Title
On Computing Schur Functions and Series Thereof

Permalink
https://escholarship.org/uc/item/5v4517z6

Authors
Chan, CP
Drensky, V
Edelman, A
et al.

Publication Date
2017-04-26

Peer reviewed
On Computing Schur Functions and Series Thereof

Cy Chan · Vesselin Drensky · Alan Edelman · Raymond Kan · Plamen Koev

Received: date / Accepted: date

Abstract We present two new algorithms for computing all Schur functions $s_\kappa(x_1, \ldots, x_n)$ for partitions $\kappa$ such that $|\kappa| \leq N$.

Both algorithms have the property that for nonnegative arguments $x_1, \ldots, x_n$ the output is computed to high relative accuracy and the cost per Schur function is $O(n^2)$.

Keywords Schur function · Hypergeometric function of a matrix argument · Computing · Accuracy

Mathematics Subject Classification (2000) 05E05 · 65F50 · 65T50

This research was partially supported by NSF Grants DMS–0314286, DMS–0411962, DMS–0608306, the Singapore–MIT Alliance (SMA), and by Grant MI-1503/2005 of the Bulgarian National Science Fund.

C. Chan
Department of Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, U.S.A.
E-mail: cychan@mit.edu

V. Drensky
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria.
E-mail: drensky@math.bas.bg

A. Edelman
Department of Mathematics, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, U.S.A.
E-mail: edelman@math.mit.edu

R. Kan
Joseph L. Rotman School of Management, University of Toronto, 105 St. George Street, Toronto, Ontario, Canada M5S 3E6.
E-mail: kan@chas.s.utoronto.ca

P. Koev
Department of Mathematics, San Jose State University, One Washington Square, San Jose, CA 95192, U.S.A.
E-mail: koev@math.sjsu.edu
1 Introduction

We consider the problem of accurately and efficiently evaluating the Schur function $s_{\kappa}(x_1, x_2, \ldots, x_n)$ and series thereof for nonnegative arguments $x_i \geq 0, i = 1, 2, \ldots, n$. The Schur functions are symmetric homogeneous polynomials

$x \times \text{a n}$

where the summation is over all partitions $\kappa$ functions corresponding to partitions $\kappa$ (see (3) in Section 2 for the exact relationship).

One way to compute the hypergeometric function of a matrix argument in practice is to truncate the series (1) for $k \leq N$ for some sufficiently large $N$. Since $s_k(x_1, x_2, \ldots, x_n) = 0$ if $\kappa_{n+1} > 0$, our goal is thus to compute, as quickly and accurately as possible, all Schur functions corresponding to partitions $\kappa$ in not more than $n$ parts and size not exceeding $N$.

Denote the set of those Schur functions by $S_{N,n}$:

$S_{N,n} \equiv \{s_{\kappa}(x_1, \ldots, x_n) | \kappa = (\kappa_1, \ldots, \kappa_n), |\kappa| \equiv \kappa_1 + \cdots + \kappa_n \leq N\}$. 
The analysis in [6] suggests that computing even a single Schur function accurately and efficiently is far from trivial. We elaborate on this briefly.

There are several determinantal expressions for the Schur function (the classical definition as quotient of generalized Vandermonde determinants, the Jacobi–Trudi identity, its dual version, the Giambelli and Lascoux–Pragacz determinants). Each one would seemingly provide a very efficient way to compute the Schur function. The problem with this approach is that the matrices involved quickly become ill conditioned as the sizes of the matrix argument (\(n\)) and the partition (\(|\kappa|\)) grow. This implies that conventional (Gaussian-elimination-based) algorithms will quickly lose accuracy to roundoff errors. The loss of accuracy is due to a phenomenon known as subtractive cancellation—loss of significant digits due to subtraction of intermediate (and thus approximate) quantities of similar magnitude.

