

ACCURATE AND EFFICIENT EVALUATION OF SCHUR AND JACK FUNCTIONS

JAMES DEMMEL AND PLAMEN KOEV

ABSTRACT. We present new algorithms for computing the values of the Schur $s_\lambda(x_1, x_2, \dots, x_n)$ and Jack $J_\lambda^\alpha(x_1, x_2, \dots, x_n)$ functions in floating point arithmetic. These algorithms deliver guaranteed high relative accuracy for positive data $(x_i, \alpha > 0)$ and run in time that is only linear in n .

1. INTRODUCTION

The Schur function $s_\lambda(x_1, x_2, \dots, x_n)$ and its generalization—the Jack function $J_\lambda^\alpha(x_1, x_2, \dots, x_n)$ —play an important role in combinatorics, representation theory, mathematical physics, and multivariate statistical analysis [16, 22, 25, 29]. Yet, when attempting to compute their numerical values, one is faced with either extremely inefficient formulas (e.g., long sums of monomials) or simple, but numerically treacherous determinants.

Our goal in this paper is to design algorithms for computing the values of the Schur and Jack functions accurately and efficiently in floating point arithmetic for positive data $x_i > 0, \alpha > 0$. By *accurately* we mean with *guaranteed* “high relative accuracy”—the output must have correct sign and leading digits. By *efficiently* we mean in time that grows as a low order polynomial in the size of the data.

Our motivation in computing the value of the Schur function stems from the fact that an accurate and efficient algorithm for computing the Schur function immediately yields new accurate and efficient algorithms for solving totally positive generalized Vandermonde linear systems [8], improving the results of [30]. It also yields new algorithms for computing the eigenvalues and the singular value decomposition of totally positive generalized Vandermonde matrices accurately [19]. The practical interest in the Vandermonde matrices coupled with their typical notorious ill conditioning [11] has rendered traditional structure-ignoring matrix algorithms nearly useless, and it has led to the development of accurate structure-exploiting

Received by the editor March 9, 2004 and, in revised form, October 14, 2004.

2000 *Mathematics Subject Classification*. Primary 65G50; Secondary 05E05.

Key words and phrases. Schur function, Jack function, zonal polynomial, high relative accuracy.

This material is based in part upon work supported by the LLNL Memorandum Agreement No. B504962 under DOE Contract No. W-7405-ENG-48, DOE Grants No. DE-FG03-94ER25219, DE-FC03-98ER25351 and DE-FC02-01ER25478, NSF Grant No. ASC-9813362, and Cooperative Agreement No. ACI-9619020.

This work was partially supported by National Science Foundation Grant No. DMS-0314286.

©2005 American Mathematical Society
Reverts to public domain 28 years from publication

algorithms [3, 6, 14]. These algorithms generalize to generalized Vandermonde matrices [8, 19] as long as we can compute the value of the Schur function accurately and efficiently, which is the topic of this paper.

Our motivation in computing the Jack function comes from its connection with the hypergeometric function of matrix argument. The latter has a variety of applications in multivariate statistical analysis [25], yet it has been notoriously difficult to evaluate even in the simplest cases [4, 12].

Our efforts are further justified by the fact that the Schur and Jack functions are accurately determined by the input data. In other words, small relative perturbations in the input arguments cause small relative perturbations in the output.

To design algorithms for computing the Schur and Jack functions, we look at various determinantal expressions and formulas that may be used for their evaluation, and we then address the accuracy and efficiency of each one.

We start with the Schur function.

Let $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_p)$, $\lambda_1 \geq \dots \geq \lambda_p > 0$ be a partition of $|\lambda| \equiv \lambda_1 + \dots + \lambda_p$ in p parts. We denote the *conjugate* partition of λ as $\lambda' = (\lambda'_1, \lambda'_2, \dots, \lambda'_{\lambda_1})$, where $\lambda'_i = \#\{\lambda_j \geq i\}$. In *Frobenius notation* [22, p. 3], λ is written as

$$\lambda = (\alpha_1, \alpha_2, \dots, \alpha_r | \beta_1, \beta_2, \dots, \beta_r).$$

Here $r = \max\{i | \lambda_j \geq i\}$ is the *rank* (also called *the length of the diagonal*) of λ [29, p. 289], $\alpha_i = \lambda_i - i$, and $\beta_i = \lambda'_i - i$. A *hook* partition $(a|b)$ is defined as $(a+1, 1^b)$ and a *ribbon* partition $[\alpha_i | \beta_j]$ is defined as

$$[\alpha_i | \beta_j] = \lambda - (\alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_r | \beta_1, \dots, \beta_{j-1}, \beta_{j+1}, \dots, \beta_r)$$

for $1 \leq i, j \leq r$. Let e_i and h_i be the elementary and complete symmetric polynomials, respectively.

The following expressions may be used to compute $s_\lambda(x_1, x_2, \dots, x_n)$:

$$(1.1) \quad \text{Quotient of alternants} \quad \det(x_i^{j-1+\lambda_n-j+1})_{1 \leq i, j \leq n} / \det(x_i^{j-1})_{1 \leq i, j \leq n}$$

$$(1.2) \quad \text{Giambelli} \quad \det(s_{(\alpha_i | \beta_j)})_{1 \leq i, j \leq r}$$

$$(1.3) \quad \text{Lascoux–Pragacz} \quad \det(s_{[\alpha_i | \beta_j]})_{1 \leq i, j \leq r}$$

$$(1.4) \quad \text{Jacobi–Trudi} \quad \det(h_{\lambda_i - i + j})_{1 \leq i, j \leq p}$$

$$(1.5) \quad \text{Dual Jacobi–Trudi} \quad \det(e_{\lambda'_i - i + j})_{1 \leq i, j \leq \lambda_1}$$

$$(1.6) \quad \text{Combinatorial definition} \quad \sum_{T\text{-SSYT}} x^T.$$

The summation in the last expression is over all semistandard Young tableaux (SSYT) of shape λ [29, p. 308].

The expressions (1.1)–(1.6) appear in [21] and [22, pp. 40, 47, 87, 41, 73]; (1.1) and (1.4)–(1.6) also appear in [29, pp. 334, 342, 344, 308]; (1.5) is also called the *Nägelsbach–Kostka formula* in [21]. We will repeatedly refer to these identities throughout this paper.

The challenge in computing the Schur function in floating point arithmetic is in achieving the *twin* goals of guaranteed accuracy *and* efficiency. As we will explain below, none of the expressions (1.1)–(1.6) fit that bill. The expressions (1.1)–(1.5) are all very efficient (it takes not more than $O(k^3)$ arithmetic operations to evaluate a k -by- k determinant), but roundoff errors may render them inaccurate. The situation is reversed with the last expression (1.6)—it is always accurate when

$x_i > 0$, but a straightforward implementation is not efficient since it represents the addition of $O(n^{|\lambda|})$ terms.

In our complexity analysis we assume that x_i and α are floating point numbers with some fixed number of significant digits, and that we want as many significant digits in the output. The number of variables n and the λ_i are integer inputs. We choose $|\lambda|$ as the measure of the size of the input partition λ , since we believe that $|\lambda| \leq \lambda_1 \cdot \lambda'_1$ better captures the size of λ than either λ'_1 (the number of parts of λ) or λ_1 (the number of parts of λ'). Our goal is an algorithm whose complexity is bounded by a polynomial function of n and $|\lambda|$. In particular the complexity must be independent of the values of the floating point inputs x_i and α .

To understand and overcome the computational challenges in evaluating the Schur and Jack functions, we look at how the efficiency of an algorithm is measured and how accuracy is lost.

