SIAM J. Matrix Anal. Appl.
Vol. 29, No. 4, pp. 1247–1266

A SUPERFAST ALGORITHM FOR TOEPLITZ SYSTEMS OF LINEAR EQUATIONS[*]

S. CHANDRASEKARAN[†], M. GU[‡], X. SUN[§], J. XIA[¶], AND J. ZHU[‡]

Abstract. In this paper we develop a new superfast solver for Toeplitz systems of linear equations. To solve Toeplitz systems many people use displacement equation methods. With displacement structures, Toeplitz matrices can be transformed into Cauchy-like matrices using the FFT or other trigonometric transformations. These Cauchy-like matrices have a special property, that is, their off-diagonal blocks have small numerical ranks. This low-rank property plays a central role in our superfast Toeplitz solver. It enables us to quickly approximate the Cauchy-like matrices by structured matrices called *sequentially semiseparable* (SSS) matrices. The major work of the constructions of these SSS forms can be done in precomputations (independent of the Toeplitz matrix entries). These SSS representations are compact because of the low-rank property. The SSS Cauchy-like systems can be solved in linear time with linear storage. Excluding precomputations the main operations are the FFT and SSS system solve, which are both very efficient. Our new Toeplitz solver is stable in practice. Numerical examples are presented to illustrate the efficiency and the practical stability.

Key words. displacement equation, SSS structure, superfast algorithm, Toeplitz matrix

AMS subject classifications. 15A06, 65F05, 65G05

DOI. 10.1137/040617200

1. Introduction.

Toeplitz systems of linear equations arise in many applications, including PDE solving, signal processing, time series analysis, orthogonal polynomials, and many others. A *Toeplitz system* is a linear system

$$Tx = b \tag{1.1}$$

with a coefficient matrix to be a *Toeplitz matrix*

$$T = \begin{pmatrix} t_0 & t_{-1} & t_{-2} & \cdots & t_{-(N-1)} \\ t_1 & t_0 & t_{-1} & \cdots & t_{-(N-2)} \\ t_2 & t_1 & t_0 & \cdots & t_{-(N-3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{N-1} & t_{N-2} & t_{N-3} & \cdots & \cdots \end{pmatrix}, \tag{1.2}$$

that is, its entries are constant along every diagonal (a matrix whose entries are constant along every antidiagonal is called a Hankel matrix). The vector $t = (t_{-(N-1)} \cdots t_{-1}\ t_0\ t_1 \cdots t_{N-1})$ is called the *Toeplitz-vector* that generates T.

There are both direct and iterative methods for solving (1.1). Direct solvers are said to be *fast* if they cost $O(N^2)$ operations; examples include Schur-type methods,

[*]Received by the editors October 19, 2004; accepted for publication (in revised form) by D. A. Bini May 8, 2007; published electronically December 13, 2007.
http://www.siam.org/journals/simax/29-4/61720.html

[†]Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 (shiv@ece.ucsb.edu).

[‡]Department of Mathematics, University of California at Berkeley, Berkeley, CA 94720 (mgu@math.berkeley.edu, zhujiang.cal@gmail.com).

[§]Department of Mechanical Engineering, University of California at Berkeley, Berkeley, CA 94720 (sunxt@cal.berkeley.edu).

[¶]Department of Mathematics, University of California at Los Angeles, Los Angeles, CA 90095 (jxia@math.ucla.edu).

Levinson-type methods, and others [32]. An important type of direct solver is the *displacement equation–type* fast solver based on Gaussian eliminations. Some known displacement equation–type methods are the Heinig [28], GKO [22], and Gu [26] methods. Those methods have complexity $O(N^2)$. Methods with complexity less than $O(N^2)$ are called *superfast*. In this paper we will present a displacement equation–type superfast algorithm.

1.1. Fast and superfast methods. Many fast and superfast methods that have been developed are numerically unstable [5, 15, 16, 7, 37]. References [5] and [15] showed that the Schur algorithm and the Levinson algorithm are weakly stable in some cases, but both may be highly unstable in the case of an indefinite and nonsymmetric matrix. Stable generalized Schur algorithms [32] and look-ahead algorithms were developed in [9, 10]. High-performance look-ahead Schur algorithms were presented [20].

Many other solvers use the FFT or other trigonometric transforms to convert the Toeplitz (or even Hankel or Toeplitz-plus-Hankel) matrices into generalized Cauchy or Vandermonde matrices, which can be done stably in $O(N \log N)$ operations. This is also the approach that we will use in this paper, with the aid of a displacement structure.

The concept of displacement structure was first introduced in [31]. The Sylvester-type *displacement equation* for a matrix $\hat{C} \in \mathbf{R}^{N\times N}$ [29] is

$$\Omega \hat{C} - \hat{C}\Lambda = UV, \tag{1.3}$$

where Ω, $\Lambda \in \mathbf{R}^{N\times N}$, $U \in \mathbf{R}^{N\times \alpha}$, $V \in \mathbf{R}^{\alpha\times N}$, and $\alpha \le N$ is the *displacement rank* with respect to Ω and Λ if $\operatorname{rank}(UV) = \alpha$. The matrix $\hat{C}$ is considered to possess a *displacement structure* with respect to Ω and Λ if $\alpha \ll N$.

With displacement structures it was shown in [19, 24, 36, 22, 28] that Toeplitz and Hankel matrices can be transformed into *Cauchy-like* matrices of the following form:

$$\hat{C} = \left(\frac{u_i^T \cdot v_j}{\eta_i - \lambda_j}\right)_{1\le i,j\le N} \qquad (u_i, v_j \in \mathbf{R}^\alpha)\ ,$$

where we assume that $\eta_i \ne \lambda_j$ for $1 \le i, j \le N$. Equivalently, a Cauchy-like matrix is the unique solution to the displacement equation (1.3) with

$$\Omega = \operatorname{diag}(\eta_1, \ldots, \eta_n),\ \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n),\ U = \begin{pmatrix} u_1^T \\ \vdots \\ u_n^T \end{pmatrix},\ \text{and } V = (v_1, \ldots, v_n)\,.$$

In particular, $\hat{C}$ is a Cauchy matrix if $u_i^T v_j = 1$ for all i and j. The displacement rank of $\hat{C}$ is at most α.

To solve Toeplitz systems through Cauchy-like matrices, many people have utilized matrix factorizations. Gohberg and Olshevsky [23] presented a fast variation of the straightforward Gaussian elimination with partial pivoting (GEPP) procedure to solve a Cauchy-like linear system of equations in $O(N^2)$ operations. Among other results, Gohberg, Kailath, and Olshevsky [22] developed algorithm GKO, an improved version of Heinig's algorithm [28], and demonstrated numerically that it is stable. In their algorithm, the Hankel matrix and the Toeplitz-plus-Hankel matrix are also transformed via fast trigonometric transforms into Cauchy-like matrices.

Gu presented a modified algorithm in [26] to avoid extra error growth. This algorithm is numerically stable, provided that the element growth in the computed factorization is not large. The algorithm takes $O(N^2)$ operations and is a fast stable method.

Superfast algorithms appeared in [34, 4, 6, 17, 35, 1, 2, 21] and many others. Superfast algorithms use divide-and-conquer strategies. Morf developed the first idea in [34]. These methods are unstable for nonsymmetric systems as they cannot deal with nearly singular leading principal submatrices.

Van Barel and Kravanja presented a superfast method for rational interpolation at roots of unity [39]. A similar idea was then applied to Toeplitz systems [38]. It provided an explicit formula for the inverse of a Toeplitz matrix. Additional techniques such as iterative refinement and downdating were still required to stabilize their algorithm.

1.2. Main results. Our new Toeplitz solver is also of the displacement equation type. Given a Toeplitz linear system, we first use the FFT to transform the associated Toeplitz matrix into a Cauchy-like matrix. Then instead of using matrix factorizations which often cost $O(N^2)$ or more, we exploit a special *low-rank property* of Cauchy-like matrices, that is, every off-diagonal block of a Cauchy-like matrix has a low numerical rank. Using this low-rank property, we then approximate the Cauchy-like matrix by a low-rank matrix structure called the *sequentially semiseparable* (SSS) matrix proposed by Chandrasekaran et al. [12, 13]. A system with the coefficient matrix in *compact* SSS form can be solved with only $O(p^2N)$ operations, where N is the matrix dimension, and p is the complexity of the semiseparable description. The SSS solver is practically stable in our numerical tests and those in [12, 13].

The SSS structure was developed to capture the low-rank property of the off-diagonal blocks of a matrix and to maintain stability or practical stability in the mean time. It is a matrix analog of semiseparable integral kernels in Kailath's paper [30]. Matrix operations with compact form SSS representations are very efficient, provided that such compact representations exist or can be easily computed. This turns out to be true for our case, as the Cauchy-like matrices are transformed from Toeplitz matrices. We use a recursive compression scheme with a shifting strategy to construct compact SSS forms for those Cauchy-like matrices. The major work of the compressions can be precomputed on some Cauchy matrices which are independent of the actual Toeplitz matrix entries.

The overall algorithm thus has the following stages:

(1) Precompute compressions of off-diagonal blocks of Cauchy matrices.
(2) Transform the Toeplitz matrix into a Cauchy-like matrix in $O(N \log N)$ operations.
(3) Construct a compact SSS representation from precomputed compressions and solve the Cauchy-like matrix system $O(p^2N)$ operations.
(4) Recover the solution of the Toepliz system in $O(N \log N)$ operations.

