and X is the Schur complement of $\Delta T_{sc}^{(k)}$ with respect to d. The next step is to search the matrix X for a 1×1 or a 2×2 pivot block using the Bunch-Kaufman pivoting strategy. In the delayed update version of the algorithm, the rows of X are not available explicitly and must be obtained by computing the corresponding row of $\Delta T_{sc}^{(k)}$ and updating it with $u^T du$. Note that the matrix $\Delta T_{sc}^{(k)}$ can be stored in factored form, and when a certain row is needed it can be computed using (4.37) in $O(kmn)$ flops. Once the first and r-th row of X are computed and tested for a pivot block, the next one or two rows of the upper triangular factor can be computed. This process is repeated until we have $2m$ rows of the upper triangular factor. This is the generator of $T_{sc}^{(k)}$ with respect to the displacement matrix Z_{22}.

This look-ahead algorithm requires $2km + 2m$ of storage for the generator $\widehat{G}$. In addition, the pivot search strategy requires reduction primitives such as determining the element with the maximum value in a vector. This primitive does not scale very well on massively parallel machines. Another potential problem with this look-ahead scheme is that in certain pathological cases, the Bunch-Kaufman algorithm may not be able to detect accurately the rank of a low rank matrix [56]. This look-ahead algorithm relies on obtaining a rank $2m$ factorization of the displacement of the Schur complement after each look-ahead step. In such cases an approximate factorization of the Toeplitz matrix is produced. The exact solution would have to be obtained using iterative refinement. In Section 4.3.2 we present an alternate look-ahead Schur algorithm that requires less storage and, in some cases, less computation than this method. It also avoids the Bunch-Kaufman pivoting strategy and its problems.

4.3.2 Look-ahead Algorithm 2

In this section, we discuss another look-ahead Schur algorithm that requires less storage than the previous scheme and avoids the reduction primitives used in the Bunch Kaufman pivoting strategy. A similar algorithm has been developed independently by Sayed and Kailath [36].

Let T be a symmetric indefinite block Toeplitz matrix of dimension $mp \times mp$ with a block size $m \times m$ whose first block row is given by $[T_1, T_2, \cdots, T_p]$. Let Z be a block down shift matrix of size mp. The displacement equation of the matrix T with respect to Z is given by

(4.25). Let us assume that T_1 is ill-conditioned. A look-ahead Schur step is needed to preserve numerical accuracy of the factorization. In addition, let us assume that the $m, 2m, \ldots, (k-1)m$ principal minors are ill-conditioned and that the km principal minor is well-conditioned. Let T and Z be partitioned as in (4.27) and (4.29), respectively. T_{11} and Z_{11} are of dimension $km \times km$ (a multiple of the block size), and T_{11} is assumed to be invertible (this is always possible by choosing k large enough). Let us also assume that all the conditions for determining the look-ahead step size of k as discussed in [30, 32] are satisfied. We now derive updating formulas to compute the generators of the Schur complement of T with respect to T_{11}. This part is related to the work of [60], but is not contained in it.

Define

$$X = T_{11}^{-1} T_{12} \qquad \text{and} \qquad U = \left[\begin{array}{c|c} I & -X \\ \hline & I \end{array}\right]; \tag{4.39}$$

then it follows that

$$U^T T U = \left[\begin{array}{c|c} T_{11} & \\ \hline & T_{sc} \end{array}\right], \qquad T_{sc} = T_{22} - T_{12}^T T_{11}^{-1} T_{12}, \tag{4.40}$$

where T_{sc} is the Schur complement of T with respect to T_{11}. To continue the block Schur algorithm after the look-ahead step, we must compute the generator of T_{sc}. Applying $U^T(\ .\ )U$ to (4.25) yields

$$U^T T U \ - \ (U^T Z U^{-T})\, U^T T U\, (U^{-1} Z^T U) \ = \ U^T G^T W G U\ . \tag{4.41}$$

Note that

$$U^{-1} Z^T U = \left[\begin{array}{c|c} Z_{11}^T & \widehat{Z}_{21}^T \\ \hline & Z_{22}^T \end{array}\right], \qquad \widehat{Z}_{21}^T = \left[\begin{array}{c|c} I & X \end{array}\right] Z^T \left[\begin{array}{c} -X \\ \hline I \end{array}\right]. \tag{4.42}$$

Using (4.40) and (4.42) we can reduce (4.41) to

$$\begin{aligned} &\left[\begin{array}{c|c} T_{11} & \\ \hline & T_{sc} \end{array}\right] - \left[\begin{array}{c|c} Z_{11} & \\ \hline \widehat{Z}_{21} & Z_{22} \end{array}\right] \left[\begin{array}{c|c} T_{11} & \\ \hline & T_{sc} \end{array}\right] \left[\begin{array}{c|c} Z_{11}^T & \widehat{Z}_{21}^T \\ \hline & Z_{22}^T \end{array}\right] \\ &\qquad = U^T G^T W G U. \end{aligned} \tag{4.43}$$

Equating the $(1,2)$ and $(2,2)$ positions in the above equation we have

$$\begin{aligned} M &= Z_{11}T_{11}\widehat{Z}_{21}^T + \left[\begin{array}{c|c} I & 0 \end{array}\right] G^T W G \left[\begin{array}{c} -X \\ \hline I \end{array}\right] = 0, \\ \Delta T_{sc} &= T_{sc} - Z_{22}T_{sc}Z_{22}^T \\ &= \widehat{Z}_{21}T_{11}\widehat{Z}_{21}^T + \left[\begin{array}{c|c} -X^T & I \end{array}\right] G^T W G \left[\begin{array}{c} -X \\ \hline I \end{array}\right]. \end{aligned} \tag{4.44}$$

Substituting for $\widehat{Z}_{21}^T$ from (4.42) we can further simplify M and ΔT_{sc} to

$$\begin{aligned} M &= \left[\begin{array}{c|c} I & 0 \end{array}\right] \left\{ Z \left[\begin{array}{c} I \\ \hline X^T \end{array}\right] T_{11} \left[\begin{array}{c|c} I & X \end{array}\right] Z^T + G^T W G \right\} \left[\begin{array}{c} -X \\ \hline I \end{array}\right] \\ &= 0 \end{aligned} \tag{4.45}$$

$$\begin{aligned} \Delta T_{sc} &= \left[\begin{array}{c|c} -X^T & I \end{array}\right] \\ &\quad \left\{ Z \left[\begin{array}{c} I \\ \hline X^T \end{array}\right] T_{11} \left[\begin{array}{c|c} I & X \end{array}\right] Z^T + G^T W G \right\} \left[\begin{array}{c} -X \\ \hline I \end{array}\right]. \end{aligned} \tag{4.46}$$

Substituting for X in the matrix in the middle of the above equations, we obtain

$$\begin{aligned} D &= Z \left[\begin{array}{c} T_{11} \\ \hline T_{12}^T \end{array}\right] T_{11}^{-1} \left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] Z^T + G^T W G \\ &= \left[\begin{array}{c|c} Z \left[\begin{array}{c} T_{11} \\ \hline T_{12}^T \end{array}\right] & G^T \end{array}\right] \left[\begin{array}{c|c} T_{11}^{-1} & 0 \\ \hline 0 & W \end{array}\right] \left[\begin{array}{c} \left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] Z^T \\ \hline G \end{array}\right]. \end{aligned} \tag{4.47}$$

This expression can now be further simplified to prove that the rank of ΔT_{sc} is at most $2m$. In order to prove this we need the following lemma.

Lemma 4.1 *Let*

$$D = \begin{bmatrix} F_{11}^T & F_{21}^T \\ F_{12}^T & F_{22}^T \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \tag{4.48}$$

where Σ_1 and $D_{11} = F_{11}^T\Sigma_1 F_{11} + F_{21}^T\Sigma_2 F_{21}$ are invertible. Then there always exists a transformation H such that

$$H^T \begin{bmatrix} \widehat{\Sigma}_1 & 0 \\ 0 & \widehat{\Sigma}_2 \end{bmatrix} H = \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \tag{4.49}$$

$$H \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} = \begin{bmatrix} \widehat{F}_{11} & \widehat{F}_{12} \\ 0 & \widehat{F}_{22} \end{bmatrix}. \tag{4.50}$$

Proof. Let $H = RQ$, where R is block upper triangular and Q is orthogonal. We choose Q such that

$$Q \begin{bmatrix} F_{11} \\ F_{21} \end{bmatrix} = \begin{bmatrix} B \\ 0 \end{bmatrix}, \tag{4.51}$$

where B is upper triangular. Moreover, since D_{11} is assumed invertible, $\begin{bmatrix} F_{11} \\ F_{21} \end{bmatrix}$ is full rank and hence B is invertible as well. Let R be partitioned conformally with D as

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \tag{4.52}$$

then H automatically satisfies (4.50) and $R_{11}B = \widehat{F}_{11}$. Also, H will satisfy (4.49) if and only if

$$\begin{bmatrix} R_{11}^T & 0 \\ R_{12}^T & R_{22}^T \end{bmatrix} \begin{bmatrix} \widehat{\Sigma}_1 & 0 \\ 0 & \widehat{\Sigma}_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} = Q \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} Q^T, \tag{4.53}$$

where the right-hand side is now known. A decomposition of the type $R_{11}^T\widehat{\Sigma}_1 R_{11}$ is known to exist if the (1,1) block of the right-hand side is invertible. From (4.51) we see that this equals $B^{-T}(F_{11}^T\Sigma_1 F_{11} + F_{21}^T\Sigma_2 F_{21})B^{-1}$, which is invertible. $\square$

To simplify (4.47) we now must apply this lemma to construct a transformation H such that

$$H^T \left[\begin{array}{c|c} \widetilde{T}_{11}^{-1} & \\ \hline & \widetilde{W} \end{array}\right] H = \left[\begin{array}{c|c} T_{11}^{-1} & \\ \hline & W \end{array}\right], \tag{4.54}$$

$$H\left[\begin{array}{c}\left[\begin{array}{cc}T_{11} & T_{12}\end{array}\right]Z^T \\ \hline G\end{array}\right] = \left[\begin{array}{c|c}\widehat{T}_{11} & \widehat{T}_{12} \\ \hline 0 & \widehat{G}_2\end{array}\right], \tag{4.55}$$

where $\widetilde{T}_{11}$ and $\widehat{T}_{11}$ are matrices of size $km \times km$, G has dimensions $2m \times mp$ and $\widehat{G}_2$ has dimensions $2m \times (mp - mk)$. To apply the above lemma we only have to show that D_{11} is invertible since T_{11} is invertible by assumption. From (4.43) it follows that

$$T_{11} = Z_{11}T_{11}Z_{11}^T + G_1^T W G_1, \qquad G_1 = G\left[\begin{array}{c} I \\ \hline 0\end{array}\right]. \tag{4.56}$$

From (4.47), D_{11} equals

$$D_{11} = \left[\begin{array}{c|c} I & 0\end{array}\right]\left\{Z\left[\begin{array}{c}T_{11}^T \\ \hline T_{12}^T\end{array}\right]T_{11}^{-1}\left[\begin{array}{c|c}T_{11} & T_{12}\end{array}\right]Z^T + G^TWG\right\}\left[\begin{array}{c} I \\ \hline 0\end{array}\right] \tag{4.57}$$

and since

$$Z = \left[\begin{array}{cc} Z_{11} & 0 \\ Z_{21} & Z_{22}\end{array}\right], \qquad G = \left[\begin{array}{cc} G_1 & G_2\end{array}\right], \tag{4.58}$$

we have

$$D_{11} = Z_{11}T_{11}T_{11}^{-1}T_{11}Z_{11}^T + G_1^TWG_1 = T_{11}, \tag{4.59}$$

which thus shows that D_{11} is invertible. Applying (4.54) and (4.55) to (4.47), we obtain

$$D = \left[\begin{array}{c|c}\widehat{T}_{11}^T & 0 \\ \hline \widehat{T}_{12}^T & \widehat{G}_2^T\end{array}\right]\left[\begin{array}{c|c}\widetilde{T}_{11}^{-1} & 0 \\ \hline 0 & \widetilde{W}\end{array}\right]\left[\begin{array}{c|c}\widehat{T}_{11} & \widehat{T}_{12} \\ \hline 0 & \widehat{G}_2\end{array}\right]. \tag{4.60}$$

Inserting (4.60) in (4.45) and (4.46) yields

$$\begin{aligned} M &= \left[\begin{array}{c|c} I & 0\end{array}\right]D\left[\begin{array}{c} -X \\ \hline I\end{array}\right] = \widehat{T}_{11}^T\widetilde{T}_{11}^{-1}\left[\begin{array}{c|c}\widehat{T}_{11} & \widehat{T}_{12}\end{array}\right]\left[\begin{array}{c} -X \\ \hline I\end{array}\right] = 0 \\ \Delta T_{sc} &= \widehat{G}_2^T\widetilde{W}\widehat{G}_2 \\ &\quad + \left[\begin{array}{c|c} -X^T & I\end{array}\right]\left[\begin{array}{c}\widehat{T}_{11}^T \\ \hline \widehat{T}_{12}^T\end{array}\right]\widetilde{T}_{11}^{-1}\left[\begin{array}{c|c}\widehat{T}_{11} & \widehat{T}_{12}\end{array}\right]\left[\begin{array}{c} -X \\ \hline I\end{array}\right]. \end{aligned} \tag{4.61}$$

Since $M = 0$ and $\widehat{T}_{11}$ and $\widetilde{T}_{11}$ are invertible, we have

$$\left[\begin{array}{c|c} \widehat{T}_{11} & \widehat{T}_{12} \end{array}\right] \left[\begin{array}{c} -X \\ \hline I \end{array}\right] = 0,$$

which yields,

$$\Delta T_{sc} = \widehat{G}_2^T \widetilde{W} \widehat{G}_2. \tag{4.62}$$

This establishes a new displacement identity where $\widetilde{W}$ and $\widehat{G}_2$ are obtained from (4.54) and (4.55).

The above description of the algorithm does not provide a method to construct the transformation H. We now outline one method to construct a matrix H such that

$$H \left[\begin{array}{c} \left[\begin{array}{c|c} T_{11} & T_{12} \end{array}\right] Z^T \\ \hline G \end{array}\right] = \left[\begin{array}{c|c} T_{11} & \widehat{T}_{12} \\ \hline 0 & \widehat{G}_2 \end{array}\right] \tag{4.63}$$

and

$$H^T \left[\begin{array}{cc} T_{11}^{-1} & \\ & \widetilde{W} \end{array}\right] H = \left[\begin{array}{cc} T_{11}^{-1} & \\ & W \end{array}\right] \tag{4.64}$$

under the assumption that T_{11} is invertible. Let $H = RQ$, where R is upper block triangular and Q is orthogonal. Let G be partitioned as $\left[\begin{array}{c|c} G_1 & G_2 \end{array}\right]$, where G_1 has dimensions $2m \times mk$. Let R be partitioned as

$$R = \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & R_{22} \end{array}\right]. \tag{4.65}$$

From (4.63), it can be seen that H must satisfy

$$H \left[\begin{array}{c} T_{11}Z_{11} \\ \hline G_1 \end{array}\right] = \left[\begin{array}{c} T_{11} \\ 0 \end{array}\right] \Rightarrow RQ \left[\begin{array}{c} T_{11}Z_{11} \\ \hline G_1 \end{array}\right] = \left[\begin{array}{c} T_{11} \\ 0 \end{array}\right]. \tag{4.66}$$

The first step involves a QR factorization :

$$\left[\begin{array}{c} T_{11}Z_{11} \\ \hline G_1 \end{array}\right] = Q^T \left[\begin{array}{c} B \\ \hline 0 \end{array}\right] = \left[\begin{array}{c|c} Q_1^T & Q_2^T \end{array}\right] \left[\begin{array}{c} B \\ \hline 0 \end{array}\right] = Q_1^T B. \tag{4.67}$$

From (4.66) and (4.67), we obtain

$$\left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & R_{22} \end{array}\right] \begin{bmatrix} B \\ 0 \end{bmatrix} = \begin{bmatrix} T_{11} \\ 0 \end{bmatrix} \Rightarrow R_{11}B = T_{11} \Rightarrow R_{11} = T_{11}B^{-1}. \tag{4.68}$$

Substituting for H in (4.64), we obtain

$$\begin{aligned} R^T \begin{bmatrix} T_{11}^{-1} & \\ & \widetilde{W} \end{bmatrix} R &= Q \begin{bmatrix} T_{11}^{-1} & \\ & W \end{bmatrix} Q^T \\ \left[\begin{array}{c|c} R_{11}^T T_{11}^{-1} R_{11} & R_{11}^T T_{11}^{-1} R_{12} \\ \hline R_{12}^T T_{11}^{-1} R_{11} & R_{12}^T T_{11}^{-1} R_{12} + R_{22}^T \widetilde{W} R_{22} \end{array}\right] &= Q \begin{bmatrix} T_{11}^{-1} & \\ & W \end{bmatrix} Q^T \end{aligned} \tag{4.69}$$

Partioning $Q^T = \left[\begin{array}{c|c} Q_1^T & Q_2^T \end{array}\right]$, and equating the (1,2) position in the above matrix equation after some simplification we obtain

$$R_{12}^T = Q_2 \begin{bmatrix} Z_{11} \\ WG_1 \end{bmatrix} \tag{4.70}$$

Equating the (2,2) position in (4.69) and rearranging the terms, we have

$$R_{22}^T \widetilde{W} R_{22} = Q_2 \begin{bmatrix} T_{11}^{-1} & \\ & W \end{bmatrix} Q_2^T - R_{12}^T T_{11}^{-1} R_{12}. \tag{4.71}$$

The matrix H is then computed as a product of R and Q.

This algorithm is of course only conceptual. It does not describe how to track the condition number of T_{11}. For this we refer to techniques such as those described in [30, 33, 32]. If no look-ahead is necessary, then the blocking scheme discussed in Section 2.4 can be used to compute H. If a look-ahead of size km is required, then H can be computed as shown in Lemma 4.1. It should be pointed out that when T_{11} is well-conditioned the transformation H and its construction should give no numerical problems.

4.3.3 Comparison of the two algorithms

In this section, we compare the two look-ahead algorithms from a computational and numerical standpoint. Consider a block Toeplitz matrix with a block size of m. Further, let us consider a look-ahead step size of km at some stage of the Schur algorithm. Let the size of the Schur complement following the look-ahead step be $lm \times lm$.

In Algorithm 1, the Bunch-Kaufman pivoting strategy is applied to obtain the generator for the Schur complement. In the worst case, we would have $2m$ steps with each step requiring two rows of ΔT_{sc} to be computed and contributing a 1×1 pivot to the factorization. This means that a total of $4m$ reduction operations, each of length lm, are done throughout the algorithm. For example, in computing one row of ΔT_{sc}, suppose the i-th row is done as $\widehat{g}_i^T \widehat{W} \widehat{G}$ ($\widehat{g}_i$ is the i-th row of $\widehat{G}$). It can be seen that the number of operations required to compute one row of ΔT_{sc} is

$$\text{total flops } = 8m^2 + 4m^2k^2 + 4m^2(k+1)l. \tag{4.72}$$

As mentioned earlier, in the worst case there are $2m$ steps requiring two rows at each step. Also, at each step the rows computed have to be updated with the portion of the factorization produced in the earlier steps of the algorithm. At the j-th step, this requires $2(j-1)lm$ operations. Hence, the total cost of the entire algorithm is

$$\begin{aligned} &= 2m\ 2\ (8m^2 + 4m^2k^2 + 4m^2(k+1)l) + 2m \sum_{j=1}^{2m} 2\ 2(j-1)lm \\ &= 16m^4l + 8m^3l + 16m^3kl + 16m^3k^2 + 32m^3. \end{aligned} \tag{4.73}$$

In the best case, only one row of ΔT_{sc} is searched for a 1×1 pivot. In this case, the computational complexity is exactly half the value of the expression (4.73).

In comparison, if we use Algorithm 2, the computation of the matrix H (described in (4.63 through 4.71)) requires a QR factorization of the matrix

$$\begin{bmatrix} T_{11}Z_{11} \\ G_1 \end{bmatrix} \tag{4.74}$$

which has a dimension of $m(k+2) \times km$. The cost of QR factorization of an $M \times N$ matrix is $4M^2N - 2MN^2 + \frac{2N^3}{3}$. For the matrix in (4.74) the computational cost is

$$\begin{aligned} &= 4m^2(k+2)^2mk - 2(mk)^2(k+2)m + (2/3)(mk)^3 \\ &= 2.67m^3k^3 + 12m^3k^2 + 16m^3k \qquad (4.75) \end{aligned}$$

We then have to compute R_{12} from (4.70). The total number of operations to compute R_{12} is

$$8m^3k + 16m^3 + 2m^2k. \qquad (4.76)$$

If we assume $R_{22} = I$, then the number of operations required to compute $\widetilde{W}$ from (4.71) is

$$2m^3k^2 + 16m^3k + 16m^3 + 8m^2. \qquad (4.77)$$

The cost of applying H to the generator of size $m(k+2) \times lm$ is equal to the cost of applying Q_2 to the generator. This yields an operation count of

$$4m^3kl + 8m^3l. \qquad (4.78)$$

The total cost of this method is found by adding together (4.75,4.76,4.77,4.78) resulting in an overall operation count of

$$4m^3kl + 8m^3l + 2.67m^3k^3 + 14m^3k^2 + 40m^3k + 2m^2k + 32m^3 + 8m^2. \qquad (4.79)$$

Comparing (4.73) and (4.79) and factoring the common multipliers, we have

$$\begin{aligned} 2m^3k^2 + 12m^3kl + 16m^4l \quad &vs. \quad 2.67m^3k^3 + 40m^3k + 2m^2k + 8m^2 \\ k^2 + 6kl + 8ml \quad &vs. \quad 1.33k^3 + 20k + \frac{k}{m} + \frac{4}{m}. \end{aligned} \qquad (4.80)$$

Consider an example where $m = 4$ and $l = 100$. It can be seen from (4.80) that, assuming the worst case for Algorithm 1, it is less expensive than Algorithm 2 for look-ahead step sizes greater than 24. If we assume the best case scenario for Algorithm 1, then it is less expensive than Algorithm 2 for look-ahead step sizes greater than 16.

Hence, for small block sizes, if the look-ahead step size is large, the Bunch-Kaufman-based look-ahead algorithm is faster than the one without pivoting. Note that in this calculation the cost of the reduction operation was not included. The results are not very different for serial machines. For parallel machines, the reduction operations give rise to several synchronization points, but the reduction is done in parallel. For Algorithm 2, the computation of H is a serial bottleneck. It is possible on some parallel machines that Algorithm 1 will have a wider range of applicability than on a sequential machine. From this it is clear that the two algorithms have distinct ranges of applicability. A study of the performance of these algorithms on parallel machines is a topic for future work.

CHAPTER 5

TRANSFORMING INDEFINITE TOEPLITZ MATRICES TO CAUCHY-LIKE MATRICES

In the previous chapter, we discussed several modifications to the block Schur algorithm to factor indefinite block Toeplitz matrices. In the presence of exact or near singular principal minors, these algorithms either produced an approximate factorization by perturbing the generator away from singularity or produced an exact factorization by looking ahead over the singularities. In both cases, however, no form of pivoting was incorporated into the Schur algorithm because pivoting destroys the displacement structure of Toeplitz matrices. In [37, 38], the authors suggest ways to overcome this problem by transforming one class of structured matrices to another using fast trigonometric transforms in such a way that pivoting may be incorporated into the factorization algorithms. These algorithms factor indefinite Toeplitz, Hankel, and Vandermonde matrices by converting them to Cauchy matrices and performing Gaussian elimination with partial pivoting. It was shown that the special displacement structure of Cauchy matrices is conducive to pivoting strategies such as partial pivoting. The algorithms suggested in [37, 38], however, did not exploit properties such as realness and symmetry simultaneously in the matrices.

In this chapter, we present Hermitian variants of the algorithms presented in [37, 38] to factor Hermitian Cauchy-like matrices. Exploiting the symmetry in Hermitian Cauchy-like matrices reduces the computational complexity by half. We also present variants that factor real, symmetric block Toeplitz matrices by converting them to real, symmetric Cauchy-like matrices and exploit both realness and symmetry simultaneously. Finally, we show how this variant may be used to convert Hermitian Toeplitz matrices into real, symmetric Cauchy-like matrices. We compare all the variants to factor Toeplitz matrices and show the reduction in complexity that results from exploiting properties such as realness, symmetry, and Hermitian symmetry. Some results from the implementation of these algorithms on the Cray J90 and Cray T90 are also presented.

We begin by reviewing the basic theory [37, 38] of transforming one class of structured matrices to another using fast trigonometric transforms. Section 5.2 reviews the Gaussian elimination method applied to structured matrices as proposed by Gohberg et al. in [38]. A variant of this algorithm that uses a symmetric form of the displacement equation, [24], is also reviewed in Section 5.2. Section 5.3 presents a variant to factor Hermitian Toeplitz matrices that exploits the Hermitian symmetry property. This section also computes the computational savings that result from exploiting the Hermitian property. Section 5.4 deals with factoring real nonsymmetric Toeplitz matrices and estimates the computational cost. Section 5.5 considers factoring real and symmetric Toeplitz matrices and estimates the savings in computation that result from exploiting both properties simultaneously. Section 5.5 also presents a new algorithm to convert a Hermitian Toeplitz matrix to a real, symmetric Cauchy-like matrix. The factorization of this resulting real, symmetric Cauchy-like matrix is significantly less expensive than the algorithm that converts a Hermitian Toeplitz matrix to a Hermitian Cauchy-like matrix before factorization. Section 5.6 discusses generalization of the algorithms to block Toeplitz matrices.

5.1 Transformations Between Classes of Structured Matrices

In [16], Kailath, Kung, and Morf introduced the idea of displacement structure to describe the structure in Toeplitz matrices that made them conducive to fast factorization schemes. Consider a real Toeplitz matrix T. Let Z be a down-shift matrix that shifts a matrix one row down when it is applied from the left. The displacement equation of T with respect to (Z, Z^T) is given by

$$T - ZTZ^T = GH^T. \tag{5.1}$$

The displacement equation may also be written as

$$ZT - TZ = GH^T. \tag{5.2}$$

We refer to Equation (5.1) as the displacement equation of type I and (5.2) as the displacement equation of type II. For real, symmetric Toeplitz matrices the displacement equation is of the form

$$T - ZTZ^T = GJG^T, \tag{5.3}$$

where J is a symmetric (diagonal) matrix and G has a displacement rank of 2. Note that in this chapter, we often use the symbols G and H to denote the generator matrices in displacement equations with different displacement matrices. This is done only for notational convenience and does not imply that the generator matrices are identical. For example, though the generators in (5.1), (5.2) and (5.3) are not identical, we use the same symbol G for convenience. Factorizations of the forms described above can be obtained analytically from the entries of the Toeplitz matrix without the need for explicit rank factorizations. The circulant down-shift matrix can be substituted for the down-shift matrix in the above displacement equations without changing the displacement structure of T and the displacement rank. This is useful, as we shall see later, because the circulant down-shift matrix can be diagonalized by the discrete Fourier transform.

In general, any matrix A that has a low displacement rank with respect to the matrices (F_l, F_r) is called a structured matrix. The displacement equation of type II of A with respect to (F_l, F_r) is given by

$$F_l A - AF_r = GH^T. \tag{5.4}$$

Cauchy-like matrices have the property that the displacement matrices F_l and F_r are diagonal. Any transform that diagonalizes the matrices F_l and F_r can, therefore, be used to convert the given structured matrix A to a Cauchy-like matrix. We show that the displacement structure of a Cauchy-like matrix is invariant to pivoting. Let the displacement equation for a Cauchy-like matrix C be

$$D_l C - CD_r = GH^T. \tag{5.5}$$

Let P be a permutation matrix ($P^T P = I$) corresponding to a partial pivoting operation. Applying this permutation to the Cauchy-like matrix C yields

$$\begin{aligned} PD_l C - PCD_r &= PGH^T \\ (PD_l P^T)(PC) - (PC)D_r &= (PG)H^T \\ \widehat{D}_l \widehat{C} - \widehat{C} D_r &= \widehat{G} H^T. \end{aligned} \tag{5.6}$$

It is clear that $\widehat{D}_l$ has the same structure as D_l (diagonal) and that the equation remains unchanged in structure. It is also easily verified that if D_l were not diagonal, such a permutation would destroy the displacement structure. For symmetric matrices with symmetric pivoting we require both D_l and D_r to be diagonal.

In particular, consider type II of the displacement equation for a Toeplitz matrix T of size n with respect to (Z, Z)

$$ZT - TZ = GH^T, \tag{5.7}$$

where Z is the circulant down-shift matrix. It is known that the discrete Fourier transform (DFT) diagonalizes the displacement matrix Z. Let F be the DFT matrix of size n defined by $F = \frac{1}{\sqrt{n}}[e^{\frac{2\pi i}{n}kj}]_{0\leq k,j\leq(n-1)}$. Then we have

$$FZF^* = \Lambda, \tag{5.8}$$

where Λ is a diagonal matrix with

$$\Lambda(j,j) = e^{\frac{2\pi i}{n}j} \qquad \text{for } j = 0\cdots(n-1), \qquad \text{where } i = \sqrt{-1}.$$

Applying the transformation $F(.)F^*$ to the displacement equation we have

$$\begin{aligned} FZTF^* - FTZF^* &= FGHF^* \\ (FZF^*)(FTF^*) - (FTF^*)(FZF^*) &= \widehat{G}\widehat{H}^* \\ \Lambda C - C\Lambda &= \widehat{G}\widehat{H}^*, \end{aligned} \tag{5.9}$$

where the displacement matrices D_l and D_r of (5.5) are both equal to Λ and C is a Cauchy-like matrix. Note that if T were a real matrix, G and H are also real. The DFT transformation converts these real matrices to complex matrices. This is undesirable because complex arithmetic is more expensive than real arithmetic.

