SUPERFAST SOLUTION OF REAL POSITIVE DEFINITE TOEPLITZ SYSTEMS*

GREGORY S. AMMAR† AND WILLIAM B. GRAGG‡

Abstract. We describe an implementation of the generalized Schur algorithm for the superfast solution of real positive definite Toeplitz systems of order $n+1$, where $n = 2^\nu$. Our implementation uses the split-radix fast Fourier transform algorithms for real data of Duhamel. We are able to obtain the nth Szegő polynomial using fewer than $8n\log_2^2 n$ real arithmetic operations without explicit use of the bit-reversal permutation. Since Levinson's algorithm requires slightly more than $2n^2$ operations to obtain this polynomial, we achieve crossover with Levinson's algorithm at $n = 256$.

Key words. Toeplitz matrix, Schur's algorithm, split-radix Fast Fourier Transform

AMS subject classifications. 65F05, 65E05

1. Introduction.

Consider the linear system of equations $Mx = b$, where

$$M = M_{n+1} = \begin{bmatrix} \mu_0 & \mu_1 & \mu_2 & \cdots & \mu_n \\ \mu_1 & \mu_0 & \mu_1 & \cdots & \mu_{n-1} \\ \mu_2 & \mu_1 & \mu_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \mu_1 \\ \mu_n & \mu_{n-1} & \cdots & \mu_1 & \mu_0 \end{bmatrix} = [\mu_{i-j}]_{i,j=0}^n$$

is a real symmetric positive definite Toeplitz matrix of order $n+1$. In contrast with the standard Gaussian and Choleski factorization techniques, which require $O(n^3)$ arithmetic operations, there are several well known *fast,* $O(n^2)$, methods for solving a Toeplitz system of equations [21,29,4,17]. More recently, several $O(n\log^2 n)$ methods have been presented [6,8,12,22,20]; we refer to these methods as *superfast* Toeplitz solvers because they require substantially less computation than the fast Toeplitz solvers for sufficiently large n.

It is well known (see, e.g., [19,18,3]) that fast Toeplitz solvers are based on ideas from the classical theory of polynomials orthogonal on the unit circle (Szegő polynomials). In particular, the Szegő polynomials can be identified with the columns of the reverse Choleski factorization of M^{-1}. This leads to the observation that the classical Szegő recursions [28,1,14] are equivalent with the Levinson-Durbin algorithm for the Yule-Walker equations [16]. Moreover, the decomposition of M^{-1} given by the Gohberg-Semencul formula is equivalent with the Christoffel-Darboux-Szegő formula. Schur's algorithm [23] provides another connection between Toeplitz solvers and classical analysis. Schur's algorithm generates a continued fraction representation of a holomorphic function mapping the unit disk in the complex plane into its closure, and is known to be closely related with the fast algorithms for finding the Choleski factorization of the positive definite Toeplitz matrix M [18,22].

A presentation of the superfast algorithm of de Hoog [12] and Musicus [22] that uses the theory of orthogonal polynomials on the unit circle is given in [2,3]. This

*This is a preprint of the paper that appears in *SIAM J. Matrix Anal. Appl.*, 9 (1), 1988, pp. 61–76. This research is supported in part by the National Science Foundation under grant DMS-8704196.

† Department of Mathematical Sciences, Northern Illinois University, DeKalb, IL 60115.

‡Department of Mathematics, University of Kentucky Lexington, KY 40506. Supported in part by the Bergen Scientific Centre, IBM.

algorithm is naturally described in terms of a generalization of Schur's classical algorithm. The generalized Schur algorithm is a doubling procedure for calculating the linear fractional transformation that results from n steps of Schur's algorithm. This formulation provides a concise and classically motivated presentation of the algorithm of de Hoog and Musicus when applied to a positive definite matrix.

The implementation of the generalized Schur algorithm for the superfast solution of a (Hermitian) positive definite Toeplitz system is described in [2]. By using standard fast Fourier transform (FFT) techniques to perform the required polynomial recursions, we can construct the linear fractional transformation that results from n steps of Schur's algorithm in $O(n \log_2^2 n)$ complex multiplications. This process yields, without extra work, all n *Schur parameters*, also known as *reflection coefficients* or *partial correlation coefficients*. These parameters are often needed in applications.

The de Hoog-Musicus algorithm consists of two phases. First the nth degree Szegő polynomial is constructed from the linear fractional transformation obtained by the generalized Schur algorithm. Second, the Gohberg-Semencul formula is used to solve the Toeplitz system in $O(n \log_2 n)$ additional multiplications. Each phase involves the computation of cyclic convolutions. These convolutions are performed using in-place FFTs without explicit use of the bit-reversal permutation by using 'dual codes' and leaving all transformed data in bit-reversed order. If we insist that the transformed data be in correct order the number of necessary data accesses increases. Our implementation of the algorithm uses $2n \log_2^2 n + O(n \log_2 n)$ complex multiplications [2]. This operation count is less than those obtained by de Hoog and Musicus. Moreover, this algorithm requires the least amount of computation among the other superfast Toeplitz solvers [6,8,20].

In this paper we describe an implementation of the generalized Schur algorithm for a real positive definite Toeplitz matrix. The implementation of this superfast Toeplitz solver for real (symmetric) positive definite matrices is conceptually the same. The essential difference is the use of FFT algorithms that exploit the inherent symmetries of the real data and their transforms. There are various ways to perform an FFT on real data in roughly half the computation as in the complex case [27]. We desire the most efficient algorithms possible since transforms of various size need to be performed repeatedly during the algorithm. We also want to be able perform the real convolutions without explicit use of the bit-reversal permutation. In Section 2, we consider some of the real FFT algorithms and show how the real split-radix FFT of Duhamel [13,24,25] suits our purpose. The generalized Schur algorithm is described in Section 3, and in Section 4 its implementation for real input data is described. In Section 5 we consider the superfast solution of a real positive definite Toeplitz system of equations by using the generalized Schur algorithm. We will see that the nth degree Szegő polynomial can be calculated in fewer than $8n \log_2^2 n$ total real operations.

2. Evaluation of Real Cyclic Convolutions. The efficient implementation of the generalized Schur algorithm relies on the use of fast Fourier transforms to evaluate cyclic convolutions. Several methods exist for calculating the Fourier transform of real data in roughly half the computation of the complex case. Each of these methods yields an efficient method for evaluating real convolutions, and results in an implementation of the generalized Schur algorithm for real data that requires roughly half the computation as in the complex case. Since convolutions of various sizes are performed repeatedly in the algorithm, we desire the most efficient real transforms possible. Moreover, we want to implement the algorithm without explicit use of the bit-reversal permutation. We avoided this permutation in the complex case, but since the transform of a real vector is not real, some additional considerations must be made to avoid the bit reversal for the case of real input data.

Recently, real FFT algorithms that can serve as dual codes to allow us to avoid the shuffling have been presented [9,13,24,25]. In this section we show how the split-radix FFT for real data suits our purpose. The algorithms are described by considering the splitting in matrix notation, and precise operation counts are given for use in the subsequent sections. We will need the following lemmas in our derivation. Assume that n and n_0 are powers of two, and let $\lg n := \log_2 n$.

LEMMA 2.1. *If* $\phi(n) = 2\phi(n/2) + 2an\lg n + bn + c$ *for* $n > n_0$*, then*

$$\phi(n) = an\lg^2 n + (a+b)n\lg n + dn - c,$$

where d *is determined by the initial condition* $\phi(n_0)$*.*

LEMMA 2.2. *If* $\phi(n) = \phi(n/2) + 2\phi(n/4) + an + b$*, then*

$$\phi(n) = \frac{2}{3}an\lg n - \frac{b}{2} + cn + d(-1)^{\lg n}$$

where c *and* d *are determined by* $\phi(n_0)$ *and* $\phi(n_0/2)$*.*

LEMMA 2.3. *If* $\phi(n) = \phi(n/2) + an\lg n + bn + c + d(-1)^{\lg n}$*, then*

$$\phi(n) = 2an\lg n + 2(b-a)n + c\lg n + \frac{d}{2}(-1)^{\lg n} + e$$

where e *is determined by* $\phi(n_0)$*.*

These lemmas are directly verified by induction and are easily derived by considering the corresponding inhomogeneous linear difference equations for $\phi_\nu := \phi(2^\nu)$.