The stages above are either stable or practically stable. Our numerical results indicate that the overall algorithm is stable in practice. The Toeplitz matrix does not have to be symmetric or positive definite, and no extra stabilizing step is necessary. After the precomputations, the total cost for the algorithm is $O(N \log N) + O(p^2N)$. This indicates that the entire algorithm is superfast.

We also point out that similar techniques are used in [33], where the low-rank property is exploited through the block columns without diagonals (called *neutered block columns* in [33]), in contrast with the off-diagonal blocks here. The compressions of either the neutered blocks or the off-diagonal blocks both give data-sparse representations which enable fast factorizations of the Cauchy-like matrices. In fact, corresponding to neutered block rows or columns, there are also matrix representations called hierarchically semiseparable (HSS) matrices [14] which are usually more complicated structures than SSS matrices.

1.3. Overview. We will discuss the displacement structure and the transformation from a Toeplitz problem to a Cauchy-like problem in section 2. The low-rank property of this Cauchy-like problem is then exploited. Section 3 then gives a linear complexity solver using the SSS structure. In section 4, we will present an algorithm for fast construction of SSS structures. We will then analyze the complexity in section 5 and use some numerical experiments to demonstrate the efficiency and the practical stability. All algorithms have been implemented in Fortran 90. Section 6 draws some conclusions.

2. Displacement structures and low-rank property.

2.1. Cauchy-like systems. Given a Toeplitz system (1.1), we can use a displacement structure to transform it into a Cauchy-like system. Define

$$Z_\delta = \begin{pmatrix} 0 & 0 & \cdots & 0 & \delta \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix},$$

and let $\Omega = Z_1$ and $\Lambda = Z_{-1}$ in (1.3). Kailath, Kung, and Morf [31] have shown that every Toeplitz matrix satisfies the displacement equation (1.3) with $A \cdot B$, having nonzero entries only in its first row and last column, to be a matrix of rank at most 2. Hence the displacement rank of a Toeplitz matrix is at most 2 with respect to Z_1 and Z_{-1}. The following result can be found in [28].

PROPOSITION 2.1. *Let $\hat{C} \in \mathbf{R}^{N\times N}$ be a matrix satisfying the displacement equation*

$$Z_1\hat{C} - \hat{C}Z_{-1} = UV, \tag{2.1}$$

where $U \in \mathbf{R}^{n\times\alpha}$ and $V \in \mathbf{R}^{\alpha\times n}$. Then $\mathcal{F}\hat{C}D_0^{-1}\mathcal{F}^H$ is a Cauchy-like matrix satisfying

$$\mathcal{D}_1(\mathcal{F}\hat{C}D_0^{-1}\mathcal{F}^H) - (\mathcal{F}\hat{C}D_0^{-1}\mathcal{F}^H)\mathcal{D}_{-1} = (\mathcal{F}U)\left(VD_0^H\mathcal{F}^H\right), \tag{2.2}$$

where $\mathcal{F} = \sqrt{\frac{1}{N}}(\omega^{2(k-1)(j-1)})_{1\le k,j\le N}$ is the normalized inverse discrete Fourier transform matrix, $\omega = e^{\frac{\pi i}{N}}$, and

$$\mathcal{D}_1 = \mathrm{diag}(1, \omega^2, \ldots, \omega^{2(N-1)}), \quad \mathcal{D}_{-1} = \mathrm{diag}(\omega, \omega^3, \ldots, \omega^{2N-1}),$$
$$D_0 = \mathrm{diag}(1, \omega, \ldots, \omega^{N-1}).$$

Here $\alpha \le 2$. This proposition suggests that for a Toeplitz matrix T, one can convert it into the Cauchy-like matrix in (2.2). Therefore the Toeplitz system (1.1) can be readily transformed into a new system

$$C\tilde{x} = \tilde{b}, \tag{2.3}$$

where C has the form

$$C = \left(\frac{u_i^T v_j}{\omega^{2i} - \omega^{2j+1}} \right)_{1 \le i,j \le N} \qquad (u_i,\ v_j \in \mathbf{R}^{\alpha}) \ . \tag{2.4}$$

In section 3 we will present a fast solver for (2.3). After obtaining $\tilde{x}$ we will then recover x with an FFT again. All the stages involving FFT are stable and cost $O(N \log N)$. The solver for (2.3) has a linear complexity and turns out to be practically stable. Thus the total cost of our algorithm is bounded by $O(N \log N) + O(Np^2)$, where p is some parameter that will be described below. This indicates our method is a superfast one with practical stability.

2.2. Low-rank property of Cauchy-like matrices. In this section, we will show a low-rank property of C, i.e., every off-diagonal block of C has a low numerical rank. This property is the basis of the superfast SSS solver in section 3.

First, a simple numerical experiment can give us an idea of the low-rank property of C. To find out the numerical ranks we can use one of the following tools:

(1) τ-*accurate SVD*: singular values less than τ are dropped if τ is an absolute tolerance, or singular values less than τ times the largest singular value are dropped if τ is a relative tolerance.

(2) τ-*accurate QR*:

$$A \approx QR, \ A : m \times n, \ \ Q : m \times k, \ R : k \times k, \ k \le l \equiv \min(m, n),$$

which is obtained in the following way. Compute the exact QR factorization of matrix $A = \hat{Q}\hat{R}$, where $\hat{Q}$ is $m \times l$ and $\hat{R}$ is $l \times n$ with diagonal entries satisfying $\hat{R}_{11} \ge \hat{R}_{22} \ge \cdots \ge \hat{R}_{ll}$. Then obtain R by dropping all rows of $\hat{R}$ with diagonal entries less than τ if τ is an absolute tolerance, or with diagonal entries less than $\tau \hat{R}_{11}$ if τ is a relative tolerance. Drop relevant columns of $\hat{Q}$ accordingly to obtain Q.

Later, by ranks we mean numerical ranks. Here we take some random Toeplitz matrices in different sizes. Then we transform them into Cauchy-like matrices C and compute the numerical ranks of their off-diagonal blocks. For simplicity, we compute the ranks for blocks $C(1:d, d+1:N)$, $C(1:2d, 2d+1:N)$, $\ldots$, $C(1:kd, kd+1:N), \ldots$, where d is a fixed integer, and these blocks are numbered as block numbers $1, 2, \ldots, k$ as shown in Figure 2.1.

Here, $k = 8$ off-diagonal blocks for each of three $N \times N$ Cauchy-like matrices are considered. See Table 2.1 for the numerical ranks, where we use the τ-accurate SVD with τ to be an absolute tolerance.

We can see that the numerical ranks are relatively small as compared to the block sizes. And when we double the dimension of the matrix, the numerical ranks do not increase much. This is more significant when a larger τ is used.

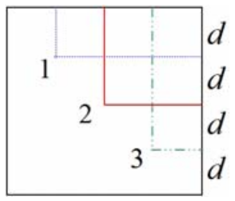

FIG. 2.1. *Off-diagonal blocks (numbered as* $1, 2, 3$*). Upper triangular part only.*

TABLE 2.1
Off-diagonal numerical ranks of Cauchy-like matrices transformed from some Toeplitz matrices with absolute tolerance $\tau = 10^{-9}$.

	$N = 640$		$N = 1280$		$N = 2560$	
Block #	Block size	Rank	Block size	Rank	Block size	Rank
1	80×560	37	160×1120	44	320×2240	52
2	160×480	43	320×960	50	640×1920	57
3	240×400	45	480×800	53	960×1600	60
4	320×320	46	640×640	53	1280×1280	61
5	400×240	46	800×480	52	1600×960	60
6	480×160	43	960×320	50	1920×640	58
7	560×80	37	1120×160	44	2240×320	52

The low-rank property can be verified theoretically in the following way. We first consider a special case of (2.4) where all $u_i^T v_j = 1$. We show that the following Cauchy matrix has low-rank off-diagonal blocks:

$$C_0 = \left(\frac{1}{\omega^{2i} - \omega^{2j+1}} \right)_{1 \le i,j \le N} \tag{2.5}$$

The central idea is similar to that in the fast multipole method [25, 8], which implies that with *well-separated* points (see, e.g., Figure 2.2) an interaction matrix is numerically low-rank.