Measuring the efficiency is straightforward. The cost of an algorithm that performs m arithmetic operations in N -digit floating point arithmetic is $O(m \cdot N \log N \log \log N)$. (Straightforward implementation of N -digit arithmetic costs $O(N^2)$ [26] or just $O(N \log N \log \log N)$ if [27] is used for multiplication and Newton’s method for division.) In double precision binary floating point arithmetic [2], $N = 53$ binary digits and the cost of an algorithm is typically said to be $O(m)$.

The issue of accuracy is more subtle. We account for roundoff errors in N -digit (binary) floating point arithmetic using the standard model [15, §2.2] in which we assume that the relative error of any arithmetic operation is small:

$$(1.7) \quad \text{fl}(x \odot y) = (x \odot y)(1 + \delta), \quad \text{where } \odot \in \{+, -, \times, /\} \text{ and } |\delta| \leq \epsilon,$$

where $\epsilon = 2^{-N}$ is called *machine precision*. This model implies that products, quotients, and sums of like-signed quantities are computed accurately, i.e., with low relative error. Also, if x and y are inputs (and so can be considered exact), then $\text{fl}(x \pm y) = (x \pm y)(1 + \delta)$ is computed accurately. Loss of relative accuracy can only occur when subtracting approximate same-sign quantities of the same magnitude. A subtraction $c = a - b$ results in the loss of $\log_2 a - \log_2 c = \log_2 a/c$ binary digits. This phenomenon is known as *subtractive cancellation* and is the reason for loss of accuracy in linear algebra algorithms.

The standard remedy for loss of accuracy is to increase the precision (i.e., increase N). The complexity of the algorithm also increases and it does so slightly superlinearly with N . Whether or not such an approach is efficient depends on whether the total amount of subtractive cancellation in an algorithm (and therefore the number of digits N that need to be used) is bounded by a polynomial in n and $|\lambda|$.

Rounding errors can cause arbitrary loss of accuracy in evaluating (1.1)–(1.4). In Sections 2 and 3 we show that these expressions will not lead to accurate and efficient algorithms for computing the Schur function. Interestingly, the determinants (1.2)–(1.4) may be inaccurate regardless of how they are evaluated—rounding errors resulting just from *storing* them in the computer cause major loss of accuracy.

In Section 4 we prove that the amount of subtractive cancellation in evaluating the dual Jacobi–Trudi determinant (1.5) is bounded. This fact leads to our first main result—an accurate and efficient algorithm for evaluating the Schur function based on (1.5). The total cost is $O((n|\lambda| + \lambda_1^3)(|\lambda|\lambda_1)^{1+\rho})$, where ρ is a tiny constant that accounts for certain logarithmic functions.

The final expression for the Schur function—its combinatorial definition (1.6)—involves no subtractions and is therefore always accurate. The only problem is efficiency. In Section 5 we use dynamic programming to design our second accurate Algorithm 5.2 for computing the Schur function. Although this algorithm is not efficient according to our definition (it is linear in n , but superpolynomial in $|\lambda|$) it is exponentially faster than an explicit evaluation of (1.6). It does not require extra precision to guarantee its accuracy.

Most importantly, Algorithm 5.2 generalizes to yield to our third main result in this paper—Algorithm 6.2 for computing the Jack function.

None of the determinantal identities (1.1)–(1.5) have known equivalents for the Jack function. Algorithm 6.2 is the fastest algorithm of any kind for computing J_κ^α and has led to the first practical algorithm [1, 10] for computing the hypergeometric function of matrix argument [20].

Algorithms 5.2 and 6.2 are accurate since they only add, multiply, and divide positive numbers. They can produce *symbolic* output when the inputs x_i are symbolic variables in a symbolic computing environment such as MAPLE [24].

In Section 7 we develop the perturbation theory and error analysis. Finally, we present numerical experiments in Section 8.

2. THE SCHUR FUNCTION AS A QUOTIENT OF ALTERNANTS

The classical definition of the Schur function (1.1) as the quotient of a generalized and ordinary Vandermonde determinants,

$$s_\lambda(x_1, \dots, x_n) = \frac{\det G}{\det V}, \text{ where } G = (x_i^{j-1+\lambda_{n-j+1}})_{1 \leq i, j \leq n} \text{ and } V = (x_i^{j-1})_{1 \leq i, j \leq n},$$

is perhaps the most unstable way of computing it. (Generalized) Vandermonde matrices are notoriously ill conditioned [11]. In order to guarantee any relative accuracy in the computed $\det G$, standard linear algebra algorithms need to use at least $\log_2 \kappa_1(G) = \log_2 \|G\|_1 \cdot \|G^{-1}\|_1$ binary digits. The condition number $\kappa_1(G)$ cannot be bounded as a function of n and $|\lambda|$; it depends on x_i and can be arbitrarily large. For example, $\lambda = (0)$, $x_i = i$, $i = 2, 3, \dots, n$, and $x_1 > n$ yield

$$\|G^{-1}\|_1 \geq |(G^{-1})_{n-1, n}| = \left| \frac{\sum_{i=1}^{n-1} x_i}{\prod_{i=1}^{n-1} (x_i - x_n)} \right| \geq \left| \frac{x_1}{(x_1 - n)(n - 2)!} \right| \geq \frac{1}{n!},$$

$\|G\|_1 \geq x_1^n$, and $\kappa_1(G) \geq x_1^n/n!$ [11, (2.6)]. Therefore, (1.1) will not lead to an accurate and efficient algorithm for computing the Schur function. In Section 8 we present numerical examples that demonstrate its numerical instability. The formula $\det V = \prod_{i>j} (x_i - x_j)$ involves no subtractive cancellation and is always accurate. If we tried to use a similar formula for $\det G$, we would arrive back where we started—at the problem of computing the Schur function accurately.

3. THE GIAMBELLI, LASCoux-PRAGACZ AND JACOBI-TRUDI DETERMINANTS

In this section we demonstrate that the standard approach of using extra precision to evaluate the determinants (1.2)–(1.4) will not lead to efficient algorithms for computing their values accurately.

In particular, we show that regardless of the amount of precision we use, there exist data (x_i) which make the floating point representations of the matrices from (1.2)–(1.4) singular (whereas $s_\lambda(x_1, x_2, \dots, x_n) > 0$ for $x_i > 0$). In other words,

roundoff errors resulting just from storing these matrices in the computer cause irrecoverable loss of accuracy.

When $\lambda = (2, 2) = (1, 0|1, 0)$ both the Giambelli (1.2) and Lascoux–Pragacz (1.3) determinants evaluate the same expression

$$\begin{aligned}
 s_\lambda(x, y) &= \det \begin{pmatrix} s_{(1|1)} & s_{(1|0)} \\ s_{(0|1)} & s_{(0|0)} \end{pmatrix} = \det \begin{pmatrix} s_{[1|1]} & s_{[1|0]} \\ s_{[0|1]} & s_{[0|0]} \end{pmatrix} = \det \begin{pmatrix} s_{(2,1)} & s_{(2)} \\ s_{(1,1)} & s_{(1)} \end{pmatrix} \\
 (3.1) \quad &= \det \begin{pmatrix} x^2 + xy + y^2 & xy(x + y) \\ x + y & xy \end{pmatrix}.
 \end{aligned}$$

Say we use m -digit binary floating point arithmetic. If $x = 2^{m+2}$ and $y = 1$, then $\text{fl}(x + y) = x$, $\text{fl}(x^2 + xy + y^2) = x^2$, and the determinant (3.1) becomes

$$(3.2) \quad \det \begin{pmatrix} x^2 & x^2 \\ x & x \end{pmatrix}.$$

Any reasonable algorithm will evaluate (3.2) as zero, whereas $s_{(2,2)}(x, y) = x^2y^2$.

