A LINEAR-TIME ALGORITHM FOR EVALUATING SERIES OF SCHUR FUNCTIONS

CY CHAN, VESSELIN DRENSKY, ALAN EDELMAN, AND PLAMEN KOEV

Abstract. We present a new algorithm for computing all Schur functions $s_\lambda(x_1, x_2, \ldots, x_n)$ for all partitions λ of integers not exceeding N in time $\mathcal{O}(n^2 K_N)$, where $K_N \equiv \#\{\lambda | \, |\lambda| \leq N\}$ is the number of those partitions.

In particular, this algorithm has optimal complexity for evaluating truncated series of Schur functions such as the hypergeometric function of a matrix argument.

1. Introduction

We present a new highly efficient algorithm for computing the finite truncation (for $k \leq N$) of the hypergeometric function of a matrix argument X in the complex case:

$$ {}_pF_q(a_1, \ldots, a_p; b_1, \ldots, b_q; X) \equiv \sum_{k=0}^{N} \sum_{\kappa \vdash k} \frac{1}{H_\kappa} \cdot \frac{(a_1)_\kappa \cdots (a_p)_\kappa}{(b_1)_\kappa \cdots (b_q)_\kappa} \cdot s_\kappa(x_1, x_2, \ldots, x_n), \tag{1} $$

where

$$ (a)_\kappa \equiv \prod_{(i,j) \in \kappa} (a - i + j) \tag{2} $$

is the generalized Pochammer symbol, s_κ is the Schur function [12], $H_\kappa \equiv \prod_{(i,j)\in\kappa} h_\kappa(i,j)$ is the product of all hook lengths $h_\kappa(i,j) \equiv \kappa_i + \kappa'_j - i - j + 1$ of κ, and $x_1, x_2, \ldots, x_n$ are the eigenvalues of X.

The efficient evaluation of (1) is a central research problem in multivariate statistical analysis [15] with a wealth of applications in wireless communications [1, 6, 7, 10, 13, 14, 16, 17, 21] as well as signal processing [20].

The challenge in evaluating (1) involves the evaluation of the Schur function s_κ and has proven extremely challenging primarily since, as a multivariate symmetric polynomial, it has exponential number of terms—$\mathcal{O}\big(n^{|\kappa|}\big)$ [4, 8, 11].

Currently, the best algorithm for evaluating (1) is `mhg` from [11] whose complexity is

$$ \mathcal{O}(nK_N^2), $$

2000 *Mathematics Subject Classification.* Primary 05E05. Secondary 65F50; 65T50.

Key words and phrases. Schur functions, Fast Fourier Transforms.

The research of the second author was partially supported by Grant MI-1503/2005 of the Bulgarian National Science Fund.

The research of the third and fourth authors was supported by NSF Grant DMS–0608306.

where $K_N \equiv \#\{\kappa|\,|\kappa| \le N\}$ is the number of terms in (1).

An estimate of K_N is Ramanujan's asymptotic formula [9, p. 116] $K_N \sim \mathcal{O}\big(e^{\pi\sqrt{2N/3}}\big)$, which is subexponential in N.

In this paper, we present a new algorithm for computing (1) whose complexity is only

$$\mathcal{O}(n^2 K_N),$$

i.e., it takes only $\mathcal{O}(n^2)$ per term instead of the $\mathcal{O}(nK_N)$ cost per term of `mhg`.

To achieve that complexity, we follow the idea in [11]: The recursive formula [12]

$$s_\lambda(x_1, x_2, \ldots, x_n) = \sum_\mu s_\mu(x_1, x_2, \ldots, x_{n-1}) x_n^{|\lambda|-|\mu|} \tag{3}$$

allows each Schur function in (1) to be computed at a cost not exceeding $\mathcal{O}(nK_M)$. The summation in (3) is over all partitions $\mu = (\mu_1, \ldots, \mu_{n-1})$ such that λ/μ is a horizontal strip, i.e.,

$$\lambda_1 \ge \mu_1 \ge \lambda_2 \ge \mu_2 \ge \cdots \ge \lambda_{n-1} \ge \mu_{n-1} \ge \lambda_n. \tag{4}$$