Here we introduce two sets of points $\{\lambda_k\}_{k=1}^N \equiv \{\omega^{2k}\}_{k=1}^N$ and $\{\eta_k\}_{k=1}^N \equiv \{\omega^{2k+1}\}_{k=1}^N$ on the unit circle. When we consider an off-diagonal block of C_0 as follows:

$$G = \left(\frac{1}{\lambda_i - \eta_j} \right)_{1 \le i \le p,\ p+1 \le j \le N}$$

we can show G is (numerically) low-rank. In fact G corresponds to two well-separated sets $\{\lambda_k\}_{k=1}^p$ and $\{\eta_j\}_{k=p+1}^N$; that is, there exists a point $c \in \mathbb{C}$ such that

$$\begin{array}{rcll} |\lambda_i - c| & > & d + e, & i = 1, \dots, p, \\ |\eta_j - c| & < & d, & j = p+1, \dots, N, \end{array} \tag{2.6}$$

where d, e are positive constants. Consider the expansion

$$\frac{1}{\lambda_i - \eta_j} = \frac{1}{\lambda_i - c} \frac{1}{1 - \frac{\eta_j - c}{\lambda_i - c}} = \sum_{k=0}^{r} \frac{(\eta_j - c)^k}{(\lambda_i - c)^{k+1}} + O\left(\left(\frac{\eta_j - c}{\lambda_i - c} \right)^{r+1} \right) \tag{2.7}$$

$$= \sum_{k=0}^{r} \frac{(\eta_j - c)^k}{(\lambda_i - c)^{k+1}} + \varepsilon, \tag{2.8}$$

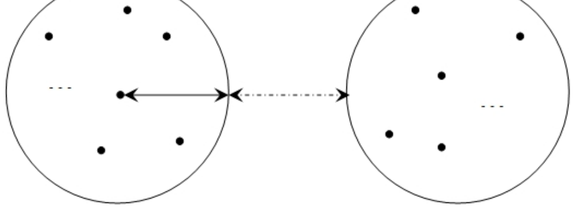

FIG. 2.2. *Well-separated sets in the plane.*

where r is a number such that the error term $|\varepsilon| = |O((\frac{\eta_j - c}{\lambda_i - c})^{r+1})|$ is bounded by a given tolerance. We have the estimate

$$\left|\frac{\eta_j - c}{\lambda_i - c}\right|^{r+1} < \left(\frac{d}{d+e}\right)^{r+1},$$

which enables us to find an appropriate r according to the tolerance. Thus

$$G = \left(\sum_{k=0}^{r} \frac{(\eta_j - c)^k}{(\lambda_i - c)^{k+1}}\right)_{1\le i\le p,\ p+1\le j\le N} + \hat{\varepsilon}$$

(2.9)

$$= \begin{pmatrix} \frac{1}{\lambda_1 - c} & \frac{1}{(\lambda_1-c)^2} & \cdots & \frac{1}{(\lambda_1-c)^{r+1}} \\ \frac{1}{\lambda_2 - c} & \frac{1}{(\lambda_2-c)^2} & \cdots & \frac{1}{(\lambda_2-c)^{r+1}} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{1}{\lambda_p - c} & \frac{1}{(\lambda_p-c)^2} & \cdots & \frac{1}{(\lambda_p-c)^{r+1}} \end{pmatrix} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ (\eta_{p+1}-c) & (\eta_{p+2}-c) & \cdots & (\eta_N - c) \\ \vdots & \vdots & \cdots & \vdots \\ (\eta_{p+1}-c)^r & (\eta_{p+2}-c)^r & \cdots & (\eta_N-c)^r \end{pmatrix} + \hat{\varepsilon}.$$

Therefore the numerical rank of G is at most $r+1$, up to an error $\hat{\varepsilon}$.

Now we can return to the Cauchy-like matrix (2.4). A similar argument shows that any off-diagonal block of C satisfies

$$\hat{G} \approx \left(\sum_{k=0}^{r} (u_i^T v_j) \frac{(\eta_j - c)^k}{(\lambda_i - c)^{k+1}}\right)_{1\le i\le p,\ p+1\le j\le N}$$

$$= \begin{pmatrix} \frac{u_1^T}{\lambda_1 - c} & \frac{u_1^T}{(\lambda_1-c)^2} & \cdots & \frac{u_1^T}{(\lambda_1-c)^{r+1}} \\ \frac{u_2^T}{\lambda_2 - c} & \frac{u_2^T}{(\lambda_2-c)^2} & \cdots & \frac{u_2^T}{(\lambda_2-c)^{r+1}} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{u_p^T}{\lambda_p - c} & \frac{u_p^T}{(\lambda_p-c)^2} & \cdots & \frac{u_p^T}{(\lambda_p-c)^{r+1}} \end{pmatrix} \begin{pmatrix} v_1 & v_2 & \cdots & v_q \\ (\eta_{p+1}-c)\,v_1 & (\eta_{p+2}-c)\,v_2 & \cdots & (\eta_N - c)\,v_N \\ \vdots & \vdots & \cdots & \vdots \\ (\eta_{p+1}-c)^r v_1 & (\eta_{p+2}-c)^r v_2 & \cdots & (\eta_N-c)^r v_N \end{pmatrix}.$$

That is, we replace the entries of the two matrix factors in (2.9) with appropriate vectors. Thus the numerical rank of $\hat{G}$ will be no larger than $\alpha(r+1)$, which will be relatively small as compared to N.

3. SSS structures and superfast SSS solver.

3.1. SSS representations.

To take advantage of the low-rank property of the Cauchy-like matrix C, we can use SSS structures introduced by Chandrasekaran et al. [12, 13]. The SSS structure nicely captures the ranks of off-diagonal blocks of a matrix such as shown in Figure 2.1.

A matrix $A \in \mathbf{C}^{M\times\tilde{M}}$ satisfies the SSS structure if there exist $2n$ positive integers $m_1, \ldots, m_n$, and $\tilde{m}_1, \ldots, \tilde{m}_n$ with $M = m_1 + \cdots + m_n$ and $\tilde{M} = \tilde{m}_1 + \cdots + \tilde{m}_n$ to block-partition A as $A = (A_{i,j})_{k\times k}$, where $A_{ij} \in \mathbb{C}^{m_i \times \tilde{m}_j}$ satisfies

$$A_{ij} = \begin{cases} D_i & \text{if } i = j, \\ U_i W_{i+1} \cdots W_{j-1} V_j^H & \text{if } i < j, \\ P_i R_{i-1} \cdots R_{j+1} Q_j^H & \text{if } i > j. \end{cases} \tag{3.1}$$

Here the superscript H denotes the Hermitian transpose and empty products are defined to be identity matrices. The matrices $\{U_i\}_{i=1}^{n-1}$, $\{V_i\}_{i=2}^{n}$, $\{W_i\}_{i=2}^{n-1}$, $\{P_i\}_{i=2}^{n}$,

TABLE 3.1
Dimensions of matrices in (3.1).

Matrix	U_i	V_i	W_i	P_i	Q_i	R_i
Dimensions	$m_i \times k_i$	$\tilde{m}_i \times k_{i-1}$	$k_{i-1} \times k_i$	$m_i \times l_i$	$\tilde{m}_i \times l_{i+1}$	$l_{i+1} \times l_i$

$\{Q_i\}_{i=1}^{n-1}$, $\{R_i\}_{i=2}^{n-1}$, and $\{D_i\}_{i=1}^{n}$ are called *generators* for the SSS structure and their dimensions are defined in Table 3.1.

As an example, the matrix A with $n = 4$ has the form

$$A = \begin{pmatrix} D_1 & U_1V_2^H & U_1W_2V_3^H & U_1W_2W_3V_4^H \\ P_2Q_1^H & D_2 & U_2V_3^H & U_2W_3V_4^H \\ P_3R_2Q_1^H & P_3Q_2^H & D_3 & U_3V_4^H \\ P_4R_3R_2Q_1^H & P_4R_3Q_2^H & P_4Q_3^H & D_4 \end{pmatrix}. \tag{3.2}$$

The SSS representation (3.2) is related to the off-diagonal blocks in Figure 2.1 in the way that the upper off-diagonal block numbers $1, 2$, and 3 are

$$U_1 \begin{pmatrix} V_2^H & W_2V_3^H & W_2W_3V_4^H \end{pmatrix}, \begin{pmatrix} U_1W_2 \\ U_2 \end{pmatrix} \begin{pmatrix} V_3^H & W_3V_4^H \end{pmatrix}, \begin{pmatrix} U_1W_2W_3 \\ U_2W_3 \\ U_3 \end{pmatrix} V_4^H.$$

Appropriate row and column bases of the off-diagonal blocks are clearly reflected.

The SSS structure depends on the sequences $\{m_i\}$ and $\{\tilde{m}_i\}$ and the SSS generation scheme. If A is a square matrix ($M = \tilde{M}$), then we can have a simpler situation $m_i = \tilde{m}_i$, $i = 1, \ldots, n$. SSS matrices are closed under addition, multiplication, inversion, etc., although the sizes of the generators may increase.

While any matrix can be represented in this form for sufficiently large k_i's and l_i's, the column dimensions of U_i's and P_i's, respectively, our main focus will be on SSS matrices which have low-rank off-diagonal blocks and have generators with k_i's and l_i's to be close to those ranks. We say these SSS matrices are *compact*. Particularly true for Cauchy-like matrices, they can have compact SSS forms. Using SSS structures, we can take advantage of the superfast SSS system solver in [12, 13] to solve the Cauchy-like systems. The solver is efficient when the SSS form is compact, and is practically stable. The solver shares similar ideas with that for banded plus semiseparable systems in [11].

Here we briefly describe the main ideas of the solver in [12, 13]. We consider solving the linear system $Ax = b$, where $A \in \mathbf{C}^{N \times N}$ satisfies (3.1) and b itself is an unstructured matrix. The solver computes an implicit ULV^H decomposition of A, where U and V are orthogonal matrices.

Before we present the formal algorithm, we demonstrate the key ideas on a 4×4 block matrix example.

3.2. SSS solver: 4×4 example. Let the initial system $Ax = b$ be partitioned as follows:

$$\begin{pmatrix} D_1 & U_1V_2^H & U_1W_2V_3^H & U_1W_2W_3V_4^H \\ P_2Q_1^H & D_2 & U_2V_3^H & U_2W_3V_4^H \\ P_3R_2Q_1^H & P_3Q_2^H & D_3 & U_3V_4^H \\ P_4R_3R_2Q_1^H & P_4R_3Q_2^H & P_4Q_3^H & D_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} - \begin{pmatrix} 0 \\ P_2 \\ P_3R_2 \\ P_4R_3R_2 \end{pmatrix} \xi, \tag{3.3}$$

where the dimensions of the generators follow those in Table 3.1 with $m_i = \tilde{m}_i$ and the vector $\xi = 0$. The extra zero vector ξ on the right-hand side of (3.3) has been added for the purpose of a general recursive pattern.