The Jacobi–Trudi determinant (1.4) is susceptible to the same problem. For example when $\lambda = (1, 1)$

$$s_\lambda(x, y) = \det \begin{pmatrix} h_1 & h_2 \\ 1 & h_1 \end{pmatrix} = \det \begin{pmatrix} x + y & x^2 + xy + y^2 \\ 1 & x + y \end{pmatrix}.$$

Again, in m -digit binary floating point arithmetic, $x = 2^{m+2}$ and $y = 1$ make the above determinant zero, far from its true value xy .

4. THE DUAL JACOBI–TRUDI DETERMINANT

In this section we prove that evaluating the *dual* Jacobi–Trudi determinant (1.5), using Gaussian elimination with no pivoting in $(2|\lambda|\lambda_1 \log_2 \lambda_1 + d)$ -digit binary floating point arithmetic, produces the value of the Schur function correct to at least d digits. This leads to our accurate and efficient Algorithm 4.3 for computing $s_\lambda(x_1, x_2, \dots, x_n)$ with complexity

$$O((n|\lambda| + \lambda_1^3)(|\lambda|\lambda_1)^{1+\rho}),$$

where ρ is tiny and accounts for certain logarithmic functions.

Theorem 4.1. *Let $A = (e_{i-i+j})_{1 \leq i, j \leq \lambda_1}$ be the matrix from the dual Jacobi–Trudi determinant (1.5) and let $A = LU$ be its LU decomposition resulting from Gaussian elimination with no pivoting. Then subtractive cancellation can result in the loss of not more than $N \equiv 2|\lambda|\lambda_1 \log_2 \lambda_1$ binary digits in $s_\lambda = \det(U)$.*

Proof. Denote $l_i = \lambda'_i$, i.e., $\lambda' = (l_1, l_2, \dots)'$.

Let $a_{ij}^{(k)}$ be the elements of the k th Schur complement of A . Using MATLAB [23] notation for the submatrices of A ,

$$a_{ij}^{(k)} = \frac{\det A([1 : k - 1, i], [1 : k - 1, j])}{\det A(1 : k - 1, 1 : k - 1)} = \frac{s_{\eta/\nu}}{s_\mu},$$

where $\eta = (l_1 + j, \dots, l_{k-1} + j, l_i + j)'$, $\mu = (l_1, l_2, \dots, l_{k-1})'$, and $\nu = (j^{k-1})'$. The matrix A is totally nonnegative when $x_i > 0$, because all minors of A have the form $s_{\kappa/\theta}$ for some partitions κ and θ . The total nonnegativity is preserved

in Schur complementation; therefore, $a_{ij}^{(k)}$ is obtained by repeatedly subtracting positive quantities from $a_{ij} = e_{l_i-j+i}$. Let $t \equiv l_i - j + i \geq 0$. Then

$$a_{ij}^{(k)} = \frac{s_{\eta/\nu}}{s_\mu} = a_{ij} - \sum_{s=1}^{k-1} l_{is} u_{sj} = e_t - \sum_{s=1}^{k-1} l_{is} u_{sj}.$$

The cumulative subtractive cancellation in forming $a_{ij}^{(k)}$ can result in the loss of at most

$$\log_2 a_{ij} - \log_2 a_{ij}^{(k)} = \log_2 \frac{a_{ij}}{a_{ij}^{(k)}} = \log_2 \frac{s_\mu e_t}{s_{\eta/\nu}}$$

digits. We will now bound $s_\mu e_t / s_{\eta/\nu}$ by the size of the largest coefficient in the expansion of $s_\mu e_t$ as a sum of monomials.

Lemma 4.2. *Let $x^T = x^\theta x^\kappa$ be a monomial participating with a nonzero coefficient in the expansion of $s_\mu e_t$ as a sum of monomials, where θ and κ are SSYT of shapes μ and $(t)'$, respectively. Then x^T also participates with a nonzero coefficient in the expansion of $s_{\eta/\nu}$ as a sum of monomials.*

Proof. It is more convenient to think of θ as a skew-SSYT of shape $\bar{\mu}/\nu$, where $\bar{\mu} = (l_1 + j, \dots, l_{k-1} + j)'$. We utilize the definition of a SSYT from Macdonald [22, p. 5]. Let θ and κ be

$$\nu = \mu^{(0)} \subset \mu^{(1)} \subset \dots \subset \mu^{(r)} = \bar{\mu}, \quad (0) = \theta^{(0)} \subset \theta^{(1)} \subset \dots \subset \theta^{(r)} = (t)'$$

where $\mu^{(i)} - \mu^{(i-1)}$ and $\theta^{(i)} - \theta^{(i-1)}$ are horizontal strips [22, pp. 71–72] for $1 \leq i \leq r$. Then the sequence of partitions

$$\nu = \mu^{(0)} + \theta^{(0)} \subset \mu^{(1)} + \theta^{(1)} \subset \dots \subset \mu^{(r)} + \theta^{(r)} = \bar{\mu} + (t)' = \eta$$

is a SSYT of shape η/ν and content $T = \theta \cup \kappa$, because every skew diagram $(\mu^{(i)} + \theta^{(i)}) - (\mu^{(i-1)} + \theta^{(i-1)})$ is a horizontal strip: $(\mu^{(i)} + \theta^{(i)})_j = \mu_j^{(i)} + \theta_j^{(i)}$ and

$$\mu_1^{(i)} + \theta_1^{(i)} \geq \mu_1^{(i-1)} + \theta_1^{(i-1)} \geq \mu_2^{(i)} + \theta_2^{(i)} \geq \mu_2^{(i-1)} + \theta_2^{(i-1)} \geq \dots$$

Therefore, x^T participates in the expansion of $s_{\eta/\nu}$ as a sum of monomials, as desired. \square

Returning to the proof of Theorem 4.1, we bound $s_\mu e_t / s_{\eta/\nu}$ by the size of the largest coefficient in front of a monomial symmetric function m_θ in the expansion of $s_\mu e_t$ in the monomial symmetric function basis.

Let $h(i, j) \equiv \lambda_i + \lambda'_j - i - j + 1$ be the hook length at $(i, j) \in \lambda$, and let $H(\lambda) \equiv \prod_{(i,j) \in \lambda} h(i, j)$. Then the number of standard Young tableaux of shape λ filled with the numbers $1, 2, \dots, |\lambda|$ is $f^\lambda = |\lambda|! / H(\lambda)$ [29, Cor. 7.21.6, p. 376]. The Kostka numbers $K_{\lambda, \alpha}$ denote the number of SSYT of shape λ and content α . Trivially $K_{\lambda, \alpha} \leq f^\lambda$.