It will be shown later that at each step of the factorization of Cauchy-like matrices of the form shown in (5.5) one has to solve Lyapunov equations derived from the displacement equation. It is well-known that the Lyapunov equation shown in (5.5) can be solved only if the eigenvalues of D_l and D_r are distinct. Since D_l and D_r are diagonal matrices, they must have distinct entries. If certain entries are very close, then retrieving the Cauchy-like matrix and factoring it may be sensitive to errors. Any loss of numerical accuracy can be restored by a few steps of iterative refinement. If they have some identical eigenvalues, then one has to compute certain additional parameters which need to be updated throughout the factorization algorithm, thereby increasing the complexity of the algorithm. With this in mind, Gohberg et al. [38] introduced a different form of the displacement equation to solve Toeplitz matrices. Consider the displacement matrices Z_1 and Z_{-1} defined by

$$Z_1 = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & & \vdots \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}, \qquad Z_{-1} = \begin{bmatrix} 0 & 0 & \cdots & 0 & -1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & & \vdots \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}. \tag{5.10}$$

The displacement equation for Toeplitz matrices can be written as

$$Z_1 T - T Z_{-1} = G H^T, \tag{5.11}$$

with a displacement rank of 2. It is well-known that the DFT matrix F diagonalizes both Z_1 and Z_{-1} as

$$\begin{aligned} F Z_1 F^* &= \Lambda_F \qquad \text{and} \\ F(D Z_{-1} D^{-1}) F^* &= \Lambda_{F_-} \qquad \text{where} \\ \Lambda_F &= diag(1, e^{\frac{2\pi i}{n}}, \cdots, e^{\frac{2\pi i}{n}(n-1)}), \\ \Lambda_{F_-} &= diag(e^{\frac{\pi i}{n}}, e^{\frac{3\pi i}{n}}, \cdots, e^{\frac{(2n-1)\pi i}{n}}) \qquad \text{and} \\ D &= diag(1, e^{\frac{\pi i}{n}}, \cdots, e^{\frac{(n-1)\pi i}{n}}). \end{aligned} \tag{5.12}$$

The displacement equation can now be rewritten as

$$\begin{aligned} (F Z_1 F^*)(F T D^{-1} F^*) - (F T D^{-1} F^*)(F D Z_{-1} D^{-1} F^*) &= (FG)(H^T D^{-1} F^*) \\ \Lambda_F C - C \Lambda_{F_-} = \widehat{G}\widehat{H}^*, \end{aligned} \tag{5.13}$$

where C is a Cauchy-like matrix defined by $C = FTD^{-1}F^*$.

It can be seen from the above displacement equation that if T is a real matrix, the DFT matrix destroys the realness property. Also, if T is symmetric, the Cauchy-like matrix that it is transformed to is no longer Hermitian. In the next few subsections we review several forms of displacement equations and the corresponding fast trigonometric transforms that convert the Toeplitz matrices to Cauchy-like matrices.

5.1.1 Non-Hermitian Toeplitz matrices

Consider a non-Hermitian Toeplitz matrix T. The displacement equation for such a matrix using the displacement matrices Z_1 and Z_{-1} mentioned above is of the form

$$Z_1T - TZ_{-1} = GH^*. \tag{5.14}$$

The Toeplitz matrix in the above equation can be converted to a Cauchy-like matrix as demonstrated in (5.13).

5.1.2 Hermitian Toeplitz matrices

The technique described in Section 5.1.1 can be applied to Hermitian Toeplitz matrices. However, doing so would convert the Hermitian Toeplitz matrix to a non-Hermitian Cauchy-like matrix. To maintain Hermitian symmetry, one may use the displacement equation

$$T - Z_1TZ_1^T = GJG^*. \tag{5.15}$$

Applying the DFT transformation $F(.)F^*$ to the above equation, we obtain

$$C - \Lambda C\Lambda^* = \widehat{G}J\widehat{G}^*, \tag{5.16}$$

where the matrix $C = FTF^*$ is a Hermitian Cauchy-like matrix.

5.1.3 Real nonsymmetric Toeplitz matrices

The techniques described in Sections 5.1.1 and 5.1.2 destroy the realness property of Toeplitz matrices. One would like to preserve the property of realness to avoid complex arithmetic, which is more expensive than real arithmetic. Just as the discrete Fourier transform was used to convert complex Toeplitz matrices to complex Cauchy-like matrices, several real trigonometric transforms such as the discrete sine, cosine, and Hartley transforms, can be used to convert real Toeplitz matrices to real Cauchy-like matrices. In this section, we demonstrate how these transforms may be used. We first review the displacement matrices and the real trigonometric transforms that diagonalize them [42]. We then show how they may be used to convert real Toeplitz matrices into real Cauchy-like matrices.

Consider the general displacement matrix $Z_{\epsilon\psi}$ of size n defined as

$$Z_{\epsilon\psi} = \begin{bmatrix} \epsilon & 1 & 0 & \cdots & 0 \\ 1 & 0 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & 1 \\ 0 & \cdots & 0 & 1 & \psi \end{bmatrix}. \tag{5.17}$$

It can be easily verified that when $\epsilon = \psi = 0$, the discrete sine transform (DST) matrix defined as

$$S_{00} = \sqrt{\frac{2}{n+1}} \left(\sin \frac{ij\pi}{n+1} \right), \qquad i, j = 1, \cdots, n \tag{5.18}$$

diagonalizes Z_{00}. Specifically, we have

$$S_{00} Z_{00} S_{00} = 2 diag(\cos \frac{j\pi}{n+1}), \qquad j = 1, \cdots, n. \tag{5.19}$$

When $\epsilon = \psi = 1$, the discrete cosine transform-II (DCT-II) S_{11} diagonalizes Z_{11}.

$$\begin{aligned} S_{11} &= \sqrt{\frac{2}{n}} \left(k_j \cos \frac{(2i+1)j\pi}{2n} \right), \qquad i, j = 0, \cdots, n-1 \\ S_{11}^T Z_{11} S_{11} &= 2 diag(\cos \frac{j\pi}{n}), \qquad j = 0, \cdots, n-1. \end{aligned} \tag{5.20}$$

Here, $k_j = 1/\sqrt{2}$ for $j = 0$, and $k_j = 1$ otherwise. For $\epsilon = \psi = -1$, the discrete sine transform-II (DST-II) S_{-1-1} diagonalizes Z_{-1-1}.

$$S_{-1-1} = \sqrt{\frac{2}{n}} \left(k_j \sin \frac{(2i-1)j\pi}{2n} \right), \qquad i, j = 1, \cdots, n$$

$$S_{-1-1}^T Z_{-1-1} S_{-1-1} = 2diag(\cos\frac{j\pi}{n}), \qquad j = 1, \cdots, n. \tag{5.21}$$

Here, $k_j = 1/\sqrt{2}$ for $j = n$, and $k_j = 1$ otherwise. If $\epsilon = -1$ and $\psi = 1$ or vice versa, then the discrete sine transform-IV (DST-IV) S_{-11} diagonalizes Z_{-11} and the discrete cosine transform-IV (DCT-IV) S_{1-1} diagonalizes Z_{1-1}.

$$\begin{aligned}
S_{-11} &= \sqrt{\frac{2}{n}}\left(\sin\frac{(2i+1)(2j+1)\pi}{4n}\right), \qquad i, j, 0, \cdots, n-1 \\
S_{-11} Z_{-11} S_{-11} &= 2diag(\cos\frac{(2j+1)\pi}{2n}), \qquad j = 0, \cdots, n-1 \\
S_{1-1} &= \sqrt{\frac{2}{n}}\left(\cos\frac{(2i+1)(2j+1)\pi}{4n}\right), \qquad i, j, 0, \cdots, n-1 \\
S_{1-1} Z_{1-1} S_{1-1} &= 2diag(\cos\frac{(2j+1)\pi}{2n}), \qquad j = 0, \cdots, n-1.
\end{aligned} \tag{5.22}$$

The displacement matrices $Z_{\epsilon\psi}$ can be used to formulate the displacement equations for nonsymmetric Toeplitz matrices. However, unlike the case of Z_1, the displacement rank with respect to $Z_{\epsilon\psi}$ will be 4 and not 2. This is the penalty one incurs for staying in the real domain. It will be shown in a later section that this increase in the displacement rank does not result in an algorithm more expensive than the one that transforms the real matrix to a complex Cauchy-like matrix.

Consider a real nonsymmetric Toeplitz matrix T. Any one of the following displacement equations may be used to convert the Toeplitz matrix to a Cauchy-like matrix.

$$\begin{aligned}
Z_{00}T - TZ_{11} &= G_1 H_1^T \\
Z_{00}T - TZ_{-1-1} &= G_2 H_2^T \\
Z_{00}T - TZ_{1-1} &= G_3 H_3^T \\
Z_{00}T - TZ_{-11} &= G_4 H_4^T.
\end{aligned} \tag{5.23}$$

Each of the equations shown above results in a displacement rank of 4. The corresponding real trigonometric transformations can be applied to obtain the displacement equation for the corresponding real Cauchy-like matrix. It must be mentioned that the matrices $G_1, \cdots, G_4$ and $H_1, \cdots, H_4$ can be calculated analytically without the need for a rank factorization of the displacement of T.

5.1.4 Real symmetric Toeplitz matrices

If the Toeplitz matrix is real and symmetric, then the techniques described in Section 5.1.3 convert it to an nonsymmetric Cauchy-like matrix. To maintain symmetry, symmetric forms of the displacement equations described in Section 5.1.3 such as

$$Z_{\epsilon\epsilon}\, T - T Z_{\epsilon\epsilon} = G_{\epsilon\epsilon}\, \Sigma {G_{\epsilon\epsilon}}^T \tag{5.24}$$

may be used, where T is a real symmetric Toeplitz matrix of size n, Σ is a skew-symmetric signature matrix, and ϵ may be either 0 or 1. By construction, it is easy to see that the displacement of T with respect to $Z_{\epsilon\epsilon}$ is of the form

$$Z_{\epsilon\epsilon}\, T - T Z_{\epsilon\epsilon} = \begin{bmatrix} 0 & -{a_\epsilon}^T & 0 \\ a_\epsilon & \mathbf{0} & E a_\epsilon \\ 0 & -{a_\epsilon}^T E & 0 \end{bmatrix}, \tag{5.25}$$

where a_ϵ is a vector of length $n-2$ that depends on the displacement matrix $Z_{\epsilon\epsilon}$, and E is the reflection permutation matrix of size $n-2$ defined by

$$E = \begin{bmatrix} & & & 1 \\ & & 1 & \\ & \iddots & & \\ 1 & & & \end{bmatrix}.$$

From (5.25), we see that the generator $G_{\epsilon\epsilon}$ and the signature matrix Σ are given by

$$G_{\epsilon\epsilon} = \begin{bmatrix} 0 & 1 & 0 & \mathbf{0}_{n-2} \\ a_\epsilon & \mathbf{0}_{n-2} & E a_\epsilon & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \Sigma = \left[\begin{array}{cc|cc} & 1 & & \\ -1 & & & \\ \hline & & & 1 \\ & & -1 & \end{array}\right], \tag{5.26}$$

where $\mathbf{0}_{n-2}$ is a vector of zeros of size $(n-2) \times 1$. The sine-I (S_{00}) or the cosine-II (S_{11}) transforms can be used to diagonalize the displacement matrices Z_{00} and Z_{11}. The corresponding displacement equation for the Cauchy-like matrix $C_{\epsilon\epsilon}$ is

$$\Lambda_{\epsilon\epsilon}\, C_{\epsilon\epsilon} \; - C_{\epsilon\epsilon}\, \Lambda_{\epsilon\epsilon} \; = \widehat{G}_{\epsilon\epsilon}\, \Sigma \widehat{G}_{\epsilon\epsilon}^{T}, \tag{5.27}$$

where $\Lambda_{\epsilon\epsilon} \; = S_{\epsilon\epsilon}{}^{T} Z_{\epsilon\epsilon}\, S_{\epsilon\epsilon}$, $C_{\epsilon\epsilon} \; = S_{\epsilon\epsilon}{}^{T} T S_{\epsilon\epsilon}$ and, $\widehat{G}_{\epsilon\epsilon} \; = S_{\epsilon\epsilon}{}^{T} G_{\epsilon\epsilon}$. The displacement rank of the above equations is 4. Interestingly, the Cauchy-like matrix $C_{\epsilon\epsilon}$ has considerable sparsity that can be exploited during the factorization algorithm. Specifically, if P is the odd-even permutation matrix (i.e., $P * x = [x_1\ x_3\ \cdots\ x_2\ x_4 \cdots]^T$), then

$$PC_{\epsilon\epsilon}\, P^T = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix} \tag{5.28}$$

where M_1 is a real symmetric Cauchy-like matrix of size $\lceil n/2 \rceil$ and M_2 is of size $\lfloor n/2 \rfloor$. In addition, it can be shown that the matrices M_1 and M_2 have a displacement rank of 2, as opposed to $C_{\epsilon\epsilon}$ that has a displacement rank of 4. We prove this for both even and odd n. First, consider the case when n is even. From the definitions of S_{00} and S_{11}, it can be seen that

$$PS_{\epsilon\epsilon}\, P^T = \begin{bmatrix} S_1 & S_2 \\ ES_1 & -ES_2 \end{bmatrix} \qquad \text{when } n \text{ is even}, \tag{5.29}$$

where S_1 and S_2 are submatrices of size $n/2$ that depend on the trigonometric transform S_{00} or S_{11}, and E is the reflection permutation matrix of size $n/2$. Let us partition the matrix PTP^T as

$$PTP^T = \begin{bmatrix} T_1 & T_2 \\ T_2^T & T_1 \end{bmatrix}. \tag{5.30}$$

Here T_1 is a symmetric Toeplitz matrix and T_2 is an nonsymmetric Toeplitz matrix of size $n/2$. It follows that

$$\begin{aligned} PC_{\epsilon\epsilon}\, P^T &= (PS_{\epsilon\epsilon}{}^{T} P^T)(PTP^T)(PS_{\epsilon\epsilon}\, P^T) \\ &= \begin{bmatrix} S_1^T & S_1^T E \\ S_2^T & -S_2^T E \end{bmatrix} \begin{bmatrix} T_1 & T_2 \\ T_2^T & T_1 \end{bmatrix} \begin{bmatrix} S_1 & S_2 \\ ES_1 & -ES_2 \end{bmatrix}. \end{aligned} \tag{5.31}$$

The $(2,1)$ entry in the above matrix equation is

$$(S_2^T T_1 - S_2^T E T_2^T) S_1 + (S_2^T T_2 - S_2^T E T_1) E S_1.$$

Rearranging the terms we have

$$S_2^T (T_1 - E T_1 E) S_1 + S_2^T (T_2 E - E T_2^T) S_1 = 0,$$

since T_1 and T_2 are Toeplitz matrices. Since the matrix $PC_{\epsilon\epsilon}\ P^T$ is symmetric, the $(1,2)$ submatrix is also zero. This proves that one can now solve two smaller systems of size $n/2$ instead of one large system of size n. Having proved that $PC_{\epsilon\epsilon}\ P^T$ is of the form shown in (5.28), we now show that M_1 and M_2 have a displacement rank of 2. Applying the permutation matrix P to the displacement Equation (5.27), we have

$$\begin{aligned}(P\Lambda_{\epsilon\epsilon}\ P^T)(PC_{\epsilon\epsilon}\ P^T) - (PC_{\epsilon\epsilon}\ P^T)(P\Lambda_{\epsilon\epsilon}\ P^T) &= (PS_{\epsilon\epsilon}{}^T P^T)(PG_{\epsilon\epsilon})\Sigma \\ & \quad (G_{\epsilon\epsilon}{}^T P^T)(PS_{\epsilon\epsilon}\ P^T). \qquad (5.32)\end{aligned}$$

From (5.26), we see that $PG_{\epsilon\epsilon}$ can be partitioned as

$$PG_{\epsilon\epsilon} = \begin{bmatrix} g_1 & g_3 & Eg_2 & Eg_4 \\ g_2 & g_4 & Eg_1 & Eg_3 \end{bmatrix}, \qquad (5.33)$$

where g_1, g_2, g_3, and g_4 are vectors of length $n/2$. From (5.33) and (5.29), we have

$$(PS_{\epsilon\epsilon}{}^T P^T)(PG_{\epsilon\epsilon}) = \\ \left[\begin{array}{c|c|c|c} S_1^T g_1 + S_1^T E g_2 & S_1^T g_3 + S_1^T E g_4 & S_1^T g_1 + S_1^T E g_2 & S_1^T g_3 + S_1^T E g_4 \\ \hline S_2^T g_1 - S_2^T E g_2 & S_2^T g_3 - S_2^T E g_4 & -S_2^T g_1 + S_2^T E g_2 & -S_2^T g_3 + S_2^T E g_4 \end{array}\right]. \qquad (5.34)$$

From the above equation, it is clear that the generators of M_1 and M_2 have rank 2. This proves that the displacement rank of M_1 and M_2 is 2.

We now outline the proof for odd n. If n is odd, then the permuted trigonometric transform $PS_{\epsilon\epsilon}\ P^T$ has the property

$$PS_{\epsilon\epsilon}\ P^T = \begin{bmatrix} S_1 & S_3 \\ S_2 & S_4 \end{bmatrix}, \tag{5.35}$$

where $S_1 = E_1S_1$, $S_2 = E_2S_2$, $S_3 = -E_1S_3$, and $S_4 = -E_2S_4$. E_1 and E_2 are reflection permutation matrices of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively. Now partitioning PTP^T conformally as

$$PTP^T = \begin{bmatrix} T_1 & T_2 \\ T_2^T & T_3 \end{bmatrix},$$

we have

$$\begin{aligned} PC_{\epsilon\epsilon}\ P^T &= (PS_{\epsilon\epsilon}{}^T P^T)(PTP^T)(PS_{\epsilon\epsilon}\ P^T) \\ &= \begin{bmatrix} S_1^T & S_2^T \\ S_3^T & S_4^T \end{bmatrix} \begin{bmatrix} T_1 & T_2 \\ T_2^T & T_3 \end{bmatrix} \begin{bmatrix} S_1 & S_3 \\ S_2 & S_4 \end{bmatrix}. \end{aligned} \tag{5.36}$$

The $(2,1)$ submatrix in the above matrix equation is

$$S_3^T T_1 S_1 + S_4^T T_2^T S_1 + S_3^T T_2 S_2 + S_4^T T_3 S_2.$$

We can show that each term in the above expression evaluates to zero and, hence, the $(2,1)$ block is zero. For example, consider the first term : $S_3^T T_1 S_1 = S_3^T T_1 E_1 S_1 = S_3^T E_1 T_1 S_1 = -S_3^T T_1 S_1 = 0$. Similarly, all other terms in the expression evaluate to zero and the $(2,1)$ block is zero. Since the Cauchy-like matrix $PC_{\epsilon\epsilon}\ P^T$ is symmetric, the $(1,2)$ block is also zero. The permutation matrix P, therefore, separates the system of equation into two systems about half the size of the original system that can be solved independently. Now we show that the displacement rank of M_1 and M_2 is 2. If n is odd, then $PG_{\epsilon\epsilon}$ can be partitioned as

$$PG_{\epsilon\epsilon} = \begin{bmatrix} g_1 & g_3 & Eg_1 & Eg_3 \\ g_2 & g_4 & Eg_2 & Eg_4 \end{bmatrix}, \tag{5.37}$$

where g_1 and g_3 are of size $\lceil n/2 \rceil$ and g_2, and g_4 are vectors of length $\lfloor n/2 \rfloor$. From (5.37) and (5.35), we have

$$\begin{aligned} & (PS_{\epsilon\epsilon}{}^T P^T)(PG_{\epsilon\epsilon}) \\ &= \left[\begin{array}{c|c|c|c} S_1^T g_1 + S_2^T g_2 & S_1^T g_3 + S_1^T g_4 & S_1^T E_1 g_1 + S_2^T E_2 g_2 & S_1^T E_1 g_3 + S_2^T E_2 g_4 \\ \hline S_3^T g_1 + S_4^T g_2 & S_3^T g_3 + S_4^T g_4 & S_3^T E_1 g_1 + S_4^T E_2 g_2 & S_3^T E_1 g_3 + S_4^T E_2 g_4 \end{array}\right] \\ &= \left[\begin{array}{c|c|c|c} S_1^T g_1 + S_2^T g_2 & S_1^T g_3 + S_2^T g_4 & S_1^T g_1 + S_2^T g_2 & S_1^T g_3 + S_2^T g_4 \\ \hline S_3^T g_1 + S_4^T g_2 & S_3^T g_3 + S_4^T g_4 & -S_3^T g_1 - S_4^T g_2 & -S_3^T g_3 - S_4^T g_4 \end{array}\right]. \end{aligned} \tag{5.38}$$

The above equation shows that the displacement rank of M_1 and M_2 is 2 because the generators of M_1 and M_2 have rank 2.

In this section, we have shown how the odd-even permutation matrix can be used to decouple a Cauchy-like matrix arising from a real symmetric Toeplitz matrix of size n into two Cauchy-like matrices of half the size and half the displacement rank. This yields substantial savings over the nonsymmetric forms of the displacement equation.

5.1.5 Converting Hermitian Toeplitz matrices to real Cauchy-like matrices

In Section 5.1.2, a displacement equation for Hermitian Toeplitz matrices was suggested using Z_1 as the displacement matrix. The discrete Fourier transform was used to convert the Hermitian Toeplitz matrix to a Hermitian Cauchy-like matrix. In this section, we show how the displacement matrix $Z_{\epsilon\epsilon}$ may be used along with the odd-even permutation matrix to convert a Hermitian Toeplitz matrix into a real, symmetric Cauchy-like matrix. The factorization of the Cauchy-like matrix can be done in real arithmetic, and the savings in computation are significant.

Consider a Hermitian Toeplitz matrix of size n. The displacement equation of T with respect to $Z_{\epsilon\epsilon}$ can be written as

$$Z_{\epsilon\epsilon}\, T - T Z_{\epsilon\epsilon} \; = G \Sigma G^*, \tag{5.39}$$

where Σ is a skew-symmetric matrix and G has a rank of 4. In the previous section on real, symmetric Toeplitz matrices, we proved that $PS_{\epsilon\epsilon}{}^T\text{Real}(T)S_{\epsilon\epsilon}\ P^T$ is of the form

$$PS_{\epsilon\epsilon}{}^T\text{Real}(T)S_{\epsilon\epsilon}\ P^T = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}, \tag{5.40}$$

where M_1 and M_2 are real symmetric Cauchy-like matrices of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively. In addition, one can prove by construction that the imaginary part of T satisfies the equation

$$PS_{\epsilon\epsilon}{}^T\text{Ixmag}(T)S_{\epsilon\epsilon}\ P^T = \begin{bmatrix} 0 & -M_3^T \\ M_3 & 0 \end{bmatrix}, \tag{5.41}$$

where M_3 is a real Cauchy-like matrix of size $\lceil n/2 \rceil \times \lfloor n/2 \rfloor$. If we define a matrix D to be of the form

$$D = \begin{bmatrix} I_1 & 0 \\ 0 & iI_2 \end{bmatrix}, \tag{5.42}$$

where I_1 and I_2 are identity matrices of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, and $i = \sqrt{-1}$, then

$$DPS_{\epsilon\epsilon}{}^T TS_{\epsilon\epsilon}\ P^T D^* = \begin{bmatrix} M_1 & M_3^T \\ M_3 & M_2 \end{bmatrix}. \tag{5.43}$$

The matrix on the right-hand side is a real symmetric Cauchy-like matrix. Since the rank of the generator G of the Hermitian Toeplitz matrix was 4, the generator matrix of the corresponding real, symmetric Cauchy-like matrix will also be of rank 4.

This section shows how a Hermitian Toeplitz matrix may be converted to a real symmetric Cauchy-like matrix. The factorization may now proceed in real arithmetic. The savings in computation resulting from this conversion will be calculated in Section 5.5.

5.2 Factorization of Cauchy-like Matrices with Pivoting

In this section, we consider the factorization algorithms for Cauchy-like matrices that allow for various pivoting strategies to be incorporated. We first discuss the algorithm due to Gohberg, Kailath, and Olshevsky to factor non-Hermitian Cauchy-like matrices with displacement matrices of the form shown in (5.5). We then present another algorithm [24] to factor Hermitian Cauchy-like matrices with displacement equations of the form $C - D_l C D_l^* = G\Sigma G^*$. These can be adapted to suit any of the Cauchy-like matrices discussed in this chapter.

5.2.1 Factoring non-Hermitian Cauchy-like matrices

In this section we discuss the algorithm proposed by Gohberg, Kailath, and Olshevsky [38] to factor non-Hermitian Cauchy-like matrices. Consider a complex non-Hermitian Cauchy-like matrix C of size n, defined by the displacement equation

$$D_l C - C D_r = GH^*, \tag{5.44}$$

where D_l and D_r are diagonal matrices and G and H are matrices of size $n \times \alpha$ with rank equal to α. Further, let us assume that D_l and D_r do not have any entries on the diagonal that are equal. We relax this restriction later. In this section we show how partial pivoting may be incorporated into the factorization algorithm.

From the above displacement equation, it is clear that any column of C can be obtained by solving the Sylvester equation

$$D_l C(:,j) - C(:,j) D_r(j,j) = GH(j,:)^*. \tag{5.45}$$

The $(i,j)^{th}$ element of C can be computed as

$$C(i,j) = \frac{G(i,:)H(j,:)^*}{D_l(i,i) - D_r(j,j)}. \tag{5.46}$$

This indicates that unless the diagonal elements of D_l and D_r are distinct, one cannot construct all elements of C. Specifically, if $D_l(k,k) = D_r(l,l)$, then the element $C(k,l)$ cannot be computed. Such elements have to be known prior to the start of the factorization. If it so happens that $D_l = D_r$, then the entire diagonal of C has to be known *a priori*.

We now proceed to describe the LU factorization algorithm with partial pivoting. The first step of the algorithm is to compute the first column of C. This can be done as described in the previous paragraph. Let the permutation matrix that brings the pivot element to the $(1,1)$

position be P_1. Applying this permutation to the displacement equation, we get

$$P_1 D_l P_1^T P_1 C - P_1 C D_r = P_1 G H^*. \tag{5.47}$$

Let us partition the matrix $P_1 C$ as

$$P_1 C = \left[\begin{array}{c|c} d & u \\ \hline l & C_1 \end{array}\right]. \tag{5.48}$$

Let us define two matrices X and Y as

$$X = \left[\begin{array}{c|c} 1 & 0 \\ \hline ld^{-1} & I \end{array}\right] \quad \text{and} \quad Y = \left[\begin{array}{c|c} 1 & d^{-1}u \\ \hline 0 & I \end{array}\right]. \tag{5.49}$$

Then $P_1 C$ can be factored as

$$P_1 C = X \left[\begin{array}{c|c} d & 0 \\ \hline 0 & C_{sc} \end{array}\right] Y. \tag{5.50}$$

Further, let $P_1 D_l P_1^T$ and D_r be conformally partitioned as

$$P_1 D_l P_1^T = \left[\begin{array}{c|c} D_{l_1} & 0 \\ \hline 0 & D_{l_2} \end{array}\right] \quad \text{and} \quad D_r = \left[\begin{array}{c|c} D_{r_1} & 0 \\ \hline 0 & D_{r_2} \end{array}\right]. \tag{5.51}$$

Let us apply the transformation $X^{-1}(.)Y^{-1}$ to (5.47). Using (5.50) and (5.51), we can write the transformed equation as

$$X^{-1}(P_1 D_l P_1^T) X (X^{-1} P_1 C Y^{-1}) - (X^{-1} P_1 C Y^{-1}) Y D_r Y^{-1} = X^{-1} P_1 G H^* Y^{-1}. \tag{5.52}$$

The above equation can be rewritten after simplification as

$$\left[\begin{array}{c|c} D_{l_1} & 0 \\ \hline D_{l_2} l d^{-1} - l d^{-1} D_{l_1} & D_{l_2} \end{array}\right] \left[\begin{array}{c|c} d & 0 \\ \hline 0 & C_{sc} \end{array}\right] - \left[\begin{array}{c|c} d & 0 \\ \hline 0 & C_{sc} \end{array}\right] \left[\begin{array}{c|c} D_{r_1} & d^{-1} u D_{r_2} - D_{r_1} d^{-1} u \\ \hline 0 & D_{r_2} \end{array}\right] = X^{-1} P_1 G H^* Y^{-1}. \tag{5.53}$$

Equating the $(2,2)$ position in the above equation, we have

$$D_{l_2}C_{sc} - C_{sc}D_{r_2} = G_1H_1^*, \tag{5.54}$$

where G_1 is the portion of $X^{-1}P_1G$ from the second row down and H_1 is the portion of $Y^{-*}H$ from the second row down. The first column of L in the LU factorization is $[1 \; d^{-*}l^*]^*$, and the first row of U is $[d \; u]$. This completes one step of the LU factorization algorithm. The process can now be repeated on the displacement equation of the Schur complement of P_1C with respect to d (C_{sc}) to get the second column of L and row of U. After n steps, one has the LU factorization of a permuted Cauchy-like matrix.

If the displacement matrices D_l and D_r have diagonal entries that are identical, then, as pointed out earlier, some elements corresponding to these entries have to be known *a priori*. In addition, these elements have to be updated with the transformation $X^{-1}(.)Y^{-1}$ to reflect their values in the Schur complement C_{sc}. To avoid this extra step in the algorithm, it is often desirable to have D_l and D_r distinct. In some cases such as Hermitian Cauchy-like matrices, however, one cannot satisfy this condition because doing so would destroy the symmetry. In such cases the extra computation at the end of each step of the algorithm to update the diagonal elements of C is unavoidable if symmetry is to be maintained.

If the permutations at each step are accumulated into the matrix P, then we see that the above algorithm produces a factorization of the Toeplitz matrix of the form $T = F^*P^TLUF$.