According to [6], the loss of accuracy in evaluating the determinantal expressions for the Schur function can be arbitrarily large in all but the dual Jacobi–Trudi identity. In the latter the amount of subtractive cancellation can be bounded independent of the values of the input arguments \(x_i\). By using extended precision one can compensate for that loss of accuracy leading to an algorithm that is guaranteed to be accurate and costs \(O((|n|\kappa| + \kappa_1^3)(|\kappa||\kappa_1|^{1+\rho}))\).

Subtraction is the only arithmetic operation that could lead to loss of accuracy; multiplication, division, and addition of same-sign quantities always preserves the relative accuracy.

In this paper we present two new algorithms for computing the Schur function. Both algorithms are subtraction-free, meaning that both are guaranteed to compute the value of the Schur function to high relative accuracy in floating point arithmetic. Both are also very efficient—the cost per Schur function, when computing all Schur functions in the set \(S_{N,n}\), is \(O(n^2)\).

This represents a major improvement over the previous best result in [6] in the sense that no extended precision arithmetic is required to achieve accuracy and the cost of computing a single Schur function is reduced from \(O((|n|\kappa| + \kappa_1^3)(|\kappa||\kappa_1|^{1+\rho}))\) to \(O(n^2)\).

While both our new algorithms have the same complexity and accuracy characteristics, each is significant in its own right for the following reasons:

- The first algorithm implements the classical definition of the Schur function as a sum of monomials over all semistandard Young tableaux. Since the coefficients in this expression are positive (integers), such an approach is subtraction-free, thus guaranteed to be accurate. The full expression of the Schur function as a sum of monomials contains exponentially many terms (\(O(n^\kappa)\) [6]) thus the similarly exponential cost of the previous algorithms based on it [6].

  We use dynamic programming and exploit various redundancies to reduce the cost to \(O(n^2)\) per Schur function as long as all Schur functions in the set \(S_{N,n}\) are computed. This algorithm is not efficient for computing individual Schur functions since most functions in \(S_{N,n}\) would need to be computed anyway. However, (unlike the second algorithm) the ideas may generalize beyond \(\alpha = 1\); we elaborate on this in Section 5.

- The second algorithm represents an accurate evaluation of the expression of the Schur function as a quotient of (generalized, totally nonnegative) Vandermonde determinants. Since virtually all linear algebra with totally nonnegative matrices can be performed efficiently and in a subtraction-free fashion [22,23], this leads to an accurate algorithm for the evaluation of individual Schur functions extremely efficiently at the cost of only \(O(n^2\kappa_1)\) each. The cost reduces further to \(O(n^2)\) each if all of \(S_{N,n}\) is computed.

\[^1\text{Here } \rho \text{ is tiny and accounts for certain logarithmic functions.}\]
This paper is organized as follows. We present background information and survey existing algorithms for this problem in Section 2. Our new algorithms are presented in Sections 3 and 4. We draw conclusions and outline open problems in Section 5.

We made software implementations of both our new algorithms available online [21].

2 Preliminaries

Algorithms for computing the hypergeometric function of a matrix argument for specific values of $p, q$, and $\alpha$ can be found in [1, 3, 4, 14, 15].

In this section we survey the approach of Koelv and Edelman [24] which works for any $\alpha > 0$. We also introduce a few improvements and set the stage for our new algorithms in the case $\alpha = 1$.

We first recall a few definitions that are relevant. Given a partition $\kappa$ and a point $(i, j)$ in the Young diagram of $\kappa$ (i.e., $i \leq \kappa'_j$ and $j \leq \kappa_i$), the upper and lower hook lengths at $(i, j) \in \kappa$ are defined, respectively, as:

$$h_\kappa^*(i, j) \equiv \kappa'_j - 1 + \alpha(\kappa_i - j);$$

$$h_\kappa^*(i, j) \equiv \kappa'_j + i + \alpha(\kappa_i - j).$$

The products of the upper and lower hook lengths are denoted, respectively, as:

$$H_\kappa^* \equiv \prod_{(i, j) \in \kappa} h_\kappa^*(i, j) \quad \text{and} \quad H_\kappa^\kappa \equiv \prod_{(i, j) \in \kappa} h_\kappa^*(i, j).$$

We introduce the “Schur” normalization of the Jack function

$$S_{\kappa}^\alpha(X) = \frac{H_\kappa^\kappa}{\alpha^{[\kappa]}|\kappa|!} C_{\kappa}^{\alpha}(X).$$

This normalization is such that $S_{\kappa}^\alpha(X) = s_{\kappa}(x_1, \ldots, x_n)$ [31, Proposition 1.2].