By the Littlewood–Richardson rule,

$$(4.1) \quad s_\mu e_t = \sum_{\gamma} s_\gamma,$$

where the summation is over all partitions γ such that γ/μ is a vertical t -strip [22, (5.17), p. 73]. By comparing the coefficients in front of $x_1 \dots x_{|\lambda|}$ in (4.1) and using

$$|\gamma| = |\eta/\nu| = l_1 + \dots + l_{k-1} + l_i + j \leq |\lambda| + \lambda_1,$$

we obtain

$$\sum_{\gamma} f^{\gamma} = \binom{|\gamma|}{t} f^{\mu} = \frac{|\gamma|!}{H(\mu)t!} \leq k^{|\gamma|} \leq \lambda_1^{|\lambda|+\lambda_1} \leq \lambda_1^{2|\lambda|}.$$

For a (skew) partition α we have $s_{\alpha} = \sum_{\theta \vdash |\alpha|} K_{\alpha, \theta} m_{\theta}$. Now $x_i > 0$ implies $m_{\theta} > 0$ and

$$\frac{s_{\mu} e_t}{s_{\eta/\nu}} = \frac{\sum_{\gamma} \sum_{\theta \vdash |\gamma|} K_{\gamma, \theta} m_{\theta}}{\sum_{\theta \vdash |\eta/\nu|} K_{\eta/\nu, \theta} m_{\theta}} \leq \max_{\theta} \sum_{\gamma} K_{\gamma, \theta} \leq \sum_{\gamma} f^{\gamma} \leq \lambda_1^{2|\lambda|}.$$

Therefore each step of Gaussian elimination can result in the loss of at most $\log_2 \lambda_1^{2|\lambda|} = 2|\lambda| \log_2 \lambda_1$ digits in any given entry of the Schur complement. Subtractive cancellation during the λ_1 steps of Gaussian elimination required to compute $s_{\lambda} = \det A$ will result in the loss of at most

$$(4.2) \quad N \equiv 2|\lambda| \lambda_1 \log_2 \lambda_1$$

digits. □

Algorithm 4.3 (Accurate dual Jacobi–Trudi determinant). The following algorithm computes the Schur function with at least d accurate binary digits.

```

In  $(2|\lambda| \lambda_1 \log_2 \lambda_1 + d)$ -digit binary floating point arithmetic:
Set  $e_1(x_1) = x_1$ 
for  $k = 2 : \lambda_1 + l_1 - 1$ 
  for  $j = 2 : k$ 
     $e_k(x_1, \dots, x_j) = e_k(x_1, \dots, x_{j-1}) + x_j \cdot e_{k-1}(x_1, \dots, x_{j-1})$ 
 $s_{\lambda} = \det (e_{l_i - i + j})_{1 \leq i, j \leq \lambda_1}$  using Gaussian elimination with no pivoting
Round  $s_{\lambda}$  back to  $d$  digits
    
```

We bound the cost of Algorithm 4.3 by using the fact that any arithmetic operation performed in N -digit arithmetic costs $O(N \log N \log \log N)$ (see the discussion in the Introduction). The elementary symmetric functions are computed accurately in a subtraction-free fashion. Since $\lambda_1 + l_1 - 1 \leq |\lambda|$, the cost of Algorithm 4.3 is only linear in n and polynomial in $|\lambda|$:

$$(4.3) \quad O((n|\lambda| + \lambda_1^3)(|\lambda| \lambda_1)^{1+\rho}),$$

where $\rho > 0$ is tiny and accounts for all the logarithmic functions.

We finish this section with a short note on the combinatorial reasons for the amount of subtractive cancellation to be bounded in the dual Jacobi–Trudi determinant (1.5), but not in determinants (1.2)–(1.4).

Let A be an m -by- m matrix from any of the determinants (1.2)–(1.5). Then $s_{\lambda} = \det A$. Let $s_{\nu} = \det A_{1:m-1, 1:m-1}$, $s_{\mu} = A_{mm}$ and $A = LU$ be the LU decomposition of A . Then, as before, the number of digits lost due to subtractive cancellation in computing U_{mm} is $\log(s_{\mu} s_{\nu} / s_{\lambda})$ and the question comes down to deciding when $s_{\mu} s_{\nu} / s_{\lambda}$ is bounded independently of the values of the x_i . Assume that n is not too small (e.g., $n \geq |\lambda|$). Fix x_2, x_3, \dots and consider $f(x_1) = s_{\mu} s_{\nu} / s_{\lambda}$ as a rational function of x_1 only. Since $\deg_{x_1} s_{\lambda} = \lambda_1$ and $\deg_{x_1} s_{\mu} s_{\nu} = \nu_1 + \mu_1$, the function f will be bounded only if $\nu_1 + \mu_1 \leq \lambda_1$. It is now trivial to verify that this only happens in the case of the dual Jacobi–Trudi determinant.

5. THE COMBINATORIAL DEFINITION OF THE SCHUR FUNCTION

In this section we design an algorithm to make the computation of the Schur function using its combinatorial definition

$$(5.1) \quad s_\lambda = \sum_{T \text{--SSYT}} x^T$$

practical. The expression (5.1) is attractive because it only involves multiplications and additions of positive numbers. There will be no need to use extra precision—the value of s_λ will be computed to high relative accuracy.

The only problem with (5.1) is efficiency, since it represents the addition of

$$(5.2) \quad s_\lambda(1^n) = \prod_{(i,j) \in \lambda} \frac{n-i+j}{h(i,j)} = O(n^{|\lambda|})$$

terms, where $h(i, j) = \lambda_i + \lambda'_j - i - j + 1$ is again the hook length at (i, j) [22, Ex. 4, p. 45].

In what follows, we use *dynamic programming* [5] and the combinatorial properties of the Schur function to derive an algorithm based on (5.1). The resulting algorithm is only linear in n and is thus exponentially faster than an explicit evaluation of (5.1).

For compactness we write $x_{1:k}$ for the vector (x_1, x_2, \dots, x_k) .

Example 5.1. We illustrate our main idea by a small example. Straightforward evaluation of $s_{(1,1)} = \sum_{i < j} x_i x_j$ costs $n^2 - n = O(n^2)$ arithmetic operations. Instead, we write

$$s_{(1,1)}(x_{1:n}) = \sum_{i=1}^{n-1} (x_1 + \dots + x_i) \cdot x_{i+1} = \sum_{i=1}^{n-1} s_{(1)}(x_{1:i}) \cdot x_{i+1},$$

and we precompute $s_{(1)}(x_{1:i}) = s_{(1)}(x_{1:i-1}) + x_i$ for $i = 1, 2, \dots, n$, thus performing $n - 1$ arithmetic operations. Finally, we compute $s_{1:n}$ as

$$(5.3) \quad s_{(1,1)}(x_{1:n}) = \sum_{i=1}^{n-1} s_{(1)}(x_{1:i}) \cdot x_{i+1}$$

performing another $2n - 1$ operations for a total complexity of $3n - 2 = O(n)$.

This idea generalizes to arbitrary partitions λ and, to Jack functions as well.

For a partition $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_p)$, the general version of (5.3) is

$$(5.4) \quad s_\lambda(x_{1:n}) = \sum_{\mu} s_\mu(x_{1:n-1}) x_n^{|\lambda/\mu|},$$

where the summation is over all $\mu \leq \lambda$ such that λ/μ is a horizontal strip, i.e.,

$$(5.5) \quad \lambda_1 \geq \mu_1 \geq \lambda_2 \geq \mu_2 \geq \dots$$

When $\lambda_{i+1} < \lambda_i$, we write

$$(5.6) \quad \lambda_{(i)} \equiv (\lambda_1, \dots, \lambda_{i-1}, \lambda_i - 1, \lambda_{i+1}, \dots, \lambda_p)$$

and simplify (5.4) further by observing that when $\lambda_2 < \lambda_1$

$$(5.7) \quad s_{\lambda_{(1)}}(x_{1:n}) = \sum_{\mu} s_\mu(x_{1:n-1}) x_n^{|\lambda_{(1)}/\mu|},$$

where the summation is over all $\mu \leq \lambda_{(1)}$ such that $\lambda_{(1)}/\mu$ is a horizontal strip. Substituting (5.7) into (5.4) we obtain

$$(5.8) \quad s_\lambda(x_{1:n}) = \begin{cases} \sum_{\mu} s_{\mu}(x_{1:n-1})x_n^{|\lambda/\mu|}, & \text{if } \lambda_2 = \lambda_1; \\ s_{\lambda_{(1)}}(x_{1:n})x_n + \sum_{\mu} s_{\mu}(x_{1:n-1})x_n^{|\lambda/\mu|}, & \text{otherwise.} \end{cases}$$

Both summations in (5.8) are over all partitions $\mu \leq \lambda$ such that $\mu_1 = \lambda_1$ and λ/μ is a horizontal strip. Using (5.5) we rewrite (5.8) as

$$(5.9) \quad s_\lambda = \begin{cases} \sum_{\mu_2=\lambda_3}^{\lambda_2} \cdots \sum_{\mu_{p-1}=\lambda_p}^{\lambda_{p-1}} \sum_{\mu_p=0}^{\lambda_p} s_{\mu}(x_{1:n-1})x_n^{|\lambda/\mu|}, & \text{if } \lambda_2 = \lambda_1; \\ s_{\lambda_{(1)}}(x_{1:n})x_n + \sum_{\mu_2=\lambda_3}^{\lambda_2} \cdots \sum_{\mu_{p-1}=\lambda_p}^{\lambda_{p-1}} \sum_{\mu_p=0}^{\lambda_p} s_{\mu}(x_{1:n-1})x_n^{|\lambda/\mu|}, & \text{otherwise.} \end{cases}$$

This is the main equation we will use to recursively compute the Schur function in Algorithm 5.2 below. The key to attaining efficiency is to store all Schur functions as they are computed and to *reuse* their values as necessary.