In this paper we improve on the result of [11] by observing that (3) represents a vector-matrix multiplication

$$s^{(n)} = s^{(n-1)} \cdot Z_n(x_n) \tag{5}$$

where $s^{(i)}$ is an (appropriately indexed) row-vector of all Schur functions $s_\kappa(x_1, x_2, \ldots, x_i)$, $|\kappa| \le N$ and $Z_n(x_n) \equiv \big(\varepsilon_{\mu,\lambda} x_n^{|\lambda|-|\mu|}\big)$ is a matrix with entries indexed with the pairs of partitions (μ, λ), with $\varepsilon_{\mu,\lambda} = 1$ if λ/μ is a horizontal strip and 0 otherwise.

Since the matrix Z_n is dense, (5) costs $\mathcal{O}(K_M^2)$, explaining the $\mathcal{O}(nK_M^2)$ complexity of `mhg` [11].

The key contribution of this paper is to recognize and exploit the structure of Z_n to perform the multiplication (5) in linear $\mathcal{O}(n^2 K_M)$ time instead of quadratic $\mathcal{O}(nK_M^2)$ time.

This work was inspired by the idea of Cookey and Tukey [3] (and later generalized [2, 5, 18, 19]) that a matrix-vector multiplication by the character table of the cyclic group of size n (i.e., by the Vandermonde matrix $V \equiv \big(e^{(i-1)(j-1)\frac{2\pi\sqrt{-1}}{n}}\big)$) can be performed in $\mathcal{O}(n\log n)$ instead of $\mathcal{O}(n^2)$ time by exploiting the structure of the cyclic group to decompose V recursively into a product of simpler structured matrices.

2. Theoretical grounds

In this section we fix also the positive integer N. For a fixed $k = 1, \ldots, n$, we extend the set of all partitions $\lambda = (\lambda_1, \ldots, \lambda_k)$ with $|\lambda| \le N$, and consider the set P_k of partitions λ satisfying the conditions

$$\lambda_1 - \lambda_2 \le N, \quad \lambda_2 - \lambda_3 \le N, \quad \ldots, \quad \lambda_{k-1} - \lambda_k \le N, \quad \lambda_k \le N. \tag{6}$$

Clearly, the number of the λs from the class P_k is $(N+1)^k$. We order $\lambda \in P_k$ in the reverse lexicographic order with respect to the k-tuple $(\lambda_1 - \lambda_2, \ldots, \lambda_{k-1} - \lambda_k, \lambda_k)$ and assume that

$\lambda < \nu$, $\lambda, \nu \in P_k$, if and only if $\lambda_k < \nu_k$ or, when $\lambda_k = \nu_k$, $\lambda_{l-1} - \lambda_l = \nu_{l-1} - \nu_l$ for $l = p+1, \ldots, k$ and $\lambda_{p-1} - \lambda_p < \nu_{p-1} - \nu_p$ for some p.

We build inductively the row vectors $F_k(x_1, \ldots, x_k)$, $k = 1, \ldots, n$,

$$F_k(x_1, \ldots, x_k) = (f_\lambda(x_1, \ldots, x_k) \mid \lambda \in P_k), \tag{7}$$

where the λs are ordered with respect to the above order. We define

$$F_1(x_1) = (1, s_{(1)}(x_1), s_{(2)}(x_1), \ldots, s_{(N)}(x_1)) = (1, x_1, x_1^2, \ldots, x_1^N),$$

and, for $k > 1$,

$$F_k(x_1, \ldots, x_k) = F_{k-1}(x_1, \ldots, x_{k-1}) Y_k(x_k), \tag{8}$$

where

$$Y_k(x_k) = \left(\varepsilon_{\mu,\lambda} x_k^{|\lambda/\mu|} \right), \quad \lambda \in P_k, \quad \mu \in P_{k-1}, \tag{9}$$

and $\varepsilon_{\mu,\lambda} = 1$ if λ/μ is a horizontal strip and $\varepsilon_{\mu,\lambda} = 0$ otherwise. (We denote the matrix by Y_k, celebrating Young because (5) expresses the Young rule which is a partial case of the Littlewood-Richardson rule for calculating the product of two Schur functions.)