The hypergeometric function of a matrix argument in terms of $S_{\kappa}^\alpha(X)$ is:

$$pF_q^\alpha(a_1, \ldots, a_p; b_1, \ldots, b_q; X) = \sum_{\kappa=0}^{\infty} \sum_{\kappa_1=0}^{\kappa} \frac{(a_1)^{\alpha}_{\kappa_1} \cdots (a_p)^{\alpha}_{\kappa_1}}{(b_1)^{\alpha}_{\kappa_1} \cdots (b_q)^{\alpha}_{\kappa_1}} \frac{\alpha^{[\kappa]}|\kappa|!}{H_\kappa^\kappa} \cdot S_{\kappa}^\alpha(X).$$

Denote the coefficient in front of $S_{\kappa}^\alpha(X)$ in (4) by:

$$Q_\kappa \equiv \frac{(a_1)^{\alpha}_{\kappa_1} \cdots (a_p)^{\alpha}_{\kappa_1}}{(b_1)^{\alpha}_{\kappa_1} \cdots (b_q)^{\alpha}_{\kappa_1}} \frac{\alpha^{[\kappa]}|\kappa|!}{H_\kappa^\kappa}.$$

Let the partition $\kappa = (\kappa_1, \kappa_2, \ldots, \kappa_6)$ have $h = \kappa'_1$ nonzero parts. When $\kappa_i > \kappa_{i+1}$, we define the partition:

$$\kappa(i) \equiv (\kappa_1, \kappa_2, \ldots, \kappa_{i-1}, \kappa_i - 1, \kappa_{i+1}, \ldots, \kappa_6).$$

The main idea in the evaluation of (4) is to update the $\kappa$ term in (4) from terms earlier in the series. In particular, we update $Q_\kappa$ from $Q_{\kappa(i)}$ and $S_{\kappa(i)}^\alpha(x_1, \ldots, x_n)$ from $S_{\mu}(x_1, \ldots, x_{n-1})$, $\mu \leq \kappa$. 

$$\sum_{\kappa_1=0}^{\kappa} \frac{(a_1)^{\alpha}_{\kappa_1} \cdots (a_p)^{\alpha}_{\kappa_1}}{(b_1)^{\alpha}_{\kappa_1} \cdots (b_q)^{\alpha}_{\kappa_1}} \frac{\alpha^{[\kappa]}|\kappa|!}{H_\kappa^\kappa} \cdot S_{\kappa}^\alpha(X).$$
In order to make the $Q_κ$ update as simple as possible, we first express $H^κ_σ$ in a way that does not involve the conjugate partition, $κ^\prime$:

$$H^κ_σ = \prod_{r=1}^h \prod_{i=1}^κ (κ_r - r + 1 + α(κ_r - c))$$
$$= \prod_{r=1}^h \prod_{j=1}^κ \prod_{i=κ_j + 1}^{κ_r} (j - r + 1 + α(κ_r - c))$$
$$= α^{|κ|} \prod_{j=1}^h (j - r + \frac{1}{α} + κ_r - κ_j)^{κ_j - κ_j + 1},$$

where $(c)_t = c(c + 1) \cdots (c + t - 1)$ is the rising factorial, the univariate version of the Pochhammer symbol defined in (2).

Defining $κ_0 ≡ ακ - i$ we obtain:

$$\frac{H^κ_σ}{H^κ_{σ|b}} = \frac{1}{ακ_0 - α + 1} \prod_{j=1}^h \frac{κ_j - κ_0}{κ_j - κ_0 + 1}. \quad (6)$$

Using (6), $Q_κ$ can be updated from $Q_{κ|b}$ as

$$Q_κ = Q_{κ|b} \prod_{j=1}^h (κ_j + 1) \times \frac{α}{ακ_0 - α + 1} \prod_{j=1}^h \frac{κ_j - κ_0}{κ_j - κ_0 + 1}, \quad (7)$$

where $κ_0 ≡ κ_0 - \frac{1}{α}.