The algorithm has two main stages, compression (or elimination) and merging, depending on the relationship between k_i (l_i) and m_i in the intermediate procedure.

3.2.1. Compression. At the beginning, $k_1 < m_1$ because of the low-rank property described earlier. We apply a unitary transformation q_1^H to U_1 so that the first $m_1 - k_1$ rows of U_1 become zeros:

$$q_1^H U_1 = \begin{array}{cc} \begin{pmatrix} 0 \\ \hat{U}_1 \end{pmatrix} & \begin{array}{c} m_1 - k_1 \\ k_1 \end{array} \end{array}. \tag{3.4}$$

Now we multiply q_1^H to the first m_1 equations of the system

$$\begin{pmatrix} q_1^H & 0 \\ 0 & I \end{pmatrix} Ax = \begin{pmatrix} q_1^H & 0 \\ 0 & I \end{pmatrix} b - \begin{pmatrix} 0 \\ P_2 \\ P_3R_2 \\ P_4R_3R_2 \end{pmatrix} \xi.$$

We pick another unitary transformation w_1^H to lower-triangularize $q_1^H D_1$, the $(1,1)$ diagonal block A, i.e.,

$$\left(q_1^H D_1\right) w_1^H = \begin{array}{c} \\ m_1 - k_1 \\ k_1 \end{array} \begin{array}{c} \begin{array}{cc} m_1 - k_1 & k_1 \end{array} \\ \begin{pmatrix} D_{11} & 0 \\ D_{21} & D_{22} \end{pmatrix} \end{array}.$$

Then system (3.3) becomes

$$\begin{pmatrix} q_1^H & 0 \\ 0 & I \end{pmatrix} A \begin{pmatrix} w_1^H & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} w_1 & 0 \\ 0 & I \end{pmatrix} x = \begin{pmatrix} q_1^H & 0 \\ 0 & I \end{pmatrix} b - \begin{pmatrix} 0 \\ P_2 \\ P_3R_2 \\ P_4R_3R_2 \end{pmatrix} \xi,$$

which can be rewritten as

$$\begin{pmatrix} D_{11} & 0 & 0 & 0 & 0 \\ D_{21} & D_{22} & \hat{U}_1 V_2^H & \hat{U}_1 W_2 V_3^H & \hat{U}_1 W_2 W_3 V_4^H \\ P_2 Q_{11}^H & P_2 \hat{Q}_1^H & D_2 & U_2 V_3^H & U_2 W_3 V_4^H \\ P_3R_2 Q_{11}^H & P_3R_2 \hat{Q}_1^H & P_3 Q_2^H & D_3 & U_3 V_4^H \\ P_4R_3R_2 Q_{11}^H & P_4R_3R_2 \hat{Q}_1^H & P_4R_3 Q_2^H & P_4 Q_3^H & D_4 \end{pmatrix} \begin{pmatrix} z_1 \\ \hat{x}_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

$$= \begin{pmatrix} \beta_1 \\ \gamma_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ P_2 \\ P_3R_2 \\ P_4R_3R_2 \end{pmatrix} \xi,$$

where we have used the partitions

$$w_1 x_1 = \begin{array}{c} m_1 - k_1 \\ k_1 \end{array} \begin{pmatrix} z_1 \\ \hat{x}_1 \end{pmatrix}, \quad q_1^H b_1 = \begin{array}{c} m_1 - k_1 \\ k_1 \end{array} \begin{pmatrix} \beta_1 \\ \gamma_1 \end{pmatrix}, \quad \text{and} \quad w_1 Q_1 = \begin{array}{c} m_1 - k_1 \\ k_1 \end{array} \begin{pmatrix} Q_{11} \\ \hat{Q}_1 \end{pmatrix}.$$

At this point, we can solve for z_1 from the system of equations $D_{11}z_1 = \beta_1$. We also subtract $D_{21}z_1$ from the right-hand side to obtain $\hat{b}_1 = \gamma_1 - D_{21}z_1$. Then we can discard the first $m_1 - k_1$ rows and columns of the coefficient matrix of the system to obtain

$$\begin{pmatrix} D_{22} & \hat{U}_1 V_2^H & \hat{U}_1 W_2 V_3^H & \hat{U}_1 W_2 W_3 V_4^H \\ P_2 \hat{Q}_1^H & D_2 & U_2 V_3^H & U_2 W_3 V_4^H \\ P_3 R_2 \hat{Q}_1^H & P_3 Q_2^H & D_3 & U_3 V_4^H \\ P_4 R_3 R_2 \hat{Q}_1^H & P_4 R_3 Q_2^H & P_4 Q_3^H & D_4 \end{pmatrix} \begin{pmatrix} \hat{x}_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} - \begin{pmatrix} 0 \\ P_2 \\ P_3 R_2 \\ P_4 R_3 R_2 \end{pmatrix} \hat{\xi},$$

where $\hat{\xi} = \xi + Q_{11}^H z_1$. This new system has a similar structure to the original one but with smaller dimension. We can continue to solve it by recursion, if further compressions of the blocks such as (3.4) are possible. Note the actual solution, say, x_1, can be recovered by

$$x_1 = w_1^H \begin{pmatrix} z_1 \\ \hat{x}_1 \end{pmatrix}.$$

3.2.2. Merging. During the recursive eliminations there are situations when k_i is no longer smaller than m_i and no further compression is possible. We are then unable to introduce more zeros into the system. Now we proceed by merging appropriate block rows and columns of the matrix. As an example we can merge the first two block rows and columns and rewrite the system of equations as follows:

$$\begin{pmatrix} \begin{pmatrix} D_1 & U_1 V_2^H \\ P_2 Q_1^H & D_2 \end{pmatrix} & \begin{pmatrix} U_1 W_2 \\ U_2 \end{pmatrix} V_3^H & \begin{pmatrix} U_1 W_2 \\ U_2 \end{pmatrix} W_3 V_4^H \\ P_3 \begin{pmatrix} Q_1 R_2^H \\ Q_2 \end{pmatrix}^H & D_3 & U_3 V_4^H \\ P_4 R_3 \begin{pmatrix} Q_1 R_2^H \\ Q_2 \end{pmatrix}^H & P_4 Q_3^H & D_4 \end{pmatrix} \begin{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ x_3 \\ x_4 \end{pmatrix}$$

$$= \begin{pmatrix} \begin{pmatrix} b_1 \\ b_2 - P_2 \hat{\xi} \end{pmatrix} \\ b_3 \\ b_4 \end{pmatrix} - \begin{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ P_3 \\ P_4 R_3 \end{pmatrix} (R_2 \hat{\xi}).$$

Hence the system becomes

$$\begin{pmatrix} \hat{D}_1 & \hat{U}_1 V_3^H & \hat{U}_1 W_3 V_4^H \\ P_3 \hat{Q}_1^H & D_3 & U_3 V_4^H \\ P_4 R_3 \hat{Q}_1^H & P_4 Q_3^H & D_4 \end{pmatrix} \begin{pmatrix} \hat{x}_1 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ b_3 \\ b_4 \end{pmatrix} - \begin{pmatrix} 0 \\ P_3 \\ P_4 R_3 \end{pmatrix} (\tilde{\xi}),$$

where

$$\hat{D}_1 = \begin{pmatrix} D_1 & U_1 V_2^H \\ P_2 Q_1^H & D_2 \end{pmatrix}, \quad \hat{U}_1 = \begin{pmatrix} U_1 W_2 \\ U_2 \end{pmatrix}, \quad \hat{Q}_1 = \begin{pmatrix} Q_1 R_2^H \\ Q_2 \end{pmatrix},$$
$$\hat{x}_1 = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \hat{b}_1 = \begin{pmatrix} b_1 \\ b_2 - P_2 \tau \end{pmatrix}, \quad \tilde{\xi} = R_2 \hat{\xi}.$$

The number of block rows/columns is reduced by one. Further compressions become possible and we can proceed to solve the system recursively. In the case $n = 1$, we have the system $D_1 x_1 = b_1 - 0\xi$, which is solved directly.

3.3. General solve algorithm. We now present a short description of the general algorithm. The procedure in the 4×4 example can be directly extended to a general system. We assume that the matrix A is in compact SSS form represented by the generators $\{U_i\}_{i=1}^{n-1}$, $\{V_i\}_{i=2}^{n}$, $\{W_i\}_{i=2}^{n-1}$, $\{P_i\}_{i=2}^{n}$, $\{Q_i\}_{i=1}^{n-1}$, $\{R_i\}_{i=2}^{n-1}$, and $\{D_i\}_{i=1}^{n}$ as in (3.1). We also partition $x = (x_i)$ and $b = (b_j)$ such that x_i and b_i have m_i rows. As in the 4×4 example, there are two stages at each step of the recursion.

In the compression stage, we perform orthogonal eliminations on both sides of A to create an $(m_1 - k_1) \times (m_1 - k_1)$ lower triangular submatrix at the top left corner of A. Then we solve a small triangular system and obtain the first few components of the solution vector. At this stage, we are left with a new system with less unknowns; hence we can carry out a recursion.