The *discrete Fourier transform (DFT)* of $x \in \mathbb{C}^n$ is defined by $F_n x$, where $nF_n := [\bar{\omega}_n^{jk}]_{j,k=0}^{n-1}$, ω_n is the principal nth root of unity $\exp(2\pi i/n)$, and $\bar{\alpha}$ denotes the complex conjugate of α. The *inverse discrete Fourier transform (IDFT)* of $y \in \mathbb{C}^n$ is then given by $W_n y$, where $W_n := F_n^{-1} = n\bar{F}_n = [\omega_n^{jk}]_0^{n-1}$. There are various ways to compute the DFT or IDFT in $O(n\log n)$ arithmetic operations. Such an algorithm is called a *fast Fourier transform (FFT)* . In the following we focus, without loss of generality, on the computation of $y = W_n x$.

Let $K_n = [e_0, e_{n-1}, e_{n-2}, ..., e_1]$ and $J_n = [e_{n-1}, e_{n-2}, ..., e_0]$, respectively, be the $n\times n$ *reflection* and *reversal matrices*, where $e_0, ..., e_{n-1}$ are the columns of the $n \times n$ identity matrix. Then we have

$$K_n W_n = [\omega_n^{-jk}] = \bar{W}_n \tag{2.1}$$

and

$$J_n W_n = [\omega_n^{(n-j-1)k}] = [\omega_n^{-jk}\omega_n^{-k}] = \bar{W}_n\bar{D}_n, \tag{2.2}$$

where $D_n := \operatorname{diag}[\omega_n^j]_0^{n-1}$. It is easily seen from (2.1) that whenever $x \in \mathbb{R}^n$, $y = F_n x$ satisfies $K_n y = \bar{y}$; that is, $\eta_{n-j} = \bar{\eta}_j$ $(j = 1, ..., n/2-1)$ and $\eta_0, \eta_{n/2} \in \mathbb{R}$. We will say the transformed vector y possesses *conjugate-even (CE) symmetry* . Thus the transform of a real vector is determined by the n real numbers that constitute its first $n/2+1$ components. There are various methods to compute the real to CE transform and its inverse in roughly half the computation as in the general (complex) case. Some of these methods are considered below.

Let $\alpha_{\mathbb{C}}, \mu_{\mathbb{C}}$, respectively, denote a complex addition and multiplication, and similarly for $\alpha_{\mathbb{R}}, \mu_{\mathbb{R}}$. Also let $\tau_{\mathbb{R}}$ denote a real arithmetic operation. We determine the number of real arithmetic operations using $\alpha_{\mathbb{C}} = 2\alpha_{\mathbb{R}}$, $\mu_{\mathbb{C}} = 2\alpha_{\mathbb{R}} + 4\mu_{\mathbb{R}} = 6\tau_{\mathbb{R}}$. In the following we ignore multiplication by 1 and $i = \sqrt{-1}$. We also count the computation of the product of a complex number with an eighth root of unity as $2\alpha_{\mathbb{R}} + 2\mu_{\mathbb{R}}$ (since, e.g., $\omega_8(\alpha + i\beta) = (\alpha - \beta)/\sqrt{2} + i(\alpha + \beta)/\sqrt{2}$).

Our interest in FFTs is motivated by the desire to efficiently compute products of polynomials, or equivalently, cyclic convolutions. The *cyclic* (or *periodic*) *convolution* $x * y =: z = [\zeta_j]_0^{n-1}$ of $x = [\xi_j]_0^{n-1}$ and $y = [\eta_j]_0^{n-1}$ is defined by $\zeta_j = \sum_{k=0}^{n-1} \xi_k \eta_{j-k}$ (where $\eta_{-k} \equiv \eta_{n-k}$). Note that from this definition, the computation of z requires $O(n^2)$ operations. It is easily verified, however, that $W_n z = (W_n x) \cdot (W_n y)$ (and $F_n z = (F_n x) \cdot (F_n y)$), where $u \cdot v$ denotes the Schur product (componentwise product) of the vectors u and v. Thus, if $\phi(n)$ denotes the computation required to compute a complex FFT of order n, $z = x * y$ can be computed using $3\phi(n) + n\mu_{\mathbb{C}}$ operations. Moreover, if $\tau(n)$ denotes the computation required by a real to CE transform or its inverse transform, then the cyclic convolution of $x, y \in \mathbb{R}^n$ can be computed in $3\tau(n) + (n/2 - 1)\mu_{\mathbb{C}} + 2\mu_{\mathbb{R}}$ $(n > 1)$.

The calculation of the Fourier and inverse Fourier transforms is typically performed using the Cooley-Tukey algorithm, which can be described as follows. We describe the inverse transform $y = W_n x$ of $x \in \mathbb{C}^n$; since $nF_n = \bar{W}_n$, the computation of $F_n x$ is completely analogous and involves the same amount of computation. We neglect division by n, which is assumed to be a power of two.

Let $m := n/2$, $D'_m := \operatorname{diag}[\omega_n^j]_0^{m-1}$, and define the permutation matrix P_n by $P_n^T x = \begin{bmatrix} x'_0 \\ x'_1 \end{bmatrix}$, where $x'_0 = [\xi_{2j}]_0^{m-1}$ and $x'_1 = [\xi_{2j+1}]_0^{m-1}$ are, respectively, the *even* and *odd parts* of x. Then it is easily seen that

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} := y = W_n P_n P_n^T x = \begin{bmatrix} W_m & D'_m W_m \\ W_m & -D'_m W_m \end{bmatrix} \begin{bmatrix} x'_0 \\ x'_1 \end{bmatrix} = \begin{bmatrix} u_0 + u_1 \\ u_0 - u_1 \end{bmatrix}, \tag{2.3a}$$

where

$$\begin{bmatrix} u_0 \\ u_1 \end{bmatrix} := \begin{bmatrix} W_m x'_0 \\ D'_m W_m x'_1 \end{bmatrix}. \tag{2.3b}$$

Thus a DFT or IDFT can be computed from the transforms of the even and odd parts of the input vector with some local work. The repeated application of this splitting leads to the *Cooley-Tukey* FFT algorithm [26]. This method for the computation of $F_n x$ is often called a *radix-two Cooley-Tukey decimation-in-time algorithm* [7]. (The output is given by two transforms of subsets of the input, and in applications the input typically corresponds with data in the time domain.) We will refer to this procedure for the computation of $F_n x$ or $W_n x$ as a *decimation-in-input (DII)* algorithm. The DII algorithm (2.3) can be performed in place (i.e., without the use of a temporary work vector) by first permuting the input vector according to the permutation

$$\Pi_n = P_n \operatorname{diag}(P_{n/2}, P_{n/2}) ... \operatorname{diag}(P_4, P_4, ..., P_4).$$

The transformation $\Pi_n x$ is referred to as the *bit-reversal permutation of order n*. Thus, in-place computation of the DII FFT is achieved by rearranging the input vector before the computations take place.

Analogously, the even and odd halves of the output are given by two transforms of order $m = n/2$. This algorithm, which is called the *Sande-Tukey* or the *radix-two Cooley-Tukey decimation-in-frequency* algorithm, is given by the recursive application of the formula

$$\text{(2.4a)} \quad \begin{bmatrix} y_0' \\ y_1' \end{bmatrix} := P_n^T W_n x = \begin{bmatrix} W_m & W_m \\ W_m D_m' & -W_m D_m' \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} W_m u_0 \\ W_m D_m' u_1 \end{bmatrix},$$

where

$$\text{(2.4b)} \qquad \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} := \begin{bmatrix} x_0 + x_1 \\ x_0 - x_1 \end{bmatrix}.$$

The resulting *decimation-in-output (DIO)* method, when implemented in place, receives input in correct order and generates output in bit-reversed order.