5.2.2 Factoring Hermitian Cauchy-like matrices

Now consider a Hermitian Cauchy-like matrix C with the displacement equation

$$C - D_lCD_l^* = G\Sigma G^*. \tag{5.55}$$

Any column of C can be obtained by solving the Lyapunov equation

$$C(:,j) - D_lC(:,j)D_l^*(j,j) = G\Sigma G(j,:)^*. \tag{5.56}$$

If $D_l^*(j,j)$ is equal to any eigenvalue of D_l, then the corresponding element of C will have to be computed *a priori* and updated during the course of the algorithm as described earlier. For the moment, let us assume that this is not the case. Further, let us assume that the pivot block is in the right location. Since we have a symmetric Cauchy-like matrix, a pivoting strategy like Bunch-Kaufman has to be used to obtain the pivot block in place. As a result, the pivot block may either be a 1×1 or a 2×2 block matrix. Let us partition the matrix C as

$$C = \left[\begin{array}{c|c} d & l^* \\ \hline l & C_1 \end{array}\right]. \tag{5.57}$$

Let us define the matrix X as

$$X = \left[\begin{array}{c|c} I & 0 \\ \hline ld^{-1} & I \end{array}\right]. \tag{5.58}$$

Then applying $X^{-1}(.)X^{-*}$ to (5.55) we obtain

$$\left[\begin{array}{c|c} d & 0 \\ \hline 0 & C_{sc} \end{array}\right] - \left[\begin{array}{c|c} A_{11} & 0 \\ \hline A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c|c} d & 0 \\ \hline 0 & C_{sc} \end{array}\right] \left[\begin{array}{c|c} A_{11}^* & A_{21}^* \\ \hline 0 & A_{22}^* \end{array}\right] = X^{-1} G \Sigma G^* X^{-*}, \tag{5.59}$$

where $A_{11} = D_{l_1}$, $A_{22} = D_{l_2}$ and $A_{21} = D_{l_2} l d^{-1} - l d^{-1} D_{l_1}$. If the Bunch-Kaufman pivoting strategy results in a 1×1 pivot, then D_{l_1} is a 1×1 matrix; otherwise, it is of size 2×2. To proceed with the factorization of the Cauchy-like matrix, we have to obtain a displacement equation of the form (5.55) for the Schur complement of C (i.e., for C_{sc}). The displacement equation will have to be of the form

$$C_{sc} - A_{22} C_{sc} A_{22}^* = G_{sc} \Sigma_{sc} G_{sc}^*. \tag{5.60}$$

Partitioning G^* conformally as $G^* = [G_1^* | G_2^*]$ and equating the $(2, 2)$ position in (5.59), we have

$$C_{sc} - A_{22} C_{sc} A_{22}^* = (G_2 - ld^{-1} G_1) \Sigma (G_2 - ld^{-1} G_1)^* + A_{21} d A_{21}^*, \tag{5.61}$$

where $A_{21} = A_{22} l d^{-1} - l d^{-1} A_{11}$. The last term of (5.61) can be expanded as

$$\begin{aligned} A_{21} d A_{21}^* &= (A_{22} l d^{-1} - l d^{-1} A_{11}) d (d^{-1} l^* A_{22}^* - A_{11}^* d^{-1} l^*) \\ &= (A_{22} l A_{11}^* - l d^{-1} A_{11} d A_{11}^*) A_{11}^{-*} d^{-1} A_{11}^{-1} \\ &\quad (A_{11} l^* A_{22}^* - A_{11} d A_{11}^* d^{-1} l^*). \end{aligned} \tag{5.62}$$

From the displacement Equation (5.55), we can also write

$$\begin{aligned} d - A_{11} d A_{11}^* &= \widehat{G}_1 \Sigma \widehat{G}_1^* & (5.63) \\ l - A_{22} l A_{11}^* &= \widehat{G}_2 \Sigma \widehat{G}_1^*. & (5.64) \end{aligned}$$

Inserting the above equations into (5.62) yields

$$A_{21} d A_{21}^* = (G_2 - l d^{-1} G_1)(\Sigma G_1^* A_{11}^{-*} d^{-1} A_{11}^{-1} G_1 \Sigma)(G_2 - l d^{-1} G_1)^*. \tag{5.65}$$

Substituting (5.65) in (5.61), $\Delta C_{sc} = C_{sc} - A_{22} C_{sc} A_{22}^*$ has the form

$$\Delta C_{sc} = (G_2 - l d^{-1} G_1)(\Sigma + \Sigma G_1^* A_{11}^{-*} d^{-1} A_{11}^{-1} \widehat{G}_1 \Sigma)(G_2 - l d^{-1} G_1)^*. \tag{5.66}$$

Using the Sherman-Morrison-Woodbury formula and (5.63) it can be shown that

$$(\Sigma + \Sigma G_1^* A_{11}^{-*} d^{-1} A_{11}^{-1} G_1 \Sigma) = (\Sigma^{-1} - G_1^* d^{-1} G_1)^{-1}. \tag{5.67}$$

Hence, the equations to update the generator and the signature matrices are

$$G_{sc} = G_2 - l d^{-1} G_1 \qquad \text{and} \qquad \Sigma_{sc}^{-1} = \Sigma^{-1} - G_1^* d^{-1} G_1. \tag{5.68}$$

At this point, all the elements of C that were computed *a priori* have to be updated to reflect their values in the Schur complement C_{sc}. Since we now have the same displacement structure for the Schur complement C_{sc}, the factorization algorithm can proceed in the same manner to the next step and eventually to completion.

5.3 Factoring Hermitian Toeplitz Matrices

In this section, we present an algorithm to compute a symmetric factorization of a Hermitian Toeplitz matrix by converting it to a Hermitian Cauchy-like matrix. We then compare the complexity of this method to the method for factoring non-Hermitian Cauchy-like matrices. A similar algorithm has been presented in [39]. In Section 5.1.5, we presented an alternate algorithm to factor Hermitian Toeplitz matrices. This was based on the conversion of a Hermitian

Toeplitz matrix to a real, symmetric Cauchy-like matrix. The factorization of this Cauchy-like matrix is then done in real arithmetic resulting in substantial savings in computation. We postpone the discussion of this algorithm to Section 5.5, however, because it is similar to the algorithm to factor real, symmetric Toeplitz matrices. The comparison of the complexity of the two methods can be found in Table 5.3 in Section 5.5.

Consider a Hermitian Toeplitz matrix T of size n. The displacement equation of type I for such a matrix and the corresponding Cauchy-like matrix were shown in Section 5.1.2 to be

$$\begin{aligned} T - Z_1\, T\, Z_1^T &= H\, \Sigma\, H^* \\ C - \Lambda\, C\, \Lambda^* &= \widehat{H}\, \Sigma\, \widehat{H}^*. \end{aligned} \tag{5.69}$$

Since the matrix T is Hermitian, the Cauchy-like matrix C is also Hermitian. The displacement matrices Λ and Λ^* are diagonal and have entries that are complex conjugates of each other. Also, from the definition of Λ we see that the (j+1,j+1)-th entry of Λ and the (n-j+1,n-j+1)-th entry of Λ^* are identical for $j = 1, \cdots, n-1$.

$$\begin{aligned} \Lambda^*(n-j+1, n-j+1) &= e^{-\frac{2\pi i(n-j)}{n}} \\ &= e^{-\frac{2\pi i n}{n}}\, e^{\frac{2\pi i j}{n}} \\ &= e^{\frac{2\pi i j}{n}} \\ &= \Lambda(j+1, j+1). \end{aligned} \tag{5.70}$$

In addition, the $(1,1)$ elements of the two displacement matrices are identical. As indicated in Section 5.2, this means that we have to compute the elements $C(1,1)$ and $C(i,j)$ for $j = 2, \cdots, n$ and $i = n - j + 2$ *a priori*. This set of elements includes some diagonal and other off-diagonal elements. If n is even, then for $i = j = 1$ and $i = j = n/2$ the elements $C(i,j)$ are diagonal and the rest are off-diagonal. If n is odd, then the only diagonal element is $C(1,1)$. We first present a fast method to compute the non-diagonal elements of C and later indicate how the diagonal elements may be computed.

To compute the off-diagonal elements of C that are needed *a priori*, we set up a non-Hermitian form of the displacement equation for T and the corresponding Cauchy-like matrix

C as

$$\begin{aligned} Z_1\, T - T\, Z_1 &= G_1\, G_2^* \\ \Lambda\, C - C\, \Lambda &= F\, G_1\, G_2^* F^*. \end{aligned} \qquad (5.71)$$

Since $\Lambda(i,i) \neq \Lambda(j,j)$ for $i \neq j$, any off-diagonal element of C can be easily computed using (5.46). To show how the diagonal elements of C are computed, we make use of the following theorem.

Theorem 5.1 *For any matrix A of size n, if F is the DFT matrix of size n and $\mathcal{C}$ is a circulant matrix that minimizes the Frobenius norm of $(A - \mathcal{C})$, then the diagonal of FAF^* is equal to the eigenvalues of the minimizer $\mathcal{C}$.*

Since the Cauchy-like matrix C is defined as FTF^*, the diagonal elements can be obtained from the eigenvalues of the circulant minimizer $\mathcal{C}$ that minimizes the Frobenius norm of $(T - \mathcal{C})$. Further, it can be easily proved that the eigenvalues of a circulant matrix are obtained from the DFT of the first column of the matrix : $\sqrt(n) F * \mathcal{C}(:,1)$. For Toeplitz matrices, the circulant minimizer $\mathcal{C}$ can be computed in $O(n)$ flops, as demonstrated in [61]. Since we only require a few diagonal elements of C (one if n is odd and two if n is even), we can use the matrix-vector product form of the DFT (instead of an FFT) to compute them. This means that the required diagonal entries of C can be computed in $O(n)$ flops.

Having computed the elements of C that are needed *a priori*, we can now proceed with a symmetric factorization of the matrix with symmetric pivoting. The Bunch-Kaufman algorithm can be used as a symmetric pivoting scheme. We outline the first step of such an algorithm. The Bunch-Kaufman pivoting scheme requires the computation of either one or two columns of C. The computation of columns of C was described in the previous section. Let P_1 be the permutation that permutes the 1×1 or 2×2 pivot block to the proper place. The displacement equation is then written as

$$(P_1\, C\, P_1^T) - (P_1\, \Lambda\, P_1^T)\, (P_1\, C\, P_1^T)(P_1\, \Lambda^*\, P_1^T) = P_1\, \widehat{H}\, \Sigma\, \widehat{H}^*\, P_1^T. \qquad (5.72)$$

The recurrence relations between the generators of the Cauchy-like matrix and its Schur complement with respect to the pivot blocks were given in Section 5.2 by (5.68). The obvious advantage in the Hermitian case is that only half of the computation has to be performed. However, we have to compute some elements of C *a priori* because Λ and Λ^* have common eigenvalues. These elements will have to be updated at each step of the factorization to obtain their values in the Schur complement of C. It can, therefore, be seen that the reduction in computation due to the Hermitian property of T is to some extent offset by the additional work one has to accomplish in the beginning to compute some elements of C *a priori* and at every step in updating these elements.

We now determine the complexity of the two algorithms to factor non-Hermitian and Hermitian Toeplitz matrices. In all the calculations we assume that a complex multiplication requires 6 flops, a complex division requires 9 flops (assuming that a real division requires 1 flop), and a complex addition requires 2 flops. We ignore the computation required to set up the displacement equation for Toeplitz matrices since this can be done in $O(n)$ flops.

Let us first consider the non-Hermitian case. Transforming the displacement equation of a Toeplitz matrix (5.11) to that of a Cauchy-like matrix (5.13) requires 2α FFTs of length n (the displacement rank $\alpha = 2$). The cost of computing these FFTs is $2\alpha K_1 n \log n$ flops. The value of K_1 is small if n is a highly composite number. If n is not so composite (or prime), then the constant K_1 can be quite large. Computing each row or column of the factorization using (5.46) at the k-th step of the factorization requires $6\alpha(n-k)+2(\alpha-1)(n-k)+2(n-k)+9(n-k) = 8\alpha(n-k)+9(n-k)$ flops. Since a column and a row have to be computed at each step, this means that the total work at each step to obtain a row and a column of the matrix is $16\alpha(n-k)+18(n-k)$ flops. Having computed the required row and column of the factorization, we must now update the generators of the k-th step to obtain the generators of the (k+1)-th step. The update of each generator requires $8\alpha(n-k)$ for a total of $16\alpha(n-k)$ flops. Thus, the total number of flops required to factor the matrix is

$$\begin{aligned}
\text{Flops} &= 2\,K_1\,\alpha\,n\,\log(n) + \sum_{k=1}^{n-1}(32\,\alpha + 18)(n-k) \\
&= 2\,K_1\,\alpha\,n\,\log(n) + (16\,\alpha + 9)(n^2 - n) \\
&\approx (16\,\alpha + 9)n^2. \qquad (5.73)
\end{aligned}$$

For Toeplitz matrices with $\alpha = 2$, the algorithm requires approximately $41n^2$ flops.

In the Hermitian case, one would use the Bunch-Kaufman pivoting scheme. At each step of the factorization, the Bunch-Kaufman algorithm checks either one or two rows of the matrix and selects either a 1×1 or a 2×2 pivot. The worst-case scenario is that at each step two rows are checked, but a 1×1 pivot is used. The best case, however, is if a 2×2 pivot is used each time that two rows are checked. We can, therefore, only provide lower and upper bounds on the complexity of the algorithm in the Hermitian case.

Since the matrix is Hermitian, the number of FFTs needed to transform the generators of the Toeplitz-like matrix to those of a Cauchy-like matrix is exactly half of that in the non-Hermitian case. The complexity for this step is $K_1 \alpha n \log(n)$. However, extra work is necessary to compute some elements of C *a priori*. Of these elements, $n-1$ $(n-2)$ elements are off diagonal if n is odd (even). The off-diagonal elements are computed by solving the corresponding Lyapunov equations in (5.71). The complexity to do this is $2K_1 \alpha n \log(n)$ to compute FG_1 and $G_2^* F^*$ and $6\alpha + 2\alpha + 2 + 9$ to solve the Lyapunov equation for each element. In addition, there are 1 (2) elements of C that are on the diagonal if n is odd (even). These elements are computed from the DFT of the first column of the circulant minimizer discussed earlier. These elements require $2n$ $(10n)$ if n is odd (even). Hence, the total complexity of computing the elements of C needed *a priori* is $2K_1 \alpha n \log(n) + 8\alpha n + O(n)$.

We now compute the complexity of the factorization algorithm. In the worst case, at the k-th step of the factorization, two rows are computed from the generators and tested for a 1×1 pivot. The work to compute two rows from the generators is $(16\alpha + 18)(n-k)$. The number of operations required to update the generators from the k-th step to those from the (k+1)-th step is $8\alpha(n-k)$. In addition to this, some extra work is required to update the elements along the diagonal of C that were computed *a priori*. This adds an extra $14(n-k)$ flops at each step. The worst-case complexity is

$$\begin{aligned}
\text{Flops} &= 3K_1 \alpha n \log(n) + 8\alpha n + O(n) + \sum_{k=1}^{n-1} (24\,\alpha + 18)(n-k) + \sum_{k=1}^{n-1} 14(n-k) \\
&= 3K_1 \alpha n \log(n) + 8\alpha n + O(n) + (12\alpha + 9 + 7)(n^2 - n) \\
&\approx (12\alpha + 16)n^2. \qquad (5.74)
\end{aligned}$$

For $\alpha = 2$, the total complexity is $40n^2$. This shows that, in the worst case, the complexity of the Hermitian algorithm is the same as that for the non-Hermitian case. However, in the best-case scenario, only one row is checked at each step and a 1×1 pivot block is used. The complexity in this situation is

$$\begin{aligned} \text{Flops} &= 3K_1 \alpha n \log(n) + 8\alpha n + O(n) + \sum_{k=1}^{n-1} (16\,\alpha + 9)(n-k) + \sum_{k=1}^{n-1} 14(n-k) \\ &= 3K_1 \alpha n \log(n) + 8\alpha n + O(n) + (8\alpha + 11.5)(n^2 - n) \\ &\approx (8\alpha + 11.5) n^2. \end{aligned} \tag{5.75}$$

For $\alpha = 2$, the complexity is $27.5n^2$.

This indicates that the complexity of the factorization algorithm using the Bunch-Kaufman pivoting scheme can vary from $27.5n^2$ to $40n^2$ depending on the pivot sequence obtained. Preserving the Hermitian structure of the factorization reduces the complexity of the factorization algorithm to some extent. Further reduction in complexity can be obtained if the Hermitian Toeplitz matrix is converted to a real, symmetric Cauchy-like matrix. This is discussed in Section 5.5. If the Toeplitz matrix is real, one would like to preserve this property as well because computation in complex arithmetic is very expensive. In the following sections we discuss algorithms to factor real Toeplitz matrices that are either symmetric or nonsymmetric. We also compare them in complexity to the complex arithmetic cases.

5.4 Real Nonsymmetric Toeplitz Matrices

In this section, we present an algorithm to factor real nonsymmetric Toeplitz matrices by converting them to real Cauchy-like matrices. We then compare this method to the algorithms discussed in the previous sections and show how maintaining the realness property leads to significant savings in computation.

Consider a real nonsymmetric Toeplitz matrix T of size n. Following the notation of Section 5.1.3, we write the displacement equation for T as

$$Z_{00} T - T Z_{11} = G H^T. \tag{5.76}$$

The displacement rank T with respect to (Z_{00}, Z_{11}) is 4. If Z_1 and Z_{-1} are used as the displacement matrices, the displacement rank is 2. Since Z_1 and Z_{-1} are diagonalized by the DFT matrix, a real Toeplitz matrix is converted to a complex Cauchy-like matrix, and all subsequent computation is done in complex arithmetic. If, however, we use (Z_{00}, Z_{11}) as the displacement matrix pair, then real trigonometric transforms can be used to convert a real Toeplitz matrix into a real Cauchy-like matrix. The subsequent factorization is done in real arithmetic. We show that the reduction in complexity due to real arithmetic more than offsets the increased displacement rank. From (5.19) and (5.20), we can write

$$\begin{aligned}(S_{00}Z_{00}S_{00})(S_{00}TS_{11}) - (S_{00}TS_{11})(S_{11}^TZ_{11}S_{11}) &= (S_{00}G)(H^TS_{11}) \\ D_lC - CD_r &= \widehat{G}\widehat{H}^T, \end{aligned} \tag{5.77}$$

where D_l is a diagonal matrix containing the eigenvalues of Z_{00} as defined in (5.19), D_r is also a diagonal matrix containing the eigenvalues of Z_{11} as defined in (5.20), $\widehat{G} = S_{00}G$, and $\widehat{H} = S_{11}^TH$. For all n, the eigenvalues of Z_{00} and Z_{11} are distinct; hence, one would not have to compute any elements of the real Cauchy-like matrix C *a priori*. A real arithmetic version of the algorithm described in Section 5.2 can be used to factor the real Cauchy-like matrix. If the permutations at every step of the factorization are accumulated in P and the upper and lower triangular factors are denoted by U and L, then we obtain a factorization of T as

$$T = S_{00}P^TLUS_{11}^T.$$

We now compute the complexity of the factorization algorithm for real Toeplitz matrices. Transforming the Toeplitz matrix to real Cauchy-like matrices involves applying the Sine-I and Cosine-II transforms to the generators. Let the displacement rank be α (= 4 for real Toeplitz matrices). The complexity of the transformation is $2K_2\alpha n\log(n)$. If n is a power of 2, then $K_2 = 2.5$. Computing a row or column of the factorization at the k-th step using a real arithmetic version of (5.46) requires $(2\alpha - 1)(n-k) + 2(n-k)$ flops. Since both a row and column of the matrix are to be computed at every step, the total number of flops for this operation is $(4\alpha - 2)(n-k) + 4(n-k)$. Having computed the row and column of the factorization, we must update the generators of the k-th step to those of the (k+1)-th step using a real arithmetic version of (5.53). The complexity of this step is $4\alpha(n-k)$. The total number of flops for the entire factorization algorithm is then

$$\text{Flops} = 2K_2\alpha n \log{(n)} + \sum_{k=1}^{n-1}(8\alpha + 2)(n-k). \tag{5.78}$$

The asymptotic complexity is, therefore, $4\alpha n^2 + n^2$. For real Toeplitz matrices, since $\alpha = 4$, the complexity is $17n^2$. If the complex arithmetic version is used, then the complexity is $41n^2$ flops. Staying in the real domain, thus, leads to significant savings in computation.

5.5 Real, Symmetric Toeplitz Matrices

In this section, we present an algorithm to factor real, symmetric Toeplitz matrices by converting them to real, symmetric Cauchy-like matrices. We also present an algorithm that converts a Hermitian Toeplitz matrix to a real, symmetric matrix and proceeds to factor it in real arithmetic.

In Section 5.1.4, we showed that a significant reduction in complexity may be obtained if we exploit simultaneously realness and symmetry in the Toeplitz matrix. It was shown that for a real symmetric Toeplitz matrix T of size n, if the symmetric form of the displacement equation was used with a displacement matrix $Z_{\epsilon\epsilon}$, then the corresponding Cauchy-like matrix $C_{\epsilon\epsilon}$ can be decoupled into two Cauchy-like matrices of half the size. Further, it was shown that the two smaller Cauchy-like matrices have a displacement rank of 2. These two smaller Cauchy-like matrices can be factored independently of each other.

Since we use the symmetric form of the displacement Equation (5.27), the diagonal elements of $C_{\epsilon\epsilon}$ cannot be obtained by solving the corresponding Lyapunov equation. One has to compute these elements *a priori*. In the following paragraphs we show how the diagonal elements of $C_{\epsilon\epsilon}$ may be computed. We demonstrate this for the case when $\epsilon = 0$. The construction for $\epsilon = 1$ is similar. The diagonal elements of $C_{00} = S_{00}TS_{00}$ can be computed using the following theorem.

Theorem 5.2 *Let* S *be a vector-space containing all* $n \times n$ *matrices that can be diagonalized by the Sine-I transform. Then, for any matrix* A *of size* n*, if we obtain a matrix* $\mathcal{S}$ *in this space that minimizes the Frobenius norm of* $(A - \mathcal{S})$*, then the diagonal of* $S_{00}AS_{00}$ *(*S_{00} *is the Sine-I transform of size* n*) is identical to the eigenvalues of* $\mathcal{S}$*.*

In addition, it was proved independently in [40], [41], and [62] that a matrix belongs to the vector-space $\mathbf{S}$ if and only if the matrix can be expressed as the sum of a special Toeplitz and a Hankel matrix. This is outlined in the following theorem.

Theorem 5.3 *Any matrix $\mathcal{S}$ in $\mathbf{S}$ can be written as $\mathcal{S} = X - Y$, where X is a symmetric Toeplitz matrix with first column $x = [x_1\ x_2\ \cdots\ x_n]^T$, and Y is a Hankel matrix with first column $[0\ 0\ x_n \cdots\ x_3]^T$ and last column $[x_3\ \cdots\ x_n\ 0\ 0]^T$.*

In [43], Chan, Ng, and Wong show how the minimizer $\mathcal{S}$ may be constructed for any matrix A in $O(n^2)$ flops. If A is Toeplitz, then they show that this computation requires only $O(n \log(n))$ flops. The algorithm proceeds by setting the partial derivative of $\|A - \mathcal{S}\|$ w.r.t. $x_1,\ x_2,\ \cdots,\ x_n$ equal to zero. We summarize the lemmas and algorithms that are important to this discussion. An important lemma due to Boman and Koltracht [41] gives a basis for the vector space $\mathbf{S}$.

Lemma 5.1 *Let Q_i, $i = 1, \cdots, n$ be $n \times n$ matrices with the (j,k) entry being given by*

$$Q_i = \begin{cases} 1 & \textit{if } |j-k| = i-1 \\ -1 & \textit{if } j+k = i-1 \\ -1 & \textit{if } j+k = 2n-i-3 \\ 0 & \textit{otherwise.} \end{cases}$$

Then $\{Q_i\}_{i=1}^n$ is a basis for $\mathbf{S}$.

Let us define a vector $r = [r_1\ r_2\ \cdots\ r_n]$, where

$$r_i = \mathbf{1}_n^T (Q_i \circ A)\mathbf{1}_n, \tag{5.79}$$

$\mathbf{1_n}$ is column vector of ones of length n and $\circ$ denotes the element-wise product. The following corollary by Chan, Ng, and Wong [43] gives an explicit formula for the entries on the first column of the minimizer $\mathcal{S}$ for any matrix A.

Corollary 5.1 *Let A be a symmetric matrix of size n and let $\mathcal{S}$ be the minimizer of $\|A - \mathcal{S}\|_F$ over all matrices in the vector space $\mathbf{S}$. Let $\mathbf{z}$ be the first column of $\mathcal{S}$ and $r_i = \mathbf{1}_n^T(Q_i \circ A)\mathbf{1}_n$. If s_o and s_e are defined to be the sum of the odd and even entries of the vector r, then we have*

$$z_1 = \frac{1}{2(n+1)}(2r_1 - r_3)$$

$$z_i = \frac{1}{2(n+1)}(r_i - r_{i+2}) \qquad i = 2, \cdots, n-2$$

and

$$\begin{aligned} z_{n-1} &= \frac{1}{2(n+1)}(s_o + r_{n-1}) \\ z_n &= \frac{1}{2(n+1)}(2s_e + r_n) \end{aligned}$$

if n is even; and

$$\begin{aligned} z_{n-1} &= \frac{1}{2(n+1)}(s_e + r_{n-1}) \\ z_n &= \frac{1}{2(n+1)}(2s_o + r_n) \end{aligned}$$

if n is odd.

The eigenvalues of the minimizer $\mathcal{S}$ can now be calculated from the first column of $\mathcal{S}$.

$$\begin{aligned} \mathcal{S} = S_{00}\Lambda S_{00} \quad &\Rightarrow \quad S_{00}\mathcal{S}e_1 = \Lambda S_{00}e_1 \\ &\Rightarrow \quad \Lambda = D^{-1}S_{00}\mathcal{S}e_1, \qquad \text{where} \qquad D = diag(S_{00}e_1). \end{aligned} \tag{5.80}$$

For any arbitrary matrix A, it is clear that the vector r can be computed in $O(n^2)$ flops and the diagonal of $S_{00}AS_{00}$ in $O(n^2 + n\log(n))$ flops. If A were Toeplitz, then r can be computed in $O(n)$ flops and the diagonal of the Cauchy-like matrix $C_{00} = S_{00}AS_{00}$ can be computed in $O(n\log(n))$ flops. In [43], the authors present the following $O(n)$ algorithm to obtain r given a symmetric Toeplitz matrix T of size n whose first column is $[t_1\ t_2\ \cdots\ t_n]^T$.

Algorithm 5.1

```
$r_1 = nt_1$
$r_2 = 2(n-1)t_2$
$w_1 = -t_1$
$v_1 = -2t_2$
for $k = 2 : \lfloor n/2 \rfloor$
    $r_{2k-1} = 2(n-2k+2)t_{2k-1} + 2w_{k-1}$
    $w_k = w_{k-1} - 2t_{2k-1}$
```

```
        $r_{2k} = 2(n-2k+1)t_{2k} + 2v_{k-1}$
        $v_k = v_{k-1} - 2t_{2k}$
    end
    if  $n$ is odd
        $r_n = 2t_n + 2w_{(n-1)/2}$
    end
```

From the above discussion, it can be seen that the total complexity of computing the diagonal elements of the Cauchy-like matrix $C_{00} = S_{00}TS_{00}$ is $O(n\log(n))$. The next step in the factorization of the Cauchy-like matrix C_{00} is the application of the odd-even sort permutation matrix P, as shown in (5.28), to expose the sparsity of C_{00} and to separate the large system of equations into two independent systems of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively. Each subsystem can be solved using the real arithmetic variant of the algorithm to factor non-Hermitian Cauchy-like matrices discussed in Section 5.2. Since the two Cauchy-like matrices are symmetric, we use the Bunch-Kaufman algorithm to search for a pivot.

We now estimate the complexity of the factorization algorithm. Consider a Cauchy-like matrix of size m with the displacement equation $D_lC - CD_l = G\Sigma G^T$. At the k-th step of the factorization algorithm, computing a row or column of the matrix requires $(2\alpha+1)(m-k)$ flops. The worst-case scenario is one in which, at each step, two rows are computed and checked and only a 1×1 pivot block is used. The complexity to update the generators for the next step is $2\alpha(m-k)$. In addition, the elements of the Cauchy-like matrix that were computed *a priori* have to be updated. The complexity to do this at the k-th step is $3(m-k)$. The total complexity in the worst-case scenario is, therefore,

$$\begin{aligned} \text{Flops} &= \sum_{k=1}^{m-1}(6\alpha+5)(m-k) \\ &= (3\alpha+2.5)m^2. \end{aligned} \tag{5.81}$$

In the best-case scenario, at each step, only one row is checked and a 1×1 pivot is used. The best case complexity is

$$\text{Flops} = \sum_{k=1}^{m-1}(4\alpha+4)(m-k)$$

$$= (2\alpha + 2)m^2. \tag{5.82}$$

As shown in Section 5.1.4, there are two independent systems each of size approximately $n/2$ with a displacement rank of $\alpha = 2$. In the worst case, the total complexity for factoring real, symmetric Toeplitz matrices of size n by converting them to Cauchy-like matrices is

$$\text{Flops} = K_2(\alpha + 1)n \log(n) + O(n) + 4.25n^2 \tag{5.83}$$

and, in the best case, the complexity is

$$\text{Flops} = K_2(\alpha + 1)n \log(n) + O(n) + 3n^2. \tag{5.84}$$

A similar algorithm can be used if we choose to use the displacement matrix Z_{11} instead of Z_{00}. The diagonal elements of C_{11} can be computed in a similar manner [63] .

5.5.1 Implementation on the Cray J90 and T90

We now present the results of some implementations of this algorithm on Cray Parallel Vector Processor (PVP) systems such as the Cray J90 and T90. The J90 can be configured with up to 32 processors. Each processor is rated as having a peak performance of 200 MFlops. The T90 system, on the other hand, is a machine with a similar architecture but a peak performance of a 2 GFlops/processor.

The algorithm presented in this section splits a large real, symmetric Toeplitz system to two real, symmetric Cauchy-like matrices of half the size. These systems may be solved independently. For a two-processor system, this yields perfect parallelism. At each step of the factorization of the two smaller systems, one has additional parallelism in the following tasks: computing a row of the factorization from the generators, searching for the pivot elements, and updating the generators for the next step of the factorization. On Cray PVP systems a list of autotasking directives is provided to the user to exploit parallelism in the program. The limitation of these directives is that once parallelism has been invoked at the higher-level by breaking the work into several tasks, each task can only be executed on a single processor. Therefore, if we choose to invoke parallelism at the highest level by considering the factorization of the Cauchy-like matrices to be two concurrent tasks, then we cannot exploit the parallelism at the lower level. Table 5.1 shows the time in milliseconds to factor a 4095×4095 real, symmetric Toeplitz matrix using two processors. It can be seen that the speedup for two processors is almost 2.

Table 5.1 Time, in milliseconds, to factor a 4095×4095 real, symmetric Toeplitz matrix while exploiting parallelism at the higher level.

Number of CPUS	J90	T90
1	872	143
2	448	74

Since two levels of parallelism cannot be exploited using the autotasking directives provided on the Cray PVP systems, we could ignore the concurrency in the two factorization tasks and exploit the parallelism at the lower level. This means that the two Cauchy-like matrices are factored one after the other and that the entire set of processors works to factor each Cauchy-like matrix. The effectiveness of this method is limited by the fact that the amount of work decreases linearly at each step of the factorization. A fixed amount of overhead is incurred each time autotasking is invoked to exploit the parallelism in computing a row of the factorization or updating the generators of the next step. Beyond a certain point in the factorization, due to reduced work and a fixed overhead, it is better to disregard the parallelism and exploit only vectorization. This means that increasing the number of processors beyond a certain point to solve the problem will only yield diminishing returns. For the same problem of factoring a 4095×4095 real, symmetric Toeplitz matrix, Table 5.2 shows the time, in milliseconds, on the J90 when autotasking is invoked at each step of the factorization of the two Cauchy-like matrices. For the sake of comparison, the time, in milliseconds, to factor the matrix using the LAPACK routine SSYTRF has also been tabulated in Table 5.2.

From Tables 5.1 and 5.2, it can be seen that exploiting the higher level parallelism with only two CPUS gives the best performance on the Cray PVP machines. This is true for larger problem sizes as well because the total amount of work in factoring a real, symmetric Toeplitz matrix using this algorithm is quite small ($3n^2$ to $4n^2$) and the parallelism reduces linearly with each step of the factorization.

Table 5.2 Time in milliseconds to factor a 4095×4095 real, symmetric Toeplitz matrix on a J90 while exploiting parallelism at each step of the factorization.