Lemma 2.1. *If $\lambda_1 \leq N$ for $\lambda \in P_k$, then*

$$f_\lambda(x_1, \ldots, x_k) = s_\lambda(x_1, \ldots, x_k). \tag{10}$$

Proof. If we delete the columns and the rows of the matrix $Y_k(x_k)$ from (9) which correspond, respectively, to partitions λ and μ with $|\lambda| \geq N$ and $|\mu| \geq N$, we shall obtain the matrix $Z_k(x_k)$ from (5). Since $F_1(x_1) = S_1(x_1)$, the proof is completed by easy induction on k. $\square$

Remark 2.2. Once we have an algorithm for a fast multiplication by Y_k we will use it to multiply by only those elements of F_{k-1} corresponding to partitions of size not exceeding N, which by the preceding lemma are exactly the Schur functions we want. Therefore even though the size of Y_k is exponential ($(N+1)^k \times (N+1)^{k-1}$) it will only cost $\mathcal{O}(n^2 K_N)$ to multiply by Y_k.

In what follows we denote by I_m the identity $m \times m$ matrix and by E_{ij} the matrix units with 1 at (i,j)-position and zeros at the other places. The size of E_{ij} will be clear from the context. If P, Q are two matrices, then we denote by $P \otimes Q$ their Kronecker product. By definition, if $P = (p_{ij})$ is an $l \times m$ matrix, then $P \otimes Q$ is an $l \times m$ block matrix with the block $p_{ij}Q$ at (i,j)-position. In particular, $I_m \otimes Q$ is a diagonal block matrix with m copies of Q on the diagonal.

We fix the $(N+1) \times (N+1)$ matrix

$$A = E_{12} + E_{23} + \cdots + E_{N,N+1} = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ 0 & 0 & 0 & \ldots & 0 \end{bmatrix}, \tag{11}$$

and define the matrices

$$B_2 = A^T = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}, \tag{12}$$

$$\begin{aligned} C_2(x_2) &= I_{N+1} + x_2 A + \dots + x_2^N A^N \\ &= (I_{N+1} - x_2 A)^{-1} \\ &= \begin{bmatrix} 1 & x_2 & \dots & \dots & x_2^N \\ 0 & 1 & \ddots & & x_2^{N-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & x_2 \\ 0 & 0 & \dots & & 1 \end{bmatrix}, \end{aligned} \tag{13}$$

$$\begin{aligned} K_2(x_2) &= I_{N+1} \otimes C_2(x_2) \\ &= \begin{bmatrix} C_2(x_2) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & C_2(x_2) \end{bmatrix}. \end{aligned} \tag{14}$$

Finally, we consider the $(N+1) \times (N+1)^2$ matrix

$$Q_2(x_2) = \left(I_{N+1} \,|\, x_2 B_2 \,|\, \dots \,|\, x_2^N B_2^N\right) \tag{15}$$

with entries indexed by pairs (μ, λ), where $\mu = (\mu_1) \in P_1$, $\lambda = (\lambda_1, \lambda_2) \in P_2$.

Lemma 2.3. *The following equation holds*

$$Y_2(x_2) = Q_2(x_2) K_2(x_2). \tag{16}$$

Proof. The matrix $Y_2(x_2)$ consists of $N+1$ blocks of size $(n+1) \times (N+1)$. The entry (p, q) of the rth block, $p, q, r = 0, 1, \dots, N$, is indexed with the pair of partitions (μ, λ), where $\mu = (p)$, $\lambda = (q+r, r)$ and is equal to x_2^{q+2r-p} if $r \le p \le q+r$ and to 0 otherwise. On the other hand, the corresponding entry of the matrix $Q_2(x_2)K_2(x_2)$ is equal to the (p, q)-entry of the matrix $x_2^r B_2^r C_2(x_2)$. The equations (11), (12) and (13) give that

$$B_2^r = E_{1+r,1} + E_{2+r,2} + \dots + E_{N+1,N+1-r},$$

$(B_2^r C_2(x_2))_{pq} = 0$ if $p < r$ and $(B_2^r C_2(x_2))_{pq}$ is equal to the $(p-r, q)$-entry $(C_2(x_2))_{p-r,q}$ of $C_2(x_2)$. Hence $(B_2^r C_2(x_2))_{pq} = x_2^{q-(p-r)}$ if $q \ge p-r$ and $(B_2^r C_2(x_2))_{pq} = 0$ if $q < p-r$. In this way, all entries of $Y_2(x_2)$ and $Q_2(x_2)K_2(x_2)$ coincide and this completes the proof. $\square$