Note that both methods require the same amount of computation $\phi(n)$. In particular, we have $\phi(1) = 0$, $\phi(2) = 2\alpha_{\mathbb{C}}$ and

$$\phi(n) = 2\phi(n/2) + n\alpha_{\mathbb{C}} + (n/2 - 2)\mu_{\mathbb{C}}$$

for $n = 2^\nu > 1$. Since two of the entries of D_m' are eighth roots of unity when $n > 4$, we have

$$\begin{aligned} &\phi(4) = 16\alpha_{\mathbb{R}}, \\ &\phi(n) = 2\phi(n/2) + (3n-4)\alpha_{\mathbb{R}} + (2n-12)\mu_{\mathbb{R}} \qquad (n > 4). \end{aligned}$$

Thus, by Lemma 2.1, the computation of the DFT and IDFT of $x \in \mathbb{C}^n$ by either the DII or DIO radix-two FFT algorithms requires at most

$$\phi(n) = (3n \lg n - 3n + 4)\alpha_{\mathbb{R}} + (2n \lg n - 7n + 12)\mu_{\mathbb{R}} \qquad (n = 2^\nu > 4).$$

In many applications we need the transformed data in correct order, so the explicit use of the bit-reversal permutation is required. However, since we are using the transforms as a computational tool for cyclic convolution evaluation, we do not need to know the true order of the components of the transformed vectors; we need only multiply corresponding components of the transformed vectors and apply the inverse transform. Thus, we can use the above two FFT algorithms as *dual codes*, one for the transform and the other for the inverse transform, in order to avoid the need to shuffle the data before or after the calculations. To calculate $z = x * y$ we use the Sande-Tukey (DIO) recursions to obtain $W_n x$ and $W_n y$ in bit-reversed order, form their Schur product to obtain $W_n z$ in bit-reversed form, and then use the Cooley-Tukey (DII) recursions to obtain z in correct order. (Note that we are using an IDFT as our transform and a DFT as our inverse transform.) While the use of dual FFT codes does not affect the amount of computation in the evaluation of cyclic convolutions, it reduces the number of data accesses.

Now consider the computation of $y = W_n x$ when $x \in \mathbb{R}^n$. Note that the DII recursion (2.3) splits the transform of x into two real transforms of half the size. These transforms also possess CE symmetry, so the redundant computations can be identified (and ignored) during each stage of the splitting. In particular, only $y_0 = [\eta_j]_0^{m-1}$ and η_m are computed from components 0 through $l = n/4$ of u_0 and u_1. The successive application of this splitting for the transform of a real vector is known as the *Edson-Bergland algorithm* [5]. This algorithm is described in a recent paper by Swarztrauber [27], where analogous algorithms for vectors with other types of symmetries are derived.

Letting $\tau(n)$ denote the amount of computation used to compute the DFT or IDFT of a real vector using the Edson-Bergland algorithm, we see that $\tau(1) = 0$, $\tau(2) = 2\alpha_{\mathbb{R}}$, $\tau(4) = 6\alpha_{\mathbb{R}}$, and for $n > 4$

$$\begin{aligned}\tau(n) &= 2\tau(n/2) + 2\alpha_{\mathbb{R}} + (n/4 - 1)(2\alpha_{\mathbb{C}} + \mu_{\mathbb{C}}) \\ &= 2\tau(n/2) + (3n/2 - 4)\alpha_{\mathbb{R}} + (n - 6)\mu_{\mathbb{R}}\end{aligned}$$

since one of the complex multiplications is by an eighth root of unity. The Edson-Bergland algorithm therefore requires at most

$$\tau(n) = \left(\frac{3n\lg n}{2} - \frac{5n}{2} + 4\right)\alpha_{\mathbb{R}} + \left(n\lg n - \frac{7n}{2} + 6\right)\mu_{\mathbb{R}}$$

operations when $n = 2^\nu > 2$, and $\tau(2) = 2\alpha_{\mathbb{R}}$.

Note that the Edson-Bergland algorithm requires the (real) input data to be in bit-reversed form. In the pursuit of our desire to eliminate the explicit use of the bit-reversal permutation, we want the real data in correct order, and the transforms, having CE symmetry, in bit-reversed order. Such an algorithm can be derived from the DIO recursions applied to a real vector.

Let us consider the DIO splitting (2.4) applied to $x \in \mathbb{R}^n$. In this case $y_0' = W_m u_0$ is the transform of a real vector, while $y_1' = W_m D_m' u_1$. A further splitting reveals the redundancies in y_1'. Let $l := n/4 \geq 1$, $u_1 =: \begin{bmatrix} t_0 \\ t_1 \end{bmatrix}$ and $D_l'' := \operatorname{diag}[\omega_{4l}^j]_{j=0}^{l-1}$. Then $(D_l'')^4 = (D_l')^2 = D_l$, $(D_l'')^{-1} = \bar{D}_l''$, $D_m' = \begin{bmatrix} D_l'' & 0 \\ 0 & iD_l'' \end{bmatrix}$ and

$$\begin{bmatrix} v_0' \\ v_1' \end{bmatrix} := P_m^T y_1' = \begin{bmatrix} W_l D_l'' z_0 \\ W_l D_l' D_l'' \bar{z}_0 \end{bmatrix} \tag{2.5}$$

where $z_0 = t_0 + it_1$. Furthermore, by (2.2) we have

$$J_l v_1' = J_l W_l D_l' D_l'' \bar{z}_0 = \bar{W}_l \bar{D}_l'' \bar{z}_0 = \bar{v}_0',$$

so the computation of v_1' is redundant. Thus, y can be calculated using one real FFT of order $n/2$, one complex FFT of order $n/4$ and some local work. This splitting strategy results in the *real split-radix FFT algorithm* of Duhamel [13,24,25]. Since the product $D_l'' z_0$ requires $l - 1$ complex multiplications, with one eighth root of unity if $l > 1$, the total work $\tau(n)$ for a split-radix FFT on a real vector of order n satisfies

$$\tau(n) = \tau(n/2) + \phi(n/4) + (3n/2 - 2)\alpha_{\mathbb{R}} + (n - 6)\mu_{\mathbb{R}} \quad (n > 4), \tag{2.6}$$

with $\tau(4) = 6\alpha_{\mathbb{R}}$, $\tau(2) = 2\alpha_{\mathbb{R}}$. Note that the corresponding DII complex to real transform (the inverse transform) is also given by the above splitting, and requires the same amount of computation.

The split-radix technique can also be applied the computation of a complex FFT. Let $x \in \mathbb{C}^n$, $y := W_n x$, and split y_1' in (2.4) to obtain

$$P_m^T y_1' =: \begin{bmatrix} v_0' \\ v_1' \end{bmatrix} = \begin{bmatrix} W_l D_l'' z_0 \\ W_l D_l' D_l'' \bar{z}_1 \end{bmatrix}, \text{where } \begin{bmatrix} z_0 \\ z_1 \end{bmatrix} := \begin{bmatrix} t_0 + it_1 \\ \bar{t}_0 + i\bar{t}_1 \end{bmatrix}.$$

In this case v_1' is not redundant, but we have $v_1' = J_l \overline{W_l D_l'' z_1}$.

Thus, the computational work $\phi(n)$ for a complex split-radix FFT of order n satisfies

$$\phi(n) = \phi(n/2) + 2\phi(n/4) + (4n-4)\alpha_{\mathbb{R}} + (2n-12)\mu_{\mathbb{R}} \quad (n > 4),$$

$$\phi(4) = 16\alpha_{\mathbb{R}}, \qquad \phi(2) = 4\alpha_{\mathbb{R}}.$$

We therefore have by Lemma 2.2 the following.

PROPOSITION 2.1. *The computation of a complex FFT of size $n = 2^\nu$ by the split-radix method requires at most*

$$\begin{aligned}\phi(n) &= \left(\frac{8}{3}n\nu - \frac{16}{9}n - \frac{2}{9}(-1)^\nu + 2\right)\alpha_{\mathbb{R}} + \left(\frac{4}{3}n\nu - \frac{38}{9}n + \frac{2}{9}(-1)^\nu + 6\right)\mu_{\mathbb{R}} \\ &= 4n\nu - 6n + 8 \text{ total real operations}\end{aligned}$$

for $n \geq 2$.