Number of CPUS	Time for Cauchy-based algorithm	Time for SSYTRF (LAPACK)
1	872	139149
2	757	74601
4	629	43540
6	596	36659
8	584	32431

5.5.2 Factoring Hermitian Toeplitz matrices

In Section 5.1.5, we showed how a Hermitian Toeplitz matrix may be converted to a real, symmetric Cauchy-like matrix with a displacement rank of 4. If T is a Hermitian Toeplitz matrix of size n, then the displacement equation with respect to $Z_{\epsilon\epsilon}$ has a rank of 4.

$$Z_{\epsilon\epsilon}\, T - T Z_{\epsilon\epsilon} \; = G\Sigma G^{*}. \tag{5.85}$$

If $\widehat{S}_{\epsilon\epsilon} \; = S_{\epsilon\epsilon}\; PD^{*}$, where D is defined in (5.42) and P is the odd-even permutation matrix, then the Cauchy-like matrix $\widehat{S}^{*}_{\epsilon\epsilon}T\widehat{S}_{\epsilon\epsilon}$ is real and symmetric with a displacement rank of 4. Since we use the symmetric form of the displacement equation, the diagonal entries of the Cauchy-like matrix will have to be computed *a priori*.

Since $\widehat{S}_{\epsilon\epsilon} \; = S_{\epsilon\epsilon}\; PD^{*}$, the diagonal of $\widehat{S}^{*}_{\epsilon\epsilon}T\widehat{S}_{\epsilon\epsilon}$ is computed by first computing the diagonal of ${S_{\epsilon\epsilon}}^{T}TS_{\epsilon\epsilon}$ and then applying the matrices P and D. Note that if T is Hermitian, then the imaginary part of T is skew-symmetric. The imaginary part of T, therefore, does not contribute anything to the vector r defined in (5.79). Hence, the diagonal of ${S_{\epsilon\epsilon}}^{T}TS_{\epsilon\epsilon}$ can be computed using only the real part of T. Having computed the diagonal of ${S_{\epsilon\epsilon}}^{T}TS_{\epsilon\epsilon}$, the diagonal of the Cauchy-like matrix $\widehat{S}^{T}_{\epsilon\epsilon}TS_{\epsilon\epsilon}$ is computed trivially. Computing the diagonal of the Cauchy-like matrix $\widehat{S}^{*}_{\epsilon\epsilon}T\widehat{S}_{\epsilon\epsilon}$ requires $O(nlog(n))$ flops.

The factorization of the real, symmetric Cauchy-like matrix then proceeds exactly as discussed earlier in this section using the Bunch-Kaufman pivoting scheme. From (5.81) and (5.82), substituting $\alpha = 4$ and $m = n$, we see that the complexity of factoring the real, symmetric Cauchy-like matrix is between $10n^2$ to $14.5n^2$ flops, which is significantly less than that of the algorithm described in Section 5.3, which converts a Hermitian Toeplitz matrix to a Hermitian Cauchy-like matrix using the DFT.

Table 5.3 summarizes the complexity of all the algorithms discussed in this chapter. It can be seen that exploiting certain properties in the Toeplitz matrix, such as realness, symmetry, or Hermitian symmetry, can yield substantial savings in computation provided the appropriate algorithm is used. For Hermitian Toeplitz matrices, there are two alternatives available. One converts it to a Hermitian Cauchy-like matrix prior to factorization, and the other converts it to a real, symmetric Cauchy-like matrix. The latter algorithm is much cheaper. However, the numerical accuracy of this algorithm needs to be studied.

Table 5.3 Comparison of cost to factor Toeplitz matrices by converting them to Cauchy-like matrices.

Input Matrix	Intermediate Matrix	Transform Used	Transform in Section	Factorization in Section	Complexity
General Toeplitz	non-Hermitian Cauchy-like	FFT	5.1.1	5.2.1	$41n^2$
Hermitian Toeplitz	Hermitian Cauchy-like	FFT	5.1.2	5.3	$27n^2$ to $40n^2$
Hermitian Toeplitz	Symmetric Cauchy-like	DST-I or DCT-II	5.1.5	5.5.2	$10n^2$ to $14.5n^2$
Non-symm. Toeplitz	Non-symm. Cauchy-like	DST or DCT	5.1.3	5.4	$17n^2$
Symmetric Toeplitz	Symmetric Cauchy-like	DST-I or DCT-II	5.1.4	5.5	$3n^2$ to $4.25n^2$

5.6 Generalization to Block Toeplitz Matrices

All the algorithms described in this chapter generalize to block Toeplitz matrices in the following manner. The block Toeplitz matrix is converted to a Toeplitz block matrix via a perfect shuffle permutation along the rows and the columns. The Toeplitz block matrix then has a displacement rank of $r\alpha$ with respect to the displacement matrix, where r is the block size of the block Toeplitz matrix and α is the displacement rank of a point Toeplitz matrix with respect to the same displacement matrix. For example, a real Toeplitz block matrix that results from a perfect shuffle permutation applied to a 1024×1024 real block Toeplitz matrix with a block size of 8 has a displacement rank of 32 with respect to the displacement matrices Z_{00} and Z_{11}.

5.7 Conclusion

In this chapter we have studied the various algorithms to factor different types of Toeplitz matrices by converting them to Cauchy-like matrices. The original algorithm of Gohberg et al. [38] does not exploit properties such as symmetry and realness in the matrix and is applicable only to non-Hermitian Toeplitz matrices. For Hermitian Toeplitz matrices, a complex arithmetic version that exploits Hermitian symmetry was proposed independently by Gallivan et al. [24], and by Kailath and Olshevsky [39]. In this chapter we have presented an algorithm that converts a Hermitian Toeplitz matrix to a real, symmetric Cauchy-like matrix that is factored using real arithmetic. This algorithm provides a significant improvement over the previous algorithms because it completely avoids complex arithmetic. For real, symmetric Toeplitz matrices, factorization algorithms that exploit realness and symmetry simultaneously have been presented. The computational complexity of all the algorithms discussed in this chapter has been calculated and compared.

While these algorithms produce numerically robust factorizations of Toeplitz matrices, they are all quite expensive. For example, in the case of an indefinite Toeplitz matrix, one could either convert the matrix to a Cauchy-like matrix and use Gaussian elimination with partial pivoting, or one could use a look-ahead scheme to jump over the singular principal minors. Another option is to perturb the singular minor away from singularity and obtain an inexact factorization. The solution is then made numerically acceptable through iterative refinement. In the presence of few singular principal minors or a small look-ahead step, the standard algorithms may produce a numerically acceptable solution in a much shorter time. Often this information is not available and, because of numerical reasons, one is forced to convert Toeplitz matrices to Cauchy matrices.

CHAPTER 6

TOEPLITZ LEAST SQUARES AND QR FACTORIZATION

6.1 Introduction

In [20], Chun et al. present an algorithm to compute the QR factorization of a Toeplitz matrix T based on a generalization of the classical Schur algorithm. This algorithm exploits the fact that the normal equation matrix T^TT has a displacement rank of 4 with respect to the down-shift matrix. The triangular factor R of the QR factorization is obtained by using the Schur algorithm to compute the Cholesky factorization of T^TT. The orthogonal factor Q is obtained by considering the matrix $[T^TT \quad T^T]$ and updating the matrix T^T with the rows of the Cholesky factor R in the same way as the Modified Gram-Schmidt algorithm. However, since T has a low displacement rank, the columns of the Q factor are obtained from the generator of T. If the Toeplitz matrix T has full column rank, then the matrix T^TT is positive definite and the generalized Schur algorithm proceeds to completion without breaking down. If T is exactly column rank deficient, then the generalized Schur algorithm breaks down because the matrix T^TT is positive semi-definite and a zero pivot element is encountered. In this chapter, we present a modification of the generalized Schur algorithm that jumps over exact singularities in the algorithm and computes a rank factorization of the Toeplitz matrix. In the presence of numerically linearly dependent columns, a drop tolerance may be used to compute an approximate rank factorization. This is true only if the drop tolerance does not cause the approximate Schur complement to lose positive definiteness. In such cases, we suggest the use of a rank revealing QR factorization algorithm based on converting the matrix $[T^TT \quad T^T]$ to a Cauchy-like matrix using the techniques described in the previous chapter. We also present algorithms to solve Toeplitz least squares problems based on converting the normal equations and the augmented system of equations to Cauchy-like matrices. Finally, some performance results of the algorithms on parallel vector processors, such as the Cray YMP and J90, are given.

6.2 QR Factorization of Rank Deficient Toeplitz Matrices

We begin this section by summarizing the generalized Schur algorithm proposed by Chun et al. to compute the QR factorization of full column rank point Toeplitz matrices [20]. If the Toeplitz matrix has linearly dependent columns, then this algorithm breaks down because the corresponding column of the generator matrix has a zero hyperbolic norm. After summarizing the generalized Schur algorithm, we show how this algorithm may be modified to jump over exactly linearly dependent columns and proceed with the factorization.

Consider a full column rank point Toeplitz matrix T of size $p \times n$. Let Z_1 be a down shift matrix of size n. The displacement rank of T^TT with respect to Z_1 is 4 [20]. The displacement equation of T^TT is given by

$$T^TT - Z_1T^TTZ_1 = G^T\Sigma G = \begin{bmatrix} G_1^T & G_2^T & G_3^T & G_4^T \end{bmatrix} \begin{bmatrix} I & & & \\ & I & & \\ & & -I & \\ & & & -I \end{bmatrix} \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix}, \tag{6.1}$$

where G_1, G_2, G_3 and G_4 are vectors of size $1 \times n$. The Cholesky factor R of T^TT can be obtained by applying the Schur algorithm to the displacement equation shown above. To obtain the orthogonal factor Q, we consider the displacement equation of the matrix $[T^TT \;\; T^T]$. Let Z_2 be a down shift matrix of size p and let Z be defined as

$$Z = \begin{bmatrix} Z_1 & \\ & Z_2 \end{bmatrix}. \tag{6.2}$$

The displacement equation is given by

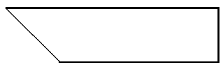
$$\begin{bmatrix} T^TT & T^T \end{bmatrix} - Z_1 \begin{bmatrix} T^TT & T^T \end{bmatrix} Z^T = G\Sigma \left[\begin{array}{c|c} G_1 & H_1 \\ G_2 & H_2 \\ G_3 & H_3 \\ G_4 & H_4 \end{array}\right], \tag{6.3}$$

where H_1, H_2, H_3 and H_4 are vectors of size $1 \times p$. The generator of $[T^TT \;\; T^T]$ can be computed in $O((p+n)\log(p+n))$ flops. At each step of the factorization, the pivot element of G_1 is used to zero out the elements below it using block hyperbolic Householder transformations similar to those discussed in Chapter 2. This hyperbolic Householder transformation is then applied to the rest of the generator. The first rows of G and H concatenated form the corresponding row of the triangular factor R and the column of the orthogonal factor Q. The shift operation, however, differs from that used in the classical Schur algorithm in that the first rows of G and H are shifted one element to the right independently of each other. For example, at the first step, if $G_1^{(1)}$ and $H_1^{(1)}$ are the first rows of G and H after the first hyperbolic Householder transformation is applied to the generator, then the shift operation is carried out as $[G_1^{(1)} \;\; H_1^{(1)}] \;\; Z^T$.

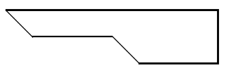
The generalized Schur algorithm described above can only be applied to Toeplitz systems with full column rank. Several applications in signal and image processing give rise to Toeplitz systems that are column rank deficient. In these least squares problems, regularization has to be applied in order to yield an acceptable solution.

A standard approach is to apply Tikhonov regularization [2], which yields a full rank matrix that can be factored using the generalized Schur algorithm. The complexity of this scheme is of $O(n^2)$ complexity. If the matrix T is large and its column rank r is small compared to the dimensions of T ($r << \min\{p, n\}$), then using Tikhonov regularization and the generalized Schur algorithm is expensive. One would like to have a regularization scheme and a corresponding factorization that has a complexity of $O(nr)$. In this section, we present a modification of the generalized Schur algorithm that jumps over exactly linearly dependent columns of T and has a complexity of $O(nr)$. After obtaining a rank factorization of the matrix $T = QR$, where Q is of size $p \times r$ and R is of size $r \times n$, an SVD of R may be computed and an SVD based regularization scheme may be used.

In this section, we only describe how to compute the upper triangular factor R. This can be extended trivially (just as in the generalized Schur algorithm) to obtain the orthogonal factor Q. If the matrix Toeplitz T is column rank deficient, then the matrix T^TT is positive semidefinite. The displacement rank of T^TT is 4. The $r \times n$ "upper-triangular" factor R of T^TT has a rank profile of type



or

.

Rank deficient matrices with displacement rank 2 have upper triangular factors that are always of the first type, whereas rank deficient matrices with displacement rank 4 have upper triangular factors that can be of both types.

Consider a column rank deficient point Toeplitz matrix T of size $p \times n$. Let the matrix have l consecutive linearly dependent columns $T(:,k), \ldots, T(:,k+l-1)$. We show that in this case a very particular property holds in the generator obtained at the start of the k-th step of the generalized Schur algorithm. Let us denote the generator of $T^T T$ at the i-th step of the generalized Schur algorithm by

$$G^{(i)} = \begin{bmatrix} G_1^{(i)} \\ G_2^{(i)} \\ G_3^{(i)} \\ G_4^{(i)} \end{bmatrix}, \tag{6.4}$$

where $G^{(i)}$ is of size $4 \times (n-i+1)$. Since the matrix $T^T T$ is positive semidefinite and the matrix T has l linearly dependent columns $k, \ldots, k+l-1$, the Schur complement of $T^T T$ with respect to the (k-1)-th principal minor has the form

$$T_{sc}^{(k-1)} = \left[\begin{array}{c|c} 0_{l,l} & 0_{l,(n-k+1-l)} \\ \hline 0_{(n-k+1-l),l} & X \end{array}\right], \tag{6.5}$$

where X is a $(n-k-l) \times (n-k-l)$ matrix with nonzero entries. The displacement of the Schur complement $T_{sc}^{(k-1)}$ also has the same sparsity pattern. The generator at the start of the k-th step of the generalized Schur algorithm is (the superscript indicating the k-th step has been dropped for convenience)

$$G = \left[\begin{array}{c|ccc} g_{11} & g_{12} & \cdots & g_{1(n-k+1)} \\ g_{21} & g_{22} & \cdots & g_{2(n-k+1)} \\ g_{31} & g_{32} & \cdots & g_{3(n-k+1)} \\ g_{41} & g_{42} & \cdots & g_{4(n-k+1)} \end{array}\right]. \tag{6.6}$$

Instead of applying a hyperbolic Householder transform to zero out the first column using g_{11}, we first apply two orthogonal transforms to zero out g_{21} and g_{41} using g_{11} and g_{31}, respectively. Let Q_1 and Q_3 be those transforms. The sparsity pattern of the generator will then be

$$\begin{bmatrix} Q_1^T & \\ & Q_3^T \end{bmatrix} G = \left[\begin{array}{c|ccccc} \hat{g}_{11} & \cdots & \hat{g}_{1l} & \hat{g}_{1(l+1)} & \cdots & \hat{g}_{1(n-k+1)} \\ 0 & \cdots & 0 & \hat{g}_{2(l+1)} & \cdots & \hat{g}_{2(n-k+1)} \\ \hat{g}_{31} & \cdots & \hat{g}_{3l} & \hat{g}_{3(l+1)} & \cdots & \hat{g}_{3(n-k+1)} \\ 0 & \cdots & 0 & \hat{g}_{4(l+1)} & \cdots & \hat{g}_{4(n-k+1)} \end{array}\right]. \tag{6.7}$$

Since the $(1,1)$ element of the Schur complement is 0, we have

$$\hat{g}_{11}^2 - \hat{g}_{31}^2 = 0 \tag{6.8}$$

and since the first row of the displacement of the Schur complement is zero we have

$$\hat{g}_{11} \begin{bmatrix} \hat{g}_{11} & \hat{g}_{12} & \cdots & \hat{g}_{1(n-k+1)} \end{bmatrix} - \hat{g}_{31} \begin{bmatrix} \hat{g}_{31} & \hat{g}_{32} & \cdots & \hat{g}_{3(n-k+1)} \end{bmatrix} = 0. \tag{6.9}$$

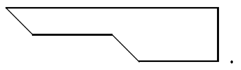
The above equations yield

$$\begin{bmatrix} \hat{g}_{11} & \hat{g}_{12} & \cdots & \hat{g}_{1(n-k+1)} \end{bmatrix} = \begin{bmatrix} \hat{g}_{31} & \hat{g}_{32} & \cdots & \hat{g}_{3(n-k+1)} \end{bmatrix}. \tag{6.10}$$

Hence, the first and third rows of the generator shown in (6.7) can be dropped. Also, since the first l columns of the reduced generator are zero, we can skip the next l steps of the generalized Schur algorithm. The generator at the start of the (k+l)-th step of the generalized Schur algorithm is

$$\begin{bmatrix} \hat{g}_{2(l+1)} & \cdots & \hat{g}_{2(n-k+1)} \\ \hat{g}_{4(l+1)} & \cdots & \hat{g}_{4(n-k+1)} \end{bmatrix}. \tag{6.11}$$

Since the matrix $T^T T$ is a positive semidefinite matrix, if the pivot column of the generator has a zero hyperbolic norm, then the $(1,1)$ element of the displacement of the Schur complement will be zero and the entire row will also be zero. A detection of a zero hyperbolic norm of the pivot column of the generator is, therefore, sufficient to drop the rank of the generator. The next time the pivot column of the generator has zero hyperbolic norm, the rank of the generator again drops by 2 causing the Schur algorithm to terminate with an upper triangular factor of the form



This reduction in the generator size and rank avoids breakdowns. The algorithm has as many steps as the number of linearly independent columns in T. The complexity of the algorithm, therefore, is $O(nr)$ (where r is the column rank of T) as opposed to $O(n^2)$ for matrices with full column rank. If a rank factorization of the Toeplitz matrix is desired, then the above algorithm could be applied to the augmented matrix $[T^TT \quad T^T]$ and the corresponding columns of the orthogonal factor Q can be computed.

If the matrix T has columns that are nearly linearly dependent on the other columns (i.e., it is nearly rank deficient), then the hyperbolic norm of the pivot column of the generator at those steps will be nonzero. In this case, a simple thresholding mechanism applied to the above algorithm can be used to obtain an approximate low rank decomposition of the matrix T^TT. Care should be taken to ensure that the positive definiteness property is not lost. If one is solving least squares problems, it is easy also to use the obtained decomposition to perform a few steps of iterative refinement on the seminormal equations.

If the matrix T is a block Toeplitz matrix of block size m, then the generator at the start of the generalized Schur algorithm has rank $4m$. Again, if the hyperbolic norm of the generator is zero, then the Schur complement will have a leading "zero." Also since the matrix T^TT is semidefinite, the entire row of the displacement of the Schur complement will be zero and the rank of the generator can be dropped by 2 by dropping the two identical rows with opposite signatures.

The algorithm proposed in this section is a significant simplification over a similar approach proposed in [47] which uses the Levinson algorithm with look-ahead. We include an example to illustrate the above algorithm. Consider a Toeplitz matrix T

$$T = \begin{bmatrix} 5 & 4 & 3 & 2 & 1 & 2 & 2 & 3 \\ 6 & 5 & 4 & 3 & 2 & 1 & 2 & 2 \\ 7 & 6 & 5 & 4 & 3 & 2 & 1 & 2 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 \\ 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 \\ 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 \\ 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 \\ 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 \\ 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 \\ 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 \end{bmatrix}. \tag{6.12}$$

Columns 3, 4, and 5 are linearly dependent on the first two columns, whereas 6, 7, and 8 are again linearly independent. The generalized Schur algorithm uses the generator

$$G^{(0)} = \begin{bmatrix} 34.7851 & 31.6228 & 28.4605 & 25.2982 & 22.1359 & 19.2611 & 16.5876 & 14.2877 \\ 0 & 4.0000 & 3.0000 & 2.0000 & 1.0000 & 2.0000 & 2.0000 & 3.0000 \\ 0 & 31.6228 & 28.4605 & 25.2982 & 22.1359 & 19.2611 & 16.5876 & 14.2877 \\ 0 & 15.000 & 14.000 & 13.000 & 12.000 & 11.000 & 10.000 & 9.0000 \end{bmatrix}$$

for the matrix $T^T T$. Two steps of the (generalized) Schur algorithm generate the first two rows of the upper triangular factor of $T^T T$. At the beginning of the third step, the first column of the generator has a Σ-norm equal to zero.

$$G^{(2)} = \begin{bmatrix} 1.0000 & 2.0000 & 3.0000 & 4.0000 & 3.9091 & 3.4545 \\ -0.7583 & -1.5166 & -2.2749 & -0.9113 & -0.5391 & 0.9020 \\ -1.2515 & -2.5030 & -3.7545 & -3.7545 & -3.4860 & -2.2985 \\ -0.0936 & -0.1873 & -0.2809 & 0.0827 & 0.4783 & 1.1324 \end{bmatrix}$$

We then use Householder transformations to eliminate $G^{(2)}(2,1)$ using $G^{(2)}(1,1)$ and $G^{(2)}(4,1)$ using $G^{(2)}(3,1)$. This gives us the generator

$$\widetilde{G}^{(2)} = \begin{bmatrix} -1.2550 & -2.5100 & -3.7650 & -3.7379 & -3.4406 & -2.2076 \\ 0 & 0 & 0 & 1.6908 & 1.9324 & 2.8060 \\ -1.2550 & -2.5100 & -3.7650 & -3.7379 & -3.4406 & -2.2076 \\ 0 & 0 & 0 & -0.3626 & -0.7370 & -1.3008 \end{bmatrix}.$$

Since the first and third rows of the generator are equal and have signatures of opposite signs, they can be removed and the generator for the next step will have only two columns. Also, it can be seen that the first three columns of this generator are zeros, which means that we can skip the corresponding rows in the upper triangular factor R. The next step would use the generator

$$G^{(5)} = \begin{bmatrix} 1.6908 & 1.9324 & 2.8060 \\ -0.3626 & -0.7370 & -1.3008 \end{bmatrix},$$

and the factorization process continues. This finally yields the triangular factor

$$R = \begin{bmatrix} -34.785 & -31.623 & -28.461 & -25.298 & -22.136 & -19.261 & -16.588 & -14.288 \\ & 1.000 & 2.000 & 3.000 & 4.000 & 3.909 & 3.455 & 2.182 \\ & & & & & -1.651 & -1.817 & -2.587 \\ & & & & & & 1.618 & 1.708 \\ & & & & & & & -1.578 \end{bmatrix}.$$

The backward error δA of the matrix $A = T^T T$ defined as

$$||\delta A|| = \frac{||A - R^T R||}{||A||},$$

is 3.57×10^{-15}, which is of the order of the machine precision ($\epsilon \approx 2.22 \times 10^{-16}$). This shows the good numerical behavior of the regularization algorithm.

The numerical behavior of this algorithm was good because the above example was exactly rank deficient. If there is a sharp drop in the singular values of the matrix, this algorithm will yield accurate results. However, if there is no sharp drop, then this algorithm may produce an inaccurate factorization due to the sensitivity of Schur complements [56]. The algorithm discussed in the next section addresses this issue.

6.3 Rank Revealing QR Factorization of Toeplitz Matrices

In the previous section, we discussed a modification of the generalized Schur algorithm to jump over singularities caused by exactly linearly dependent columns in a Toeplitz matrix. If the Toeplitz matrix were nearly column rank deficient, then it was suggested that a drop tolerance be used to obtain an approximate rank factorization of the matrix, because no form of pivoting can be incorporated into the generalized Schur algorithm. If the matrix $[T^T T \; T^T]$ were converted to a Cauchy-like matrix using the techniques described in Chapter 5, then diagonal pivoting can be used in the factorization and a rank revealing QR factorization of the matrix T can be obtained. This would significantly improve the numerical accuracy of the problem.

Consider the Toeplitz matrix in (6.12). Columns 3, 4, and 5 are exactly linearly dependent on the first two columns and 6, 7, and 8 are again linearly independent. For this matrix, the modification of the generalized Schur algorithm discussed in Section 6.2 works well because the columns are exactly linearly dependent. If the third element in the first row of the matrix T is slightly perturbed from 3 to 3.001, then the columns 3, 4, and 5 are no longer exactly dependent on the first two columns. The singular values of this slightly perturbed matrix T are

$$\text{S.V.} = \begin{bmatrix} 70.97803763483665 \\ 5.83839918699007 \\ 3.07175336231008 \\ 0.92475222044967 \\ 0.88115779119142 \\ 0.00000106271294 \\ 0.00000045293333 \\ 0.00000001416288 \end{bmatrix}. \tag{6.13}$$

It can be argued that the numerical rank of this matrix is still 5 because of the sharp drop from the fifth to the sixth singular values. Since the last singular value of the matrix is almost equal to the square-root of machine precision (1.49×10^{-8}), the matrix $T^T T$ is numerically singular and the generalized Schur algorithm breaks down. This can be seen when the following errors are computed:

$$\begin{aligned} \|I - Q^TQ\| &= 0.9981 \\ \|T - QR\| &= 2.4473 \times 10^{-5} \\ \|T^TT - R^TR\| &= 4.5362 \times 10^{-7}. \end{aligned} \tag{6.14}$$

Even the modified generalized Schur algorithm does not yield a good factorization because the columns of T are no longer exactly linearly dependent. We therefore seek an algorithm that allows us to incorporate pivoting into the QR factorization process. One alternative is to use the Householder transformation-based rank revealing QR factorization algorithm [64], but it is very expensive. In this section, we present a fast rank revealing QR factorization algorithm that has significantly better numerical performance than the generalized Schur algorithm. This algorithm is essentially derived by adapting the generalized Schur algorithm to Cauchy-like matrices.

Let T be a real Toeplitz matrix of size $m \times n$. We present an algorithm to obtain a rank revealing QR factorization of this matrix. The rank revelation process is carried out by performing a pivoted factorization of the extended matrix A defined by

$$A = \begin{bmatrix} T^TT & T^T \\ T & 0 \end{bmatrix}. \tag{6.15}$$

A pivoted factorization of T^TT is carried out by converting it to a symmetric Cauchy-like matrix to obtain the upper triangular factor R, and the rows of R are used to update the matrix T^T (just as it is done in the modified Gram-Schmidt algorithm) to obtain the corresponding columns of the orthogonal factor Q.

Let us consider a displacement matrix Z of the form shown in (6.40), where Z_{00} is of size n and Z_{11} is of size m. The displacement equation can then be written as

$$\begin{bmatrix} Z_{00} & 0 \\ 0 & Z_{11} \end{bmatrix} \begin{bmatrix} T^TT & T^T \\ T & 0 \end{bmatrix} - \begin{bmatrix} T^TT & T^T \\ T & 0 \end{bmatrix} \begin{bmatrix} Z_{00} & 0 \\ 0 & Z_{11} \end{bmatrix}$$

$$= \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \Sigma \begin{bmatrix} G_1^T & G_2^T \end{bmatrix}. \tag{6.16}$$

The displacement rank of A with respect to (Z, Z) is 8. Since we know that the sine-I transform, S_{00}, diagonalizes Z_{00} and that the cosine-II transform, S_{11}, diagonalizes Z_{11}, we have

$$\Lambda = \begin{bmatrix} \Lambda_{00} & 0 \\ 0 & \Lambda_{11} \end{bmatrix} = \begin{bmatrix} S_{00} & 0 \\ 0 & S_{11}^T \end{bmatrix} \begin{bmatrix} Z_{00} & 0 \\ 0 & Z_{11} \end{bmatrix} \begin{bmatrix} S_{00} & 0 \\ 0 & S_{11} \end{bmatrix} \tag{6.17}$$

and

$$\begin{aligned} \Lambda \begin{bmatrix} S_{00}T^TTS_{00} & S_{00}T^TS_{11} \\ S_{11}^TTS_{00} & 0 \end{bmatrix} &- \begin{bmatrix} S_{00}T^TTS_{00} & S_{00}T^TS_{11} \\ S_{11}^TTS_{00} & 0 \end{bmatrix} \Lambda \\ &= \begin{bmatrix} S_{00}G_1 \\ S_{11}^TG_2 \end{bmatrix} \Sigma \begin{bmatrix} G_1^TS_{00} & G_2^TS_{11} \end{bmatrix}. \end{aligned} \tag{6.18}$$

Let us also assume that the eigenvalues of Z_{00} (Λ_{00}) and Z_{11} (Λ_{11}) are distinct. We later discuss the case when they have some identical eigenvalues. Let us consider the first n rows of the Cauchy-like matrix.

$$\begin{aligned} \Lambda_{00} \begin{bmatrix} S_{00}T^TTS_{00} & S_{00}T^TS_{11} \end{bmatrix} &- \begin{bmatrix} S_{00}T^TTS_{00} & S_{00}T^TS_{11} \end{bmatrix} \begin{bmatrix} \Lambda_{00} & 0 \\ 0 & \Lambda_{11} \end{bmatrix} \\ &= S_{00}G_1\Sigma \begin{bmatrix} G_1^TS_{00} & G_2^TS_{11} \end{bmatrix} \end{aligned} \tag{6.19}$$

The Cauchy-like matrix $S_{00}T^TTS_{00}$, denoted by C, is positive semidefinite because the matrix T^TT is positive semidefinite. The rows of the upper triangular factor R are obtained from a pivoted factorization of C. Since it is a positive semidefinite matrix, a diagonal pivoting strategy is sufficient. We now outline the rank revealing QR factorization algorithm. The first step in this algorithm is to search for a pivot element in the diagonal of C. The diagonal elements of this matrix would have to be computed from the Lyapunov equation

$$\Lambda_{00}C - C\Lambda_{00} = S_{00}G_1\Sigma G_1^TS_{00}. \tag{6.20}$$

Since the displacement matrices Λ_{00} are the same, the diagonal elements cannot be computed using the above Lyapunov equation. This means that the diagonal elements of C will have to be computed *a priori* just as they were in the algorithm presented in Section 5.5. In Section 5.5, we showed that if the Cauchy-like matrix C were obtained from a Toeplitz matrix, then the diagonal of C can be computed in $O(n \log(n))$ flops. If, however, the matrix C is obtained from a general matrix, the computation of the diagonal elements of C takes $O(n^2 + n\log(n))$ flops. We now present such an algorithm.

In Section 5.5, it was shown that the eigenvalues of the minimizer $\mathcal{S}$ among all matrices in the vector space $\mathbf{S}$ of all matrices diagonalizable by the sine-I transform are the diagonal elements of C. If we compute the vector r (5.79) for the matrix T^TT, then using Corollary 5.1 we can compute the first column of the minimizer $\mathcal{S}$. The eigenvalues of the minimizer $\mathcal{S}$ are then computed using (5.80).