The Jack function $S_κ^{(α)}(X)$ can be dynamically updated using the formula of Stanley [31, Proposition 4.2] (see also [24, (3.8)] and (3)):

$$S_κ^{(α)}(x_1, \ldots, x_n) = \sum_{μ} s_μ^{(α)}(x_1, \ldots, x_{n-1}) x_n^{κ/μ} \sigma_{κμ}, \quad (8)$$

where the summation is over all partitions $μ ≤ κ$ such that the skew shape $κ/μ$ is a horizontal strip (i.e., $κ_1 ≥ μ_1 ≥ κ_2 ≥ μ_2 ≥ \cdots ≥ 32, p. 339)). The coefficients $σ_{κμ}$ are defined as

$$σ_{κμ} = \prod_{\{i,j\} ∈ κ}^{h^κ_σ(i,j)} h^μ_κ(i,j) \prod_{\{i,j\} ∈ μ}^{h^μ_κ(i,j)} h^κ_μ(i,j), \quad (9)$$

where both products are over all $(i,j) ∈ κ$ such that $κ'_j = μ'_j + 1$. For $α = 1$, clearly, $σ_{κμ} = 1$ for all $κ$ and $μ$.

Once again, instead of computing the coefficients $σ_{κμ}$ in (8) from scratch, it is much more efficient to start with $σ_{κκ} = 1$ and update the next coefficient in the sum (8) from the previous ones. To this end, let $μ$ be a partition such that $κ'_j = μ'_j$ for $j = 1, 2, \ldots, μ_k - 1$, and $κ/μ$ is a horizontal strip. Then we update $σ_{κμ|b}$ from $σ_{κμ}$ using:

$$\frac{σ_{κμ|b}}{σ_{κμ}} = \prod_{r=1}^h h^κ_r(r, μ_r) \prod_{r=1}^k h^μ_r(r, μ_k) = \prod_{r=1}^k 1 + \frac{κ_r - μ_r + α - 1}{κ_r - μ_r + 1} = \prod_{r=1}^h \frac{1 + \bar{κ}_r - \bar{μ}_r + α - 1}{1 + \bar{μ}_r}, \quad (10)$$

which is obtained directly from (9).
We use (8) to compute $S_{\kappa}^{(a)}(x_1, \ldots, x_k)$ for $i = h + 1, \ldots, n$. For $i = h$, the result of Stanley [31, Propositions 5.1 and 5.5] allows for a very efficient update:

\[
S_{\kappa}^{(a)}(x_1, \ldots, x_k) = (x_1 \cdots x_k)^{\kappa_1} \cdot S_{\kappa - \kappa_0}^{(a)}(x_1, \ldots, x_k) \prod_{j=1}^{h} \frac{h - i + 1 + \alpha(\kappa_j - j)}{h - i + \alpha(\kappa_j - j + 1)},
\]

where $\kappa - \kappa_0 I \equiv (\kappa_1 - \kappa_0, \kappa_2 - \kappa_0, \ldots, \kappa_{h-1} - \kappa_0)$.

The new results in this section comprise of the updates (7) and (10), which are more efficient than the analogous ones in [24, Lemmas 3.1 and 3.2]. These new updates do not require the conjugate partition to be computed and maintained by the algorithm and cost $2(p + q) + 4h$ and $9h$, down from $2(p + q) + 11\kappa_0 + 9h - 11$ and $12k + 6\mu_k - 7$, respectively.

Additionally, the use of (11) reduces the cost of an evaluation of a truncation of (1) by a factor of about $N/2$.