Thus, the split-radix procedure obtains the DFT with roughly 80% of the computation of the radix-two method. A proportionate amount of computational savings for the evaluation of a real FFT and its (complex to real) inverse is obtained when the real split-radix FFT is performed using a split-radix complex FFT. In particular, by (2.6), Proposition 2.1 and Lemma 2.3, we obtain the following.

PROPOSITION 2.2. *The computation $\tau(n)$ required to compute the FFT or IDFT of a real vector of order $n = 2^\nu$ and the FFT or IDFT of a vector with CE symmetry is at most*

$$\begin{aligned}\tau(n) &= \left(\frac{4}{3}n\nu - \frac{17}{9}n - \frac{1}{9}(-1)^\nu + 3\right)\alpha_{\mathbb{R}} + \left(\frac{2}{3}n\nu - \frac{19}{9}n + \frac{1}{9}(-1)^\nu + 3\right)\mu_{\mathbb{R}} \\ &= 2n\nu - 4n + 6 \text{ total real operations}\end{aligned}$$

for $n \geq 2$.

These operation counts agree with those reported in [24,25].

Thus, a real cyclic convolution of order n can be computed without explicit use of the bit-reversal permutation by using the real to complex DIO split-radix FFT and its DII inverse as dual codes. These algorithms can be performed in place using real arithmetic and n real storage locations. Moreover, the split-radix FFT has the smallest operation count among the known FFT algorithms for real transforms of length equal to a power of two [25].

We remark that we investigated the use of other methods for real transforms before becoming aware of the split-radix method. We first considered the use of the common method of evaluating the transform of a real vector by the transform of a complex vector of half the size and postprocessing (see, e.g., [26,27]). However, this method requires the explicit use of the bit-reversal permutation, and moreover, is not as efficient as the Edson-Bergland algorithm. As we remarked above, the Edson-Bergland algorithm is not appropriate for our use either. We considered the use of Hartley transforms [10], and we in fact initially implemented the algorithm using dual codes for the Hartley transform. However, while it requires less computation than the common pre- and post-processing procedure, the radix-two computation of a Hartley transform requires more computation than the Edson-Bergland algorithm. Moreover, the Hartley transform of a convolution is not equal to the Schur product of Hartley transforms, so slightly more computation is needed to form the transform of the convolution from the transforms of the input data. Direct observation shows that the Fourier transform of $x \in \mathbb{R}^4$ requires $6\alpha_{\mathbb{R}}$ while the Hartley transform requires $8\alpha_{\mathbb{R}}$. We therefore do not expect the use of the Hartley transform to be as efficient as Fourier transforms in the evaluation of real convolutions. We have not considered, however, the relative performance of Fourier and Hartley transforms in the evaluation of real convolutions where the input have additional symmetry (e.g., real even or real odd input vectors).

In our study of the development of DIO real to complex FFT analogous with the Edson-Bergland FFT, we found the procedure described by (2.5), and afterward were made aware that this procedure is in fact the real split-radix FFT of Duhamel, which when applied to the real and complex FFT together, uses 20% less computation than the standard radix-two methods. In the meantime we have also become aware of the paper by Briggs [9], where DIO methods are derived that are analogous with the symmetric DII FFTs presented in [27], including the Edson-Bergland algorithm.

3. The Generalized Schur Algorithm. Let ϕ be a *Schur function*, which is to say that ϕ is a holomorphic function that maps the open unit disk D in the complex plane into its closure. Schur's algorithm [23] is a procedure that generates a sequence of Schur functions by the successive application of linear fractional transformations (LFTs) to ϕ. It is defined as follows.

Schur's Algorithm.
input: an initial Schur function ϕ_0,
$\gamma_1 := \phi_0(0)$,
for $n = 1, 2, 3, \cdots$ **while** $|\gamma_n| < 1$

$$\phi_n(\lambda) := \frac{1}{\lambda}\frac{\phi_{n-1}(\lambda) - \gamma_n}{1 - \bar{\gamma}_n \phi_{n-1}(\lambda)},$$

$$\gamma_{n+1} := \phi_n(0).$$

If $|\gamma_n| = 1$, then $\phi_{n-1}(\lambda) \equiv \gamma_n$ and Schur's algorithm terminates. On the other hand, if $|\gamma_n| < 1$ then ϕ_n is a Schur function. Thus, Schur's algorithm generates a possibly finite sequence of Schur functions ϕ_n that satisfy $\phi_{n-1} = t_n(\phi_n)$, where

$$t_n(\tau) = (\gamma_n + \lambda\tau)/(1 + \bar{\gamma}_n \lambda\tau).$$

We can therefore view the algorithm as the generation of a continued fraction representation of ϕ_0 (since continued fractions are related with compositions of LFTs). In particular, $\phi_0 = T_n(\phi_n)$, where $T_n = t_1 \circ t_2 \circ ... \circ t_n$. The function $T_n(0)$ is referred to as the *nth approximant* of ϕ_0, and ϕ_n is called the *nth tail* of ϕ_0.

In the standard implementation of Schur's algorithm, each ϕ_n is written as a quotient of formal power series,

$$\phi_n =: \frac{\alpha_n(\lambda)}{\beta_n(\lambda)} = \frac{\sum_k \alpha_{n,k}\lambda^k}{\sum_k \beta_{n,k}\lambda^k},$$

with $\beta_n(0) > 0$ as a *partial normalization.* The computations are then arranged so that the coefficient pairs $(\alpha_{0,k}, \beta_{0,k})$ are entered and processed in a sequential manner. This results in the following algorithm.

Progressive Schur Algorithm:

for $k = 1, 2, 3, \cdots$ **while** $|\gamma_k| < 1$

enter $\alpha_{0,k-1}$, $\beta_{0,k-1}$

for $j = 1, 2, 3, ..., k-1$

$$\begin{bmatrix} \alpha_{j,k-j-1} \\ \beta_{j,k-j} \end{bmatrix} = \begin{bmatrix} 1 & -\gamma_j \\ -\bar{\gamma}_j & 1 \end{bmatrix} \begin{bmatrix} \alpha_{j-1,k-j} \\ \beta_{j-1,k-j} \end{bmatrix}$$

$\gamma_k = \alpha_{k-1,0}/\beta_{k-1,0}$,

$\beta_{k,0} = \beta_{k-1,0}(1 - |\gamma_k|^2)$

Of course, in practice a finite number of coefficients are input. Let $\alpha^{(n)}$ denote the polynomial of degree less than n formed by the first n terms of the power series α. If $\alpha_0^{(n)}$, $\beta_0^{(n)}$ are input, then the progressive Schur algorithm calculates $\alpha_k^{(n-k)}$, $\beta_k^{(n-k)}$ and γ_k for $k = 1, ..., n$. using at most $n^2\alpha_{\mathbb{R}} + n(n+2)\mu_{\mathbb{R}}$.

Schur's algorithm can also be formulated in terms of the LFTs T_n. In particular, it is easily seen that

$$T_n(\tau) = \frac{\xi_n + \tilde{\eta}_n\tau}{\eta_n + \tilde{\xi}_n\tau},$$

where ξ_n, η_n are the polynomials that satisfy the recurrence relations

$$\begin{bmatrix} \xi_n \\ \eta_n \end{bmatrix} = \begin{bmatrix} \tilde{\eta}_{n-1} & \xi_{n-1} \\ \tilde{\xi}_{n-1} & \eta_{n-1} \end{bmatrix} \begin{bmatrix} \gamma_n \\ 1 \end{bmatrix}, \qquad \begin{bmatrix} \xi_0 \\ \eta_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{3.1}$$

where $\tilde{\xi}_n(\lambda) := \lambda^n\bar{\xi}_n(1/\lambda)$ and $\tilde{\eta}_n(\lambda) := \lambda^n\bar{\eta}_n(1/\lambda)$. We refer to ξ_n and η_n as the nth *Schur polynomials* associated with the Schur function ϕ_0. Note that (3.1) implies (for $n \geq 1$)

$$\deg \xi_n < n, \quad \deg \eta_n < n,$$

$$\xi_n(0) = \gamma_1, \quad \eta_n(0) = 1,$$

as well as the *determinant formula*

$$\eta_n\tilde{\eta}_n - \xi_n\tilde{\xi}_n = \delta_n\lambda^n,$$

where $\delta_n = (1 - |\gamma_1|^2)...(1 - |\gamma_n|^2)$.