We now show how the vector r may be computed in $O(n^2)$ flops. Consider the displacement equation of T^TT with respect to the displacement matrix Z_1 (5.10)

$$\Delta T^TT = T^TT - Z_1T^TTZ_1^T = H\Sigma_1H^T, \tag{6.21}$$

where H is of rank 4 and Σ_1 is a symmetric matrix. This displacement equation can be computed in $O((m+n)\log(m+n))$ flops. Now, from the above displacement equation it follows that

$$T^TT = \sum_{i=0}^{n-1} Z_1^i(\Delta T^TT)(Z_1^T)^i. \tag{6.22}$$

The k-th diagonal of ΔT^TT, denoted by b_k, can be computed from the displacement equation (6.21) in $7(n-k+1)$ flops. An additional $(n-k)$ flops are needed to compute the k-th diagonal of T^TT from b_k using (6.22). Let us denote this by d_k. Since there are n distinct diagonals in T^TT (symmetry), we need approximately $4n^2$ flops to compute all diagonals of T^TT. The contribution of each diagonal to the vector r is then calculated using the following algorithm.

Algorithm 6.1

```
for $i = 1 : n$
    $\text{dsum}_i = \sum_{j=1}^{n-i+1} d_i(j)$
    $m = \lfloor \frac{n-i}{2} \rfloor$
    $k = n - i + 1$
    for $j = 1 : m$
        $d_i(j) = d_i(j) + d_i(k - j + 1)$
    end
    if   $(i = 1)$
        $r_1 = \mathrm{dsum}_1$
        for $l = 1 : m$
            $j = i + 2l$
            $r_j = r_j - b_i(l)$
        end
    else
        $r_i = r_i + 2\mathrm{dsum}_i$
        if   $(i <= (n - 2))$
            for $l = 1 : m$
                $j = i + 2l$
                $r_j = r_j - 2b_i(l)$
            end
        end
    end
end
```

The total number of flops to compute the contribution of all diagonals $d_1, \cdots, d_n$ of $T^T T$ to the vector r is approximately equal to $1.25n^2$ flops. Hence, the total flops to compute the diagonal of C using the above algorithm is

$$\text{Flops} = O((m + n) \log (m + n)) + 5.25n^2. \tag{6.23}$$

Having computed the diagonal elements of C *a priori*, we proceed with the rank revealing QR factorization algorithm. The first step is to search for a pivot element in the diagonal of C. Let P_1 be the permutation matrix that brings the pivot element into place. This matrix is applied to the Cauchy-like matrix in the following manner.

$$P_1 \begin{bmatrix} S_{00}T^TTS_{00} & S_{00}T^TS_{11} \end{bmatrix} \begin{bmatrix} P_1 & 0 \\ 0 & I \end{bmatrix} \tag{6.24}$$

The first row of the permuted matrix is used to obtain the first row of R in the QR factorization and the first row of Q^T. Let the first row of the permuted Cauchy-like matrix be $[u_1\ u_2]$ and let the first element of u_1 be d_1. The first row of the upper triangular factor R, denoted by r_1, is then given by $r_1 = u_1/\sqrt{d_1}$ and the first column of Q, denoted by q_1, is given by $q_1^T = u_2/\sqrt{d_1}$. The next step is to obtain a low rank displacement equation for the Schur complement. The generator of the Schur complement is found by taking the last $(n+m-1)$ rows of the product

$$\begin{bmatrix} 1 & 0 \\ -l & I \end{bmatrix} S_{00}G_1, \tag{6.25}$$

where $[1 \quad l^T] = [u_1 d_1^{-1} \quad u_2 d_1^{-1}]$. Having computed the generator for the next step of the algorithm, the diagonal of C is updated to compute the diagonal of the Schur complement of C as

$$\mathrm{diag}(C) - \begin{bmatrix} 1 \\ -l \end{bmatrix} d_1 \begin{bmatrix} 1 & -l^T \end{bmatrix}. \tag{6.26}$$

The same procedure is repeated to compute the second row of R and the second column of Q. After n steps the rank revealing QR factorization is obtained as

$$\begin{aligned} PS_{00}T^TS_{11} &= R^TQ^T \\ S_{11}TS_{00} &= QRP. \end{aligned} \tag{6.27}$$

If the matrix T is rank-deficient, then the factorization algorithm is stopped when we encounter a near zero pivot element.

We now study the numerical accuracy of this rank revelation process by considering the Toeplitz matrix T discussed earlier in this section. The pivot element after the start of the sixth step is 2.01×10^{-12}, which is much smaller than the pivot element at the start of the fifth step, 0.37669. Hence, the algorithm is stopped after five steps. To assess the numerical behavior of this algorithm we compute the following errors :

$$\begin{aligned} ||I - Q^TQ|| &= 1.2266 \times 10^{-12} \\ ||S_{11}TS_{00} - QRP|| &= 1.1657 \times 10^{-6} \\ ||S_{00}T^TTS_{00} - P^TR^TRP|| &= 2.5574 \times 10^{-12}. \end{aligned} \tag{6.28}$$

Comparing (6.14) and (6.28), we see that the rank revealing QR factorization algorithm significantly improves the orthogonality of Q and the accuracy of the upper triangular factor. The error $||S_{11}TS_{00} - QRP||$ is of the same order as the sixth singular value of T (1.06×10^{-6}). Note that since the matrix T is squared in the algorithm, this rank revelation algorithm fails to detect the rank accurately if the matrix T has singular values less than $\sqrt{\epsilon}$ where ϵ is the machine precision. The Householder transformation based rank revealing QR factorization algorithm does not use the matrix T^TT and, hence, yields a better estimate of the rank profile of the matrix T.

In the discussion of the above algorithm we assume that Z_{00} and Z_{11} do not have any common eigenvalues. If Z_{00} and Z_{11} have some identical eigenvalues, then the corresponding elements of the Cauchy-like matrix $S_{00}T^TS_{11}$ have to be computed *a priori*. From (5.19) and (5.20), we see that if the numbers $(n+1)$ and m are relatively prime, then Λ_{00} and Λ_{11} will have no common eigenvalues and no elements of $S_{00}T^TS_{11}$ will be needed *a priori*. In (6.16), we have chosen Z_{00} to be the $(1,1)$ block in the displacement matrix Z and Z_{11} to be the $(2,2)$ block. This results in the condition that $gcd(n+1,m) = 1$ for no elements of $S_{00}T^TS_{11}$ to be computed *a priori*. Since we can choose between Z_{00} and Z_{11} for the two diagonal blocks of Z, we have four choices for the displacement matrix. The corresponding condition that has to be satisfied so that no elements of the Cauchy-like matrix have to be computed *a priori* are shown below.

$$\begin{bmatrix} S_{00} & \\ & S_{00} \end{bmatrix} \Rightarrow \text{gcd}\,(n+1, m+1) = 1$$

$$\begin{bmatrix} S_{00} & \\ & S_{11} \end{bmatrix} \Rightarrow \text{gcd}\,(n+1, m) = 1$$

$$\begin{bmatrix} S_{11} & \\ & S_{00} \end{bmatrix} \Rightarrow \gcd(n, m+1) = 1$$

$$\begin{bmatrix} S_{11} & \\ & S_{11} \end{bmatrix} \Rightarrow \gcd(n, m) = 1. \tag{6.29}$$

If none of the four conditions in (6.29) are satisfied, then one can choose the condition with the smallest GCD. This is done because the number of off-diagonal elements one has to compute *a priori* is equal to the GCD of the two numbers minus 1. For example, if $m = 35$ and $n = 14$, we would choose the $(1,1)$ block of Z to be Z_{11} and the $(2,2)$ block to be Z_{00} since $\gcd(n, m+1) = 2$, which is the least. In this case, the (i,j)-th $(i = 8, j = 18)$ element of $S_{11}T^TS_{00}$ will have to be computed *a priori*. If the least GCD among the four choices is small, then the amount of computation to obtain the off-diagonal elements is on the order of $O((m+n)\log(m+n))$ flops.

For a Toeplitz matrix of rank r, the complexity of the rank revealing algorithm (considering only the highest order terms) is

$$\begin{aligned} \text{Flops} &\approx 5.25n^2 + \sum_{k=1} r\{(4\alpha+4)(n-k) + (4\alpha+1)m\} \\ &\approx 5.25n^2 + 33mr + 36nr - 18r^2. \end{aligned} \tag{6.30}$$

If the matrix T has full column rank, then the complexity is $23.25n^2 + 33mn$.

Figure 6.1 shows the performance of the routine to compute a rank revealing QR factorization of a real Toeplitz matrix of size 4096×2024. The two plots are for the Cray J90 (solid line) and the Cray YMP (dashed line). The amount of work to compute the triangular factor R reduces linearly with each step of the factorization. The work to compute the orthogonal factor Q, however, remains constant. The parallel performance of this routine on the Cray PVP systems is, therefore, slightly better than that of the factorization of the normal equations. Table 6.1 lists the performance of this routine on a two-processor Cray T90.

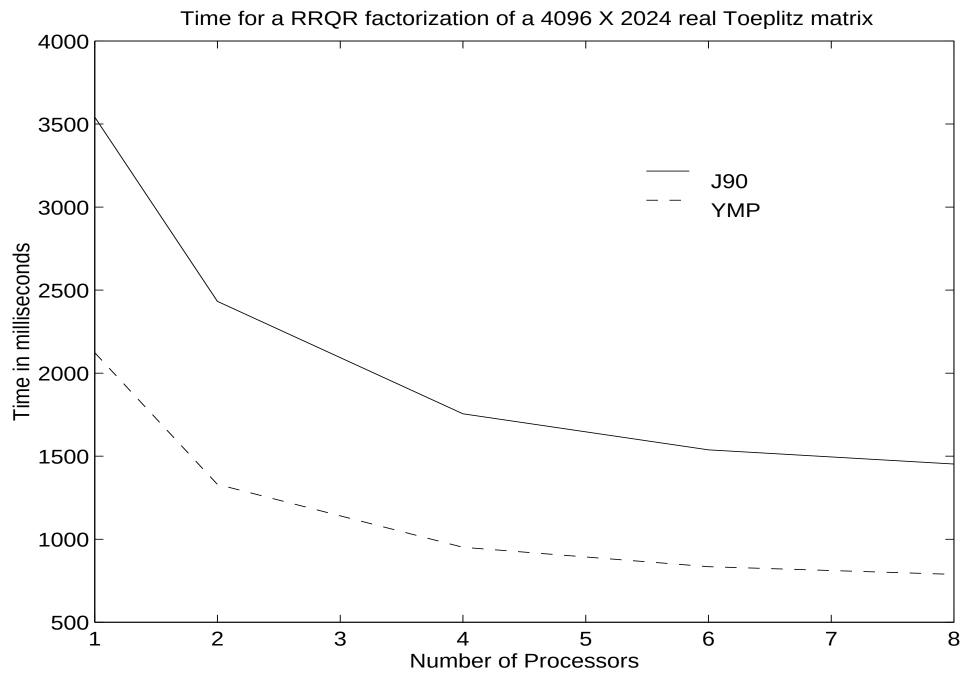


Figure 6.1 Time in milliseconds to obtain a RRQR factorization of a 4096×2024 real Toeplitz matrix on the Cray J90 and Cray YMP.

Table 6.1 Time in milliseconds to obtain a RRQR factorization of a 4096×2047 real Toeplitz matrix on a Cray T90.

Number of CPUS	Time for RRQR on T90 (in ms)
1	491
2	384

6.4 Augmented System Method for Solving Least Squares Problems

In the previous sections, we discussed the generalized Schur algorithm and several variants to obtain the QR factorization of a rank deficient Toeplitz matrix. In the case of near column rank deficiency we suggested the use of a fast rank revealing QR factorization algorithm. However, the rank revealing QR factorization algorithm discussed in the previous section computes the upper triangular factor R by factoring the normal equation matrix $T^T T$. If the matrix T is ill-conditioned, then it was seen (in the example discussed in the previous chapter) that the rank profile of the matrix T cannot be accurately determined. This in turn limits the accuracy of the least squares problem because the singular values of T that are less than the square root of the machine precision are lost in the process of squaring the matrix T. All fast Toeplitz least squares algorithms, whether Levinson- or Schur-based, suffer from this problem. One way to avoid this problem is to use the augmented system of equations [65]. For general matrices, solving the augmented system of equations is prohibitively expensive. For Toeplitz matrices, this complexity can be significantly reduced if the Toeplitz structure is exploited. In this section, we show how the augmented system method for solving least squares problems can be adapted to obtain a fast Toeplitz least squares algorithm that does not suffer from the problem of squaring the condition number. We also illustrate by way of an example how this method may yield superior numerical solutions to Toeplitz least squares problems.

Let us consider a Toeplitz least squares problem of the form

$$min \; \|Tx - b\|_2, \tag{6.31}$$

where T is of size $m \times n$. The augmented system formulation of this least squares problem [65] can be written as

$$\begin{bmatrix} \alpha I_{m\times m} & T \\ T^T & 0_{n\times n} \end{bmatrix} \begin{bmatrix} \alpha^{-1} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0_{n\times 1} \end{bmatrix}, \tag{6.32}$$

where r is the residual vector defined as $r = b - Tx$ and α is a parameter that must be chosen to induce appropriate pivoting. We will refer to the augmented matrix in the above equation as A_α.

The augmented matrix A_α in the above equation is symmetric and indefinite. The Bunch-Kaufman pivoting strategy may be used to solve the system of equations. If the value of α is chosen to be sufficiently large, then the first m pivots are α and the Schur complement at the end of m steps is $-\alpha^{-1}T^T T$. This is then equivalent to solving the normal equations. If the value of α is chosen to be sufficiently small, so as to pick either 1×1 or 2×2 pivots using the Bunch-Kaufman pivoting strategy, then the accuracy of the solution does not depend on the square of the condition number of T. If we define

$$z_\alpha = \begin{bmatrix} \alpha^{-1} r \\ x \end{bmatrix}, \tag{6.33}$$

then since the Bunch-Kaufman algorithm is backward stable for general symmetric systems, the forward error can be bounded by

$$||\tilde{z}_\alpha - z_\alpha||_2 \leq \frac{\epsilon\kappa_2(A_\alpha)}{1-\epsilon\kappa_2(A_\alpha)}||z_\alpha||_2. \tag{6.34}$$

In [65, 66], Björck showed that an optimal α is one that minimizes the forward error. The forward error is minimized when the condition number of A_α is minimized. If the smallest singular value of T is σ_n, then it can be shown that [65, 66] an α equal to $\sigma_n/\sqrt{2}$ minimizes the condition number of A_α and therefore is the optimal value.

Unfortunately, in most problems one may not have an estimate of the smallest singular value of T. Our empirical studies of the augmented system method indicate that, in practice, for values of α between the smallest singular value of T and machine precision ϵ, the Bunch-Kaufman pivoting strategy produces a forward error in the solution vector x (not z_α) that varies very little. We offer an explanation for this using the SVD of T. Let the SVD of T be

$$T = U\Sigma V^T, \tag{6.35}$$

and let the singular values contained in Σ be $\sigma_1, \sigma_2, \cdots, \sigma_n$. The augmented system can now be transformed as

$$\begin{aligned} \begin{bmatrix} U^T & \\ & V^T \end{bmatrix} \begin{bmatrix} \alpha I & T \\ T^T & 0 \end{bmatrix} \begin{bmatrix} U & \\ & V \end{bmatrix} \begin{bmatrix} U^T & \\ & V^T \end{bmatrix} \begin{bmatrix} \alpha^{-1} \\ x \end{bmatrix} &= \begin{bmatrix} U^T & \\ & V^T \end{bmatrix} \begin{bmatrix} b \\ 0 \end{bmatrix} \\ \begin{bmatrix} \alpha I & \Sigma \\ \Sigma^T & 0 \end{bmatrix} \hat{z}_\alpha &= \hat{b}. \end{aligned} \tag{6.36}$$

A perfect shuffle permutation P applied to this equation transforms it to

$$\begin{bmatrix} \begin{bmatrix} \alpha & \sigma_1 \\ \sigma_1 & 0 \end{bmatrix} & & & & & & \\ & \ddots & & & & & \\ & & \ddots & & & & \\ & & & \begin{bmatrix} \alpha & \sigma_n \\ \sigma_n & 0 \end{bmatrix} & & & \\ & & & & \alpha & & \\ & & & & & \ddots & \\ & & & & & & \ddots \\ & & & & & & & \alpha \end{bmatrix} P\hat{z}_\alpha = P\hat{b}. \qquad (6.37)$$

This system of equations can be solved to obtain $P\hat{z}_\alpha$. The solution vector x is then obtained by undoing the perfect shuffle permutation and applying the orthogonal transformation V. It can very easily be seen that the accuracy of the elements of x depends on the condition number of the block diagonal matrix consisting of the first n 2×2 diagonal blocks, whereas that of the residual $\alpha^{-1}r$ depends on the condition number of the entire matrix A_α. The condition number of the block diagonal matrix comprising the first n 2×2 blocks is

$$\frac{|\alpha + \sqrt{\alpha^2 + 4\sigma_1^2}|}{|\alpha - \sqrt{\alpha^2 + 4\sigma_n^2}|}.$$

For any $\alpha \le \sigma_n$, this condition number is approximately equal to the condition number of T. Hence, we can conclude that for any $\alpha \le \sigma_n$, the relative forward error in x is approximately equal to that obtained from a backward stable algorithm. The relative forward error for the residual $\alpha^{-1}r$, however, depends on the condition number of the entire matrix A_α and for very small α this residual can be very inaccurate. The optimal value of α for the smallest error in the residual is $\alpha = \sigma_n/\sqrt{2}$, as shown in [65, 66].

In practice, however, when the Bunch-Kaufman algorithm is used as the pivoting strategy to solve the augmented system, we find that choosing $\alpha \le \sigma_n$ ensures a suitable choice of pivots (i.e., about n 2×2 pivots and $m - n$ 1×1 pivots). In such cases, the relative forward error in x is comparable to the case when $\alpha = \sigma_n/\sqrt{2}$. It can, therefore, be concluded that the only significant effect α has is to ensure a suitable sequence of pivots in the factorization algorithm.

We illustrate this through an example. Consider the following Toeplitz matrix T.

$$T = \begin{bmatrix} 5 & 4 & 3.001 & 2 & 1 & 1 & 2 & 3 \\ 6 & 5 & 4 & 3.001 & 2 & 1 & 1 & 2 \\ 7 & 6 & 5 & 4 & 3.001 & 2 & 1 & 1 \\ 8 & 7 & 6 & 5 & 4 & 3.001 & 2 & 1 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3.001 & 2 \\ 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3.001 \\ 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 \\ 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 \\ 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 \\ 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 \\ 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 \end{bmatrix} \tag{6.38}$$

The singular values of this matrix are

$$S.V. = \begin{bmatrix} 70.88227257736855 \\ 6.26334054820666 \\ 3.17683456259507 \\ 0.62966200031027 \\ 0.14578536299351 \\ 0.00000503553004 \\ 0.00000060350491 \\ 0.00000001797212 \end{bmatrix}. \tag{6.39}$$

Though the effect of α on the accuracy of x can be illustrated using any arbitrary matrix, we use this example because the generalized Schur algorithm (factoring a small displacement rank matrix) breaks down due to the singularity of $T^T T$. Let the solution vector x be $[1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]^T$. The exact right-hand side vector b is then

$$[21.001\ 24.001\ 29.001\ 36.001\ 44.001\ 52.001\ 60\ 68\ 76\ 84\ 92]^T.$$

Choosing the optimal value of α as derived in [65, 66] ($\alpha = 0.00000001797212/\sqrt{2}$) yields a forward relative error of 5.7226×10^{-08} in x. If we choose $\alpha \ll \sigma_n$, e.g., $\alpha = 1 \times 10^{-10}$, we obtain a relative forward error of 6.5977×10^{-08} in x. By further reducing α, we see that $\alpha = 1 \times 10^{-14}$ also yields a relative forward error of 6.5977×10^{-08}. This shows that the relative accuracy of the solution vector x remains unchanged for $\alpha \leq \sigma_n$.

Note that α cannot be set to zero because the augmented system of equations becomes singular and cannot be factored. In general, for solving least squares problems involving ill-conditioned coefficient matrices, one can choose $\alpha \approx \epsilon$ (machine precision) to ensure that it is less than the the smallest singular value of the coefficient matrix.

For least squares problems with condition numbers on the order of $1/\sqrt{\epsilon}$ or more, the generalized Schur algorithm breaks due to the singularity of $T^T T$. The augmented systems method discussed above may be used to solve such problems because it does not square the condition number of the problem. The sensitivity of the least squares problem, however, remains unchanged.

If the Toeplitz structure of T is ignored, then the complexity of computing an LDL^T factorization of A_α using the Bunch-Kaufman algorithm is $O((m+n)^3)$. However, this can be significantly reduced if the Toeplitz structure is exploited. We now present a fast algorithm to factor the augmented system of equations. Let us consider a displacement matrix Z of the form

$$Z = \begin{bmatrix} Z_{00} & \\ & Z_{11} \end{bmatrix}, \tag{6.40}$$

where Z_{00} is of size m and Z_{11} is of size n. The augmented system matrix A_α shown in (6.32) is a real symmetric quasi-Toeplitz matrix with a displacement rank of 8 with respect to the displacement matrix Z.

$$ZA_\alpha - A_\alpha Z = G\Sigma G^T \tag{6.41}$$

The corresponding displacement equation for the Cauchy-like matrix is

$$\begin{bmatrix} \Lambda_{00} & \\ & \Lambda_{11} \end{bmatrix} \begin{bmatrix} \alpha I & S_{00} T S_{11} \\ S_{11}^T T S_{00} & 0 \end{bmatrix} - \begin{bmatrix} \alpha I & S_{00} T S_{11} \\ S_{11}^T T S_{00} & 0 \end{bmatrix} \begin{bmatrix} \Lambda_{00} & \\ & \Lambda_{11} \end{bmatrix}$$

$$= \begin{bmatrix} S_{00} & \\ & S_{11}^T \end{bmatrix} G \Sigma G^T \begin{bmatrix} S_{00} & \\ & S_{11} \end{bmatrix}. \qquad (6.42)$$

Since the Cauchy-like matrix is symmetric, we have to compute the diagonals *a priori*. From the above equation, it can be seen that he diagonal elements are obtained trivially. Further, if Λ_{00} and Λ_{11} have common eigenvalues, then the corresponding elements of $S_{00}TS_{11}$ will have to be computed *a priori*. From (5.19) and (5.20), we see that if the numbers $(m+1)$ and n are relatively prime, then Λ_{00} and Λ_{11} will have no common eigenvalues and no elements of $S_{00}TS_{11}$ will be needed *a priori*. In the above example, we have chosen Z_{00} to be the $(1,1)$ block in the displacement matrix Z and Z_{11} to be the $(2,2)$ block. This results in the condition that $gcd(m+1,n) = 1$ for no elements of $S_{00}TS_{11}$ to be computed *a priori*. Since we can choose between Z_{00} and Z_{11} for the two diagonal blocks of Z, we have four choices for the displacement matrix. The corresponding condition that has to be satisfied so that no off-diagonal elements of the Cauchy-like matrix have to be computed *a priori* are shown in (6.29).

If none of the conditions in (6.29) can be satisfied, we suggest the use of Z_{11} or Z_{00} instead of Z. The displacement rank of A_α with respect to either Z_{00} or Z_{11} is still 8. However, we require some extra computation in order to compute the diagonal elements of the Cauchy-like matrix. The complexity of doing this is $O((m+n)\log(m+n))$ and, hence, can be ignored in the total complexity of the algorithm.

In Section 5.5, we discussed a method for factoring real symmetric Toeplitz matrices. That method can be extended to the augmented system method by using the displacement equation shown above to convert A_α to a Cauchy-like matrix. The complexity of the factorization algorithm is between $18(m+n)^2$ and $26.5(m+n)^2$.

Though this method for solving ill-conditioned Toeplitz least squares problems may be numerically more accurate than the generalized Schur algorithm, one obvious disadvantage of this method is the increased complexity of the method (now $O((m+n)^2)$ as opposed to $O(n^2)$). While this is the case, one could use this method if $m \approx n$. As a final note, it must be mentioned that all algorithms discussed in this chapter that are based on the conversion to Cauchy-like matrices can be extended to block Toeplitz matrices following the principles discussed earlier in Section 5.6.

CHAPTER 7

ITERATIVE DECONVOLUTION OF IMAGES USING THE SCHUR ALGORITHM

In earlier chapters, we have discussed several algorithms to factor various Toeplitz matrices and solve Toeplitz least squares problems. Toeplitz matrices arise in several problems in signal and image processing. In this chapter we study one such application. Consider the problem of restoring images that have been blurred by a space-invariant point spread function (PSF). The blurring operator can be expressed as a Toeplitz matrix (or block Toeplitz with Toeplitz blocks in two-dimensional signal restoration); the problem of deblurring is a Toeplitz least squares problem. In the presence of noise, a regularization scheme such as Tikhonov regularization is commonly employed. This problem is well-studied and in recent years several preconditioners have been proposed to solve the problem iteratively. We study the applicability of direct methods based on the Schur algorithm in solving the deconvolution problems. In cases where direct methods prove to be very expensive, we suggest the use of preconditioners based on the direct methods. We show that these methods can be well-implemented to scale on distributed memory multiprocessors such as the Cray T3D; we also present some examples of deconvolution with typical PSFs.

7.1 Introduction

Consider a one-dimensional signal a consisting of n samples, $a_1, \cdots, a_n$, which has been blurred by a linear space-invariant point spread function (PSF) h consisting of m samples to yield a blurred signal b. The convolution operation that results in the blurred signal b can be expressed in matrix terms as

$$Ha = \begin{bmatrix} h_1 & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & & & \vdots \\ h_m & \ddots & \ddots & & 0 \\ 0 & \ddots & \ddots & \ddots & h_1 \\ \vdots & & & & \vdots \\ 0 & \cdots & \cdots & 0 & h_m \end{bmatrix} a = b. \tag{7.1}$$

In the above equation, the size of the matrix H is $(n + m - 1) \times n$ and the size of the blurred signal b is $n - m + 1$. The problem of deconvolution is the inverse problem, and the "original" signal a is to be determined in the least squares sense. The matrix H is usually ill-conditioned $(\sigma_1 \gg \sigma_n)$. In addition, several sources of noise, such as sensor noise, atmospheric fluctuations, film grain irregularities, and quantization noise, cause errors in the data acquisition process. This noise is usually modeled as additive Gaussian or Poisson noise.

If the signal b is noisy, then standard regularization schemes such as Tikhonov regularization and iterative regularization [2] are usually employed in the process of deconvolution. In this chapter, we use the Tikhonov regularization approach. In this particular problem, the regularization term is incorporated as a physical constraint on the signal such as smoothness of the signal.

The least squares formulation of the 1D deconvolution problem with Tikhonov regularization is

$$\min \left\| \begin{pmatrix} H \\ \sqrt{\mu} L \end{pmatrix} a - \begin{pmatrix} b \\ 0 \end{pmatrix} \right\|, \tag{7.2}$$

where L is any difference operator and $\sqrt{\mu}$ is the regularization factor. In most practical problems, L is chosen to be either the identity matrix I or the first-order difference operator

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -1 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & -1 \end{bmatrix}. \tag{7.3}$$

Before the least squares problem is solved, one has to determine the regularization factor. This in itself is a tough albeit well-studied problem; a survey of existing algorithms can be found in [2]. In our examples, we choose $\sqrt{\mu}$ based on trial and error. Intuitively however, it is clear that if the image is very noisy, then one would have to choose a large value for μ to enforce smoothness of the image and thereby reduce noise to a greater extent. If the image is not too noisy, we could choose a small value for μ. If, however, the matrix H is very ill-conditioned, then a large value of μ may be used to improve the numerical accuracy of the computation.

The normal equation for this least squares problem is given by

$$\left(H^T H + \mu L^T L\right) a = H^T b. \tag{7.4}$$

The coefficient matrix in the above equation is a banded Toeplitz matrix with a half-bandwidth of m. Fast direct methods that exploit the structure of the coefficient matrix (Toeplitz) include the Levinson and Schur algorithms. The Levinson algorithm factors the inverse of the Toeplitz matrix and, hence, cannot exploit the band structure, whereas the Schur algorithm factors the Toeplitz matrix itself and hence exploits the bandedness as well. The Schur algorithm can, therefore, be used to factor the coefficient Toeplitz matrix, and the "deblurred" signal a can be obtained after a forward and back-solve.

For a banded Toeplitz matrix of size n with a half-bandwidth of m, the complexity of the Schur algorithm is approximately $4mn$. If the matrix is dense, then the complexity of the Schur algorithm is $2n^2$. Recently, the stability of the Schur algorithm has been studied by Bojanczyk, Brent, de Hoog, and Sweet [67], and by Stewart and Van Dooren [56]. They show that when stabilized hyperbolic transformations are used, we have

$$T = \tilde{R}^T \tilde{R} + \Delta T \text{ where } ||\Delta T|| \leq \epsilon t_0 n^2. \tag{7.5}$$

When ordinary hyperbolic downdating is used, they prove that

$$||\Delta T|| \leq \epsilon t_0 n^3. \tag{7.6}$$

This indicates that the Schur algorithm is backward stable.

In the following sections we consider deconvolving images (2D signals) blurred with separable and non-separable PSFs. Section 7.2 shows how the Schur algorithm may be used to deconvolve 2D signals (images) that have been blurred with separable linear space-invariant PSFs. Section 7.3 presents two preconditioners to solve the problem of deconvolution with non-separable PSFs iteratively. Section 7.4 presents some results on astronomical images. Section 7.5 compares the preconditioners described in Section 7.3 to other commonly used preconditioners. Section 7.6 discusses the implementation of the 2D deconvolution algorithm on distributed memory multiprocessors and compares the complexity of our preconditioning schemes to other preconditioners based on block circulant matrices.

7.2 Deconvolution with a Separable 2D PSF

Consider a two-dimensional signal (an image) A of size $n_1 \times n_2$ that has been blurred by a two-dimensional PSF h of size $m_1 \times m_2$ to yield a blurred image of size $(n_1+m_1-1) \times (n_2+m_2-1)$. Let the PSF h be

$$h = \begin{bmatrix} h_{11} & \cdots & \cdots & h_{1m_2} \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ h_{m_1 1} & \cdots & \cdots & h_{m_1 m_2} \end{bmatrix}. \tag{7.7}$$

Further, let the image A be converted to a 1D vector a by stacking the columns of the image one over the other. Let b be a 1D vector derived similarly from B. Then the two-dimensional convolution problem is expressed in matrix terms as

$$Ha = \begin{bmatrix} H_1 & & & \\ \vdots & \ddots & & \\ H_{m_2} & & \ddots & \\ & \ddots & & H_1 \\ & & \ddots & \vdots \\ & & & H_{m_2} \end{bmatrix} a = b, \tag{7.8}$$

where

$$H_1 = \begin{bmatrix} h_{11} & & \\ \vdots & \ddots & \\ h_{m_1 1} & & \ddots & \\ & \ddots & & h_{11} \\ & & \ddots & \vdots \\ & & & h_{m_1 1} \end{bmatrix} \cdots \; H_{m_2} = \begin{bmatrix} h_{1m_2} & & \\ \vdots & \ddots & \\ h_{m_1 m_2} & & \ddots & \\ & \ddots & & h_{1m_2} \\ & & \ddots & \vdots \\ & & & h_{m_1 m_2} \end{bmatrix}. \tag{7.9}$$

It can be seen that H is a rectangular block Toeplitz matrix with rectangular Toeplitz blocks. The size of each block $H_1 \cdots H_{m_2}$ is $(n_1 + m_1 - 1) \times n_1$. The size of the Toeplitz matrix H is $(n_1 + m_1 - 1)(n_2 + m_2 - 1) \times n_1 n_2$.