In the merging stage, we merge the first two block rows and columns of A while still maintaining the SSS structure. The numbers of block rows and columns are reduced by one.

Combining these two stages, we can proceed with recursion to solve the system. When $n = 1$ we can solve the linear system directly with standard solvers.

The SSS solver has a complexity $O(Np^2)$ [12, 13], where p is the maximum numerical rank of the off-diagonal blocks of A, as compared to the traditional $O(N^3)$ cost for a general dense $N \times N$ matrix. We use only orthogonal transformations and a single substitution in the SSS solver. Although a formal proof for the backward stability is not yet available, the solver is shown to be practically stable. The reader is referred to [12, 13] for more discussions on the stability.

4. Fast construction of SSS representation for *C*. According to section 3, a system in compact SSS form can be solved very efficiently. We can thus use that algorithm to solve the Cauchy-like system (2.3), provided that C can be quickly written in SSS form. Therefore we try to find an efficient construction scheme. Here we provide a divide-and-conquer SSS construction scheme by using the fast merging and splitting strategy in [13]. If we know further that the matrix has low-rank off-diagonal blocks and it is easy to compress the off-diagonal blocks, then the construction can be superfast. Here we will concentrate on this situation as Cauchy-like matrices have this low-rank property.

We first present a general divide-and-conquer construction algorithm and then describe a fast shifting strategy to compress the off-diagonal blocks of C.

4.1. General divide-and-conquer construction algorithm. In this section, we discuss the construction of the SSS structure of a matrix A, when the partition sequence $\{m_i\}_{i=1}^{n}$ is given. The general construction methods can be applied to any unstructured matrix, thus proving that any matrix has an SSS structure. (Of course, k_i and l_i will usually be large in this case, precluding any speed-ups.) These methods can be viewed as specific ways to make the realization algorithm of [18] more efficient.

Suppose we are given an $N \times N$ matrix A and a partition sequence $\{m_i\}_{i=1}^{n}$ with $\sum_{i=1}^{n} m_i = N$. Starting with an appropriate sequence $\{\tilde{m}_1, \tilde{m}_2\}$, where $\sum_{i=1}^{k} m_i = \tilde{m}_1$ and $\sum_{i=k+1}^{n} m_i = \tilde{m}_2$, we can first partition A into a 2×2 block matrix and then construct a simple SSS form

$$A = \begin{array}{c|cc} & \tilde{m}_1 & \tilde{m}_2 \\ \hline \tilde{m}_1 & D_1 & B \\ \tilde{m}_2 & E & D_2 \end{array} = \begin{array}{cc} \tilde{m}_1 & \tilde{m}_2 \\ \left(\begin{matrix} D_1 \\ P_2 Q_1^H \end{matrix}\right. & \left.\begin{matrix} U_1 V_2^H \\ D_2 \end{matrix}\right) \end{array}, \tag{4.1}$$

where

$$B = U_1 V_2^H \equiv U_1 \left(\Sigma_1 F_1^H\right), \quad F = P_2 Q_1^H \equiv P_2 \left(\Sigma_1 F_1^H\right) \tag{4.2}$$

are the low-rank SVDs of the off-diagonal blocks B and E. Note if the compressions are done by τ-accurate SVD approximations or rank-revealing QR decompositions, then appropriate "$=$" signs should be replaced by "$\approx$" signs. We now split either the $(1,1)$ block or the $(2,2)$ block to obtain a 3×3 block SSS matrix. For instance, we can split the $(1,1)$ block according to an appropriate new sequence $\{\hat{m}_1,\hat{m}_2,\hat{m}_3\}$ as follows, where $\hat{m}_1+\hat{m}_2=m_1,\ \hat{m}_3=m_2$:

$$\begin{pmatrix} D_1^{\mathbf{new}} & U_1^{\mathbf{new}}(V_2^{\mathbf{new}})^H \\ P_2^{\mathbf{new}}(Q_1^{\mathbf{new}})^H & D_2^{\mathbf{new}} \end{pmatrix} = D_1,\ \begin{pmatrix} U_1^{\mathbf{new}}W_2^{\mathbf{new}} \\ U_2^{\mathbf{new}} \end{pmatrix} = U_1,\ (V_3^{\mathbf{new}})^H = V_2^H,$$

$$(R_2^{\mathbf{new}}(Q_1^{\mathbf{new}})^H (Q_2^{\mathbf{new}})^H) = Q_1^H,\ P_3^{\mathbf{new}} = P_2,\ D_3^{\mathbf{new}} = D_2,$$

where the new generators (marked by the superscript **new**) introduced based on (4.1) can be determined in the following way. First, we partition the matrices for the old first block conformally with the two new blocks as

$$\begin{pmatrix} D_1^{11} & D_1^{12} \\ D_1^{21} & D_1^{22} \end{pmatrix} = D_1,\ \begin{pmatrix} U_1^1 \\ U_1^2 \end{pmatrix} = U_1,\ ((Q_1^1)^H (Q_1^2)^H) = Q_1^H.$$

We can then identify from these and the previous equations that

$$D_1^{\mathbf{new}} = D_1^{11},\ D_2^{\mathbf{new}} = D_1^{22},\ U_2^{\mathbf{new}} = U_1^2,\ V_3^{\mathbf{new}} = V_2,\ Q_2^{\mathbf{new}} = Q_1^2,\ P_3^{\mathbf{new}} = P_2,\ D_3^{\mathbf{new}} = D_2.$$

The remaining matrices satisfy

$$(D_1^{12}U_1^1) = U_1^{\mathbf{new}}((V_2^{\mathbf{new}})^H W_2^{\mathbf{new}}),\ \begin{pmatrix} D_1^{21} \\ (Q_1^1)^H \end{pmatrix} = \begin{pmatrix} P_2^{\mathbf{new}} \\ R_2^{\mathbf{new}} \end{pmatrix} (Q_1^{\mathbf{new}})^H. \tag{4.3}$$

By factorizing the left-hand side matrices using numerical tools such as the SVD and rank-revealing QR, these two equations allow us to compute those remaining matrices for the new blocks.

A is thus in the new form

$$A = \begin{array}{c} \\ \hat{m}_1 \\ \hat{m}_2 \\ \hat{m}_3 \end{array} \begin{array}{c} \begin{array}{ccc} \hat{m}_1 & \hat{m}_2 & \hat{m}_3 \end{array} \\ \begin{pmatrix} D_1^{\mathbf{new}} & U_1^{\mathbf{new}}(V_2^{\mathbf{new}})^H & U_1^{\mathbf{new}}W_2^{\mathbf{new}}(V_3^{\mathbf{new}})^H \\ P_2^{\mathbf{new}}(Q_1^{\mathbf{new}})^H & D_2^{\mathbf{new}} & U_2^{\mathbf{new}}(V_3^{\mathbf{new}})^H \\ P_2^{\mathbf{new}}R_2^{\mathbf{new}}(Q_1^{\mathbf{new}})^H & P_3^{\mathbf{new}}(Q_2^{\mathbf{new}})^H & D_3^{\mathbf{new}} \end{pmatrix} \end{array}.$$

We can use similar techniques if we want to split the second row and column of (4.1).

We can continue this by either splitting the first block row and block column or the last ones using the above techniques, or splitting any middle block row and block column similarly. Then we will be able to construct the desired SSS representation according to the given sequence $\{m_i,\ i=1,2,\ldots,n\}$.

The general construction can be organized with bisection. The major cost is in compressions of off-diagonal blocks of the form

$$D_i^{12} = X_i Y_i^H, \tag{4.4}$$

where X_i and Y_i are tall and thin, and D_i^{12} is an off-diagonal block of

$$D_i = \begin{pmatrix} D_i^{11} & D_i^{12} \\ D_i^{21} & D_i^{22} \end{pmatrix}.$$

The compression (4.4) can be achieved by a τ-accurate QR factorization.

The construction is also practically stable in our implementation.

4.2. Compression of off-diagonal blocks. For a general matrix with low-rank off-diagonal blocks, the SSS construction can cost $O(N^2p)$ as a compression such as (4.2), (4.3), and (4.4) can take $O(K^2p)$, where K is the dimension of the block being compressed. However, for the Cauchy-like matrix C in (2.3) the compressions can be precomputed.

THEOREM 4.1. *The compression of the off-diagonal block*

$$G = \left(\frac{u_i^T \cdot v_j}{\lambda_i - \eta_j}\right)_{1\le i\le p,\ q\le j\le N} = XY^H \tag{4.5}$$

of C can be obtained by the compression of the corresponding off-diagonal block

$$G = \left(\frac{1}{\lambda_i - \eta_j}\right)_{1\le i\le p,\ q\le j\le N} = X_0Y_0^H$$

of C_0, where the column dimension of X is no larger than twice the size of the column dimension of X_0.