The generalized Schur algorithm is based on a doubling procedure for generating T_{2n} from T_n. The idea can be described in general terms as follows. Let $\phi_n = T_n^{-1}(\phi_0)$ be the nth tail of ϕ_0, and let $T_{n,n}$ denote the LFT that results from n steps of Schur's algorithm applied to the Schur function ϕ_n. The LFT T_{2n} is then given by the composition $T_n \circ T_{n,n}$.

For each n, let α_n and β_n be formal power series such that $\phi_n = \alpha_n/\beta_n$. The nth tail of ϕ_0 is then given by

$$\phi_n = \frac{\alpha_n}{\beta_n} = T_n^{-1}(\phi_0) = \frac{\alpha_0\eta_n - \beta_0\xi_n}{\beta_0\tilde{\eta}_n - \alpha_0\tilde{\xi}_n}.$$

It is shown in [2,3] that both the numerator and denominator in the last expression are formal power series that are divisible by λ^n. In particular,

$$\alpha_0\eta_n - \beta_0\xi_n = \gamma_{n+1}\delta_n\lambda^n + O(\lambda^{n+1}),$$

$$\beta_0\tilde{\eta}_n - \alpha_0\tilde{\xi}_n = \delta_n\lambda^n + O(\lambda^{n+1}).$$

We can therefore take

$$\alpha_n = (\alpha_0\eta_n - \beta_0\xi_n)/\lambda^n, \qquad \beta_n = (\beta_0\tilde{\eta}_n - \alpha_0\tilde{\xi}_n)/\lambda^n. \tag{3.2}$$

These formulas constitute the first component of the generalized Schur algorithm: the computation of α_n and β_n (i.e., ϕ_n) from ξ_n, η_n, α_0, and β_0.

The next step in the generalized Schur algorithm is the doubling step. Since $\phi_n = \alpha_n/\beta_n$ is a Schur function we can obtain the nth Schur polynomials $\xi_{n,n}$ and $\eta_{n,n}$ (i.e., $T_{n,n}$) from α_n and β_n using the same procedure that $\xi_{0,n} = \xi_n$ and $\eta_{0,n} = \eta_n$ were obtained from α_0 and β_0.

The third step of the algorithm is the computation of $\xi_{0,2n}$ and $\eta_{0,2n}$ from the composition of T_n and $T_{n,n}$. In particular, for any $k > 0$ we have $\phi_n = T_{n,k}(\phi_{n+k})$; that is, the kth tail of ϕ_n is the $(n+k)$th tail of ϕ_0. We therefore have $\phi_0 = T_{n+k}(\tau) = T_n(T_{n,k}(\tau))$, or equivalently,

$$\xi_{0,n+k} = \tilde{\eta}_{0,n}\xi_{n,k} + \xi_{0,n}\eta_{n,k}, \qquad \eta_{0,n+k} = \tilde{\xi}_{0,n}\xi_{n,k} + \eta_{0,n}\eta_{n,k}. \tag{3.3}$$

Equations (3.2) and (3.3) form the basis of the generalized Schur algorithm. The algorithm is easiest to describe in its recursive form.

Generalized Schur Algorithm:

input: $n = 2^\nu$ and polynomials $\alpha_0^{(n)}, \beta_0^{(n)}$, where α_0, β_0 are power series such that $\phi_0 := \alpha_0/\beta_0$ is a Schur function;

$\xi_{0,1} = \gamma_1 = \alpha_0^{(1)}$, $\eta_{0,1} = 1$;

for $m = 1, 2, 4, ..., n/2$

1. use (3.2) to obtain $\alpha_m^{(m)}$, $\beta_m^{(m)}$ from $\xi_{0,m}$, $\eta_{0,m}$, $\alpha_0^{(2m)}$, $\beta_0^{(2m)}$;
2. use the generalized Schur algorithm to compute $\xi_{m,m}$, $\eta_{m,m}$ from $\alpha_m^{(m)}$ and $\beta_m^{(m)}$ as $\xi_{0,m}$ and $\eta_{0,m}$ were obtained from $\alpha_0^{(m)}$ and $\beta_0^{(m)}$;
3. use (3.3) to compute $\xi_{0,2m}$, $\eta_{0,2m}$ from $\xi_{0,m}$, $\eta_{0,m}$, $\xi_{m,m}$, $\eta_{m,m}$;

output: the Schur polynomials $\xi_n = \xi_{0,n}$, $\eta_n = \eta_{0,n}$ and the Schur parameters $[\gamma_j]_1^n$.

Note that the classical Schur algorithm generates $\alpha_k^{(n-k)}, \beta_k^{(n-k)}$ using the n Schur parameters γ_k as intermediate values. In the generalized Schur algorithm, however, the number of coefficients of α_k, β_k to be calculated is equal to the largest power of two that divides k. For example, $n/2$ coefficients are calculated when $k = n/2$, $n/4$ coefficients are calculated when $k = n/4$ and $k = 3n/4$, and only the constant terms are calculated when k is odd. Nevertheless, *every Schur parameter is generated in the generalized Schur algorithm.* These parameters are often of physical and mathematical significance; if they are not needed for the output, however, they do not need to be stored in the above algorithm.

4. Implementation of the Generalized Schur Algorithm for Real Data. In order to implement the generalized Schur algorithm, we write steps 1 and 3 as convolutions and apply FFT techniques. Let $\mathbb{R}_n[\lambda]$ denote the set of real polynomials of degree less than n. For any polynomial $\xi(\lambda) = \sum_0^{n-1} \xi_j\lambda^j \in \mathbb{R}_n[\lambda]$ we write

$x \leftrightarrow \xi$ for the associated vector $x = [\xi_j]_0^{n-1} \in \mathbb{R}^n$. Define vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$ $(m = n/2 \geq 1)$ by

$$x_0, x_1, x \leftrightarrow \xi_m, \xi_{m,m}, \xi_n, \qquad a_0, a_1, a \leftrightarrow \alpha^{(m)}, \alpha_m^{(m)}, \alpha^{(n)}$$

and similarly for $y \leftrightarrow \eta, b \leftrightarrow \beta$. Also define $\tilde{x} = J_n \bar{x}$, so that $\tilde{x} \leftrightarrow \tilde{\xi}_n/\lambda$.

Step 3 involves the products of polynomials in $\mathbb{R}_m[\lambda]$ and $\mathbb{R}_{m+1}[\lambda]$, which are equivalent with convolutions of size n. Specifically,

$$x = E_n \begin{bmatrix} \tilde{y}_0 \\ 0 \end{bmatrix} * \begin{bmatrix} x_1 \\ 0 \end{bmatrix} + \begin{bmatrix} x_0 \\ 0 \end{bmatrix} * \begin{bmatrix} y_1 \\ 0 \end{bmatrix},$$

$$y = E_n \begin{bmatrix} \tilde{x}_0 \\ 0 \end{bmatrix} * \begin{bmatrix} x_1 \\ 0 \end{bmatrix} + \begin{bmatrix} y_0 \\ 0 \end{bmatrix} * \begin{bmatrix} y_1 \\ 0 \end{bmatrix},$$

where $E_n = [e_1, e_2, ..., e_{n-1}, e_0]$ is the $n \times n$ *cyclic downshift matrix.*

While Step 1 involves the polynomials in $\mathbb{R}_{3m}[\lambda]$, only the *middle* m coefficients of these polynomials are needed. We can therefore obtain a_1, b_1 from the last m components of convolutions of order n. In particular,

$$\begin{bmatrix} * \\ a_1 \end{bmatrix} = a * \begin{bmatrix} y_0 \\ 0 \end{bmatrix} - b * \begin{bmatrix} x_0 \\ 0 \end{bmatrix},$$

$$\begin{bmatrix} * \\ b_1 \end{bmatrix} = b * E_n \begin{bmatrix} \tilde{y}_0 \\ 0 \end{bmatrix} - a * E_n \begin{bmatrix} \tilde{x}_0 \\ 0 \end{bmatrix}.$$

We can therefore perform one step of the generalized Schur algorithm, the calculation of $[u, v] := W_n[x, y]$ from a, b and $[u_0, v_0] := W_m[x_0, y_0]$ as follows. The number of real arithmetic operations used follows each substep in brackets.