The problem of deconvolution however, is the inverse problem and has to be solved in the least-squares sense using a regularization scheme. If we use Tikhonov regularization, then

$$\left(H^T H + \mu L^T L\right) a = H^T b, \tag{7.10}$$

where $L^T L$ could either be an identity matrix or any difference operator such as the 2D Laplacian.

The coefficient matrix $(H^T H + \mu L^T L)$ in the above equation is now a banded block Toeplitz matrix with banded Toeplitz blocks. This is a Toeplitz matrix with two levels of "Toeplitzness." Block generalizations of the Schur algorithm exist [23], but they exploit only one level of Toeplitzness. If we use a block Schur algorithm to solve the 2D deconvolution problem, the complexity of the algorithm is very high $(O(m_2 n_1^3 n_2))$. However, if the P.S.F matrix h were of rank 1, then the complexity could be drastically reduced. In signal processing literature this rank-1 condition on the 2D PSF h is usually referred to as separability of the two dimensions as we shall demonstrate.

Let h be a rank-1 matrix whose factorization is given as

$$h = \begin{bmatrix} h_{11} & \cdots & \cdots & h_{1m_2} \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ h_{m_1 1} & \cdots & \cdots & h_{m_1 m_2} \end{bmatrix} = \begin{bmatrix} g_1 \\ \vdots \\ \vdots \\ g_{m_1} \end{bmatrix} \begin{bmatrix} f_1 & \cdots & \cdots & f_{m_2} \end{bmatrix}. \tag{7.11}$$

The matrix H is now simplified to a great extent.

$$H = \begin{bmatrix} f_1 G & & & \\ \vdots & \ddots & & \\ f_{m_2} G & & \ddots & \\ & \ddots & & f_1 G \\ & & \ddots & \vdots \\ & & & f_{m_2} G \end{bmatrix} = F \otimes G, \tag{7.12}$$

where

$$F = \begin{bmatrix} f_1 & & & \\ \vdots & \ddots & & \\ f_{m_2} & & \ddots & \\ & \ddots & & f_1 \\ & & \ddots & \vdots \\ & & & f_{m_2} \end{bmatrix} \quad \text{and} \quad G = \begin{bmatrix} g_1 & & & \\ \vdots & \ddots & & \\ g_{m_1} & & \ddots & \\ & \ddots & & g_1 \\ & & \ddots & \vdots \\ & & & g_{m_1} \end{bmatrix}. \tag{7.13}$$

From (7.8) and (7.12), we get

$$H\ a = b \quad \Longleftrightarrow \quad G\ A\ F^T = B \tag{7.14}$$

where A is the original image and B is the blurred image. It can be seen from the above equation that the two dimensions of the image (rows and columns) have been separated. The deconvolution problem can now be broken into two subproblems: deconvolution of the columns followed by deconvolution of the rows (much like most separable transforms such as DFTs and DSTs). This is equivalent to solving the deconvolution problem of (7.10).The deconvolution of the columns is carried out by solving

$$\left(G^TG + \sqrt{\mu}L^TL\right)A_1 = G^TB \tag{7.15}$$

where A_1 is an intermediate matrix. The deconvolution of the rows now is

$$\left(F^TF + \sqrt{\mu}L^TL\right)A^T = F^TA_1^T. \tag{7.16}$$

In (7.15) and (7.16), L is either the identity matrix or any suitable difference operator. Since the coefficient matrices in (7.15) and (7.16) are banded Toeplitz matrices, the Schur algorithm could be used to obtain an inexpensive factorization. The total complexity of the deconvolution algorithm using this scheme is

$$O(n_1m_1) + O(n_2n_1m_1) + O(n_2m_2) + O(n_2n_1m_2) \approx O(n_2n_1(m_1 + m_2)). \tag{7.17}$$

If m_1 and m_2 are small compared to n_1 and n_2, then the complexity of the algorithm is almost linear in $n_1 \times n_2$.

In addition, parallelization of this algorithm for implementation on distributed memory machines such as the Cray T3D is fairly easy and can be done using standard and commonly available computational kernels. We present some results of experiments on the Cray T3D in a later section.

To demonstrate the numerical behavior of the Schur algorithm in this deconvolution scheme, we present some example images. Consider an example image shown in Figure 7.1 that is blurred with a separable 11×11 Gaussian filter of the form $h(i,j) = \exp\left(-0.1(i-6)^2\right)\exp\left(-0.1(j-6)^2\right)$. The blurred image is shown in Figure 7.2. The deconvolution scheme discussed in this section was used and the result is shown in Figure 7.3. It can be seen that most of the blurring has been removed but there are some disagreeable edge artifacts in the form of dark and white bands, which are caused by the abrupt truncation of the blurred image. There are several heuristics that one can use to eliminate these edge effects. Most of them rely on extending the blurred image in all directions by suitably padding it. The deconvolution scheme is then carried out on the padded image. We use a commonly employed heuristic (that we shall soon describe) to obtain the deblurred image shown in Figure 7.4. It must be mentioned that the

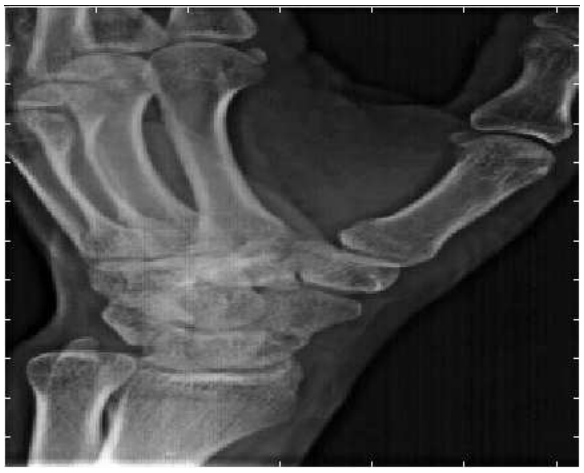


Figure 7.1 Original fractured wrist image.

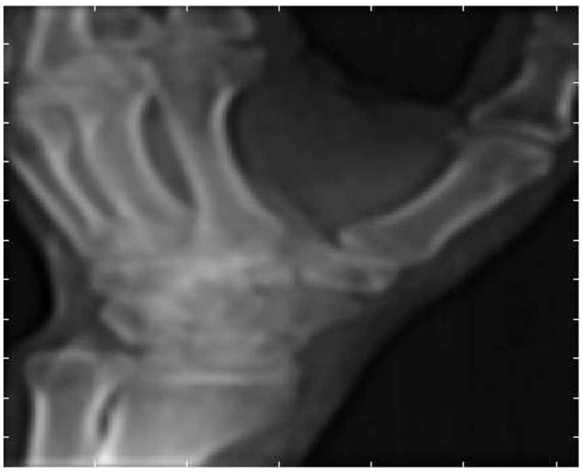


Figure 7.2 Blurred with a Gaussian filter.

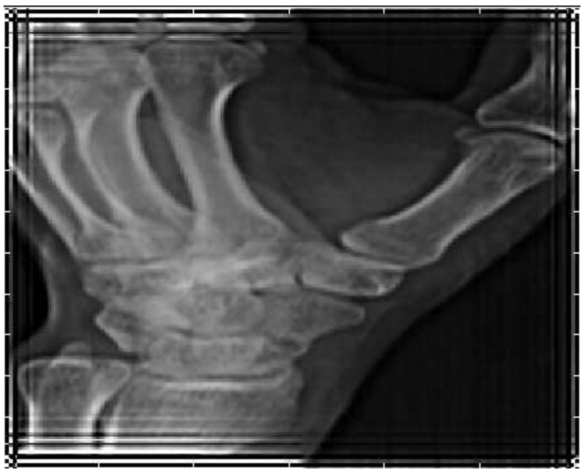


Figure 7.3 Restored without padding.

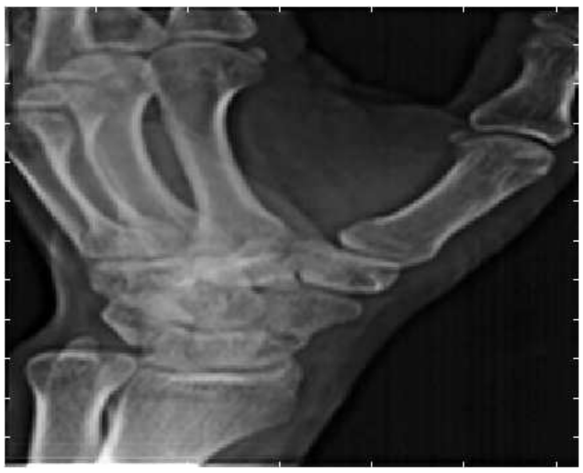


Figure 7.4 Restored with padding.

regularization factor in both cases was 0.001. A low value was used since there was no noise added to the image. A very low value of μ causes severe ringing in the deblurred image and hence was avoided.

We now outline the heuristic scheme commonly employed to reduce edge artifacts. Consider a 1D blurred signal of size n_1. Using the 1D deconvolution scheme discussed earlier with a P.S.F of length m_1, the deblurred signal is of length $n_1 - m_1 + 1$ and has the edge artifacts seen in Figure 7.3. If the blurred signal is padded in some way with $(m_1 - 1)/2$ samples on either side, then the padded blurred signal is of length $n_1 + m_1 - 1$ and the deblurred signal is of length n_1

(equal to the size of the blurred image we started out with). In addition, suitably padding the blurred image will reduce the edge artifacts significantly.

To determine what the padding around the blurred signal should be, we extend the original signal (that is blurred to produce the blurred signal given to us) by $(m_1 - 1)/2$ samples in either direction. The extended signal is assumed to be constant in that region with a value equal to the extremal samples of the original image. The corresponding samples of the blurred image are now computed using the definition of the convolution product. To illustrate this better, we consider an example where $n_1 = 7$ and $m_1 = 3$. Let the blurred signal given to us be b of length $n_1 = 7$ and the PSF be of length $m_1 = 3$. If we carried out the deconvolution scheme without the padding, let us assume that we get the deblurred signal a of length $n_1 - m_1 + 1 = 5$ in samples $a_1, \cdots, a_5$. Since we had to pad the blurred signal with the $(m_1 - 1)/2 = 1$ sample on either side, we must first pad the deblurred signal with one sample on either side. The padded samples are shown in (7.18) as being boxed.

$$\left[\begin{array}{c|ccccc|c} h_1 & & & & & & \\ h_2 & h_1 & & & & & \\ h_3 & h_2 & h_1 & & & & \\ & h_3 & h_2 & h_1 & & & \\ & & h_3 & h_2 & h_1 & & \\ & & & h_3 & h_2 & h_1 & \\ & & & & h_3 & h_2 & h_1 \\ & & & & & h_3 & h_2 \\ & & & & & & h_3 \end{array}\right] \left[\begin{array}{c} \boxed{a_0} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ \boxed{a_6} \end{array}\right] = \left[\begin{array}{c} \boxed{b_0} \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ \boxed{b_8} \end{array}\right] \tag{7.18}$$

The padded samples of a are a_0 and a_6 while that of b are b_0 and b_8. We assume that $a_0 = a_1$ and $a_5 = a_6$, then

$$\begin{aligned} (h_1 + h_2)a_1 &= b_1 \text{ and } a_1 = a_0 = \frac{b_1}{(h_1 + h_2)} && (7.19) \\ (h_2 + h_3)a_5 &= b_7 \text{ and } a_5 = a_6 = \frac{b_7}{(h_2 + h_3)} && (7.20) \\ b_0 &= \frac{h_1 b_1}{(h_1 + h_2)} && (7.21) \end{aligned}$$

$$b_8 = \frac{h_3 b_7}{(h_2 + h_3)}. \tag{7.22}$$

Now that the samples b_0 and b_8 are computed, we can consider the extended blurred signal $\hat{b} = (b_0, \cdots, b_8)$ and proceed with the deconvolution algorithm to obtain the deblurred signal which will now be of length $n_1 = 7$. Since the extended blurred signal $\hat{b}$ does not end abruptly but slowly reduces to 0 via the padded samples, the edge artifacts are significantly reduced. This scheme can be extended to two dimensions and was employed in the deconvolution scheme that resulted in the deblurred image shown in Figure 7.4.

7.3 Deconvolution with a Nonseparable 2D PSF

In Section 7.2, we considered the 2D deconvolution problem when the PSF had rank 1. This resulted in the two dimensions being separable. This property was exploited and a direct method based on the Schur algorithm was employed to solve the deconvolution problem. In this section, we address the problem of deconvolution when the 2D PSF in not separable, i.e., the PSF has a rank greater than 1.

Consider the problem of deconvolving an image of size $(n_1 + m_1 - 1) \times (n_2 + m_2 - 1)$ that has been blurred by a 2D PSF h of size $m_1 \times m_2$ (7.7). The deconvolved image would be of size $n_1 \times n_2$ and the problem of deconvolution is expressed as solving (7.8) in the least squares sense. Equation (7.10) gives the normal equations for this problem. In this section, we make the assumption that h is not a rank 1 matrix but has a low rank. Most problems in signal and image processing yield PSFs that have low rank. We shall illustrate this through some examples. Since we no longer have a rank 1 factorization of h as in (7.11), we can no longer factor H as the Kronecker product of two banded lower triangular Toeplitz matrices as in (7.12). If one chose to use a direct method such as the block Schur algorithm [23] to solve the normal equations, then only one level of Toeplitzness in the coefficient matrix can be exploited and the computational complexity of the problem would be prohibitively expensive even for moderately sized problems.

In recent years, there have been several preconditioners proposed to solve Toeplitz and block Toeplitz systems iteratively. A survey of preconditioning schemes for Toeplitz and block Toeplitz systems can be found in [68]. Most of these preconditioners have been used in the iterative deconvolution of images. All the preconditioners proposed thus far have been motivated solely by the structure of the problem, i.e., a block Toeplitz matrix with Toeplitz blocks. In this section, we propose a preconditioner that is based on approximating the PSF and, hence, in some sense is motivated by the application (image deconvolution).

The idea central to this new preconditioner is the fact that since most 2D non-separable PSFs are of low rank, one may construct a preconditioner based on a rank 1 approximation of the 2D PSF. This approximation may be done either in the space domain or in the frequency domain. In most practical problems, since the PSF has to be obtained by the blurring of an impulse, one usually has an estimate of the PSF in the space domain and the approximation has to be done in this domain. We consider an example PSF and show how the preconditioner may be constructed.

Consider a circularly symmetric ideal low-pass filter whose frequency response is shown in Figure 7.5. This is an example of a 2D non-separable filter. The impulse response of this filter is given by

$$h_{i,j} = \frac{\omega_c}{2\pi\sqrt{i^2+j^2}} J_1(\omega_c\sqrt{i^2+j^2}), \tag{7.23}$$

where J_1 is the Bessel function of the first kind and the first order and ω_c is the cutoff frequency of the filter. A plot of the impulse response (sampled at 41×41 points) for $\omega_c = 0.2\pi$ is shown in Figure 7.6. If we consider the singular values of the impulse response matrix resulting from sampling (7.23) at 41×41 points, then the plot of the singular values is shown in Figure 7.8. It can be seen from this plot that the PSF has a low rank. As mentioned earlier, one can approximate this PSF in either the frequency or the space domain. In the frequency domain, a square plateau having the same area as the circular frequency response (see Figure 7.5) is a separable approximation. The impulse response corresponding to this approximation is given by

$$h(i,j) = h_1(i)h_2(j) = \frac{\sin(b\,i)}{\pi\,i}\frac{\sin(b\,j)}{\pi\,j}, \quad \text{where} \quad b = \frac{\sqrt{\pi}\omega_c}{2}. \tag{7.24}$$

Figure 7.7 is a plot of this separable approximate PSF sampled at 41×41 points. The PSF can be approximated in the spatial domain by taking the outer product of the singular vectors corresponding to the first singular value of the impulse response matrix. Figure 7.9 is a plot of the separable PSF obtained from the singular vectors corresponding to the first singular value of the 2D non-separable PSF.

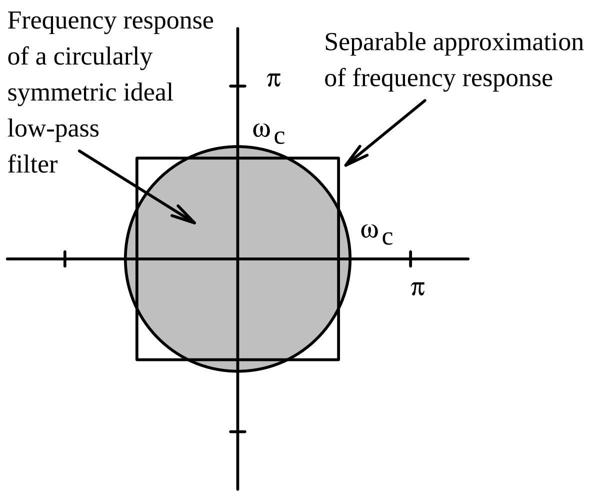


Figure 7.5 Circularly symmetric low-pass filter.

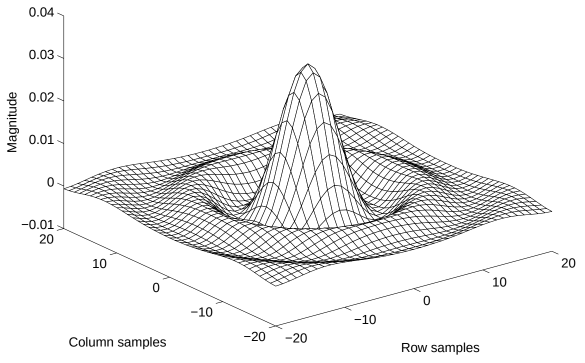


Figure 7.6 Impulse response of circularly symmetric filter.

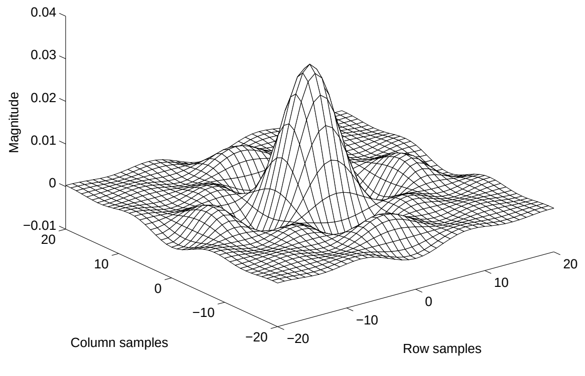


Figure 7.7 Impulse response from frequency domain approximation.

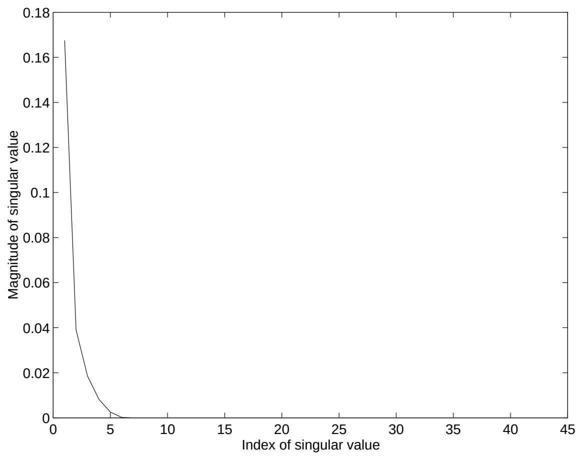


Figure 7.8 Singular values of impulse response shown in Figure 7.6.

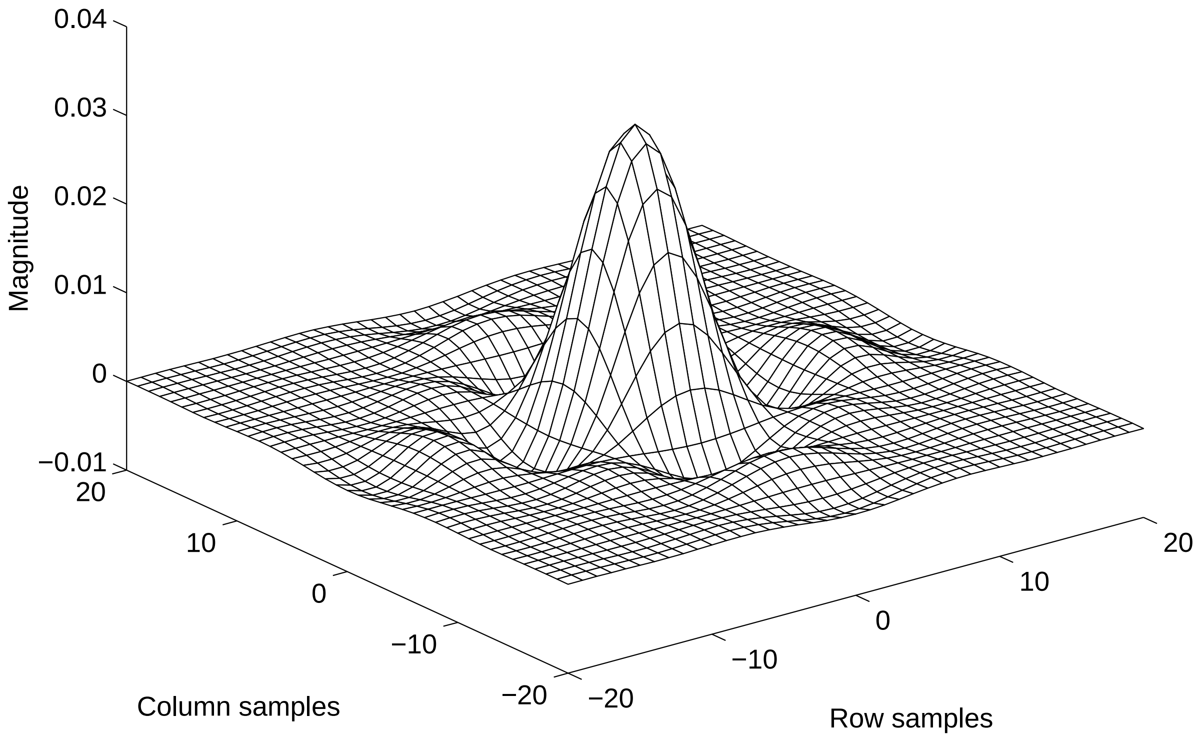


Figure 7.9 Impulse response from space domain approximation.

In most deconvolution problems, we have an estimate of the PSF in the space domain. Hence, an S.V.D. of the PSF matrix is used to obtain a separable approximation. Using MATLAB notation, the following code is used to compute the separable approximation in the space domain.

```
[m1,m2] = size(h);
[u,s,v] = svd(h);
g = u(:,1) * sqrt(s(1,1));
f = v(:,1) * sqrt(s(1,1));
```

This yields a rank 1 approximation to h of the form $h \approx gf^T$. If we construct rectangular banded lower triangular Toeplitz matrices F and G from f and g, respectively, then one can approximate the coefficient matrix of (7.10) by

$$(H^T H + \mu I) \approx (F^T F + \sqrt{\mu} I) \otimes (G^T G + \sqrt{\mu} I). \tag{7.25}$$

The above approximation is valid, of course, for small values of μ. In the above equation, we have chosen the matrix L to be the identity matrix. The matrices $\left(F^T F + \sqrt{\mu} I\right)$ and $\left(G^T G + \sqrt{\mu} I\right)$ are banded Toeplitz matrices with half bandwidths m_2 and m_1. Using the property that the inverse of the Kronecker products of two matrices is the Kronecker product of their inverses, $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$, one can obtain a very inexpensive preconditioner based on the factorization of the two banded Toeplitz matrices using the Schur algorithm. Further, since the entries of the two matrices in the Kronecker product die out as you move away from the diagonal, one could obtain more computational savings if a "drop-tolerance" were used to zero out entries close to zero.

If we use the preconditioned conjugate gradient (PCG) algorithm on the normal equations (7.10), then the Kronecker product-based approximation of the coefficient matrix can be used as a preconditioner. Applying this preconditioner is very inexpensive if $m_1 \ll n_1$ and $m_2 \ll n_2$. For example, at every step of the PCG algorithm [64] one has to solve the following system of equations

$$M z_k = r_k, \tag{7.26}$$

where M is the preconditioner. In our problem $M = \left(F^TF + \sqrt{\mu}I\right) \otimes \left(G^TG + \sqrt{\mu}I\right)$, and z_k and r_k are of size $n_1 \times n_2$. If we rewrite z_k to be a matrix Z_k of size $n_1 \times n_2$ and r_k to be a matrix R_k of size $n_1 \times n_2$, then (7.26) can be written as

$$\left(G^TG + \sqrt{\mu}I\right) Z_k \left(F^TF + \sqrt{\mu}I\right) = R_k. \tag{7.27}$$

$\left(G^TG + \sqrt{\mu}I\right)$ is a banded Toeplitz matrix of size $n_1 \times n_1$ with a half bandwidth of m_1 and $\left(F^TF + \sqrt{\mu}I\right)$ is a banded Toeplitz matrix of size $n_2 \times n_2$ with a half bandwidth of m_2. The total cost of applying this preconditioner is, therefore, $O((m_1 + m_2)n_1n_2)$.

In the preceding paragraphs, we described how one may construct a preconditioner based on approximating the PSF h. The resulting preconditioner, a Kronecker product of two banded Toeplitz matrices, was expected to approximate the matrix $(H^TH+\mu I)$ in the normal equations. This preconditioner was obtained from the PSF. One could construct another preconditioner very similar to the preconditioner described above by approximating the matrix $(H^TH + \mu I)$ directly by the Kronecker product of two Toeplitz matrices. We use Toeplitz matrices to approximate the coefficient matrix because it is a block Toeplitz matrix with Toeplitz blocks. We outline the steps by which such an approximation may be constructed.

The matrix $(H^TH+\mu I)$ in our problem is a $n_1n_2 \times n_1n_2$ banded block Toeplitz matrix with Toeplitz blocks. The half bandwidth of the block Toeplitz matrix is m_2 and the half bandwidth of the Toeplitz blocks is m_1. We attempt to construct symmetric banded Toeplitz matrices A and B of sizes $n_2 \times n_2$ and $n_1 \times n_1$ and half bandwidths m_1 and m_2, respectively, such that

$$(H^TH + \mu I) \approx A \otimes B. \tag{7.28}$$

Let the matrix $T = (H^TH + \mu I)$ be written as

$$T = (H^TH + \mu I) = \begin{bmatrix} T_1 & \cdots & T_{m_2}^T & 0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & & \\ T_{m_2} & & \ddots & & \ddots & \\ 0 & \ddots & & \ddots & & T_{m_2}^T \\ \vdots & & \ddots & & \ddots & \vdots \\ 0 & \cdots & 0 & T_{m_2} & \cdots & T_1 \end{bmatrix}, \tag{7.29}$$

and the matrices A and B be

$$A = \begin{bmatrix} a_1 & \cdots & a_{m_2} & 0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & & \\ a_{m_2} & & \ddots & & \ddots & \\ 0 & \ddots & & \ddots & & a_{m_2} \\ \vdots & & \ddots & & \ddots & \vdots \\ 0 & \cdots & 0 & a_{m_2} & \cdots & a_1 \end{bmatrix}, \quad B = \begin{bmatrix} b_1 & \cdots & b_{m_1} & 0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & & \\ b_{m_1} & & \ddots & & \ddots & \\ 0 & \ddots & & \ddots & & b_{m_1} \\ \vdots & & \ddots & & \ddots & \vdots \\ 0 & \cdots & 0 & b_{m_1} & \cdots & b_1 \end{bmatrix}. \tag{7.30}$$

To construct the preconditioner we must satisfy the following constraints to the extent possible.

$$\begin{aligned} T_1 &\approx a_1 B \\ &\vdots \\ T_{m_2} &\approx a_{m_2} B \end{aligned} \tag{7.31}$$

Since the matrix T is derived from a blurring operator, the Frobenius norm of the blocks $T_1 \cdots T_{m_2}$ decreases rapidly from the diagonal. We begin constructing the preconditioner by choosing a_1 and B such that $T_1 = a_1 B$. T_1 is a banded symmetric Toeplitz matrix. Let the first column of T_1 be $(t_{11}, \cdots, t_{1m_1}, 0, \cdots, 0)$. We choose $a_1 = b_1 = \sqrt{t_{11}}$ and $b_i = t_{1i}/a_1$ for $i = 2, \cdots, m_1$. This gives us the matrix B and the main diagonal of A. The remaining elements of A, namely, $a_2, \cdots, a_{m_2}$ are obtained by minimizing the Frobenius norm of $(T_i - a_i B)$ for $i = 2, \cdots, m_2$.

This procedure gives us an approximation to the coefficient matrix $(H^T H + \mu I)$ of the normal equations of the form $A \otimes B$. This approximation can then be used as a preconditioner in the PCG algorithm in exactly the same manner as the preconditioner derived from the 2D non-separable PSF.

7.4 Experimental Results

In this section, we present the results of some experiments on deconvolving astronomical images iteratively using the two preconditioners described in Section 7.3.

The true (original) image in these experiments is a simulation of a star cluster with a globular clusterlike luminosity function and spatial distribution. Spatially variant and invariant PSFs were constructed to simulate the blurring effects due to errors in construction of mirrors in the Wide-Field Planetary Camera on the Hubble Space Telescope (HST). These images (all of which are of size 256×256) are a part of STScI's Image Restoration Project [69] and were obtained via anonymous FTP from `ftp.stsci.edu`, in the directory `software/stsdas/testdata/restore/sims/star_cluster`. For our experiments, we use the spatially invariant PSF only. For problems with piecewise constant PSFs, the problem of deconvolution can be broken into subproblems where each region is deconvolved using a separate PSF.

Figure 7.10 shows the true image of the star cluster. The spatially invariant PSF of the filter that simulates the blurring effect of the HST is shown in Figure 7.11. Figure 7.12 shows the blurred image of the star-cluster. Read-out noise and Poisson noise have been added to the image. To illustrate the low rank nature of the PSF matrix, the singular values of the 41×41 PSF matrix have been plotted in Figure 7.13. The ratio of the first singular value to the second was $1.19/0.25 = 4.76$. A separable PSF was constructed using the singular vectors of the first singular value. This separable approximation was used as a preconditioner in the iterative deconvolution scheme. Figure 7.14 is a plot of this separable approximation. At this point, we mention that though the number of non-zero samples in the approximate PSF was about 41 in each dimension, samples that were less than 1% of the maximum were discarded. This translates directly to a reduction in complexity since the bandwidth of the Toeplitz matrices is smaller. The half bandwidths of the two Toeplitz matrices $(F^TF + \sqrt{\mu}I)$ and $(G^TG + \sqrt{\mu}I)$ were 15 and 18, respectively.