Now, for $n \geq 3$ we define inductively the matrices

$$U_n(x_n) = I_{(N+1)^{n-1}} + x_n(A \otimes B_{n-1}) + \cdots + x_n^N(A^N \otimes B_{n-1}^N) \tag{17}$$
$$= \left(I_{(N+1)^{n-1}} - x_n(A \otimes B_{n-1})\right)^{-1},$$
$$V_n(x_n) = K_{n-1}(x_n) = I_{N+1} \otimes C_{n-1}(x_n), \tag{18}$$
$$C_n(x_n) = U_n(x_n)V_n(x_n), \tag{19}$$
$$K_n(x_n) = I_{N+1} \otimes C_n(x_n), \tag{20}$$
$$B_n = B_{n-1} \otimes I_{N+1} = B_2 \otimes I_{(N+1)^{n-2}}, \tag{21}$$
$$Q_n(x_n) = \left(I_{(N+1)^{n-1}} \,|\, x_nB_n \,|\, \ldots \,|\, x_n^NB_n^N\right). \tag{22}$$

The following theorem generalizes Lemma 2.3 for any $n \geq 2$.

Theorem 2.4. *The following equation holds for any $n \geq 2$*

$$Y_n(x_n) = Q_n(x_n)K_n(x_n). \tag{23}$$

Proof. We mimic the proof of Lemma 2.3.

Consider the partitions

$$\lambda = (a_1 + \cdots + a_n, a_2 + \cdots + a_n, \ldots, a_n);$$
$$\mu = (b_1 + \cdots + b_{n-1}, b_2 + \cdots + b_{n-1}, \ldots, b_{n-1}).$$

Then the entry in Y_n corresponding to (μ, λ) should equal

$$x_n^{a_1+2a_2+\cdots+na_n-b_1-2b_2-\cdots-(n-1)b_{n-1}}$$

if λ/μ is a horizontal strip, i.e., if

$$a_1 + \cdots + a_n \geq b_1 + \cdots + b_{n-1} \geq a_2 + \cdots + a_n \geq \cdots \geq a_{n-1} + a_n \geq b_{n-1} \geq a_n,$$

and 0 otherwise.

Since $Y_n(x_n) = Q_n(x_n)K_n(x_n) = \left(I_{(N+1)^{n-1}} \mid x_nB_nC_n(x_n) \mid \cdots \mid x_n^NB_n^NC_n(x_n)\right)$, the (μ, λ) entry of $Y_n(x_n)$ is in the $(1, a_n)$ block of Y_n, i.e.,

$$x_n^{a_n}B_n^{a_n}C_n(x_n)$$

Call this matrix T_n. Since $B_n = B_2 \otimes I_{(N+1)^{n-2}}$, we have $B_n^{a_n} = B_2^{a_n} \otimes I_{(N+1)^{n-2}}$ and

$$\begin{aligned} T_n &= x_n^{a_n}B_n^{a_n}U_n(x_n)V_n(x_n) \\ &= x_n^{a_n}B_n^{a_n}U_n(x_n)(I_{N+1} \otimes C_{n-1}(x_n)) \\ &= x_n^{a_n}B_n^{a_n}\left(I_{N+1} \otimes I_{(N+1)^{n-2}} + x_nA \otimes (B_{n-1}C_{n-1}(x_n))\right. \\ &\quad \left. + x_n^2A^2 \otimes (B_{n-1}^2C_{n-1}(x_n)) + \cdots + x_N^NA^N \otimes (B_{n-1}^NC_{n-1}(x_n))\right) \\ &= x_n^{a_n}\left(B_2^{a_n} \otimes I_{(N+1)^{n-2}} + x_nB_2^{a_n}A \otimes (B_{n-1}C_{n-1}(x_n))\right. \\ &\quad \left. + x_n^2B_2^{a_n}A^2 \otimes (B_{n-1}^2C_{n-1}(x_n)) + \cdots + x_n^NB_2^{a_n}A^N \otimes (B_{n-1}^NC_{n-1}(x_n))\right). \end{aligned}$$