Proof. Assume the off-diagonal block $G_0 = (\frac{1}{\lambda_i - \eta_j})_{1\le i\le p,\ q\le j\le N}$ is in compressed form $X_0Y_0^H$, and the (i,j) entry is $\frac{1}{\lambda_i-\eta_j} = X_{i,:}Y_{j,:}^H$, where $X_{i,:}$ ($Y_{j,:}$) denotes the ith row of X (Y). As $\alpha \le 2$ for simplicity we fix $\alpha = 2$ in (2.1). Then the corresponding off-diagonal block in C is

$$\begin{aligned}
G &= \left(\frac{u_i^T \cdot v_j}{\lambda_i - \eta_j}\right)_{1\le i\le p,\ q\le j\le N} = \left(\frac{u_{i1}v_{j1} + u_{i2}v_{j2}}{\lambda_i - \eta_j}\right)_{1\le i\le p,\ q\le j\le N} \\
&= \left((u_{i1}v_{j1} + u_{i2}v_{j2})\, X_{i,:}Y_{j,:}^H\right)_{1\le i\le p,\ q\le j\le N} \\
&= \left(\begin{pmatrix} u_{i1}X_{i,:} & u_{i2}X_{i,:}\end{pmatrix}\begin{pmatrix} v_{j1}Y_{j,:}^H \\ v_{j2}Y_{j,:}^H\end{pmatrix}\right)_{1\le i\le p,\ q\le j\le N} \\
&= \begin{pmatrix} u_{11}X_{1,:} & u_{12}X_{1,:} \\ \vdots & \vdots \\ u_{p1}X_{i,:} & u_{p2}X_{i,:}\end{pmatrix}\begin{pmatrix} v_{q1}Y_{q,:}^H & \cdots & v_{N1}Y_{N,:}^H \\ v_{q2}Y_{q,:}^H & \cdots & v_{N2}Y_{N,:}^H\end{pmatrix} \\
&\equiv XY^H.
\end{aligned}$$

That is, we get a compression of G. ◻

Theorem 4.1 indicates that we can convert the compressions of the off-diagonal blocks of C to be the compressions of those of C_0 which is independent of the actual entries of the Toeplitz matrix T. This means the compressions of off-diagonal blocks of C_0 can be precomputed. The precomputation can be done in $O(N^2p)$ flops by a rank-revealing QR factorization such as in [27]. It is possible to reduce the cost to $O(N \log N)$ due to the fact that the compression of a large off-diagonal block can be obtained by that of small ones. This can be seen implicitly from the following subsection.

4.3. Compressions of off-diagonal blocks in precomputation. We further present a shifting strategy to reduce the cost of the compressions in the precomputation. The significance of this shifting strategy is to relate the compressions of large off-diagonal block of C_0 to those of small ones. That is, in different splitting stages of the SSS construction of C, the compressions of off-diagonal blocks with different sizes can be related.

For simplicity, we look at the example of partitioning C_0 into 4×4 blocks. Assume in the first cut that we can partition C_0 into 2×2 blocks with equal dimensions as in the following:

$$
\begin{aligned}
C_0 &= \left(\frac{1}{\omega^{2i}-\omega^{2j+1}}\right)_{1\le i,\ j\le N} = \left(\begin{array}{c||c} C_{0;1,1} & C_{0;1,2} \\ \hline\hline C_{0;2,1} & C_{0;2,2} \end{array}\right) \\
&= \left(\begin{array}{ccc||ccc}
\frac{1}{\omega^2-\omega^3} & \cdots & \frac{1}{\omega^2-\omega^{4k+1}} & \frac{1}{\omega^2-\omega^{4k+3}} & \cdots & \frac{1}{\omega^2-\omega^{2N+1}} \\
\vdots & & \vdots & \vdots & \cdots & \vdots \\
\frac{1}{\omega^{4k}-\omega^3} & \cdots & \frac{1}{\omega^{4k}-\omega^{4k+1}} & \frac{1}{\omega^{4k}-\omega^{4k+3}} & \cdots & \frac{1}{\omega^{4k}-\omega^{2N+1}} \\
\hline\hline
\multicolumn{3}{c||}{C_{0;2,1}} & \multicolumn{3}{c}{C_{0;2,2}}
\end{array}\right),
\end{aligned}
$$

where $k=\frac{N}{4}$, and without loss of generality, we consider only the block upper triangular part. Assume that we have obtained a compression $X_1Y_1^H$ of $C_{0;1,2}$ such that

$$
C_0 = \left(\begin{array}{c||c} C_{0;1,1} & X_1Y_1^H \\ \hline\hline C_{0;2,1} & C_{0;2,2} \end{array}\right),
$$

where X_1 and Y_1 are tall and thin and can be computed with τ-accurate SVD or rank-revealing QR factorization. Next, we split $C_{0;1,1}$, the $(1,1)$ block of C_0, into 2×2 blocks and compress its off-diagonal block. Suppose the resulting off-diagonal block of $C_{0;1,1}$ has size $k\times(\frac{N}{2}-l)$. We will compress it by shifting certain parts of X_1 and Y_1. That is, we partition X_1 and Y_1 conformally as

$$
X_1 = \begin{pmatrix} X_{1,1} \\ X_{1,2} \end{pmatrix},\ Y_1 = \begin{pmatrix} Y_{1,1} \\ Y_{1,2} \end{pmatrix},
$$

and also pick a $k\times(\frac{N}{2}-l)$ block from the lower left corner of $C_{0;1,2}=X_1Y_1^H$ (that is, $X_{1,2}Y_{1,1}^H$)

$$
\begin{aligned}
C_0 &= \left(\begin{array}{c||c} C_{0;1,1} & X_1Y_1^H \\ \hline\hline C_{0;2,1} & C_{0;2,2} \end{array}\right) \\
&= \left(\begin{array}{c|ccc||ccc|c}
 & \frac{1}{\omega^2-\omega^{2l+3}} & \cdots & \frac{1}{\omega^2-\omega^{N+1}} & & & & \\
 & \vdots & \cdots & \vdots & & & & \\
 & \frac{1}{\omega^{2k}-\omega^{2l+3}} & \cdots & \frac{1}{\omega^{2k}-\omega^{N+1}} & & & & \\
\hline
 & & & & \frac{1}{\omega^{2k+2}-\omega^{N+3}} & \cdots & \frac{1}{\omega^{2k+2}-\omega^{2(N-l)+1}} & \\
 & & & & \vdots & \cdots & \vdots & \\
 & & & & \frac{1}{\omega^{4k}-\omega^{N+3}} & \cdots & \frac{1}{\omega^{N}-\omega^{2(N-l)+1}} & \\
\hline\hline
\multicolumn{4}{c||}{C_{0;2,1}} & \multicolumn{4}{c}{C_{0;2,2}}
\end{array}\right) \\
&= \left(\begin{array}{c|c||c|c}
 & X_2Y_2^H & X_{1,1}Y_{1,1}^H & X_{1,1}Y_{1,2}^H \\
\hline
 & & X_{1,2}Y_{1,1}^H & X_{1,2}Y_{1,2}^H \\
\hline\hline
\multicolumn{2}{c||}{C_{0;2,1}} & \multicolumn{2}{c}{C_{0;2,2}}
\end{array}\right),
\end{aligned}
$$

where $X_2Y_2^H$ is an unknown compression of the upper right submatrix of $C_{0;1,1}$, and the blocks that don't concern us are left blank. At this point we do not need another factorization to get $X_2Y_2^H$; instead we can directly derive the compression $X_2Y_2^H$ of $C_{0;1,1}$ from $X_{1,2}$ and $Y_{1,1}$. Clearly we have

$$X_2Y_2^H = \begin{pmatrix} \frac{1}{\omega^2-\omega^{2l+3}} & \cdots & \frac{1}{\omega^2-\omega^{N+1}} \\ \vdots & \cdots & \vdots \\ \frac{1}{\omega^{2k}-\omega^{2l+3}} & \cdots & \frac{1}{\omega^{2k}-\omega^{N+1}} \end{pmatrix},$$

$$X_{1,2}Y_{1,1}^H = \begin{pmatrix} \frac{1}{\omega^{2k+2}-\omega^{N+3}} & \cdots & \frac{1}{\omega^{2k+2}-\omega^{2(N-l)+1}} \\ \vdots & \cdots & \vdots \\ \frac{1}{\omega^{4k}-\omega^{N+3}} & \cdots & \frac{1}{\omega^{N}-\omega^{2(N-l)+1}} \end{pmatrix} = \frac{1}{\omega^{2k}}X_2Y_2^H.$$

That means we can get $X_2Y_2^H$ by shifting a subblock of $X_1Y_1^H$. A similar situation holds for the splitting of the $(2,2)$ block of C after the first splitting. For the successive compressions in later splittings, a similar shifting can be used. For splitting with general block sizes the shifting is also similar.

The shifting scheme indicates that, in different levels of divide-and-conquer SSS constructions, the compressions of large blocks and small blocks are related. This can be used to further save the compression cost.

5. Performance and numerical experiments.

It is well known that the FFT transformation of the Toeplitz matrix T to a Cauchy-like matrix C, and the recovery from the solution $\tilde{x}$ of the Cauchy-like system to the solution x of the Toeplitz system, are stable. In addition, the fast divide-and-conquer construction and the SSS system solver in section 3 are both practically stable as we will see in our numerical results. Thus our overall algorithm is stable in practice. Here no extra steps are needed to stabilize the algorithm as required in some other superfast methods [38].

After the precomputations discussed in sections 4.2 and 4.3 are finished, the cost of each SSS construction is only $O(Np^2)$, where p is the maximum of the off-diagonal ranks of the Cauchy matrix C_0. The SSS solver also costs $O(Np^2)$. The total cost is thus no more than $O(N\log N) + O(Np^2)$ flops. The total storage requirement is $O(Np)$. For the convenience of coding, we use an $O(N^2p)$ cost precomputation routine as discussed in subsection 4.2.