Figures 7.15 - 7.18 show the results of the CG algorithm applied to the normal equations with a regularization parameter of $\mu = 0.001$. The iterative deconvolution algorithm in the absence of a preconditioner converges in approximately 40 iterations.

If the separable approximation to the original PSF based on the first singular vectors is chosen as a preconditioner, then the iterative deconvolution process converges in eight iterations. Figures 7.19 - 7.22 show some intermediate iterations during the convergence process.

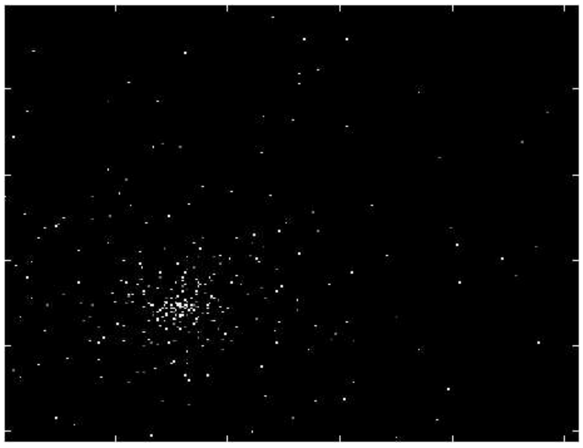


Figure 7.10 Original star-cluster image.

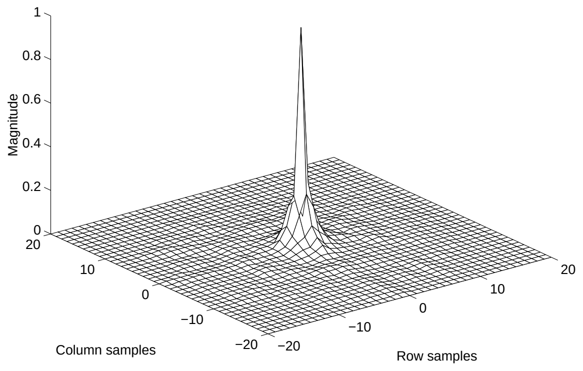


Figure 7.11 PSF of HST camera.

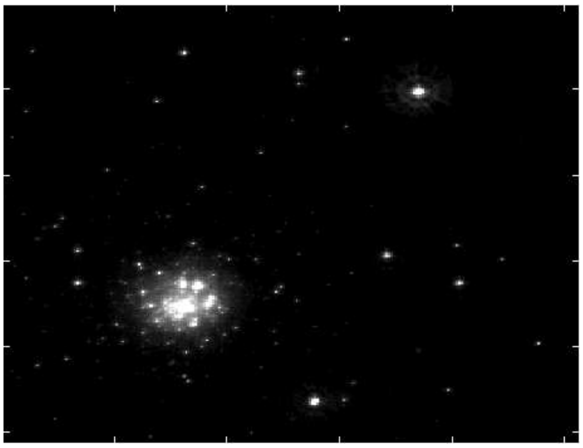


Figure 7.12 Image blurred by the PSF of the HST camera.

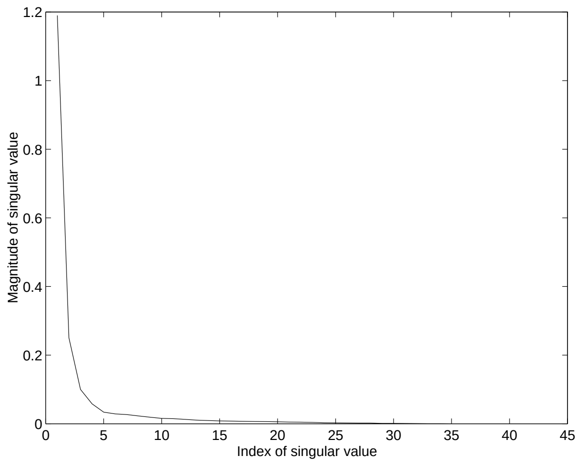


Figure 7.13 Singular values of non-separable PSF.

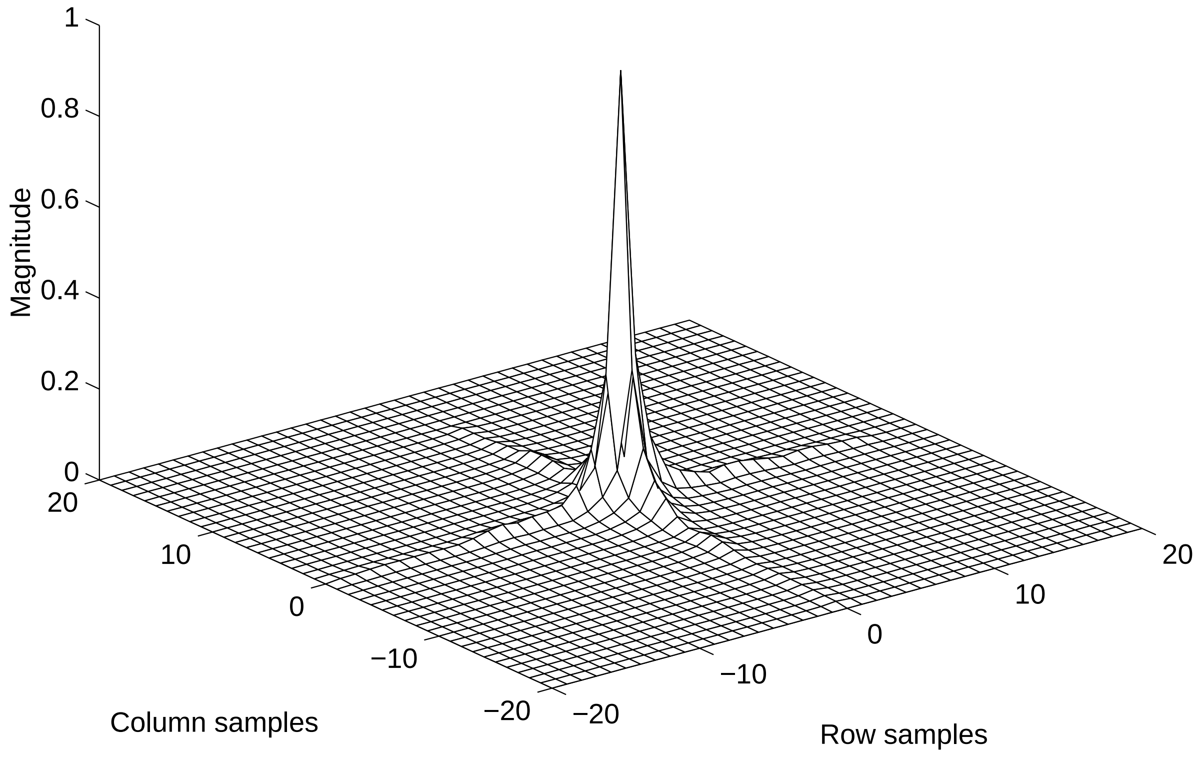


Figure 7.14 PSF from space domain approximation.

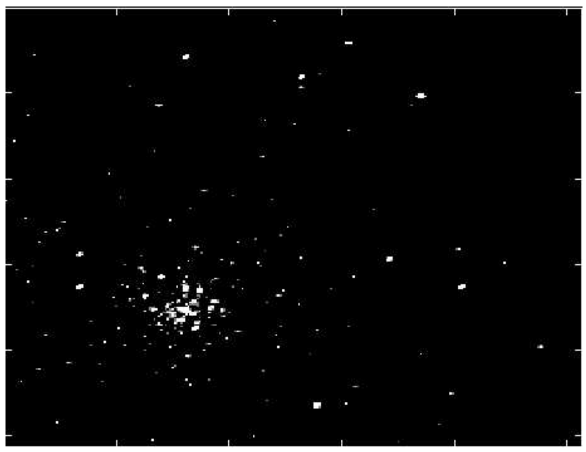


Figure 7.15 CG after 10 iterations.

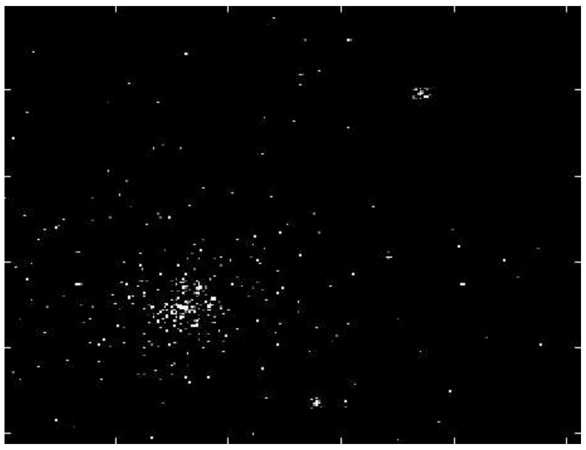


Figure 7.16 CG after 20 iterations.

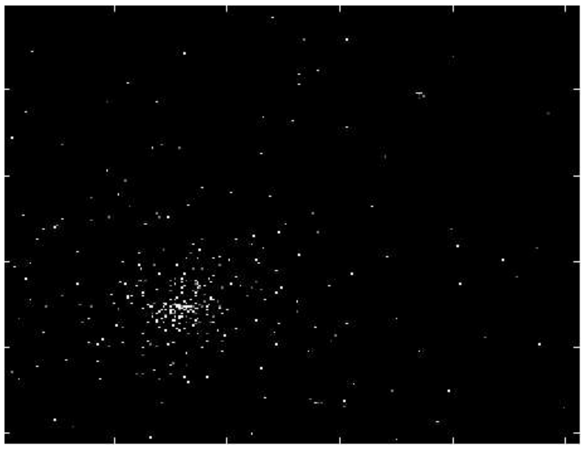


Figure 7.17 CG after 30 iterations.

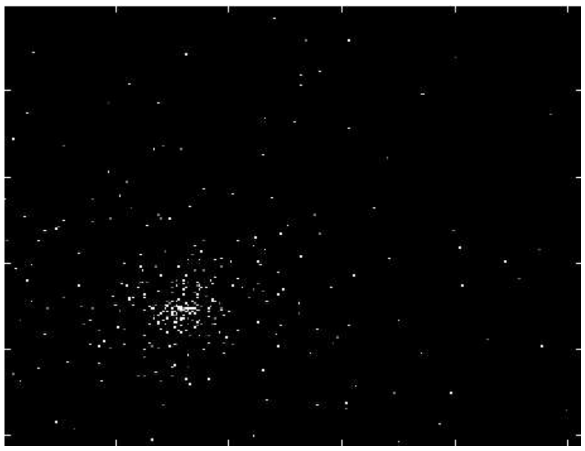


Figure 7.18 CG after 40 iterations.

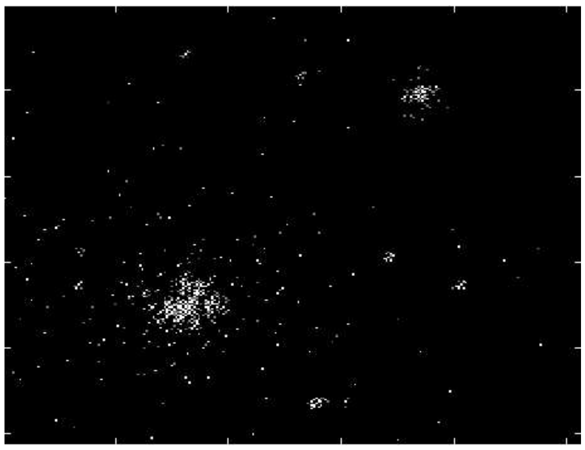


Figure 7.19 PCG after 2 iterations.

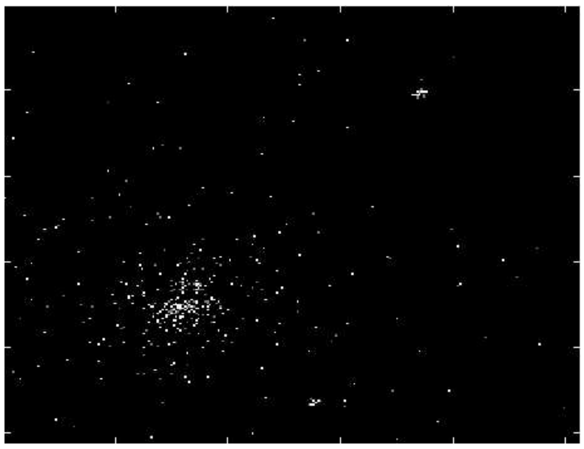


Figure 7.20 PCG after 4 iterations.

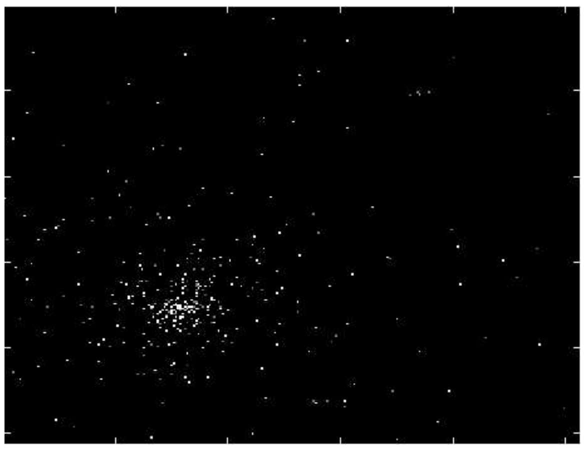


Figure 7.21 PCG after 6 iterations.

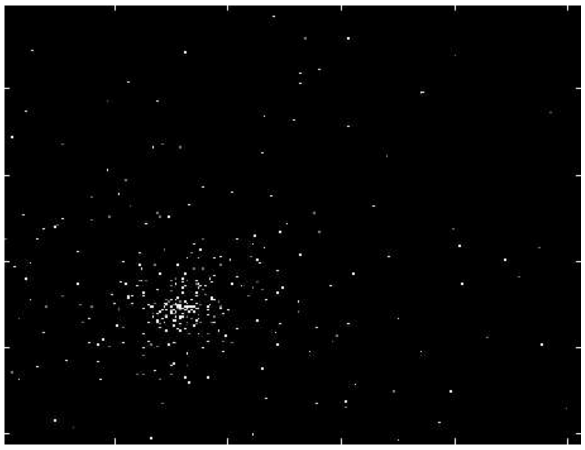


Figure 7.22 PCG after 8 iterations.

If we approximate the coefficient matrix $(H^T H + \mu I)$ directly with the Kronecker product of two banded Toeplitz matrices as described in the previous section and use it to precondition the system, then the iterative deconvolution process converges in ten iterations.

7.5 Comparison to Commonly Used Preconditioners

A commonly used preconditioner in iterative deconvolution problems is based on approximating the block Toeplitz matrix with Toeplitz blocks (BTTB) by a block circulant matrix with circulant blocks (BCCB). Inverting the BCCB matrix is then done using the 2D discrete Fourier transform. The construction of such a preconditioner has been described in [70]. Applying such a preconditioner M [70] to a vector r_k in the PCG algorithm involves computing two 2D FFTs and an element-wise division

$$M^{-1} r_k = \texttt{ifft2}\left(\Lambda^{-1}\texttt{fft2}(r_k)\right). \tag{7.32}$$

Here Λ is the set of eigenvalues of the BCCB matrix and `fft2` and `ifft2` are the forward and inverse 2D FFTs applied to a vector by considering it to be a 2D matrix. For an image of size $n_1 \times n_2$ the complexity of applying this preconditioner would be $O(n_1 n_2 \log(n_1 n_2))$.

The cost of applying the preconditioners described in Section 7.3 would critically depend on the half bandwidth of the two Toeplitz matrices making up the Kronecker product. Applying the preconditioners involves factoring the two banded Toeplitz matrices and carrying out a series of multiple right-hand side forward and backward solves. The Schur algorithm used to factor the two banded Toeplitz has lower complexity than the forward and backward solves. If the half bandwidths of the two Toeplitz matrices are m_1 and m_2, then the cost of applying this preconditioner would be $O((m_1 + m_2) n_1 n_2)$.

If m_1 and m_2 are small compared to n_1 and n_2, then it may be less expensive to apply the preconditioners based on direct methods. In addition, data locality in the computation of an FFT is poor compared to that of forward and back solves.

We now present experimental results using three different point spread functions to compare the convergence properties of the preconditioner based on a circulant approximation with the preconditioner based on a separable approximation. In the first experiment we used a Gaussian point spread function $h(x, y) = e^{-0.03(x^2+y^2)}$ sampled at a grid of 21×21 points. Since the point spread function is separable, the preconditioned conjugate gradient (PCG) algorithm using the separable preconditioner converged in two iterations. When the circulant approximation-based preconditioner was used, the PCG algorithm failed to converge even after ten iterations.

In the second experiment, we used the point spread function shown in Figure 7.11. The PCG algorithm using the circulant preconditioner converged in five iterations while the separable approximation-based preconditioner resulted in convergence in eight iterations. The ratio of the first two singular values of the PSF shown in Figure 7.11 was 4.751. When this ratio was increased to 15 (making the PSF more separable), then both preconditioners converged in four iterations. When the ratio was further increased to 30, the PCG algorithm using the separable preconditioner converged in three iterations while that using the circulant preconditioner converged in four iterations. This shows that as the ratio of the first two singular values increases the performance of the separable preconditioner improves.

In the third experiment we consider a non-separable PSF (see Figure 7.23) with two singular values (ratio = 6.1726) and a slightly wider support (implying greater blurring) than the one in Figure 7.11.The PCG algorithm using the separable preconditioner converged in seven iterations while the circulant preconditioner resulted in convergence in ten iterations. In addition, the solution obtained using the circulant preconditioner had two stars in the upper right corner of the image missing. When a PSF based only on the first singular value is used, the PCG algorithm using the separable preconditioner converged in 1 iteration while that using the circulant preconditioner took six iterations to converge.

7.6 Complexity and Other Implementation Issues

In this section, we discuss some implementation issues and compare the proposed preconditioners to some commonly used preconditioners. We also present some performance numbers of the deconvolution scheme on distributed memory machines such as the Cray T3D.

7.6.1 Implementing the matrix-vector product needed in the CG algorithm

If we choose to deconvolve the image iteratively by applying the CG or PCG algorithm to the normal equation (7.10), then at each iteration one would have to compute the product $(H^TH + \mu I)x$. It was shown earlier that the matrix $(H^TH + \mu I)$ is a banded block Toeplitz

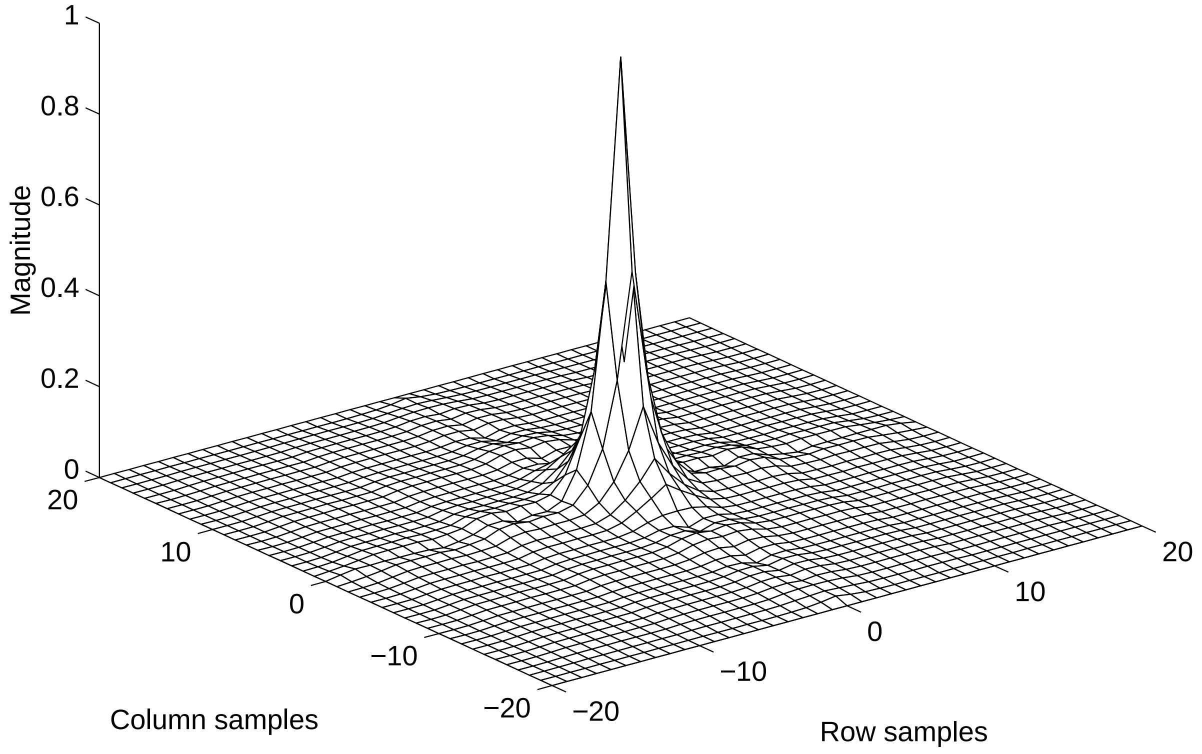


Figure 7.23 PSF with two singular values having a ratio of 6.1726.

matrix with banded Toeplitz blocks. In this section we show how the product $(H^T H + \mu I)x$ may be computed. We assume that all variables have the usual meanings and the size of the deconvolution problem is as described in Section 7.3.

We first show how the product $(H^T H + \mu I)x$ may be computed in the space domain, i.e., using the definition of convolution. Assuming the usual problem size, the vector x is of length $n_1 n_2$. Let X be a 2D image of size $n_1 \times n_2$ derived from the vector x. The matrix-vector product Hx is the 2D convolution operation of X with the PSF h expressed in a linear algebraic form. Since h is of size $m_1 \times m_2$, the zero padded convolution operation yields a matrix $\hat{X}$ of size $(n_1 + m_1 - 1) \times (n_2 + m_2 - 1)$. Let $\hat{x}$ be the vector corresponding to the matrix $\hat{X}$. The product $H^T \hat{x}$ is a non-zero padded 2D convolution product of the 2D matrix $\hat{X}$ with a reflected version of h, i.e., with $e_1 h e_2$ (where e_1 and e_2 are reflection matrices of size m_1 and m_2, respectively). Using MATLAB notation, the product $(H^T H + \mu I)x$ is computed as

```
Xhat = conv2(X,h)
Y = conv2(Xhat,h(m1:-1:1,m2:-1:1),'valid') + mu * X.
```

If m_1 and m_2 are small compared to n_1 and n_2, then the complexity of the entire product would be approximately $4 m_1 m_2 n_1 n_2$.

If m_1 and m_2 were not small compared to n_1 and n_2, then it is less expensive to formulate the product $(H^T H + \mu I)x$ as a circular convolution operation and compute it using the 2D discrete Fourier transform. We now show how such a computation may be carried out.

Let the matrix $(H^T H + \mu I)$ be partitioned as shown in (7.29). The matrices $T_1, \cdots, T_{m_2}$ are banded Toeplitz matrices with a half bandwidth of m_1. The entries of these Toeplitz matrices can be computed very inexpensively from the 2D PSF matrix h. Let $\hat{h}$ be the zero padded convolution product of h with $e_1 h e_2$, where e_1 and e_2 are reflection matrices. In MATLAB notation, this would be computed as

```
hhat = conv2(h,h(m1:-1:1,m2:-1,1)).
```

Since h is of size $m_1 \times m_2$, the matrix $\hat{h}$ is of size $(2m_1 - 1) \times (2m_2 - 1)$. The effect of the regularization term μI can be incorporated by adding μ to $\hat{h}_{m_1 m_2}$. The entries of $T_1, \cdots, T_{m_2}$ can now be obtained from the columns of $\hat{h}$ numbering $m_2, \cdots, 2m_2 - 1$, respectively. The following MATLAB code illustrates how this is accomplished.

```
Ti = toeplitz( [ hhat(m1:2*m1-1,m2+i-1); zeros(n1-m1,1) ], ...
               [ hhat(m1:-1:1,m2+i-1); zeros(n1-m1,1) ] ).
```

Having computed the banded Toeplitz matrices $T_1, \cdots, T_{m_2}$, we show how a banded block Toeplitz matrix with banded Toeplitz blocks may be multiplied with a vector. We begin by illustrating the 1D case : multiplication of a banded Toeplitz matrix with a vector.

Let T be a square banded Toeplitz matrix of size n_1 with a half bandwidth of m_1. Let x be a vector of size n_1 that is to be multiplied by T. The Toeplitz matrix T is embedded in a circulant matrix C_T and the matrix vector product is expressed as a circulant convolution. The circulant convolution is then carried out using the discrete Fourier transform. We illustrate this with an example. Let $n_1 = 5$ and $m_1 = 3$,

$$\left[\begin{array}{ccccc|cc} t_0 & t_1 & t_2 & & & t_{-2} & t_{-1} \\ t_{-1} & t_0 & t_1 & t_2 & & & t_{-2} \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 & & \\ & t_{-2} & t_{-1} & t_0 & t_1 & t_2 & \\ & & t_{-2} & t_{-1} & t_0 & t_1 & t_2 \\ \hline t_2 & & & t_{-2} & t_{-1} & t_0 & t_1 \\ t_1 & t_2 & & & t_{-2} & t_{-1} & t_0 \end{array}\right] \left[\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ \hline 0 \\ 0 \end{array}\right] = \left[\begin{array}{c} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ \hline b_5 \\ b_6 \end{array}\right]. \tag{7.33}$$

The augmented matrix vector product in (7.33) is a circulant convolution product. If we denote the first column of C_T by c_1, then the circulant convolution is computed using DFTs as

$$b = \texttt{ifft}\left(\texttt{fft}(c_1). * \texttt{fft}(\hat{x})\right), \tag{7.34}$$

where $\hat{x}$ is the augmented vector constructed from x and $.*$ denotes the Hadamard product. The first $n_1 = 5$ elements of b yield the matrix-vector product Tx that we desire. In addition, if T is symmetric, then c_1 is mirror symmetric about its center. This results in $\texttt{fft}(c_1)$ being real and some savings in computation can be obtained.

This procedure can be extended quite naturally to the case where a banded block Toeplitz matrix with banded Toeplitz blocks is to be multiplied with a vector. Let us consider a banded block Toeplitz matrix T of size $n_1 n_2 \times n_1 n_2$ with a block size of $n_1 \times n_1$. Let the block Toeplitz matrix have a half bandwidth of m_2. The non-zero blocks of the matrix would then range from

$T_{-m_2+1}, \cdots, T_0, \cdots, T_{m_2-1}$. Let each of these blocks in turn be banded Toeplitz matrices with half bandwidth equal to m_1. As described in (7.33) each banded Toeplitz block is padded suitably to convert them into circulant matrices. Let these matrices be $C_{-m_2+1}, \cdots, C_0, \cdots, C_{m_2-1}$ of size $(n_1 + m_1 - 1) \times (n_1 + m_1 - 1)$. The padding is then completed at the block level to obtain a block circulant matrix with circulant blocks as

$$C_T = \left[\begin{array}{ccccc|cc} C_0 & C_1 & C_2 & & & C_{-2} & C_{-1} \\ C_{-1} & C_0 & C_1 & C_2 & & & C_{-2} \\ C_{-2} & C_{-1} & C_0 & C_1 & C_2 & & \\ & C_{-2} & C_{-1} & C_0 & C_1 & C_2 & \\ & & C_{-2} & C_{-1} & C_0 & C_1 & C_2 \\ \hline C_2 & & & C_{-2} & C_{-1} & C_0 & C_1 \\ C_1 & C_2 & & & C_{-2} & C_{-1} & C_0 \end{array}\right]. \qquad (7.35)$$

In this example $n_2 = 5$ and $m_2 = 3$. Multiplication of a vector x with the Toeplitz block Toeplitz matrix is done as follows. First, the image X of size $n_1 \times n_2$ that is derived from x is padded with zeros to obtain an augmented matrix $\hat{X}$ of size $(n_1 + m_1 - 1) \times (n_2 + m_2 - 1)$. The first column of the block circulant matrix C_T (of length $(n_1 + m_1 - 1)(n_2 + m_2 - 1)$) is rearranged as a matrix of size $(n_1 + m_1 - 1) \times (n_2 + m_2 - 1)$ by stacking the first columns of the circulant blocks side by side. Let this matrix be $\hat{C}$. The 2D discrete Fourier transform is then used to compute the 2D circulant convolution.

$$B = \texttt{ifft2}\left(\texttt{fft2}(\hat{C}) .* \texttt{fft2}(\hat{X})\right). \qquad (7.36)$$

The upper left $n_1 \times n_2$ elements of B then yield the matrix vector product of T with x. Again, if T is symmetric, then $\texttt{fft2}(\hat{C})$ is real and one can save some computation.

On machines provided by most high performance computer (HPC) vendors, a fast implementation of the DFT is available for vectors whose lengths can be expressed as products of powers of limited radixes such as $2, 3, 4$ and 5. On such machines, one would have to pad the 2D PSF h with zeros on the boundary (to size $\hat{m}_1 \times \hat{m}_2$) such that $n_1 + \hat{m}_1 - 1$ and $n_2 + \hat{m}_2 - 1$ are of lengths that can be suitably factored.

7.6.2 Implementation on the Cray T3D

The preconditioners described in Section 7.3 are quite amenable to a scalable implementation on distributed memory machines such as the Cray T3D. On such machines, the image would have to be distributed in a panel distribution across a linear array of processors. Each processor would store and process a certain number of contiguous columns of the image. Applying the preconditioner at each step of the deconvolution process involves solving (7.27). This is done in three main steps. In the first step, each processor uses the Cholesky factors of $\left(G^TG + \sqrt{\mu}I\right)$ to solve the set of multiple right-hand sides allotted to it. The processors then collaborate to perform a distributed 2D matrix transpose to flip the rows and columns after which the factors of $\left(F^TF + \sqrt{\mu}I\right)$ are used to solve the multiple right-hand sides. The image is transposed again to restore its original shape.

The other important step at each iteration is the matrix vector product of the coefficient matrix with one of the conjugate directions. This is done using 2D FFTs. The main computational kernels in this iterative deconvolution process are, therefore, the Schur algorithm, multiple forward and backward solves with banded lower and upper triangular matrices, distributed transpose algorithm and 2D FFTs. The factorization of the two Toeplitz matrices $\left(G^TG + \sqrt{\mu}I\right)$ and $\left(F^TF + \sqrt{\mu}I\right)$ forming the Kronecker product is done only once using the Schur algorithm. The Schur algorithm to factor banded Toeplitz matrices has been optimized for a single processing element (PE) of the Cray T3D. Figure 7.24 shows the performance of the Schur algorithm on a single PE of the Cray T3D. The x-axis indicates the half bandwidth of the banded Toeplitz matrix and the y-axis is the performance in Mflops.