Algorithm 5.2 (Schur function). The following algorithm computes the Schur function $s_\lambda(x_1, x_2, \dots, x_n)$ given $x = (x_1, x_2, \dots, x_n)$ and $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_p)$, $\lambda_1 \geq \dots \geq \lambda_p > 0$, using only additions and multiplications. All functions are called by value. The variables x and λ are *global*. All other variables are local.

```
function schur(x, λ)
    n = length(x)
    allocate S(1 : N_λ, 1 : n)                                (N_λ is defined in (5.10))
    for every ν ≤ λ set S(N_ν, 1 : n) = "empty"
    return sch(n, 1, λ).

function s = sch(m, k, ν)
    if ν_1 = 0 or m = 0 then return s = 1
    if ν_{m+1} > 0 then return s = 0
    if m = 1 then return s = x_1^{ν_1}
    if S(N_ν, m) ≠ "empty" then return s = S(N_ν, m)
    s = sch(m - 1, 1, ν)
    if ν_1 > ν_2 then
        s = s + x_m · sch(m, 1, ν_{(1)})
        i = 2
    while ν_i > 0
        if ν_i > ν_{i+1} then
            if ν_i > 1 then s = s + x_m · sch(m, i, ν_{(i)})
            else s = s + x_m · sch(m - 1, 1, ν_{(i)})
            i = i + 1
    if k = 1 then S(N_ν, m) = s
```

Algorithm 5.2 implements (5.9) by recursively generating all partitions $\mu \leq \lambda$ such that $\mu_1 = \lambda_1$ and λ/μ is a horizontal strip.

In order to be able to store and recall the values of the computed Schur function, we assign a distinct integer index N_μ to every partition $\mu \leq \lambda$ according to

$$(5.10) \quad N_\mu \equiv \sum_{i=1}^p \mu_i \cdot M_i, \quad \text{where } M_i \equiv \prod_{j=i+1}^p (\lambda_j + 1).$$

For example, if $\lambda = (9, 9, 9)$, this scheme would assign (distinct) three-digit integers to all partitions $\mu \leq (9, 9, 9)$ with $N_{(5,4,1)} = 541$, etc.

There are, of course, other possible indexing schemes. The advantage of the scheme above is that partitions that differ in only one part have a very easy relationship between their indexes: $N_{\mu_{(i)}} = N_\mu - M_i$. Also, it is easy to recover the parts of μ from N_μ at $O(1)$ operations per part:

$$\mu_1 = \left\lfloor \frac{N_\mu}{M_1} \right\rfloor \quad \text{and} \quad \mu_i = \left\lfloor \frac{\text{mod}(N_\mu, M_{i-1})}{M_i} \right\rfloor \quad \text{for } i > 1,$$

where $\text{mod}(q, r)$ is the remainder of q modulo r ;

In our MATLAB implementation of Algorithm 5.2 (available from [18]) we never manipulate the actual partitions, just their indexes.

The function `sch`(m, k, N_ν) computes $s_\nu(x_1, x_2, \dots, x_m)$ recursively. The parameter k keeps track of the depth of the recursion and ensures that any subsequent partition (call it η) generated further into the recursion satisfies $\eta_i = \nu_i$, $i = 1, 2, \dots, k$.

The recursion terminates when $\lambda = 0$ (then $s_\lambda = 1$); $\lambda = (1)$ and $n = 1$ (then $s_\lambda = x_1$); or $n = 0$ (then $s_\lambda = 0$).

Computed Schur functions are stored in the array `S` and reused as needed without being recomputed.

To compute the sum (5.9), Algorithm 5.2 recursively generates all $\mu \leq \lambda$ such that $\lambda_1 = \mu_1$, and λ/μ is a horizontal strip (i.e., $\lambda_{i+1} \leq \mu_i \leq \lambda_i$ for $i \geq 2$).

Next, we bound the cost of Algorithm 5.2.

A single call to `sch` costs at most $O(p) = O(\lambda_1)$ work. To bound the number of calls to `sch`, let P_m be the number of partitions $|\mu| \leq m$. In the course of recursion at most $P_{|\lambda|} \cdot n$ Schur functions are computed (along with $s_\lambda(x_1, x_2, \dots, x_n)$ whose value we want). These are $s_\mu(x_1, x_2, \dots, x_i)$, $\mu \leq \lambda$, $i = 1, 2, \dots, n$.

There are at most $P_{|\lambda|}$ terms in (5.9) (in fact a lot less). Therefore the cost of Algorithm 5.2 is bounded by $O(P_{|\lambda|}^2 \cdot \lambda_1 \cdot n)$.

There is no explicit formula for the number of partitions $\nu \leq \lambda$ or the number P_m of partitions $|\nu| \leq m$. We can however use Ramanujan's asymptotic formula [13, p. 116] for number of partitions of m

$$p(m) \sim \frac{1}{4m\sqrt{3}} \exp(\pi\sqrt{2m/3})$$

to obtain

$$P_m = \sum_{i=1}^m p(i) \sim O\left(\exp(\pi\sqrt{2m/3})\right) \quad \text{and} \quad P_m^2 \sim O\left(\exp(2\pi\sqrt{2m/3})\right).$$

Therefore the complexity of Algorithm 5.2 is only linear in n and subexponential in $|\lambda|$; it is bounded by

$$(5.11) \quad O\left(e^{5.2|\lambda|^{1/2}} \cdot \lambda_1 \cdot n\right).$$

In a previous work [7], we presented an algorithm in which we bounded the cost of computing $s_\lambda(x_1, x_2, \dots, x_n)$ by

$$O(n(1 + \lambda_1)^{\log n} \dots (1 + \lambda_p)^{\log n})$$

exponentially faster than (5.2). Comparing this to bound (5.11) we see that our new Algorithm 5.2 can be exponentially faster again, indeed only linear in n (although not polynomial in $|\lambda|$ as Algorithm 4.3 is).

Note that the objects stored in the table $\mathbf{S}()$ or returned by function $\mathbf{sch}()$ could be numbers (if the x_i are numbers) or symbolic polynomials (if the x_i are indeterminates). Thus the algorithm above can be used to compute Schur polynomials numerically or symbolically. But there are (at least) two ways to implement symbolic evaluation. The simplest way is to fully evaluate each entry of $\mathbf{S}()$ or value of $\mathbf{sch}()$ as an explicit sum of monomials. In this case the cost is at least as large as the size of the output, or $O(n^{|\lambda|})$. The second, faster way is to exploit previously computed subexpressions in the table $\mathbf{S}()$, and not expand them. In other words, each entry of $\mathbf{S}()$ or value of $\mathbf{sch}()$ is represented as a polynomial in the x_i and previously computed table entries in $\mathbf{S}()$. If we were to write out the symbolic expression for the final result $\mathbf{sch}(1, n, \lambda)$, we would get a sequence of assignment statements, where each statement assigns an explicit polynomial expression in previously computed quantities to a new variable name. This symbolic evaluation of a Schur polynomial as a sequence of assignment statements performing only additions and multiplications (called a DAG or straight line code in computer science language) can be exponentially smaller than an explicit sum of monomials, as the next proposition shows.