The (μ, λ) entry of Y_n is thus in the (b_{n-1}, a_{n-1}) block of T_n, which equals the (b_{n-1}, a_{n-1}) block of

$$x_n^{a_n}\left(x_n^{a_{n-1}+a_n-b_{n-1}}B_2^{a_n}A^{a_n+a_{n-1}-b_{n-1}} \otimes (B_{n-1}^{a_{n-1}+a_n-b_{n-1}}C_{n-1}(x_n))\right),$$

i.e.,

$$x_n^{a_{n-1}+2a_n-b_{n-1}}B_{n-1}^{a_{n-1}+a_n-b_{n-1}}C_{n-1}(x_n),$$

if $a_{n-1}+a_n \geq b_{n-1} \geq a_n$ and 0 otherwise. The inductive step is thus clear and we continue by induction to conclude that that the (μ, λ) entry of Y_n equals $x_n^{a_1+2a_2+\cdots+na_n-b_1-2b_2-\cdots-(n-1)b_{n-1}}$ if λ/μ is a horizontal strip and 0 otherwise. □

3. The Algorithm

3.1. **Computing only the desired Schur functions.** As mentioned in Remark 2.2, a straightforward implementation of $Y_n(x_n)$ would yield Schur functions corresponding to the entire set of partitions P_n. The number of such functions is $(N+1)^n$. However, we only wish to compute the Schur functions over the partitions of size at most N, and P_n contains many more partitions than those.

In the algorithm presented in this section, we devise a method that leverages the structure explored thus far, but only to compute the Schur functions on the desired partitions. The way we do this is by keeping track of the indices of the desired partitions within the vectors $F_k(x_1, \ldots, x_k)$ for $k = 1, \ldots, n$, and only maintaining the Schur functions over those partitions. Whenever we do a multiplication by a matrix, we do only the work necessary to compute the next set of desired Schur functions.

The key reason we are able to do so efficiently is that the structured matrix $Y_k(x_k)$ requires us only to reference partitions that are smaller than the ones being processed during computation. Thus, by maintaining all partitions up to a certain size, we guarantee the ability to proceed in computing the solution.

3.2. **Pseudocode notation.** We will use psedo-code based on MATLAB notation. Data arrays are indexed with parentheses "`()`" and are 1-indexed. Note that this is different from the indexing of partitions in the set P_n, which will be 0-indexed, as this is more elegant mathematically. Further, the use of the colon "`:`" in the notation `array(index1:index2)` indicates taking all values in array between `index1` and `index2`, inclusive.

3.3. **Algorithm and analysis.**

3.3.1. *Helper functions.* Let $\Phi(N, n)$ be the set of partitions of size at most N that use at most n rows. Let $\phi(N, n) = |\Phi(N, n)|$. We define a function `computeIndices(`N`,`n`)` that finds the (0-indexed) indices of $\Phi(N, n)$ within the list of partitions in P_n (sorted, as before, in reverse lexicographic order). Note that all indices generated by `computeIndices(`N`,`n`)` must be less than $(N+1)^n$ since that is the number of partitions in P_n.