3 The first algorithm

In this section we present the first of our two new algorithms for computing all Schur functions of the set $\mathcal{S}_{N,n}$. This algorithm is based on the classical definition of the Schur function [32, Section 7.10]:

\[
s_{\kappa}(x_1, \ldots, x_k) = \sum_{T \in A_{\kappa}} X^T,
\]

where the summation is over the set $A_{\kappa}$ of all semistandard $\kappa$-tableaux $T$ filled with the numbers $1, 2, \ldots, k$. Also, $X^T = x_1^{c_1} \cdots x_k^{c_k}$, and $(c_1, \ldots, c_k)$ is the content of $T$. Extending the notation, let:

\[
S_{\kappa}^{(m)}(x_1, \ldots, x_k) = \sum_{T \in A_{\kappa,m}} X^T,
\]

where the summation is over the set $A_{\kappa,m}$, which equals $A_{\kappa}$ with the additional restriction that $k$ does not appear in the first $m$ rows.

Note that $s_{\kappa}(x_1, \ldots, x_k) = S_{\kappa}^{(1)}(x_1, \ldots, x_k)$ and $s_{\kappa}(x_1, \ldots, x_{k-1}) = S_{\kappa}^{(k)}(x_1, \ldots, x_k)$.

Lemma 1 The following identity holds for all $S_{\kappa}^{(m-1)}(x_1, \ldots, x_k)$:

\[
s_{\kappa}^{(m-1)}(x_1, \ldots, x_k) = \begin{cases} 
S_{\kappa}^{(m)}(x_1, \ldots, x_k), & \text{if } \kappa_m = \kappa_{m+1}; \\
S_{\kappa}^{(m-1)}(x_1, \ldots, x_k) \cdot x_k + S_{\kappa}^{(m)}(x_1, \ldots, x_k), & \text{otherwise},
\end{cases}
\]

where the partition $\kappa_{(m)}$ is defined as in (5).

Proof In the first case ($\kappa_m = \kappa_{m+1}$), no $k$ is allowed in the $m$th row of $T$ because of the strictly increasing property of each column of $T$. Therefore, the restriction that no $k$ appear in the first $m - 1$ rows of $T$ is equivalent to the restriction that no $k$ appear in the first $m$ rows of $T$, and $A_{\kappa,m} = A_{\kappa,m-1}$.

In the second case, there are two possibilities for the $\kappa$-tableau $T \in A_{\kappa,m-1}$. If the entry in position $(m, \kappa_m)$ is not equal to $k$, then none of the entries in the $m$th row can equal $k$ due to the nondecreasing nature of each row. Thus, the tableaux fitting this description are exactly the set $A_{\kappa,m}$.

If the entry in position $(m, \kappa_m)$ is equal to $k$, then removal of that square of the tableau clearly results in an element of $A_{\kappa_{(m)},m-1}$. Further, for every tableau in $A_{\kappa_{(m)},m-1}$, the addition of a square containing $k$ to the $m$th row results in a valid semistandard tableau in $A_{\kappa,m-1}$. The tableau retains its semistandardness because every element in the $m$th row (and in the
entire table as well) can be no larger than \( k \), and every element in the \( \kappa_m \)th column above the new square can be no larger than \( k - 1 \) due to the restriction that every tableau in \( A_{K_{n,m},m-1} \) cannot have \( k \) in the first \( m - 1 \) columns.

We have thus constructed a bijection \( f \) mapping \( A_{K_{n,m},m-1} \) to the set (call it \( B \)) of tableaux in \( A_{K,m-1} \) where the entry in position \((m, \kappa_m)\) equals \( k \). Clearly, for each \( T \in A_{K_{n,m},m-1} \), \( X^T \cdot x_k \) and \( \sum_{T \in B} X^T \cdot x_k \).

Combining these two possibilities for \( T \in A_{K,m-1} \), we obtain

\[
S_{m-1}^m(x_1, \ldots, x_k) = \sum_{T \in A_{K,m-1}} X^T
= \sum_{T \in A_{K,m}} X^T + \sum_{T \in A_{K_{n,m}-1}} X^T \cdot x_k
= S_k^m(x_1, \ldots, x_k) + S_{K_{n,m}}^{m-1}(x_1, \ldots, x_k) \cdot x_k,
\]

concluding our proof.