In addition, a fast distributed memory transpose routine has been written in assembly language for the Cray T3D. This routine is also used in the parallel 2D and 3D FFT routines that are a part of the scientific libraries on the Cray T3D. The performance of this routine in terms of MBytes/sec for a 512×512 image transpose is shown in Figure 7.25. The number of PEs varies from 1 to 32 along the x-axis.

Figure 7.26 is a plot of the time per iteration (in seconds) to solve the example astronomical image deconvolution problem of Section 7.3. The image size was 256×256, i.e., $n_1 = n_2 = 256$. $m_1 = m_2 = 41$. Since $n_1 + m_1 - 1 = 296$ is not factorizable into powers of $2, 3, 4$ and 5, we pad the 2D PSF to be of size 45×45. The preconditioner was constructed from the first singular value of the the 2D PSF. Further, all elements of $(F^TF + \sqrt{\mu}I)$ and $(G^TG + \sqrt{\mu}I)$ that were

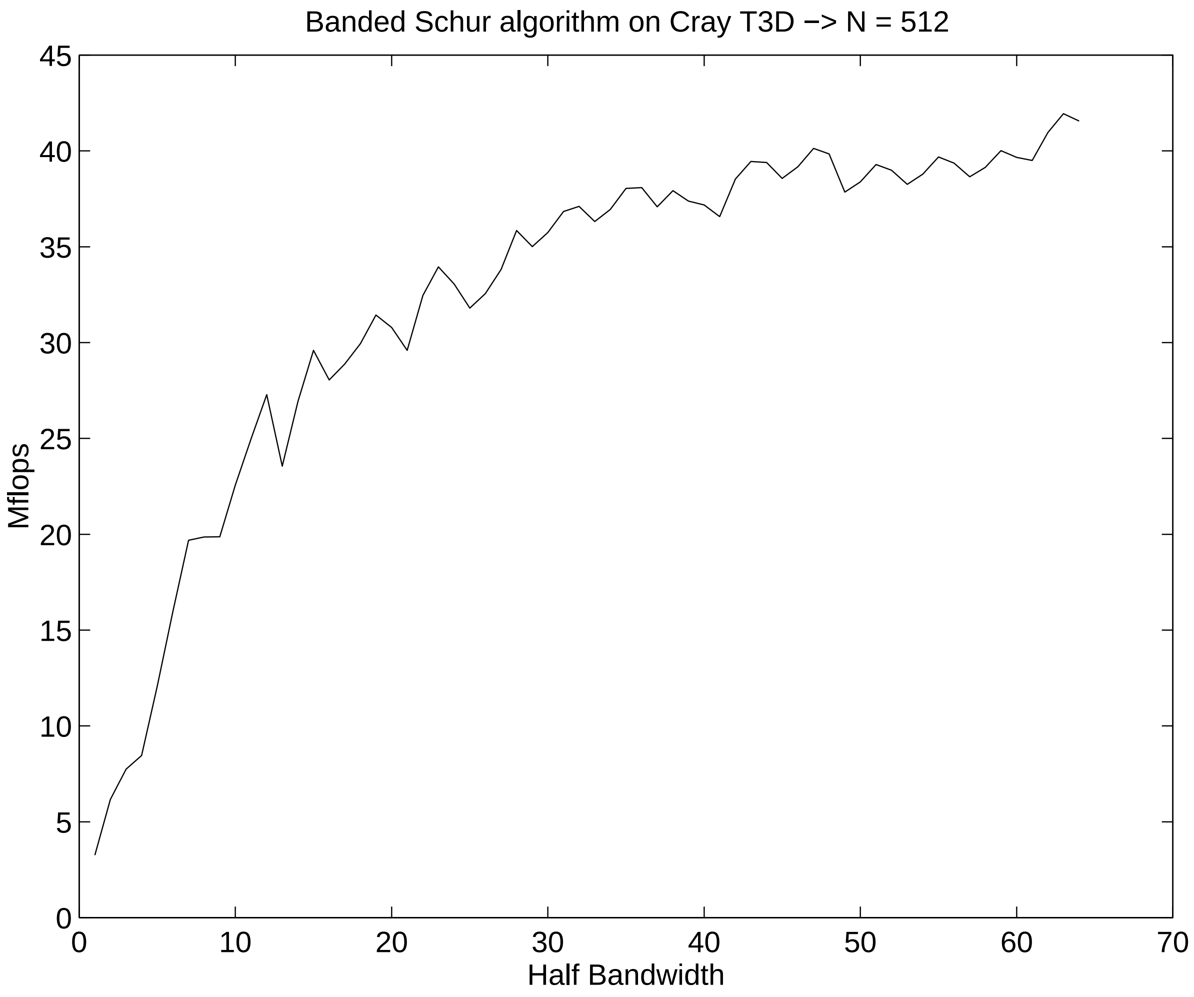


Figure 7.24 Banded Schur algorithm on a single PE of the Cray T3D.

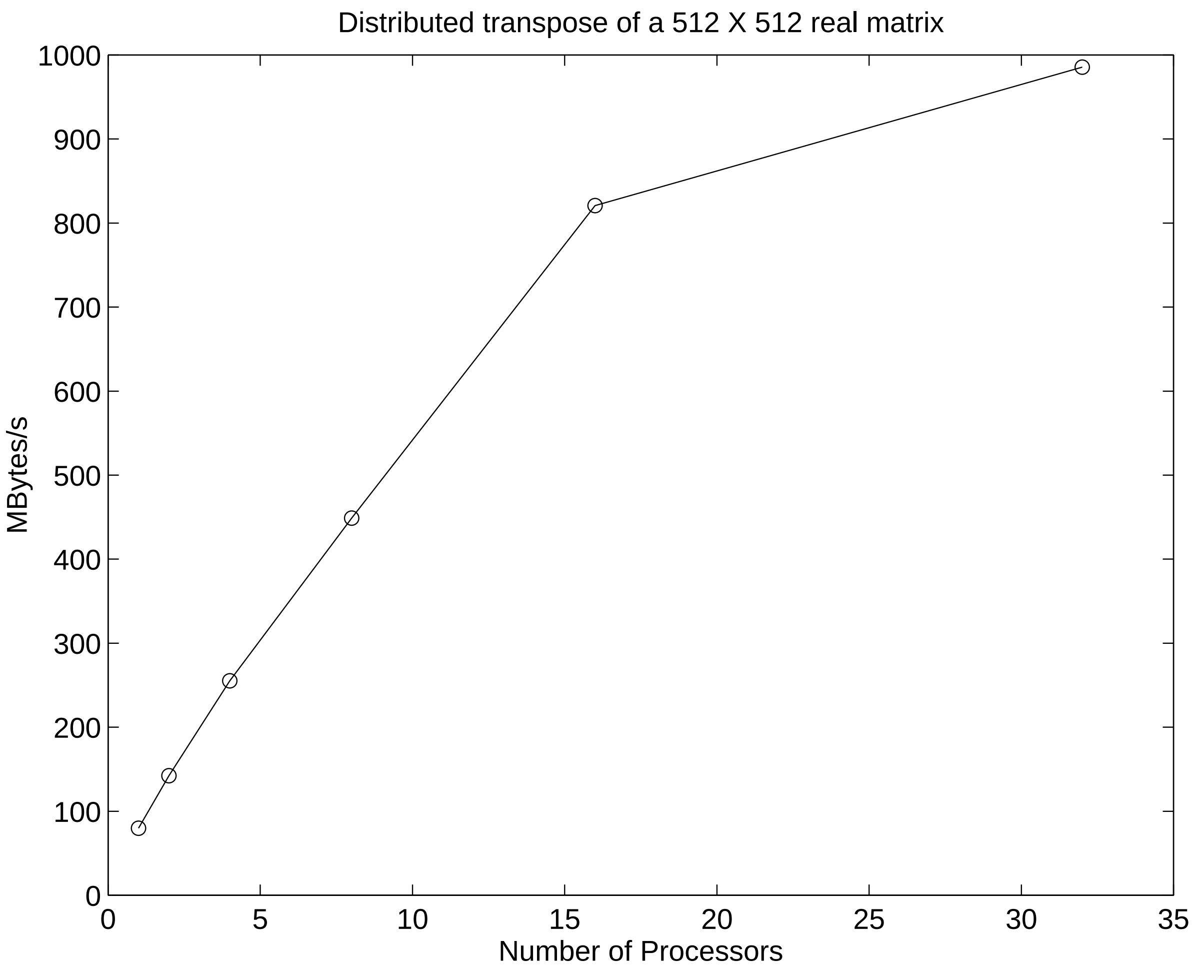


Figure 7.25 Distributed transpose of a 512×512 matrix on the Cray T3D.

less than 1 % of the diagonal were set to zero. This reduced the half bandwidths of the two banded Toeplitz matrices to 15 and 18, respectively. Figure 7.26 plots the time to apply the preconditioner, the time for the matrix-vector product and the total time in seconds versus the number of PEs in the machine. The machine size was varied from 1 to 64. The times are also listed in tabular form in Table 7.1. From Table 7.1 it can be seen that both the preconditioner

Table 7.1 Time, in seconds, per iteration to solve the deconvolution problem.

Number of processors	Time per iteration	Time to apply preconditioner	Time for matvec
1	0.872	0.429	0.409
2	0.458	0.219	0.217
4	0.232	0.111	0.110
8	0.118	0.056	0.056
16	0.060	0.028	0.029
32	0.031	0.014	0.015
64	0.017	0.007	0.009

and the iterative scheme scale well on the T3D. As was pointed out in Section 7.4, the PCG algorithm converged in eight iterations with this preconditioning scheme. From Figure 7.26, it can be concluded that without the preconditioning step, the iteration of a normal CG would require approximately half the time. For this particular problem, the CG algorithm converges in 40 iterations. The PCG algorithm, therefore, requires the same time as 16 iterations of CG and, in real terms, using the proposed preconditioner, one obtains a savings of around 2.5 over the CG algorithm.

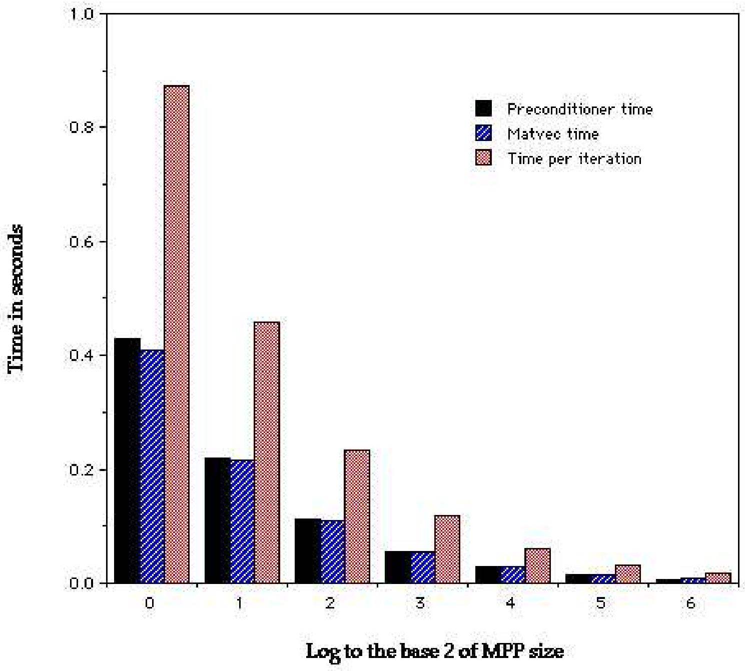


Figure 7.26 Time, in seconds, per iteration to solve the deconvolution problem.

CHAPTER 8

CONCLUSION

Algorithms to solve Toeplitz systems can be broadly classified into two main classes, namely, Levinson-like algorithms that factor the inverse of the Toeplitz matrix and Schur-like algorithms that factor the Toeplitz matrix itself. In this dissertation we have proposed several new algorithms based on generalizing the classical Schur algorithm to solve Toeplitz and block Toeplitz linear systems and least squares problems. Some obvious advantages of Schur-like algorithms over Levinson-like algorithms are that Levinson-like algorithms fail to exploit certain properties such as bandedness of the Toeplitz matrix since the inverse of a banded matrix is dense. Schur-like algorithms on the other hand exploit this property quite naturally. Other less obvious advantages are that Levinson-like algorithms use non-scalable primitives such as dotproducts in the computation of the factorization while the Schur-like algorithms only rely on scalable primitives such as triads. In addition, it has been shown that the classical Schur algorithm is backward stable while the Levinson algorithm has only been shown to be weakly stable. In this chapter, we summarize the contributions of this dissertation to the area of direct Schur-like algorithms to solve Toeplitz and block Toeplitz linear systems and least squares problems.

The main contributions of this dissertation are

- Several new blocking schemes to block hyperbolic Householder transformations that are used in a block generalization of the Schur algorithm to factor symmetric positive-definite block Toeplitz matrices.
- Implementation results to show the improvement in performance of the block Schur algorithm on machines with a hierarchical memory structure.
- Generalization of hyperbolic Householder transformations to the indefinite case in the absence of breakdown.

- A new algorithm that produces an approximate factorization of symmetric indefinite block Toeplitz matrices by perturbing the generator when a singularity is encountered. Accuracy of the solution is enhanced by the use of iterative refinement. An optimal value for the perturbation and a bound on the number of steps of iterative refinement have been derived.

- Two new look-ahead Schur algorithms to compute the exact factorization of symmetric indefinite block Toeplitz matrices. One algorithm is based on retrieving the generator of the Schur complement using the Bunch-Kaufman algorithm while the other is based on obtaining the generator by completing squares.

- Modifications to existing algorithms that convert indefinite Toeplitz matrices to Cauchy-like matrices. These modified algorithms exploit properties such as realness and symmetry simultaneously. We also show how Hermitian Toeplitz matrices may be converted to real Cauchy-like matrices, thereby avoiding expensive calculations in complex arithmetic.

- Implementation results of Cauchy-like methods on high performance architectures such as parallel vector processing systems.

- Modification of the generalized Schur algorithm to obtain a rank factorization of rectangular block Toeplitz matrices with exactly linearly dependent columns.

- Extension of Cauchy-like methods to solve Toeplitz least squares problems using the normal equations method or the augmented system of equations.

- Extension of Cauchy-like methods to compute a rank-revealing QR factorization of block Toeplitz matrices.

- New direct method-based preconditioners to solve regularization problems in the iterative deconvolution of images.

The research work that has contributed to this dissertation was motivated by the need for an LAPACK-like suite of routines to solve Toeplitz and block Toeplitz linear systems and least squares problems. The last few years have seen a tremendous improvement in the state-of-the-art for Schur-like algorithms. The most significant contribution to this field by this dissertation and the work of other researchers has been block Toeplitz look-ahead solvers that handle exact and near singularities and the new Cauchy-like methods that were first proposed by Gohberg, Kailath and Olshevsky. Based on these and other algorithms, Toeplitz matrix equivalents of LAPACK routines such as SPORTF (factorization of symmetric positive-definite matrices), SSYTRF (factorization of symmetric indefinite matrices), SGEQRF (QR factorization without pivoting) and SGEQPF (QR factorization with pivoting) are now available. One area of future work would be to consolidate all existing Schur-like algorithms into a portable LAPACK-like suite of routines for Toeplitz systems. Such a library, if provided by high-performance computer vendors, would be of great use to several scientific and engineering researchers.

Such a library will have to be accompanied by advances in the stability analysis of Toeplitz and block Toeplitz factorization algorithms and in the understanding of the tradeoffs between higher performance algorithms with can have stability problems in certain situations and slower but more reliable algorithms. The resulting library codes would be hybrids which rely on instability detection strategies to switch to poorer performing but more reliable methods when difficulties are encountered. Progress has been made in such stability analysis recently [67, 56], but further work in rank revealing factorization and library design situations is needed.

A class of matrices for which fast direct Schur-like algorithms do not exist consists of Toeplitz matrices with multiple levels of Toeplitzness such as block Toeplitz matrices with Toeplitz blocks. Block Schur algorithms only exploit one level of Toeplitzness and for some 2D and 3D applications this could be prohibitively expensive. Iterative methods can exploit all levels in their matrix-vector multiplication primitives. However, as with more general situations the convergence of these methods depends heavily on the development of preconditioners. While a large amount of research has already been conducted in this area we believe that the insights developed for direct methods in this thesis will aid in the development of novel preconditioners for Toeplitz-block-Toeplitz matrices.

REFERENCES

[1] Z.-P. Liang and P. C. Lauterbur, "A generalized series approach to MR spectroscopic imaging," *IEEE Trans. Med. Imaging*, vol. 10, pp. 132–137, June 1991.

[2] M. Hanke and P. C. Hansen, "Regularization methods for large-scale problems," *Surv. Math. Ind.*, vol. 3, pp. 253–315, 1993.

[3] J. H. McClellan, "Parametric signal modeling," in *Advanced Topics in Signal Processing* (J. S. Lim and A. V. Oppenheim, eds.), Englewood Cliffs, NJ: Prentice Hall, 1988.

[4] P. Van Dooren, "Numerical linear algebra for signals, systems and control." Class Notes for ECE 497, Electrical Engineering Dept., University of Illinois at Urbana-Champaign, 1993.

[5] E. Berlekamp, *Algebraic Coding Theory.* New York: McGraw-Hill, 1968.

[6] J. Massey, "Shift-register synthesis and BCH coding," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 122–127, 1969.

[7] I. Schur, "Über potenzreihen, die im Inneren des Einheitskreises beschränkt sind," *Z. Reine Angew. Math.*, vol. 147, pp. 205–232, 1917.

[8] N. Levinson, "The Weiner RMS error criterion in filter design and prediction," *J. Math. Phys.*, vol. 25, pp. 261–278, 1947.

[9] J. Durbin, "The fitting of time series models," *Rev. Inst. Int. Stat.*, vol. 28, pp. 233–243, 1960.

[10] W. F. Trench, "An algorithm for the inversion of finite Toeplitz matrices," *J. SIAM*, vol. 12, pp. 261–278, 1964.

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

[12] J. Rissanen, "Algorithms for triangular decomposition of block Hankel and Toeplitz matrices with application to factoring positive matrix polynomials," *Math. Comp.*, vol. 27, pp. 147–154, 1973.

[13] N. I. Akhiezer, *The Classical Moment Problem.* London: Oliver and Boyd, 1965.

[14] P. Delsarte, Y. Genin, and Y. Kamp, "Schur parametrization of positive definite block-Toeplitz systems," *SIAM J. Appl. Math.*, vol. 36, no. 1, pp. 34–46, 1979.

[15] I. Gohberg and A. Semencul, "On the inversion of finite Toeplitz matrices and their continuous analogs," *Mat. Issled.*, vol. 2, pp. 201–233, 1972.

[16] T. Kailath, S.-Y. Kung, and M. Morf, "Displacement ranks of matrices and linear equations," *J. Math. Anal. and Appl.*, vol. 68, pp. 395–407, 1979.

[17] H. Lev-Ari, *Nonstationary lattice filter modeling.* PhD thesis, Stanford University, December 1983.

[18] H. Lev-Ari and T. Kailath, "Triangular factorization of structured Hermitian matrices," *Operator Theory: Advances and Appl.*, vol. 18, pp. 301–324, 1986.

[19] J. Chun, *Fast array algorithms for structured matrices.* PhD thesis, Stanford University, 1989.

[20] J. Chun, T. Kailath, and H. Lev-Ari, "Fast parallel algorithms for QR and triangular factorization," *J. Sci. Stat. Comput.*, vol. 8, pp. 899–913, 1987.

[21] T. Kailath and J. Chun, "Generalized displacement structure for block-Toeplitz, Toeplitz-block and Toeplitz-derived matrices," *SIAM J. Matrix Anal. Appl.*, vol. 15, pp. 114–128, 1994.

[22] G. Cybenko and M. Berry, "Hyperbolic Householder algorithms for factoring structured matrices," *SIAM J. Matrix Anal. Appl*, vol. 11, pp. 499–520, October 1990.

[23] K. Gallivan, S. Thirumalai, and P. Van Dooren, "On solving block Toeplitz matrices using a block Schur algorithm," in *Proc. 1994 Int. Conf. Parallel Processing*, (St. Charles, IL), pp. III–274–III–281, August 1994.

[24] K. A. Gallivan, S. Thirumalai, P. Van Dooren, and V. Vermaut, “High performance algorithms to solve Toeplitz and block Toeplitz systems,” *Linear Algebra Appl.*, April 1996. To appear.

[25] G. Heinig and K. Rost, *Algebraic Methods for Toeplitz-like Matrices and Operators.* Boston, MA: Birkhaüser, 1984.

[26] P. Delsarte, Y. V. Genin, and Y. G. Kamp, “A generalization of the Levinson algorithm for Hermitian Toeplitz matrices with any rank profile,” *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 33, pp. 964–971, August 1985.

[27] D. Pal and T. Kailath, “Fast triangular factorization and inversion of Hermitian, Toeplitz, and related matrices with arbitrary rank profiles,” *SIAM J. Matrix Anal. Appl.*, vol. 14, pp. 1016–1042, October 1993.

[28] S. Pombra, H. Lev-Ari, and T. Kailath, “Levinson and Schur algorithms for Toeplitz matrices with singular minors,” in *Proc. 1988 Int. Conf. Acoust., Speech, Signal Process.*, (New York, NY), pp. 1643–1646, April 11-14 1988.

[29] C. J. Zarowski, “Schur algorithms for Hermitian Toeplitz, and Hankel matrices with singular leading principal submatrices,” *IEEE Trans. Signal Process.*, vol. 39, pp. 2464–2480, November 1991.

[30] T. F. Chan and P. C. Hansen, “A look-ahead Levinson algorithm for indefinite Toeplitz systems,” *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 490–506, April 1992.

[31] S. Cabay and R. Meleshko, “A weakly stable algorithm for Padé approximants and the inversion of Hankel matrices,” *SIAM J. Matrix Anal. Appl.*, vol. 14, pp. 635–765, July 1993.

[32] R. Freund and H. Zha, “Formally biorthogonal polynomials and a look-ahead Levinson algorithm for general Toeplitz systems,” *Linear Algebra Appl.*, vol. 188/189, pp. 255–304, 1993.

[33] M. H. Gutknecht and M. Hochbruck, “Look-ahead Levinson- and Schur-type recurrences in the Padé table,” *ETNA*, vol. 2, pp. 104–129, September 1994.

[34] K. A. Gallivan, S. Thirumalai, and P. Van Dooren, "A new look-ahead Schur algorithm," in *Proc. Fifth SIAM Conf. Applied Linear Algebra*, (Snowbird, UT), pp. 450–454, SIAM, June 1994.

[35] K. A. Gallivan, S. Thirumalai, and P. Van Dooren, "A block Toeplitz look-ahead Schur algorithm," in *SVD in Signal Processing III: Algorithms, Architectures and Applications* (M. Moonen and B. D. Moor, eds.), pp. 199–206, Amsterdam: Elsevier, 1995.

[36] A. H. Sayed and T. Kailath, "A look-ahead block Schur algorithm for Toeplitz like matrices," *SIAM J. Matrix Anal. Appl.*, vol. 16, pp. 388–414, 1995.

[37] G. Heinig, "Inversion of generalized Cauchy matrices and other classes of structured matrices," in *Linear Algebra for Signal Processing* (A. Bojanczyk and G. Cybenko, eds.), vol. 69 of *The IMA Volumes in Mathematics and its Applications*, pp. 95–114, New York: Springer-Verlag, 1994.

[38] I. Gohberg, T. Kailath, and V. Olshevsky, "Gaussian elimination with partial pivoting for structured matrices," tech. rep., Information Systems Lab., Stanford University, 1994.

[39] T. Kailath and V. Olshevsky, "Symmetric and Bunch-Kaufman pivoting for partially structured Cauchy-like matrices with application to Toeplitz-like matrices," tech. rep., Information System Laboratory, Stanford University, Stanford, CA, 1995.

[40] D. Bini and F. D. Benedetto, "A new preconditioner for the parallel solution of positive definite Toeplitz systems," in *Proc. Second ACM Symp. Parallel Algorithms and Architectures*, (Crete, Greece), pp. 220–223, 1990.

[41] E. Boman and I. Koltracht, "Fast transform based preconditioner for Toeplitz equations," *SIAM J. Matrix Anal. Appl.*, vol. 16, pp. 628–645, 1995.

[42] E. Bozzo and C. D. Fiore, "On the use of certain matrix algebras associated with discrete trigonometric transforms in matrix displacement decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 1995, pp. 312–, January 16.

[43] R. Chan, M. Ng, and C. Wong, "Sine transform based preconditioner for symmetric Toeplitz systems," *Linear Algebra Appl.* To appear.

[44] D. R. Sweet, "Fast Toeplitz orthogonalization," *Numer. Math.*, vol. 43, pp. 1–21, 1984.

[45] A. W. Bojanczyk, R. P. Brent, and F. R. de Hoog, "QR factorization of Toeplitz matrices," *Numer. Math.*, vol. 49, pp. 81–94, 1986.

[46] G. Cybenko, "Fast Toeplitz orthogonalization using inner products," *SIAM J. Sci. Stat. Comput.*, vol. 8, pp. 734–740, September 1987.

[47] P. C. Hansen and H. Gesmar, "Fast orthogonal decomposition of rank deficient Toeplitz matrices," *Numerical Algorithms*, vol. 4, pp. 151–166, 1993.

[48] T. Kailath and A. H. Sayed, "Displacement structure: Theory and applications," *SIAM Review*, vol. 37, pp. 297–386, September 1995.

[49] R. P. Brent and F. T. Luk, "A systolic array for the linear-time solution of Toeplitz systems of equations," *J. VLSI and Computer Systems*, vol. 1, pp. 1–22, 1983.

[50] J.-M. Delosme and I. C. F. Ipsen, "Parallel solution of symmetric positive definite systems with hyperbolic rotations," *Linear Algebra Appl.*, vol. 77, pp. 75–111, 1986.

[51] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel algorithms for dense linear algebra computations," *SIAM Review*, vol. 32, pp. 54–135, 1990.

[52] C. M. Rader and A. O. Steinhardt, "Hyperbolic Householder transformations," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 34, pp. 1589–1602, 1986.

[53] C. Bischof and C. Van Loan, "The WY representation for products of Householder matrices," *SIAM J. Sci. Stat. Comput.*, vol. 8, pp. s2–s13, 1987.

[54] R. Schreiber and C. Van Loan, "A storage-efficient WY representation for products of Householder transformations," *SIAM J. Sci. Stat. Comput.*, vol. 10, pp. 53–57, January 1989.

[55] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* Baltimore: SIAM, 1992.

[56] M. Stewart and P. Van Dooren, "Stability issues in the factorization of structured matrices," *SIAM J. Matrix Anal. Appl.* To appear.

[57] P. Concus and P. Saylor, "A modified direct preconditioner for indefinite symmetric Toeplitz systems," *Journal of Lin. Alg. & Appl.* To appear.

[58] J. H. Wilkinson, *The Algebraic Eigenvalue Problem.* Oxford, England: Oxford University Press, 1965.

[59] J. R. Bunch and L. Kaufman, "Some stable methods for calculating inertia and solving symmetric linear systems," *Math. Comp.*, vol. 31, pp. 163–179, January 1977.

[60] T. Kailath and A. Sayed, "Fast algorithms for generalized displacement structures," in *Recent Advances in Mathematical Theory of Systems* (H. Kimura and S. Kodoma, eds.), pp. 27–32, 1992. Proc. MTNS-91.

[61] T. Chan, "An optimal circulant preconditioner for Toeplitz systems," *SIAM J. Sci. Statist. Comput.*, vol. 9, pp. 766–771, 1988.

[62] T. Huckle, "Fast transforms for tridiagonal linear equations," *BIT*, vol. 34, pp. 99–112, 1994.

[63] R. H. Chan, T. F. Chan, and C. Wong, "Cosine transform based preconditioners for total variation minimization problems in image processing," Tech. Rep. 95-8, Chinese University of Hong Kong, Shatin, Hong Kong, 1995.

[64] G. H. Golub and C. F. Van Loan, *Matrix Computations.* The John Hopkins University Press, 1989.

[65] Å. Björck, "Pivoting and stability in the augmented system," Tech. Rep. LiTH-MAT-R-1991-30, University of Linköping, Dept. of Mathematics, June 1991.

[66] Å. Björck, "Component-wise perturbation analysis and error bounds for linear least squares solutions," Tech. Rep. LiTH-MAT-R-1989-13, University of Linköping, Dept. of Mathematics, December 1990.

[67] A. W. Bojanczyk, R. P. Brent, F. R. D. Hoog, and D. R. Sweet, "On the stability of the Bareiss and related Toeplitz factorization algorithms," *SIAM J. Matrix Anal. Appl.*, vol. 16, pp. 40–57, January 1995.

[68] R. Chan and K. P. Ng, "Conjugate gradient methods for Toeplitz systems," *SIAM Review*, 1995. To appear.

[69] R. J. Hanisch, "WF/PC simulation data sets," in *Newsletter of STScI's Image Restoration Project* (R. J. Hanisch, ed.), pp. 76–77, 1993.

[70] M. Hanke, J. G. Nagy, and R. J. Plemmons, "Preconditioned iterative regularization," in *Numerical Linear Algebra* (L. Reichel, A. Ruttan, and R. S. Varga, eds.), pp. 141–163, Berlin: de Gruyter, 1993.

VITA

The author was born on February 6, 1969 to Mr. B. R. Thirumalai and Mrs. Vijayalakshmi Thirumalai in Bombay, India. After spending seventeen fun-filled years in the bustling metropolis of Bombay, the author's academic career in Electrical Engineering began with an oft-remembered train journey across the Indian subcontinent to the quiet university town of Kharagpur (located 75 miles to the west of the city of Calcutta) where he received the B.Tech. degree in Electronics and Electrical Communication Engineering from the Indian Institute of Technology in 1990.

In the fall of 1990, after completing the B.Tech. degree, the author found himself embarking on another trip, this time across several continents, to the quiet, easy-paced midwestern town of Urbana where he would spend the next few years in graduate school at the University of Illinois at Urbana-Champaign. He received the M.S. degree in Electrical Engineering in 1992. He is currently a candidate for the Ph.D. degree in Electrical Engineering at the University of Illinois at Urbana-Champaign.

In 1984, the author received the National Talent Search scholarship (India's most highly regarded precollege scholarship) from the National Council for Educational Research and Training, Govt. of India. He was also awarded the Outstanding Paper Award jointly with Kyle A. Gallivan and Paul Van Dooren for their paper titled "On Solving Block Toeplitz Systems Using a Block Schur Algorithm", presented at the 1994 International Conference on Parallel Processing in St. Charles, IL. The author is a member of the Institute of Electrical and Electronics Engineers, Inc. and also holds an Advanced Class Amateur Radio License (KB0TRX).

After completing his doctoral dissertation, the author will continue working at Cray Research, Inc. where he is currently a member of the Scientific and Math. Libraries Group. His research interests include parallel numerical algorithms, linear algebra for signals, systems and control, signal processing and image processing.