`computeIndices(`N, n`)`

```
  partitions ← enumeratePartitions(N,n,0)
  for m ← 1 to length(partitions) do
    indices(m) ← partitionToIndex(partitions(m),N)
  end for
  Return indices

enumeratePartitions(N,n,min)
  for m ← min to ⌊N/n⌋ do
    if n = 1 then
      Add (m) to partitions
    else
      subPartitions ← enumeratePartitions(N − m,n − 1,m)
      Add m to the n^th position of all partitions in subPartitions
      Add subPartitions to partitions
    end if
  end for
  Return partitions

partitionToIndex(partition,N,n)
  index ← partition(n)
  for m ← (n − 1) down to 1 do
    index ← index · (N + 1) + ( partition(m) − partition(m + 1) )
  end for
  Return index
```

The function `enumeratePartitions` enumerates all partitions in the set $\Phi(N, n)$ in reverse lexicographic order. It works by stepping through possible values for the last element and recursively enumerating the rest of the partition. To analyze its complexity, we simply observe that a constant amount of work is done per recursion level per partition enumerated. Thus, the running time of `enumeratePartitions(`N`,`n`,0)` is simply $\mathcal{O}(n \cdot \phi(N, n))$.

The function `partitionToIndex` takes a partition and returns its index within the sorted set P_n. It simply takes the difference between consecutive elements and interprets the result as an $(N+1)$-ary number with the elements increasing in significance from first to last. This function clearly takes $\mathcal{O}(n)$ time per partition, so its running time on $\phi(N, n)$ partitions is $\mathcal{O}(n \cdot \phi(N, n))$. Thus, `computeIndices(`$N$`,`$n$`)` also takes $\mathcal{O}(n \cdot \phi(N, n))$ time. Note that the output of `computeIndices` is sorted in ascending order.

3.3.2. *Matrix functions.* In this section we will describe five matrix functions `mulY`, `mulQ`, `mulK`, `mulC`, and `mulU`, which simulate multiplication with the matrices $Y_n(x_n)$, $Q_n(x_n)$, $K_n(x_n)$, $C_n(x_n)$, and $U_n(x_n)$, respectively.

We first present an algorithm `mulY` that simulates multiplication by $Y_n(x_n) = Q_n(x_n)K_n(x_n)$, but that only computes the Schur functions corresponding to partitions in $\Phi(N, n)$. Suppose $Y_n(x_n)$ contains a non-zero element at (i, j). Recall from (9) that this implies the i^{th} partition in P_{n-1}, call it μ, is a subpartition of the j^{th} partition in P_n, call it λ. Thus, $|\mu| \leq |\lambda|$, and if λ is in $\Phi(N, n)$, then, μ must be in $\Phi(N, n-1)$.

From the above argument, we see that the partitions in $\Phi(N, n)$ will only depend on partitions in $\Phi(N, n-1)$, so we only need to simulate a very sparse part of the original $Y_n(x_n)$ matrix. `mulY` takes as input the Schur functions over $\Phi(N, n-1)$, the indices of $\Phi(N, n-1)$ in P_{n-1} (as computed by `computeIndices(`N`,`$n-1$`)`), and x_n. `mulY` outputs the Schur functions over $\Phi(N, n)$ (as well as their indices).

```
mulY(input,inputIndices,x,n,N)
  (output,outputIndices) ← mulQ(input,inputIndices,x,n,N)
  output ← mulK(output,outputIndices,x,n,N)
  Return (output,outputIndices)

mulQ(input,inputIndices,x,n,N)
  blockLength ← (N + 1)^(n−1)
  offset ← (N + 1)^(n−2)

  # compute the indices in the output
  outputIndices ← computeIndices(N,n)
  H ← constructHashTable(outputIndices)

  for m ← 1 to length(outputIndices) do
    curIndex ← outputIndices(m)
    if curIndex < blockLength then
      # this works because the input and output indices less than (N + 1)^(n−1) match
      output(m) ← input(m)
    else if curIndex (mod blockLength) < blockLength − offset then
      output(m) ← x · output(H(curIndex − blockLength + offset))
    end if
  end for
  Return (output,outputIndices)
```

`mulY` simply runs `mulQ` and `mulK`. From (22), we designed `mulQ` to process its input in blocks of length $(N+1)^{n-1}$. The first block is simply copied from the input. Note that since $\Phi(N, n)$ is a superset of $\Phi(N, n-1)$, and the partitions are ordered in reverse lexicographic order, the first $\phi(N, n-1)$ entries of `outputIndices` are exactly equal to `inputIndices`. For the remaining blocks, we note that each block is a shifted (by `offset`) version of the previous block multiplied by x_n. Since we need to lookup values in `output` based on their indices,

we place all of the indices into a hash table so we can do constant time lookups within the `output` array. The function `constructHashTable(outputIndices)` just constructs a hash table that maps each index to its location within the array `outputIndices`.