Our algorithm, based on Lemma 1, is very simple.

**Algorithm 1** The following algorithm computes all Schur functions in \( \mathcal{F}_{N,m} \)

for all \( \kappa \in \mathcal{F}_{N,k} \) initialize \( s_\kappa = x_1^{|\kappa|} \) if \( \kappa \in \mathcal{F}_{N,1} \) and \( s_\kappa = 0 \) otherwise

for \( k = 2 \) to \( n \) *(Loop 1)*

for \( m = k \) down to \( 1 \) *(Loop 2)*

for all \( \kappa \in \mathcal{F}_{N,k} \) such that \( \kappa_m > \kappa_{m+1} \), in reverse lexicographic order

\( s_\kappa = s_\kappa + s_{\kappa_m} \cdot x_k \)

endfor

endfor

After the first line Algorithm 1, the variables \( s_\kappa \) contain \( s_\kappa(x_1) \). During each iteration of Loop 1, the values stored in \( s_\kappa \) for \( \kappa \in \mathcal{F}_{N,k} \) are updated from \( s_\kappa(x_1, \ldots, x_{k-1}) = s_\kappa(x_1, \ldots, x_k) \) to \( s_\kappa(x_1, \ldots, x_k) = s_\kappa(x_1, \ldots, x_k) \). During each iteration of Loop 2, the values in \( s_\kappa \) for \( \kappa \in \mathcal{F}_{N,k} \) are updated from \( s_\kappa(x_1, \ldots, x_k) \) to \( s_\kappa(x_1, \ldots, x_k) \).

The last line of the algorithm implements Lemma 1. Since the partitions are processed in reverse lexicographic order, \( s_{K_{n,m}} \) will have already been updated for each \( \kappa \) when this line is executed. Thus, at the time \( s_\kappa \) is updated, \( s_{K_{n,m}} \) contains \( s_{K_{n,m}}^{m-1}(x_1, \ldots, x_k) \), and \( s_\kappa \) is updated from \( s_\kappa(x_1, \ldots, x_k) \) to \( s_\kappa^{m-1}(x_1, \ldots, x_k) \). Our algorithm updates the Schur functions “in place” using a single memory location for each partition.

In order to complete the algorithm in time \( \mathcal{O}(n^3) \) per Schur function, we must be able to lookup the memory location of \( s_{K_{n,m}} \) in constant time for each \( \kappa \in \mathcal{F}_{N,m} \) and \( 1 \leq m \leq n \).

In our implementation, we keep all of the Schur variables \( \{s_\kappa \}_{\kappa \in \mathcal{F}_{N,m}} \) in an array and use a lookup table of size \( |\mathcal{F}_{N,m}| \cdot n \) to find the appropriate array index for each \( s_{K_{n,m}} \). Since this lookup table is invariant over all possible inputs \( x_1, \ldots, x_n \), we simply precompute it (or load it from disk) and keep it in persistent memory for future calls to our algorithm.

It is also possible to compute the indexes of each \( s_{K_{n,m}} \) on the fly rather than having them stored in a lookup table. This modification reduces the memory requirement from \( \mathcal{O}(|\mathcal{F}_{N,m}| \cdot n) \) to \( \mathcal{O}(|\mathcal{F}_{N,k}|) \), but increases the time complexity by a factor of \( n \) to \( \mathcal{O}(n^3) \) per Schur function. In practice, the constant hidden by the big-\( \mathcal{O} \) notation (for time complexity) is greatly increased when computing indices on the fly; therefore, since \( n \ll N \) we have found the lookup table approach to be much more efficient. Further discussion of implementation issues and MATLAB code for both methods are available online [21].
4 The second algorithm

Our second algorithm is based on the expression of the Schur function as a quotient of totally nonnegative generalized Vandermonde determinants:

\[ s_κ(x_1, \ldots, x_n) = \frac{\det G}{\det V_{n,m}}, \]  

(12)

where

\[ G \equiv (x_i^{j-1+κ_{i,j-1}})_{i,j=1}^n \quad \text{and} \quad V_{n,m} \equiv (x_i^{j-1})_{i,j=1}^n \]

are \( n \times n \) generalized and ordinary Vandermonde matrices, respectively.