A preliminary implementation of our superfast solver in Fortran 90 is available at http://www.math.ucla.edu/~jxia/work/toep. We did experiments on an Itanium 2 1.4 GHz SGI Altix 350 server with a 64-bit Linux operating system and Intel MKL BLAS. Our method (denoted by `NEW`) is compared to Gaussian elimination with partial pivoting (denoted by `GEPP`) (via the LAPACK linear system solver ZGESV [3]). We consider real $N \times N$ Toeplitz matrices T whose entries are random and uniformly distributed in $[0,1]$. The right-hand sides b are obtained by $b = Tx$, where the exact solution x has random entries uniformly distributed in $[-1,1]$. Matrices of size $N = 2^k \times 100$, $k = 2,3,\ldots$, are considered. These matrices are moderately ill-conditioned. For example, for $N = 2^k \times 100$ with $k = 2,\ldots,7$, the one-norm condition numbers of the matrices increase from the order of about 10^3 to 10^6.

For the `NEW` solver two important parameters are involved, the SSS block size d (Figure 2.1) and the tolerance τ in the compressions of off-diagonal blocks. Here we use the τ-accurate QR factorization with τ as a relative tolerance (section 2.2).

Figure 5.1 shows the execution time (in seconds) for `GEPP` and our `NEW` solver with different d and τ. For the `NEW` solver only the time for solving the Cauchy-like

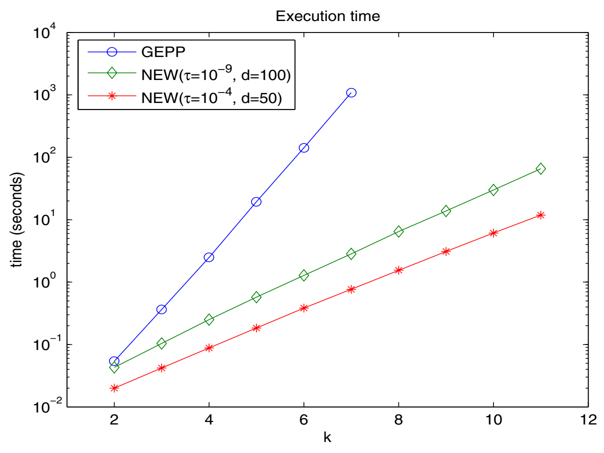

FIG. 5.1. *Computation time (in seconds) versus $N = 2^k \times 100$. For the* NEW *solver, time for the precomputation is excluded.*

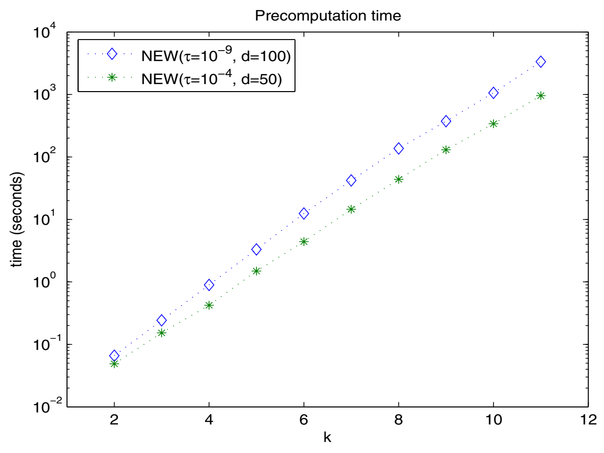

FIG. 5.2. *Time (in seconds) for precomputations in the* NEW *solver.*

system (2.3) is reported, as the compressions of off-diagonal blocks can be done in the precomputation, as described in section 4.3, and the SSS construction time is nearly the precomputation time. The precomputation time is shown in Figure 5.2, although the precomputation only needs to be done once for a particular matrix size N.

We also consider the *time scaling factor*, that is, the factor by which the time is multiplied when the matrix size doubles; see Figure 5.3. We observe that the time scaling factors for the NEW solver are near 2, that is to say the NEW solver is close to being a linear time solver.

Figures 5.4 and 5.5 present the errors $\varepsilon_1 = \frac{||\hat{x}-x||_2}{||x||_2}$ and $\varepsilon_2 = \frac{||T\hat{x}-b||_2}{|||T|\,|\hat{x}|+|b|\,||_2}$, respectively, where $\hat{x}$ denotes the numerical solution.

A significant point of the new solver is that we can use a relatively large tolerance τ in the precomputation and solving, and then use iterative refinement to improve the accuracy. A relatively large τ leads to relatively small off-diagonal ranks (and thus

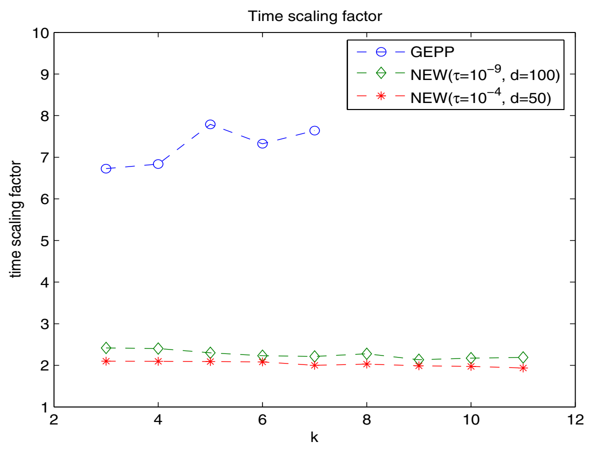

FIG. 5.3. *Time scaling factors.*

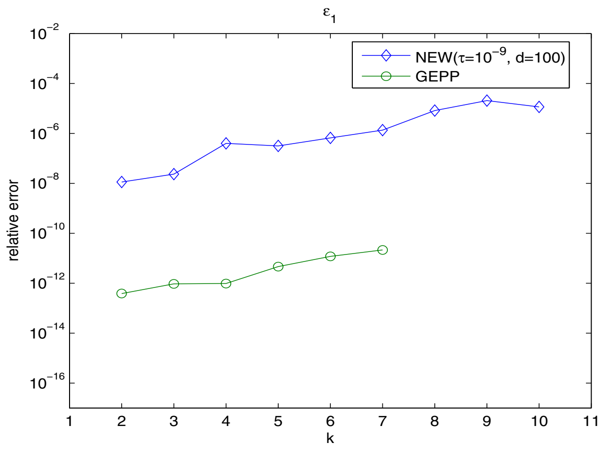

FIG. 5.4. $\varepsilon_1 = \frac{||\hat{x}-x||_2}{||x||_2}$ *versus* $N = 2^k \times 100$.

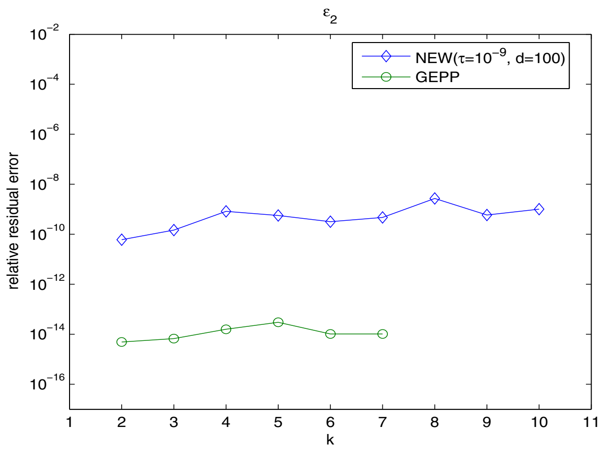

FIG. 5.5. $\varepsilon_2 = \frac{||T\hat{x}-b||_2}{||\,|T|\,|\hat{x}|+|b|\,||_2}$ *versus* $N = 2^k \times 100$.

small p in the operation counts above). Iterative refinements are very cheap due to the facts that no precomputation is needed and the solver itself is very fast (Figure 5.1). For $\tau = 10^{-4}$ and $d = 50$ the accuracy results after 2 to 8 steps of iterative refinement are displayed in Figure 5.6. In fact the number of iterative refinement steps required to reach $\varepsilon_2 < 10^{-13}$ is listed in Table 5.1.

Thus it is also clear that our NEW solver can also perform well as a preconditioner.

The block size d also affects the performance. Figure 5.7 indicates that for a

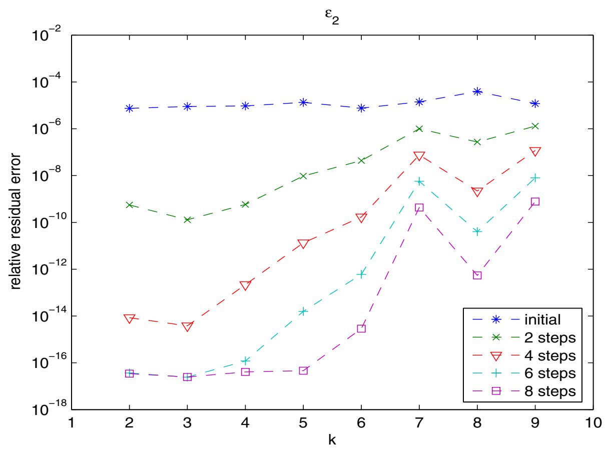

FIG. 5.6. $\varepsilon_2 = \frac{||T\hat{x}-b||_2}{|||T||\hat{x}|+|b|||_2}$, *initial values (before iterative refinements), and after some steps of iterative refinement.*

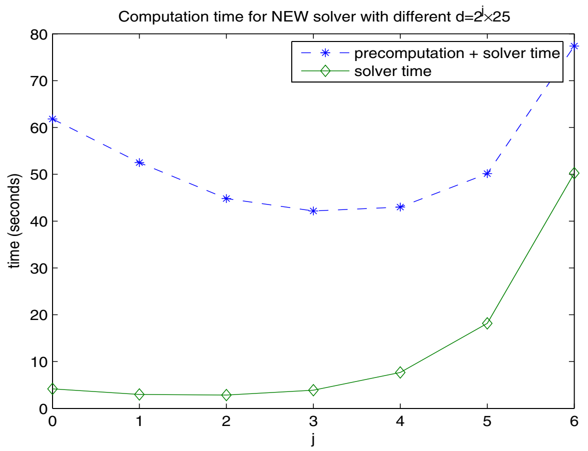

FIG. 5.7. *Computation time of the* NEW *solver with* $N = 12800$, $\tau = 10^{-9}$, *and different block size* $d = 2^j \times 25$.