For a partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ at `curIndex` in $\Phi(N, n)$, we know we will never miss on a hash table lookup in the `for` loop because the partition located at `curIndex` $-$ `blockLength` $+$ `offset` is just $\lambda^\dagger = (\lambda_1, \lambda_2, \ldots, \lambda_{n-1}, \lambda_n - 1)$. This fact can be derived from the reverse lexicographic ordering. Since $|\lambda^\dagger| < |\lambda|$, we know that $\lambda^\dagger$ is also in $\Phi(N, n)$.

As argued before, `computeIndices`(N, n) takes $\mathcal{O}(n \cdot \phi(N, n))$ time. Constructing the hash table costs linear time in the number of entries, or $\mathcal{O}(\phi(N, n))$. The for loop of `mulQ` takes time $\mathcal{O}(\phi(N, n))$ since hash table look-ups take constant time, therefore the total time to multiply by $Q_n(x_n)$ using `mulQ` is $\mathcal{O}(n \cdot \phi(N, n))$.

The following function simulates multiplying its input by $K_n(x_n) = I_{N+1} \otimes C_n(x_n)$, which simply multiplies each of its input's $(N+1)$ blocks of size $(N+1)^{n-1}$ by $C_n(x_n)$. The values in each block are found by scanning pointers across the `indices` array, which is sorted in ascending order.

```
mulK(input,indices,x,n,N)
  blockLength ← (N + 1)^(n−1)
  minPointer ← 1
  maxPointer ← 1
  for blockIndex ← 0 to ⌊ max(indices) / blockLength ⌋ do
    # figure out which indices are in the current block
    curIndicesMin ← blockIndex · blockLength
    curIndicesMax ← curIndicesMin + blockLength − 1

    # scan pointers forward
    Scan minPointer to point at the smallest index in indices that is at least curIndicesMin
    Scan maxPointer to point at the largest index in indices that is at most curIndicesMax

    # extract the data for the current block
    curData ← input(minPointer:maxPointer)
    # extract the indices for the current block and subtract the block offset
    curIndices ← indices(minPointer:maxPointer) − curIndicesMin

    # run mulC on block of data
    output(minPointer:maxPointer) ← mulC(curData,curIndices,x,n,N)
  end for
  Return output
```

Since `mulK`, `mulC`, and `mulU` are called recursively on inputs of varying length, we will analyze their complexity in terms of the number of input elements. Let l be the length of the input

to a call to `mulK`, and let l_i be the number of input indices in the i^{th} block of the `for` loop. Note $\sum_{i=0}^{N} l_i = l$. Excluding calls to `mulC`, multiplying by $K_n(x_n)$ using `mulK` takes $\mathcal{O}(l)$ time for pointer scanning and data manipulation. If we let $T_C(n, l)$ denote the time taken to multiply a vector of length l by $C_n(x_n)$ using `mulC`, then the time to multiply by $K_n(x_n)$ is $\mathcal{O}(l) + \sum_{i=0}^{N} T_C(n, l_i)$.

We define `mulC` to simulate multiplying by $C_n(x_n) = U_n(x_n)K_{n-1}(x_n)$ by calling `mulU` and `mulK`. Note that K_1 is the identity matrix, so we can skip the `mulK` step when $n = 2$.

```
mulC(input,indices,x,n,N)
  output ← mulU(input,indices,x,n,N)
  if n > 2 then
    output ← mulK(output,indices,x,n − 1,N)
  end if
  Return output
```

Finally, we define `mulU`:

```
mulU(input,indices,x,n,N)
  blockLength ← (N + 1)^{n−2}
  if n = 2 then
    offset ← 0
  else
    offset ← (N + 1)^{n−3}
  end if
  H ← constructHashTable(indices)
  for m ← 1 to length(input) do
    curIndex ← indices(m)
    if curIndex ≥ blockLength AND curIndex (mod blockLength) < blockLength −
    offset then
      output(m) ← x · output(H(curIndex − blockLength + offset)) + input(m)
    else
      output(m) ← input(m)
    end if
  end for
  Return output
```