Since \( s_κ(x_1, x_2, \ldots, x_n) \) is a symmetric polynomial, we can assume that the \( x_i \)'s are sorted in increasing order: \( 0 ≤ x_1 ≤ x_2 ≤ \cdots ≤ x_n \). This choice of ordering makes \( G \) and \( V_{n,m} \) totally nonnegative [9, p. 76] thus the methods of [23, Section 6] can be used to evaluate (12) with guaranteed accuracy in \( O(n^2κ) \) time. The matrices \( G \) and \( V_{n,m} \) are notoriously ill conditioned [11] meaning that conventional Gaussian-elimination-based algorithms will quickly lose all accuracy to roundoff [6].

The contribution of this section is to show how to eliminate the removable singularity at \( x_i = x_j, i ≠ j \) and to arrange the computations in such a way that the cost per Schur function is only \( O(n^2) \) when evaluating all of \( S_{N,κ} \).

It is convenient to see \( G \) as a submatrix of the rectangular Vandermonde matrix

\[ V_{n,m} \equiv (x_i^{j-1+n,m})_{i,j=1}^n. \]

\( m = n - 1 + κ_1 \), consisting of columns \( 1 + κ_1, 2 + κ_{n-1}, \ldots, n - 1 + κ_1 \).

Consider the LDU decomposition \( V_{n,n-1+κ_1} = LDU \), where \( L \) is a unit lower triangular \( n \times n \) matrix, \( D \) is a diagonal \( n \times n \) matrix, and \( U \) is a unit upper triangular \( n \times (n - 1 + κ_1) \) matrix.

The critical observation here is that the value of \( \det G \) is unaffected by \( L \), namely

\[ \det G = \det D \cdot \det U, \]

where \( U \) is the \((n \times n)\) submatrix of \( U \) consisting of its columns \( 1 + κ_1, 2 + κ_{n-1}, \ldots, n - 1 + κ_1 \).

However, \( \det D = \det V_{n,m} \), thus

\[ s_κ(x_1, x_2, \ldots, x_n) = \det U. \]  

(13)

The explicit form of \( U \) is known [8, Section 2.2], [33, eq. (2.3)], allowing us to write (13) also as:

\[ s_κ(x_1, \ldots, x_n) = \det \left( h_{i-j+κ_{n-j+1}}(x_1, \ldots, x_j) \right)_{i,j=1}^n, \]

where \( h_k, k = 1, 2, \ldots, \) are the complete symmetric polynomials and, by default, \( h_k \equiv 0 \) for \( k < 0 \).

In order to apply the algorithms of [23] to evaluate (13), we need the bidirectional decomposition of \( U \), which has a particularly easy form:

\[ B D(U) = \begin{pmatrix} 1 & x_1 & x_1 & \ldots & x_1 & x_1 & \ldots \\ 0 & 1 & x_2 & \ldots & x_2 & x_2 & \ldots \\ 0 & 0 & 1 & \ldots & x_3 & x_3 & \ldots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ 0 & 0 & 0 & \ldots & 1 & x_n & x_n \ldots \end{pmatrix} \]  

(14)
For example, for \( n = 3, m = 4 \) [22, Section 3]:

\[
U = \begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & x_2 & 1 & x_3 \\
1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1
\end{bmatrix}.
\]

Therefore computing the Schur function consists of using Algorithm 5.6 from [23] \( \kappa_1 \) times to remove the appropriate \( \kappa_1 \) columns in \( U \) and obtain the bidiagonal decomposition of \( \bar{U} \). The determinant of \( \bar{U} \), i.e., the Schur function, is easily computable by multiplying out the diagonal of the diagonal factor in the bidiagonal decomposition of \( \bar{U} \).

The total cost is \( O(n^2 \kappa_1) \).