0. Suppose $[u_0, v_0]$ has been computed $\quad [\omega(m)]$.
1. Compute $\quad [a_1, b_1]$:
   (a) $[x_0, y_0] = F_m[u_0, v_0] \quad [2\tau(m)]$,
   (b) $[p, q] = W_n \begin{bmatrix} x_0 & y_0 \\ 0 & 0 \end{bmatrix} \quad [2\phi(m/2) + (2m-4)\alpha_{\mathbb{R}} + (4m-12)\mu_{\mathbb{R}}]$,
   (c) $[r, s] = W_n E_n \begin{bmatrix} \tilde{x}_0 & \tilde{y}_0 \\ 0 & 0 \end{bmatrix} \quad [\mathit{free}]$,
   (d) $[c, d] = W_n[a, b] \quad [2\tau(2m)]$,
   (e) $\begin{bmatrix} * \\ a_1 \end{bmatrix} = F_n(c \cdot q - d \cdot p)$, $\begin{bmatrix} * \\ b_1 \end{bmatrix} = F_n(d \cdot s - c \cdot r)$,
   $[2\tau(2m) + (12m-8)\alpha_{\mathbb{R}} + (16m-8)\mu_{\mathbb{R}}]$.
2. Compute $[u_1, v_1]$ from $[a_1, b_1]$ as $[u_0, v_0]$ was computed from $[a_0, b_0] \quad [\omega(m)]$.
3. Computation of $[u, v]$:
   (a) $[x_1, y_1] = F_m[u_1, v_1] \quad [2\tau(m)]$,
   (b) $[p_1, q_1] = W_n \begin{bmatrix} x_1 & y_1 \\ 0 & 0 \end{bmatrix} \quad [2\phi(m/2) + (2m-4)\alpha_{\mathbb{R}} + (4m-12)\mu_{\mathbb{R}}]$,
   (c) $u = s \cdot p_1 + p \cdot q_1, \quad v = r \cdot p_1 + q \cdot q_1 \quad [(12m-8)\alpha_{\mathbb{R}} + (16m-8)\mu_{\mathbb{R}}]$.

We compute p, q from u_0, v_0 as follows. We have

$$\begin{bmatrix} p_0' \\ p_1' \end{bmatrix} := P_n^T W_n \begin{bmatrix} x_0 \\ 0 \end{bmatrix} = \begin{bmatrix} W_m x_0 \\ W_m D_m' x_0 \end{bmatrix},$$

so that $p_0' = u_0$ and $p_1' = W_m D_m' F_m u_0$. Moreover, if $l := n/4 \geq 1$, let $\begin{bmatrix} t_0 \\ t_1 \end{bmatrix} := x_0$. Then it follows from the split-radix splitting that

$$P_m^T p_1' = \begin{bmatrix} z_0 \\ J_l \bar{z}_0 \end{bmatrix},$$

where $z_0 = W_l D_l''(t_0 + it_1)$. Note that the computation of z_0 from x_0 requires $(l-1)\mu_{\mathbb{C}}$, including the one eighth root of unity when $l > 1$. Thus, p is calculated from u_0 in 1(a) and 1(b) using $\tau(m) + \phi(l) + (2l-2)\alpha_{\mathbb{R}} + (4l-6)\mu_{\mathbb{R}}$ operations, and likewise for q.

We now show that r and s can be obtained by negating the odd parts of $\bar{p}$ and $\bar{q}$. We have

$$r = W_n E_n \begin{bmatrix} J_m x_0 \\ 0 \end{bmatrix} = W_n E_n \begin{bmatrix} 0 & J_m \\ J_m & 0 \end{bmatrix} \begin{bmatrix} 0 \\ x_0 \end{bmatrix} = W_n E_n J_n \begin{bmatrix} 0 \\ x_0 \end{bmatrix}.$$

It is easily verified that $E_n J_n = K_n$ and $W_n E_n J_n = \bar{W}_n$, so that

$$\bar{r} = W_n \begin{bmatrix} 0 \\ x_0 \end{bmatrix}.$$

Hence,

$$\begin{bmatrix} \bar{r}_0' \\ \bar{r}_1' \end{bmatrix} := P_n^T \bar{r} = \begin{bmatrix} W_m & W_m \\ W_m D_m' & -W_m D_m' \end{bmatrix} \begin{bmatrix} 0 \\ x_0 \end{bmatrix},$$

and so

$$\bar{r}_0' = W_m x_0 = p_0', \qquad \bar{r}_1' = -W_m D_m' x_0 = -p_1'.$$

The same relationship holds for s and q. Thus $[r, s]$ can be obtained from $[p, q]$ with no additional computation.

The counts for the computations in steps 1(e) and 3(c) follow from the fact that the transforms are determined by $m-1$ complex and two real numbers. Also note that, since $\Pi_n p = \begin{bmatrix} \Pi_m p_0' \\ \Pi_m p_1' \end{bmatrix}$, the above manipulations are easily performed when the transforms are stored in bit-reversed order.

The amount of computation $\omega(n)$ required to obtain u, v by the generalized Schur algorithm with real input data therefore satisfies, for $n > n_0 > 2$

$$\begin{aligned} \omega(n) &= 2\omega(n/2) + 4\tau(n) + 4\tau(n/2) + 4\phi(n/4) + (14n-24)\alpha_{\mathbb{R}} + (20n-40)\mu_{\mathbb{R}} \\ &= 2\omega(n/2) + 16n \lg n - 8n + 16 \quad \text{total real operations,} \end{aligned}$$

with roughly twice as many additions as multiplications. By Lemma 2.1, we obtain

$$\omega(n) = 8n \lg^2 n + Cn - 16 \qquad (n \geq n_0),$$

where C is determined by $\omega(n_0)$.

Note that

$$\xi_{0,1} = \gamma_1 = \alpha_{0,0}/\beta_{0,0}, \qquad \eta_{0,1} = 1,$$

so that $\omega(1) = 1\mu_{\mathbb{R}}$. If we use the doubling procedure at this stage, we obtain $\omega(2) = 16\alpha_{\mathbb{R}} + 18\mu_{\mathbb{R}} = 34\tau_{\mathbb{R}}$, $\omega(4) = 180$ and $C = 17$. We can improve on this by considering more direct methods to carry out the recursions in the early stages of the algorithm.

Note that we can obtain $\xi_{0,2}$, $\eta_{0,2}$ from $\alpha_0^{(2)}$, $\beta_0^{(2)}$ using $7\tau_{\mathbb{R}}$ from

$$\xi_{0,2} \leftrightarrow x_0 := \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix}, \qquad \eta_{0,2} \leftrightarrow y_0 := \begin{bmatrix} 1 \\ \gamma_1\gamma_2 \end{bmatrix}$$

where

$$\gamma_1 = \alpha_{0,0}/\beta_{0,0}, \qquad \gamma_2 = \frac{\alpha_{1,0}}{\beta_{1,0}} = \frac{\beta_{0,1}\gamma_1 - \alpha_{0,1}}{\alpha_{0,1}\gamma_1 - \beta_{0,0}}.$$

More generally, k steps of the progressive Schur algorithm can be used to generate the first k Schur parameters in $2k(k+1)\tau_{\mathbb{R}}$. Then ξ_k and η_k can be obtained in $(2k^2-5k+3)\tau_{\mathbb{R}}$ using (3.1), and $u, v \in \mathbb{R}^k$ are obtained in $2\tau(k)$ additional operations. Using this procedure, we obtain $\omega(4) = 67\tau_{\mathbb{R}}$ and $C = -45/4$ in (4.1).