TABLE 5.1
Number of iterative refinement steps required to reach $\varepsilon_2 < 10^{-13}$ *for different matrix dimensions* $N = 2^k \times 100$.

k	2	3	4	5	6	7	8	9
Number of steps	4	4	5	6	7	15	9	21

given N if $d = 2^j \times 25$ is too large or too small, then both the solver time and the precomputation time can be relatively large. In fact by counting the operations in the algorithm it is possible to determine an optimal d; see [12] for the derivation of the optimal choice of d in the SSS solver. We can do similar derivations for both the SSS construction and the SSS solver. We just point out that when τ gets larger, the off-diagonal ranks decrease and a smaller d should be chosen. In the previous experiments we used two sets of parameters: $(\tau = 10^{-9}, d = 100)$ and $(\tau = 10^{-4}, d = 50)$.

Finally, it turns out that our current preliminary implementation of the solver is slower than the implementation of the algorithm in [38], likely due to our inefficient data structure and nonoptimized codes for SSS matrices. The complicated nature of SSS matrices needs more careful coding and memory management. An improved software implementation is under construction.

6. Conclusions and future work.

In this paper we have presented a superfast and practically stable solver for Toeplitz systems of linear equations. A Toeplitz matrix is first transformed into a Cauchy-like matrix, which has a nice low-rank property, and then the Cauchy-like system is solved by a superfast solver. This superfast solver utilizes this low-rank property and makes use of an SSS representation of the Cauchy-like matrix. A fast construction procedure for SSS structures is presented. After a one-time precomputation the solver is very efficient $(O(N \log N) + O(Np^2)$ complexity). Also the algorithm is efficient in that only linear storage is required. In future work we hope to further reduce the precomputation cost and finish a better-designed version of the current Fortran 90 codes in both the computations and the coding.

Acknowledgments. Many thanks to the referees for their valuable comments which have greatly improved the development and presentation of this paper. The authors are also grateful to the editors and the editorial assistant for their patience and thoughtful suggestions.

REFERENCES

[1] G. S. Ammar and W. B. Gragg, *Superfast solution of real positive definite Toeplitz systems*, SIAM J. Matrix Anal. Appl., 9 (1988), pp. 61–76.

[2] G. S. Ammar and W. B. Gragg, *Numerical experience with a superfast real Toeplitz solver*, Linear Algebra Appl., 121 (1989), pp. 185–206.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, 2nd ed., SIAM, Philadelphia, PA, 1994.

[4] R. R. Bitmead and B. D. O. Anderson, *Asymptotically fast solution of Toeplitz and related systems of linear equations*, Linear Algebra Appl., 34 (1980), pp. 103–116.

[5] A. W. Bojanczyk, R. P. Brent, F. R. de Hoog, and D. R. Sweet, *On the stability of the Bareiss and related Toeplitz factorization algorithms*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 40–57.

[6] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, J. Algorithms, 1 (1980), pp. 259–295.

[7] J. R. Bunch, *Stability of methods for solving Toeplitz systems of equations*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 349–364.

[8] J. Carrier, L. Greengard, and V. Rokhlin, *A fast adaptive multipole algorithm for particle simulations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 669–686.

[9] T. F. Chan and P. C. Hansen, *A look-ahead Levinson algorithm for general Toeplitz systems*, IEEE Trans. Signal Process., 40 (1992), pp. 1079–1090.

[10] T. F. Chan and P. C. Hansen, *A look-ahead Levinson algorithm for indefinite Toeplitz systems*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 490–506.

[11] S. Chandrasekaran and M. Gu, *Fast and stable algorithms for banded plus semiseparable matrices*, SIAM J. Matrix Anal. Appl., 25 (2003), pp. 373–384.

[12] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, *Fast Stable Solvers for Sequentially Semi-Separable Linear Systems of Equations and Least Squares Problems*, Technical report, University of California, Berkeley, CA, 2003.

[13] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, *Some fast algorithms for sequentially semiseparable representations*, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 341–364.

[14] S. Chandrasekaran, M. Gu, and W. Lyons, *A Fast and Stable Adaptive Solver for Hierarchically Semi-Separable Representations*, Technical report, UCSB Math 2004-20, University of California, Santa Barbara, CA, 2004.

[15] G. Cybenko, *The numerical stability of the Levinson-Durbin algorithm for Toeplitz systems of equations*, SIAM J. Sci. Statist. Comput., 1 (1980), pp. 303–319.

[16] G. Cybenko, *Error Analysis of Some Signal Processing Algorithms*, Ph.D. thesis, Princeton University, Princeton, NJ, 1978.

[17] F. R. deHoog, *On the solution of Toeplitz systems*, Linear Algebra Appl., 88/89 (1987), pp. 123–138.

[18] P. Dewilde and A. van der Veen, *Time-Varying Systems and Computations*, Kluwer Academic Publishers, Boston, MA, 1998.

[19] M. Fiedler, *Hankel and Loewner matrices*, Linear Algebra Appl., 58 (1984), pp. 75–95.

[20] K. A. Gallivan, S. Thirumalai, P. Van Dooren, and V. Vermaut, *High performance algorithms for Toeplitz and block Toeplitz matrices*, Linear Algebra Appl., 241/243 (1996), pp. 343–388.

[21] L. Gemignani, *Schur complements of Bezoutians and the inversion of block Hankel and block Toeplitz matrices*, Linear Algebra Appl., 253 (1997), pp. 39–59.

[22] I. Gohberg, T. Kailath, and V. Olshevsky, *Fast Gaussian elimination with partial pivoting for matrices with displacement structure*, Math. Comp., 64 (1995), pp. 1557–1576.

[23] I. Gohberg and V. Olshevsky, *Fast state space algorithms for matrix Nehari and Nehari-Takagi interpolation problems*, Integral Equations Operator Theory, 20 (1994), pp. 44–83.

[24] I. Gohberg and V. Olshevsky, *Complexity of multiplication with vectors for structured matrices*, Linear Algebra Appl., 202 (1994), pp. 163–192.

[25] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comput. Phys., 73 (1987), pp. 325–348.

[26] M. Gu, *Stable and efficient algorithms for structured systems of linear equations*, SIAM J. Matrix Anal. Appl., 19 (1998), pp. 279–306.

[27] M. Gu and S. C. Eisenstat, *Efficient algorithms for computing a strong rank-revealing QR factorization*, SIAM J. Sci. Comput., 17 (1996), pp. 848–869.

[28] G. Heinig, *Inversion of generalized Cauchy matrices and other classes of structured matrices*, in Linear Algebra for Signal Processing, IMA Vol. Math. Appl. 69, Springer, New York, 1995, pp. 63–81.

[29] G. Heinig and K. Rost, *Algebraic Methods for Toeplitz-like Matrices and Operators*, Oper. Theory Adv. Appl. 13, Birkhäuser Verlag, Basel, 1984, pp. 109–127.

[30] T. Kailath, *Fredholm resolvents, Wiener-Hopf equations, and Riccati differential equations*, IEEE Trans. Inform. Theory, 15 (1969), pp. 665–672.

[31] T. Kailath, S. Kung, and M. Morf, *Displacement ranks of matrices and linear equations*, J. Math. Anal. Appl., 68 (1979), pp. 395–407.

[32] T. Kailath and A. H. Sayed, eds., *Fast Reliable Algorithms for Matrices with Structure*, SIAM, Philadelphia, 1999.

[33] P. G. Martinsson, V. Rokhlin, and M. Tygert, *A fast algorithm for the inversion of general Toeplitz matrices*, Comput. Math. Appl., 50 (2005), pp. 741–752.

[34] M. Morf, *Fast Algorithms for Multivariable Systems*, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, 1974.

[35] B. R. Musicus, *Levinson and Fast Choleski Algorithms for Toeplitz and Almost Toeplitz Matrices*, Technical report, Res. Lab. of Electronics, M.I.T., Cambridge, MA, 1984.

[36] V. Pan, *On computations with dense structured matrices*, Math. Comp., 55 (1990), pp. 179–190.

[37] D. R. Sweet, *The use of pivoting to improve the numerical performance of algorithms for Toeplitz matrices*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 468–493.

[38] M. Van Barel, G. Heinig, and P. Kravanja, *A stabilized superfast solver for nonsymmetric Toeplitz systems*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 494–510.

[39] M. Van Barel and P. Kravanja, *A stabilized superfast solver for indefinite Hankel systems*, Linear Algebra Appl., 284 (1998), pp. 335–355.