Proposition 5.3. *The Schur polynomial can be represented symbolically as a DAG (using only addition and multiplication) whose size is bounded by (5.11).*

Proof. We can place the Schur polynomials $s_\mu(x_1, x_2, \dots, x_i)$ for all $\mu \leq \lambda$ and all $i \leq n$ on the nodes of the graph. We will have at most $P_{|\lambda|}$ edges coming out of each node. Similar to the derivation of (5.11), the total size of the DAG will not exceed (5.11). □

6. COMPUTING THE JACK FUNCTION

The Jack symmetric function

$$J_\lambda^\alpha(x_1, \dots, x_n) = \sum_{T\text{-SSYT}} f_T(\alpha)x^T,$$

is a generalization of the Schur function. It depends on a parameter α and its coefficients are polynomials in α , not just integers [17, 28]. When $\alpha = 1$ we obtain the (normalized) Schur function, and when $\alpha = 2$ we obtain the zonal polynomial.

In this section we use an approach similar to that of Algorithm 5.2 to compute the value of the Jack function. The resulting Algorithm 6.2 is the fastest algorithm of any kind for computing the value of the Jack function. It represents an exponential improvement over the approach taken in [12] for the computation of the zonal polynomials C_κ .

We use a generalization of the recursive formula (5.4).

Proposition 6.1. *For a partition λ let*

$$h_\lambda^*(i, j) \equiv \lambda'_j - i + \alpha(\lambda_i - j + 1) \quad \text{and} \quad h_\lambda^\lambda(i, j) \equiv \lambda'_j - i + 1 + \alpha(\lambda_i - j)$$

be the upper and lower hook lengths at $(i, j) \in \lambda$, respectively. Define

$$(6.1) \quad \beta_{\lambda\mu} \equiv \frac{\prod_{(i,j) \in \lambda} B_{\lambda\mu}^\lambda(i, j)}{\prod_{(i,j) \in \mu} B_{\lambda\mu}^\mu(i, j)}, \quad \text{where} \quad B_{\lambda\mu}^\nu(i, j) \equiv \begin{cases} h_\nu^*(i, j), & \text{if } \lambda'_j = \mu'_j; \\ h_\nu^\nu(i, j), & \text{otherwise.} \end{cases}$$

Then

$$(6.2) \quad J_\lambda^\alpha(x_1, x_2, \dots, x_n) = \sum_{\mu \leq \lambda} J_\mu^\alpha(x_1, x_2, \dots, x_{n-1}) x_n^{|\lambda/\mu|} \beta_{\lambda\mu},$$

where the summation is over all $\mu \leq \lambda$ such that λ/μ is a horizontal strip.

Proof. Define

$$j_\mu \equiv \prod_{(i,j) \in \mu} h_*^\mu(i, j) h_\mu^*(i, j) \quad \text{and} \quad A_{\lambda\mu}^\nu(i, j) \equiv \begin{cases} h_*^\nu(i, j), & \text{if } \lambda'_j = \mu'_j; \\ h_\nu^*(i, j), & \text{otherwise.} \end{cases}$$

Then [28, (20)] $J_\lambda^\alpha(x_1, x_2, \dots, x_n) = \sum_{\mu \leq \lambda} J_\mu^\alpha(x_1, x_2, \dots, x_{n-1}) J_{\lambda/\mu}^\alpha(x_n) j_\mu^{-1}$, where the summation is over all $\mu \leq \lambda$ such that λ/μ is a horizontal strip. Let $m = |\lambda/\mu|$. From [28, Thms. 5.8 and 6.1, and Lemma 6.2] we obtain

$$\begin{aligned} J_{\lambda/\mu}^\alpha(x_n) j_\mu^{-1} &= \frac{x_n^m}{\alpha^m m!} \cdot g_{\mu m}^\lambda \cdot j_\mu^{-1} \\ &= \frac{x_n^m}{\alpha^m m!} \left(\prod_{(i,j) \in \mu} A_{\lambda\mu}^\mu(i, j) \right) \left(\prod_{(i,j) \in \lambda} B_{\lambda\mu}^\lambda(i, j) \right) m! \alpha^m j_\mu^{-1} \\ &= x_n^m \left(\prod_{(i,j) \in \mu} \frac{A_{\lambda\mu}^\mu(i, j)}{h_*^\mu(i, j) h_\mu^*(i, j)} \right) \left(\prod_{(i,j) \in \lambda} B_{\lambda\mu}^\lambda(i, j) \right) \\ &= x_n^m \left(\prod_{(i,j) \in \mu} B_{\lambda\mu}^\mu(i, j) \right)^{-1} \left(\prod_{(i,j) \in \lambda} B_{\lambda\mu}^\lambda(i, j) \right), \end{aligned}$$

which yields (6.2). □

Algorithm 6.2 (Jack function). The following algorithm computes the Jack function $J_\lambda^\alpha(x_1, x_2, \dots, x_n)$ given $x = (x_1, x_2, \dots, x_n)$ and $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_p)$, $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_p > 0$. All functions are called by value. The variables x_i , λ_i , and α are *global variables*. All other variables are local.

```
function jack(x, λ, α)
    n = length(x)
    allocate S(1 : N_λ, 1 : n)           (N_λ is defined in (5.10))
    For every ν ≤ λ set S(N_ν, 1 : n) = "empty"
    return jack(n, 0, λ, λ)

function s = jac(m, k, μ, ν)
    if ν_1 = 0 or m = 0 then return s = 1
    if ν_{m+1} > 0 then return s = 0
    if m = 1 then return s = x_1^{ν_1} · ∏_{i=1}^{ν_1-1} (iα + 1)
    if k = 0 and S(N_ν, m) ≠ "empty" then return s = S(N_ν, m)
    i = k; if k = 0 then i = 1
    s = jac(m - 1, 0, ν, ν) · β_{μν} · x_m^{μ-|ν|}           (β_{μν} is computed using (6.1))
    while ν_i > 0
        if ν_i > ν_{i+1} then
            if ν_i > 1 then s = s + jac(m, i, μ, ν_{(i)})
            else s = s + jac(m - 1, 0, ν_{(i)}, ν_{(i)}) · β_{μν_{(i)}} · x_m^{μ-|ν_{(i)}|}
        i = i + 1
    if k = 0 then S(N_ν, m) = s
```

Algorithm 6.2 computes the Jack function by implementing (6.2). The recursion terminates when $\lambda = (0)$ (then $J_{(0)}^\alpha = 1$); $\lambda = (k)$ and $n = 1$ (then $J_{(k)}^\alpha(x_1) = x_1^k(1 + \alpha)(1 + 2\alpha) \cdots (1 + (k - 1)\alpha)$); or $\lambda_{n+1} > 0$ (then $J_\lambda^\alpha(x_1, \dots, x_n) = 0$). In computing the Jack function we do not make use of a relationship of type (5.7) and do not know if such a relationship exists.

We confirmed the correctness of the implementation of Algorithm 6.2 by comparing its output for symbolic input x_i and α to that of the package [9]. We made no speed comparison with the implementation [9] because the latter produces *symbolic* output with complexity independent of n .