In this process of computing \( s_\kappa(x_1, \ldots, x_n) \) a total of \( \kappa_1 \) (intermediate) Schur functions are computed, therefore all functions in the \( S_{N, n} \) can be computed at the cost of \( O(n^2) \) each.

5 Open problems

It is natural to ask if the ideas of this paper can extend beyond \( \alpha = 1 \) and in particular to \( \alpha = 2 \), the other value of \( \alpha \) of major practical importance [27].

None of the determinantal expressions for the Schur function are believed to have analogues for \( \alpha \neq 1 \), thus we are skeptical of the potential of the ideas in Section 4 to generalize.

The results of Section 3, however, may extend beyond \( \alpha = 1 \).

Consider the (column) vector \( s^{(n)} \) consisting of all Schur functions in \( S_{N, n} \) ordered in reverse lexicographic order. Let \( s^{(n-1)} \) be the same set, but on \( n - 1 \) variables \( x_1, \ldots, x_{n-1} \). Then

\[
s^{(n)} = M s^{(n-1)}
\]

where \( M \) is an \( |S_{N,n}| \times |S_{N,n-1}| \) matrix whose entries are indexed by partitions and \( M_{\mu \nu} = x^{(x)-y} \) if \( \mu / \nu \) is a horizontal strip and 0 otherwise. The contribution of Section 4 was to recognize that \( M \) consists of blocks of the form

\[
A = \begin{bmatrix}
1 \\
x \\
x^2 \\
x^3
\end{bmatrix}.
\]

Since \( A^{-1} \) is bidiagonal:

\[
A^{-1} = \begin{bmatrix}
1 \\
x \\
-x \\
-x
\end{bmatrix},
\]

given a vector (call it \( z \)), the matrix-vector product \( y = Az \) can be formed in linear (instead of quadratic) time by solving instead the bidiagonal linear system \( A^{-1}y = z \) for \( y \).

This was our original approach in designing Algorithm 1. Ultimately we found the much more elegant proof which we presented instead.
The question is whether this approach can be generalized to other values of $\alpha$. Unfortunately the matrix $A$ in general has the form:

$$A^{(\alpha)} = \begin{bmatrix}
1 & x & \frac{1}{\Gamma(\alpha+1)} & \frac{1}{\Gamma(1+\alpha)(1+2\alpha)} & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} \\
x & 1 & x & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} \\
\frac{1}{\Gamma(1+\alpha+1)} & x & 1 & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} \\
\frac{1}{\Gamma(1+\alpha+1)} & x & \frac{1}{\Gamma(1+\alpha+1)} & 1 & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} \\
\frac{1}{\Gamma(1+\alpha+1)} & x & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} & 1 & \frac{1}{\Gamma(1+\alpha+1)} \\
\frac{1}{\Gamma(1+\alpha+1)} & x & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} & \frac{1}{\Gamma(1+\alpha+1)} & 1 \\
\end{bmatrix},$$

where the general expression for the entries $a_{ij}^{(\alpha)}$ is

$$a_{ij}^{(\alpha)} = \frac{x^{i-j}}{(i-j)!} \prod_{k=0}^{i-j-1}(k\alpha + 1), i > j.$$

The matrix $(A^{(\alpha)})^{-1}$ is not bidiagonal for $\alpha \neq 1$ thus the approach of Section 3 cannot be carried over directly. One could consider exploiting the Toeplitz structure of $A^{(\alpha)}$ to form a matrix-vector product with it in $O(k \log k)$ instead of $k^2$ time (assuming $A^{(\alpha)}$ is $k \times k$) [12, p. 193]. Current computing technology, however, limits $N$ to about 200 and since $k \leq N$, this approach does not appear feasible in practice at this time.

**Acknowledgements** We thank James Demmel for posing the problem of computing the Schur function accurately and efficiently [5, Section 9.1(2)] as well as Iain Johnstone and Richard Stanley for many fruitful discussions during the development of this material.

**References**

3. Chen, W.R.: Table for upper percentage points of the largest root of a determinantal equation with five roots