Note that $\omega(n)$ is the amount of computation for $u, v \in \mathbb{C}^n$. An additional $2\tau(n)$ is needed to obtain ξ_n, η_n. We therefore have the following.

PROPOSITION 4.1. *Given the first n terms of formal power series α, β such that α/β is a Schur function, the amount of computation required by the generalized Schur algorithm to obtain the nth Schur polynomials ξ_n, η_n is at most*

$$8n \lg^2 n + 4n \lg n - \frac{77}{4} n - 4$$

real arithmetic operations for $n = 2^\nu > 2$.

5. Application to Real Toeplitz Systems of Equations. The real positive definite Toeplitz matrix $M = M_{n+1} = [\mu_{j-k}]_{j,k=0}^n = M^T$ defines an inner product on $\mathbb{R}_{n+1}[\lambda]$ by $\langle \lambda^j, \lambda^k \rangle := \mu_{j-k}$, and the monic polynomials $\chi_k(\lambda)$ that are orthogonal under this inner product are the monic *Szegő polynomials* determined by M. Let $\delta_k := \langle \chi_k, \chi_k \rangle$, and define R to be the unit right triangular matrix whose kth column contains the coefficients of χ_k $(0 \le k \le n)$. Then we have

$$R^T M R = D := \operatorname{diag}[\delta_k]_0^n,$$
$$M^{-1} = R D^{-1} R^{-T},$$

so the Szegő polynomials determine the *reverse Choleski factorization* of M^{-1}. These polynomials satisfy the recurrence relations below, which comprise the first phase of many fast Toeplitz solvers, particularly those of Levinson and Trench (see, e.g., [21,29,16]). Let $\chi_k \leftrightarrow \begin{bmatrix} r_k \\ 1 \end{bmatrix} \in \mathbb{R}^{k+1}$ and $m_k := [\mu_j]_1^k \in \mathbb{R}^k$.

Levinson's Algorithm (Szegő Recursions):
input: $[\mu_j]_0^n$
$\delta_0 := \mu_0$, $\gamma_1 := -\mu_1/\mu_0$,
$r_1 := \gamma_1$, $\delta_1 = (1-\gamma_1^2)\delta_0$,
for $k = 1, ..., n-1$ **do**

$$\gamma_{k+1} = \frac{-1}{\delta_k} m_{k+1}^T \begin{bmatrix} r_k \\ 1 \end{bmatrix},$$

$$r_{k+1} = \begin{bmatrix} 0 \\ r_k \end{bmatrix} + \begin{bmatrix} 1 \\ \tilde{r}_k \end{bmatrix} \gamma_{k+1},$$

$$\delta_{k+1} = \delta_k(1-\gamma_{k+1}^2).$$

Thus, $\chi_k, \delta_k, \gamma_k (0 < k \leq n)$ can be obtained using $n^2\alpha_{\mathbb{R}} + n(n+2)\mu_{\mathbb{R}}$.

In Levinson's algorithm the χ_k, δ_k are used to solve $Mx = b$ using the inverse Choleski factorization. In Trench's algorithm the nth degree polynomial χ_n and its squared norm δ_n are used to construct M_n^{-1} by means of the classical Christoffel-Darboux-Szegő formula; the matrix interpretation is the *Gohberg-Semencul formula:*

$$\delta_n M_{n+1}^{-1} = T_1 T_1^T - T_0^T T_0$$

where

$$T_0 = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ \rho_0 & 0 & \ddots & & 0 \\ \rho_1 & \rho_0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \rho_{n-1} & \rho_{n-2} & \cdots & \rho_0 & 0 \end{bmatrix} \qquad T_1^T = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ \rho_{n-1} & 1 & \ddots & & 0 \\ \rho_{n-2} & \rho_{n-1} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \rho_0 & \rho_1 & \cdots & \rho_{n-1} & 1 \end{bmatrix}$$

and $\chi_n(\lambda) = \sum_0^n \rho_j \lambda^j$ (see, e.g., [15]).

In contrast with the Szegő recursions, the progressive Schur algorithm is used to obtain the Choleski decomposition of M, $M = LDL^T$ where L is unit left triangular. In particular, this factorization is obtained in $O(n^2)$ operations by performing n steps of Schur's classical algorithm applied to a Schur function $\phi_0 = \alpha_0/\beta_0$ with

$$\alpha_0^{(n)} = -\sum_{j=0}^{n-1} \mu_{j+1}\lambda^j, \qquad \beta_0^{(n)} = \sum_{j=0}^{n-1} \mu_j \lambda^j. \tag{5.1}$$

Moreover, the Schur parameters generated by this process are the same as the Schur parameters generated by the Szegő recursions.

The generalized Schur algorithm with the above initialization does not generate the Choleski factorization, but only pieces of it. Nevertheless, all the Schur parameters are generated, and moreover, it follows from the recursions that the Schur polynomials determine the Szegő polynomials [3]; specifically,

$$\chi_n = \tilde{\eta}_n + \tilde{\xi}_n/\lambda.$$

Thus, we can use the generalized Schur algorithm to obtain ξ_n, η_n, compute χ_n and use the Gohberg-Semencul formula to obtain $M^{-1}b$. In this way the Toeplitz system is solved using $8n \lg^2 n + O(n \lg n)$ total real arithmetic operations.

The operation count for the generalized Schur algorithm can be reduced for this superfast Toeplitz solver. Some observations to this effect follow.

Note that since we are not interested in the Schur polynomials per se, we can get χ_n from the transforms $[u, v] = W_n[x, y]$, saving one FFT of order n. Let $x, y \leftrightarrow \xi_n, \eta_n$. Then

$$\chi_n \leftrightarrow \begin{bmatrix} r_n \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ J_n y_n \end{bmatrix} + \begin{bmatrix} J_n x_n \\ 0 \end{bmatrix},$$

and since $\eta_n(0) = 1$,

$$r_n = J_n x + E_n J_n y_n - e_0.$$

It follows that

$$r_n = F_n \overline{(D_n u + v} - e_0.$$

Note that $D_n u$ has CE symmetry, so this product involves two multiplications by eighth roots of unity when $n > 4$. We can therefore obtain χ_n from u, v using $\tau(n) + (2n-3)\alpha_{\mathbb{R}} + (2n-12)\mu_{\mathbb{R}}$ operations. However, multiplication by D_n requires the use of the bit-reversal permutation. In particular, letting $w_n = [\omega_n^j]_0^{n-1} \in \mathbb{C}^n$, we have $\Pi_n D_n u = \Pi_n(w_n \cdot u) = \Pi_n w_n \cdot \Pi_n u$, so we need the nth roots of unity in bit-reversed form. We can either use an additional storage vector for the roots of unity in bit-reversed order, or the correctly ordered w_n can be shuffled when needed. Since this shuffling replaces one FFT, it does not increase the number of data accesses, but it does provide some computational savings.

The relationship between $a \leftrightarrow \alpha_0^{(n)}$ and $b \leftrightarrow \beta_0^{(n)}$ provides more opportunity to reduce the amount of computation if one is content to shuffle data in order to avoid an FFT. We have $b = (\mu_0 - \mu_n)e_0 - E_n a$, and since $W_n E_n = D_n W_n$,

$$d = (\mu_0 - \mu_n)e - D_n c,$$

where $e := W_n e_0$ is the vector of all ones, and $[c, d] := W_n[a, b]$. Thus, when $n \geq 8$, we can replace $1\tau(n)$ with $(n/2-2)\mu_{\mathbb{C}}$ including two multiplications by eighth roots of unity and $(n/2+2)\alpha_{\mathbb{R}}$, or $(7/2)n - 14\tau_{\mathbb{R}}$. This results in a savings of $2n \lg n - (15/2)n + 20$ total real operations, but we must have $\Pi_n w_n$ with $\Pi_n c$ to obtain $\Pi_n D_n c$. Clearly, this procedure can be used each time the elements of α_0, β_0 are accessed; however, it cannot be used in the later stages of the algorithm because α_m, β_m $(m > 0)$ are not related as α_0, β_0 are. Since $\alpha_0^{(k)}, \beta_0^{(k)}$ are used for each $4 < k = 2^\kappa \leq n$, the total savings of using this procedure is (by Lemma 2.3)

$$4n \lg n - 19n + 20 \lg n + 4\tau_{\mathbb{R}} \qquad (n > 4).$$

The above observations yield the following.