To bound the complexity of Algorithm 6.2, we can repeat the complexity analysis of Algorithm 5.2. Since $\beta_{\lambda\mu}$ costs at most $10|\lambda|$ to compute; the cost of Algorithm 6.2 is bounded by

$$(6.3) \quad O\left(e^{5.2|\lambda|^{1/2}} \cdot |\lambda| \cdot \lambda_1 \cdot n\right).$$

Just as in the case of the Schur function, the bound (6.3) is exponentially better than the cost of evaluating the Jack function explicitly.

When $\alpha > 0$ and $x_i > 0$, Algorithm 6.2 computes $J_\lambda^\alpha(x_1, \dots, x_n)$ to high relative accuracy, since the implementation only adds, multiplies, or divides positive numbers.

7. PERTURBATION THEORY AND ERROR ANALYSIS

We will now prove that the values of the Schur and Jack functions are determined accurately by the data (x_i, α) and that Algorithms 5.2 and 6.2, respectively, compute these values to high relative accuracy.

We will accumulate relative errors and relative perturbations in the style of Higham [15]. If $|\delta_i| \leq \delta < 1/k$ and $\rho_i = \pm 1$, then

$$\left| \prod_{i=1}^k (1 + \delta_i)^{\rho_i} - 1 \right| \leq \frac{k\delta}{1 - k\delta},$$

which implies the following perturbation result for polynomials with positive coefficients.

Proposition 7.1. *Let $f(x) = f(x_1, x_2, \dots, x_n) = \sum_T a_T x^T$ be a polynomial of total degree k with positive coefficients a_T (such as $f = s_\lambda$ or $f = J_\lambda^\alpha$). If $\hat{x}_i = x_i(1 + \delta_i)$ are small perturbations of $x_i > 0$ ($|\delta_i| \leq \delta < 1/k$), then*

$$|f(\hat{x}) - f(x)| \leq \frac{k\delta}{1 - k\delta} f(x).$$

In other words small relative perturbations in $x_i > 0$ cause small relative perturbations in $f(x_1, x_2, \dots, x_n)$.

Next, we prove that Algorithms 5.2 and 6.2 compute the Schur and Jack functions to high relative accuracy.

Theorem 7.2. *Let $f_\lambda(x_1, \dots, x_n)$ equal either $s_\lambda(x_1, \dots, x_n)$ or $J_\lambda^\alpha(x_1, \dots, x_n)$, where $x_i > 0$, $i = 1, 2, \dots, n$, and $\alpha > 0$. Let \hat{f}_λ be the value of f_λ computed by Algorithm 5.2 or Algorithm 6.2 (as appropriate) in floating point arithmetic with machine precision ϵ . Then*

$$|f_\lambda - \hat{f}_\lambda| \leq \frac{F\epsilon}{1 - F\epsilon} f_\lambda,$$

where F is the number of arithmetic operations performed to compute f_λ .

Proof. Let $f_\lambda = \sum a_T x^T$, where the summation is over all SSYT T of shape λ , and let $\hat{f}_\lambda = \sum \hat{a}_T x^T$ be the value f_λ computed in floating point arithmetic. The goal is to prove that \hat{a}_T are small perturbations of a_T . The only arithmetic operations performed by Algorithms 5.2 and 6.2 are addition, multiplication, or division. Each of these operations results in only one factor of the form $(1 + \delta_{T,j})^{\rho_i}$, $\rho_i = \pm 1$, contributing to a_T (we assume that the x_i and α are floating point numbers). Therefore

$$a_T = \prod_{j=1}^k (1 + \delta_{T,j})^{\rho_i}, \quad |\delta_{T,j}| \leq \epsilon,$$

where $k \leq F$ is the number of arithmetic operations that x^T participates in. Thus,

$$|a_T - \hat{a}_T| \leq \frac{F\epsilon}{1 - F\epsilon} a_T$$

and

$$|f_\lambda - \hat{f}_\lambda| = \left| \sum_T (a_T - \hat{a}_T) x^T \right| \leq \sum_T |a_T - \hat{a}_T| x^T \leq \frac{F\epsilon}{1 - F\epsilon} f_\lambda,$$

as desired. \square

8. NUMERICAL EXAMPLES

We performed extensive numerical tests to verify the correctness and accuracy of Algorithms 4.3 and 5.2. We present two numerical examples where the performance of these algorithms is fairly typical.

We contrast the accuracy of our Algorithms 4.3 and 5.2 with that of the classical definition of the Schur function as a quotient of alternants (1.1) which becomes inaccurate on very small examples.

In our first example (in the left plot of Figure 1) we demonstrate how quickly the accuracy of (1.1) deteriorates. We computed $s_{(1)}(x_1, \dots, x_n) = x_1 + \dots + x_n$, where $x_i = 1 + (i - 1)/100$ for different values of n .

As expected, both Algorithms 4.3 and 5.2 computed this relatively simple Schur function to 16 decimal digits, whereas (1.1) failed spectacularly to compute a single correct digit beyond $n = 20$. This failure was expected as we discussed in Section 2.

In our second example we computed $s_{(k,3,2,1)}(x_{1:51})$, where $x_i = 1 + (i - 1)/100$ for $k = 10, 15, \dots, 50$, and we plotted the results in the right plot of Figure 1. Algorithm 4.3 delivered the correct result to 16 decimal digits in each case (since the condition number of any matrix in this computation did not exceed 10^{50} and we used at least 100-decimal-digit arithmetic). The evaluation of (1.1) yielded no correct digits. Algorithm 5.2 was always accurate to least 15 correct digits, as expected.

In our experiments the loss of accuracy by Algorithm 4.3 appeared to depend only linearly on $|\lambda|$; it was less dramatic than the upper bound (4.2) would suggest.

We finish this section with a few comments on the relative instability of formulas (1.1)–(1.4). The classical definition (1.1) fared the worst in our experiments delivering accurate results only for very modest values of n and $|\lambda|$ and quickly deteriorating subsequently. In Section 3 we showed that the Giambelli (1.2), the Lascoux–Pragacz (1.3), and the Jacobi–Trudi (1.4) determinants can fail to produce a single correct digit of the Schur function on small examples. Over an entire range of test problems, however, these formulas proved fairly accurate in most

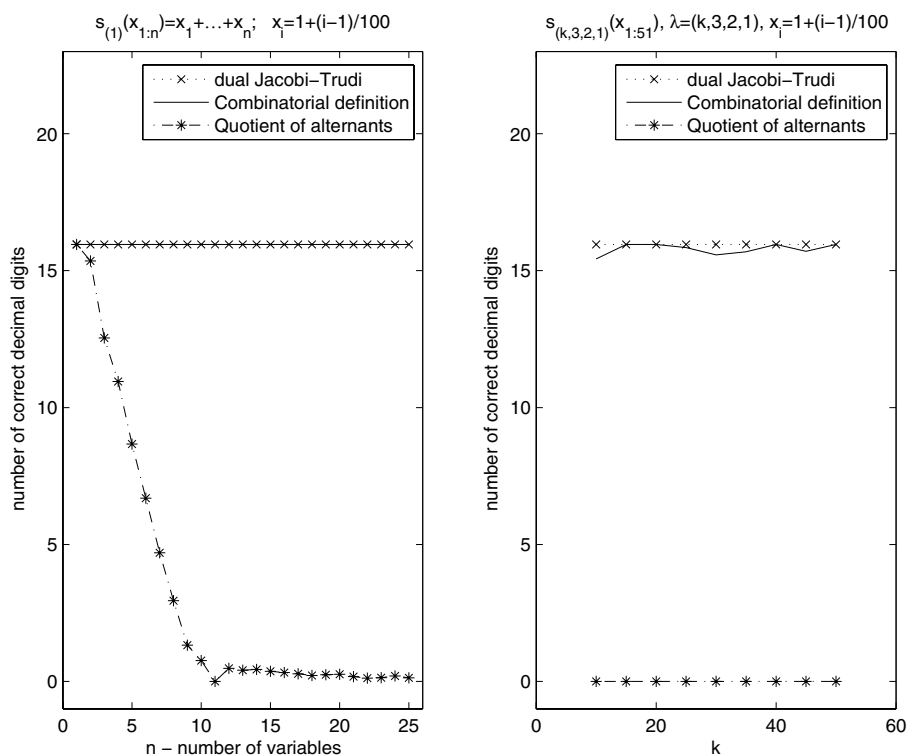


FIGURE 1. The accuracy of Algorithm 4.3 (based on the dual Jacobi–Trudi identity (1.5)), Algorithm 5.2 (based on the combinatorial definition (1.6)), and the classical definition as a quotient of alternants (1.1) evaluated using MATLAB’s function `det`.

cases. Since we have shown examples where all three expressions (1.2)–(1.4) fail to meet our criteria of guaranteed accuracy and efficiency, we did not pursue this observation further.

9. ACKNOWLEDGMENTS

The authors would like to thank Ioana Dumitriu and Alan Edelman for many helpful discussions in the process of adapting our Algorithm 5.2 to compute Jack polynomials, Richard Stanley and Sergey Fomin for their help in bounding the amount of cancellation in the dual Jacobi–Trudi determinant, and the anonymous referees for the careful reading of our manuscript and for making many useful suggestions that resulted in substantial improvements to our presentation.

REFERENCES

1. P.-A. Absil, A. Edelman, and P. Koev, *On the largest principal angle between random subspaces*, <http://www.csit.fsu.edu/~absil/Publi/RandomSubspaces.htm>, Submitted to *Linear Algebra Appl.*
2. ANSI/IEEE, New York, *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 ed., 1985.

3. Å. Björck and V. Pereyra, *Solution of Vandermonde systems of equations*, Math. Comp. **24** (1970), 893–903. MR0290541 (44:7721)
4. R. W. Butler and A. T. A. Wood, *Laplace approximations for hypergeometric functions with matrix argument*, Ann. Statist. **30** (2002), no. 4, 1155–1177. MR1926172 (2003h:62076)
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, second ed., MIT Press, Cambridge, MA, 2001. MR1848805 (2002e:68001)
6. J. Demmel, *Accurate SVDs of structured matrices*, SIAM J. Matrix Anal. Appl. **21** (1999), no. 2, 562–580. MR1742810 (2001h:65036)
7. J. Demmel and P. Koev, *Necessary and sufficient conditions for accurate and efficient rational function evaluation and factorizations of rational matrices*, Structured matrices in mathematics, computer science, and engineering. II (Boulder, CO, 1999), Amer. Math. Soc., Providence, RI, 2001, pp. 117–143. MR1855508 (2002h:65055)
8. ———, *The accurate and efficient solution of a totally positive generalized Vandermonde linear system*. SIAM J. Matrix Anal. Appl., **27** (2005), no. 1, 142–152.
9. I. Dumitriu, A. Edelman, and F. Shuman, *A Maple Package for Computing Multivariate Orthogonal Polynomials*, <http://www.math.berkeley.edu/~dumitriu>.
10. A. Edelman and B. Sutton, *Tails of condition number distributions*, submitted to SIAM J. Matrix Anal. Appl.
11. W. Gautschi, *How (un)stable are Vandermonde systems?*, Asymptotic and computational analysis (Winnipeg, MB, 1989), Lecture Notes in Pure and Appl. Math., vol. 124, Dekker, New York, 1990, pp. 193–210. MR1052434 (91f:65080)
12. R. Gutiérrez, J. Rodríguez, and A. J. Sáez, *Approximation of hypergeometric functions with matricial argument through their development in series of zonal polynomials*, Electron. Trans. Numer. Anal. **11** (2000), 121–130. MR1799027 (2002b:33004)
13. G. H. Hardy, *Ramanujan: twelve lectures on subjects suggested by his life and work.*, AMS Chelsea Publishing Company, New York, 1999. MR0106147 (21:4881)
14. N. J. Higham, *Stability analysis of algorithms for solving confluent Vandermonde-like systems*, SIAM J. Mat. Anal. Appl. **11** (1990), no. 1, 23–41. MR1032215 (91d:65062)
15. ———, *Accuracy and stability of numerical algorithms*, second ed., Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002. MR1927606 (2003g:65064)
16. G. James and A. Kerber, *The representation theory of the symmetric group*, Encyclopedia of Mathematics and its Applications, vol. 16, Addison-Wesley Publishing Co., Reading, Mass., 1981, With a foreword by P. M. Cohn, With an introduction by Gilbert de B. Robinson. MR0644144 (83k:20003)
17. F. Knop and S. Sahi, *A recursion and a combinatorial formula for Jack polynomials*, Invent. Math. **128** (1997), no. 1, 9–22. MR1437493 (98k:33040)
18. P. Koev, <http://www-math.mit.edu/~plamen>.
19. P. Koev, *Accurate eigenvalues and SVDs of totally nonnegative matrices*, SIAM J. Matrix Anal. Appl., **27** (2005), no. 1, 1–23.
20. P. Koev and A. Edelman, *The efficient evaluation of the hypergeometric function of matrix argument*, Math. Comp., to appear.
21. I. G. Macdonald, *Schur functions: theme and variations*, Séminaire Lotharingien de Combinatoire (Saint-Nabor, 1992), Publ. Inst. Rech. Math. Av., vol. 498, Univ. Louis Pasteur, Strasbourg, 1992, pp. 5–39. MR1308728 (95m:05245)
22. ———, *Symmetric functions and Hall polynomials*, 2nd ed., Oxford University Press, 1995. MR1354144 (96h:05207)
23. The MathWorks, Inc., Natick, MA, *MATLAB reference guide*, 1992.
24. M. Monagan, K. Geddes, K. Heal, G. Labahn, and S. Vorkoetter, *Maple V Programming guide for release 5*, Springer-Verlag, 1997.
25. R. J. Muirhead, *Aspects of multivariate statistical theory*, John Wiley & Sons Inc., New York, 1982, Wiley Series in Probability and Mathematical Statistics. MR0652932 (84c:62073)
26. D. Priest, *Algorithms for arbitrary precision floating point arithmetic*, Proceedings of the 10th Symposium on Computer Arithmetic (Grenoble, France) (P. Kornerup and D. Matula, eds.), IEEE Computer Society Press, June 26–28 1991, pp. 132–145.
27. A. Schönhage and V. Strassen, *Schnelle Multiplikation grosser Zahlen*, Computing (Arch. Elektron. Rechnen) **7** (1971), 281–292. MR0292344 (45:1431)
28. R. Stanley, *Some combinatorial properties of Jack symmetric functions*, Adv. Math. **77** (1989), no. 1, 76–115. MR1014073 (90g:05020)

29. ———, *Enumerative combinatorics. Vol. 2*, Cambridge Studies in Advanced Mathematics, vol. 62, Cambridge University Press, Cambridge, 1999. MR1676282 (2000k:05026)
30. H. Van de Vel, *Numerical treatment of a generalized Vandermonde system of equations*, Linear Algebra Appl. **17** (1977), 149–179. MR0474757 (57:14390)

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE DIVISION, UNIVERSITY OF CALIFORNIA, BERKELEY, CALIFORNIA 94720

E-mail address: demmel@eecs.berkeley.edu

DEPARTMENT OF MATHEMATICS, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, CAMBRIDGE, MASSACHUSETTS 02139

E-mail address: plamen@math.mit.edu