THEOREM 5.1. *The computation of χ_n and δ_n from the real positive definite Toeplitz matrix $M = [\mu_{i-j}] = M^T$ using the generalized Schur algorithm as described above requires at most*

$$8n \lg^2 n - 2n \lg n + \frac{31}{4}n - 20 \lg n - 29$$

real arithmetic operations for $n = 2^\nu > 4$.

Thus, the total operation count is slightly less than $8n \lg^2 n$, while that of Levinson's algorithm is slightly more than $2n^2$. These bounds are equal at $n = 256$, so the amount of computation for this algorithm is less than that of Levinson's algorithm for $n = 2^\nu \geq 256$. Of course, this difference rapidly becomes substantial as n doubles. We remark that the algorithm uses roughly twice as many additions as multiplications.

6. Concluding Remarks. The fact that Levinson's algorithm is more efficient for $n < 256$ indicates some improvement in this superfast Toeplitz solver must be possible in its early stages. Moreover, the split Levinson recursions [11], in which redundancies in Levinson's algorithm are removed, requires about $3n^2/2$ real operations. This indicates the likelihood of improving the implementation of the generalized Schur algorithm for the superfast solution of Toeplitz systems. In fact, we made little use of the relationship between α_0 and β_0 given by (5.1).

While the algorithm may vectorize well, the above description is inherently sequential because the computation of $\xi_{m,m}$ cannot proceed until $\alpha_m^{(m)}$ and $\beta_m^{(m)}$ are calculated, which in turn cannot be computed until $\xi_{0,m}$ and $\eta_{0,m}$ are computed. These bottlenecks in the doubling strategy show how this algorithm is not a splitting into independent subproblems as in the case of an FFT. Thus, apart for the obvious independent quantities to be calculated within a step (e.g., p, q can be calculated from u_0, v_0 simultaneously), any parallel implementation of the algorithm is likely to be inherently different from the one presented here.

With regard to the use of the generalized Schur algorithm in the case that n is not a power of two, the recursions (3.2), (3.3) can be used to derive the appropriate convolution formulas for a given factorization of n. For example, if $n = 3m$ a decomposition of ξ_n, η_n into three smaller Schur polynomials that correspond with Schur functions ϕ_0, ϕ_m, ϕ_{2m} could be derived. In this manner the development of multiple radix implementations of the generalized Schur algorithm may proceed analogously with that of FFT algorithms. In a sense, the generalized Schur algorithm is one level of complexity higher than an FFT. It is not unlikely that a family of implementations of the generalized Schur algorithm will be useful in practice, each one tailored to specific lengths of and symmetries in the input data.

We finally remark that we have strived for the lowest operation count for aesthetic and theoretical rather than practical reasons. Some of the fine tuning described in Section 5 will not appear in code for the algorithm because it will make the code too long and tedious. We will, nevertheless, use the dual split-radix codes to reduce the amount of computation and avoid the bit-reversal permutation.

Acknowledgement. We are indebted to Avideh Zakhor for helpful discussions on the split-radix FFT algorithms.

REFERENCES

[1] N. I. Akhiezer, *The Classical Moment Problem,* Oliver and Boyd, Edinburgh, 1965.

[2] G. S. Ammar and W. B. Gragg, *Implementation and Use of the Generalized Schur Algorithm,* in Computational and Combinatorial Methods in Systems Theory, C. I. Byrnes and A. Lindquist, eds., North-Holland, Amsterdam, 1986, pp. 265-280.

[3] G. S. Ammar and W. B. Gragg, *The Generalized Schur Algorithm for the Superfast Solution of Toeplitz Systems,* in Rational Approximation and its Applications in Mathematics and Physics, J. Gilewicz, M. Pindor and W. Siemaszko, eds., Lecture Notes in Mathematics 1237, Springer, Berlin, 1987, pp. 315-330.

[4] E. H. Bareiss, *Numerical Solution of Linear Equations with Toeplitz and vector Toeplitz Matrices,* Numer. Math., 13 (1969), pp. 404-424.

[5] G. D. Bergland, *A fast Fourier transform algorithm for real-valued series,* Comm. ACM, 11 (1968), pp. 703-710.

[6] 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.

[7] R. E. Blahut, *Fast Algorithms for Digital Signal Processing,* Addison-Wesley, Reading, MA, 1985.

[8] 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.
[9] W. L. Briggs, *Further Symmetries of In-Place FFTs,* SIAM J. Sci. Stat. Comput., 8 (1987), 644-654.
[10] O. Buneman, *Conversion of FFTs to fast Hartley transforms,* SIAM J. Sci. Statist. Comput. 7 (1986), pp. 624-638.
[11] P. Delsarte and Y. V. Genin, *The Split Levinson Algorithm,* IEEE Trans. Acoust. Speech Signal Process., 34 (1986), pp. 470-478.
[12] F. de Hoog, *A New Algorithm for Solving Toeplitz Systems of Equations,* Lin. Algebra Appl., 88/89 (1987), pp. 122-138.
[13] P. Duhamel, *Implementation of "split-radix" FFT algorithms for complex, real, and real-symmetric data,* IEEE Trans. Acoust. Speech Signal Process., 34 (1986), pp. 285-295.
[14] L. Y. Geronimus, *Orthogonal Polynomials*, Consultants Bureau, New York, 1961.
[15] I. C. Gohberg and I. A. Fel'dman, *Convolution Equations and Projection Methods for their Solution,* American Mathematical Society, Providence, RI, 1974.
[16] G. H. Golub and C. F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, Baltimore, MD, 1984.
[17] J. R. Jain, *An Efficient Algorithm for a Large Toeplitz Set of Linear Equations,* IEEE Trans. Acoust. Speech Signal Process., 27 (1979), pp. 612-615.
[18] T. Kailath, *A Theorem of I. Schur and Its Impact on Modern Signal Processing,* in I. Schur Methods in Operator Theory and Signal Processing, I. Gohberg, editor, Birkhäuser-Verlag, Basel, 1986, pp. 9-30.
[19] T. Kailath, A. Vieira and M. Morf, *Inverses of Toeplitz Operators, Innovations, and Orthogonal Polynomials,* SIAM Rev., 20 (1978), pp. 106-119.
[20] R. Kumar, *A Fast Algorithm for Solving a Toeplitz System of Equations,* IEEE Trans. Acoust. Speech Signal Process., 33 (1985). pp. 254-267.
[21] N. Levinson, *The Wiener RMS (Root-Mean-Square) Error Criterion in Filter Design and Prediction,* J. Math. Phys., 25 (1947), pp. 261-278.
[22] B. R. Musicus, *Levinson and Fast Choleski Algorithms for Toeplitz and Almost Toeplitz Matrices,* Report, Res. Lab. of Electronics, M.I.T., 1984.
[23] I. Schur, *Uber Potenzreihen, die in Innern des Einheitskrises Beschränkt Sind,* J. Reine Angew. Math., 147 (1917), pp. 205-232.
[24] H. V. Sorensen, M. T. Heideman and C. S. Burrus, *On computing the split-radix FFT,* IEEE Trans. Acoust. Speech Signal Process., 34 (1986), pp. 152-156.
[25] H. V. Sorensen. D. L. Jones, M. T. Heideman and C. S. Burrus, *Real-Valued Fast Fourier Transform Algorithms,* IEEE Trans. Acoust. Speech Signal Process., 35 (1987), pp. 849-863.
[26] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis,* Springer-Verlag, New York, 1980.
[27] P. N. Swarztrauber, *Symmetric FFTs,* Math. Comp. 47 (1986), pp. 323-346.
[28] G. Szegő, *Orthogonal Polynomials,* American Mathematical Society, Providence, RI, 1939.
[29] W. Trench, *An Algorithm for the Inversion of Finite Toeplitz Matrices,* SIAM J. Appl. Math., 12 (1964), pp. 515-522.