The function `mulU` simulates multiplication by $U_n(x_n)$ by computing a linear time backsolve using $U_n(x_n)^{-1}$. Suppose we are given vector v and wish to compute $w = v \cdot U_n(x_n) \iff v = w \cdot U_n(x_n)^{-1}$. Let $w = (w_1, w_2, \ldots, w_{(N+1)^{n-1}})$ and $v = (v_1, v_2, \ldots, v_{(N+1)^{n-1}})$, where each vector is split into $(N + 1)$ blocks of length $(N + 1)^{n-2}$. Recalling (17), we see that the first

block of v is equal to the first block of w, and then within each remaining block, we have the relation $v_i = w_i - x \cdot w_{i-(N+1)^{n-2}+(N+1)^{n-3}}$ if i is in the first $(N+1)^{n-2} - (N+1)^{n-3}$ elements of the block, and $v_i = w_i$ otherwise. Rearranging terms, we have

$$w_i = \begin{cases} x \cdot w_{i-(N+1)^{n-2}+(N+1)^{n-3}} + v_i & \text{, if property } A \text{ is satisfied} \\ v_i & \text{, otherwise} \end{cases}, \tag{24}$$

where property A is satisfied if and only if i is not in the first block, and $i \pmod{(N+1)^{n-2}} < (N+1)^{n-2} - (N+1)^{n-3}$. The above pseudocode for `mulU` precisely implements (24), yielding a linear time algorithm for multiplication by $U_n(x_n)$.

Again, we know that the hash table will always hit on lookup because it will always be looking for a partition smaller than the current one being processed in the `for` loop. If a partition is in the set being processed, all partitions smaller than it must also be in the set.

The complexity analysis for `mulU` is similar to that for `mulQ`. Assuming the length of the input is l, `constructHashTable` takes $\mathcal{O}(l)$ time, and the `for` loop takes $\mathcal{O}(l)$ time since it does constant work per input element. Thus, the total running time of `mulU` is $\mathcal{O}(l)$, which, combined with the running time for `mulK`, implies that the running time of `mulC` is

$$T_C(n,l) = \begin{cases} \mathcal{O}(l) & \text{, if } n = 1 \\ \mathcal{O}(l) + \sum_{i=0}^{N} T_C(n-1, l_i) & \text{, otherwise} \end{cases}. \tag{25}$$

Clearly, for each level of recursion (each value of n), a linear amount of work is done in the size of the original input. Thus, solving this recurrence yields:

$$T_C(n,l) = \mathcal{O}(nl). \tag{26}$$

Finally, this implies that multiplying by $Y_n(x_n)$ takes $\mathcal{O}(n \cdot \phi(N,n)) + \mathcal{O}(\phi(N,n)) + \mathcal{O}(n \cdot \phi(N,n)) = \mathcal{O}(n \cdot \phi(N,n))$ time.

To compute the Schur functions for (1) from scratch, note that $\Phi(N,1)$ is just the set of partitions $\{(0),(1),\ldots,(N)\}$, and the vector of Schur functions over those partitions is just $(1, x_1, x_1^2, \ldots, x_1^N)$, which takes $\mathcal{O}(N)$ time to compute. Then, computing the Schur functions over $\Phi(N,k)$ for $k = \{2,3,\ldots,n\}$ requires multiplication by $Y_2(x_2), Y_3(x_3), \ldots, Y_n(x_n)$, which takes time at most

$$\sum_{k=2}^{n} \mathcal{O}\left(k \cdot \phi(N,k)\right) < \mathcal{O}\left(n^2 \cdot \phi(N,n)\right). \tag{27}$$

As mentioned in the introduction, $\phi(N,n) \leq \phi(N,N) = K_N = \mathcal{O}(e^{\pi\sqrt{2N/3}})$ [9, p. 116], so the running time can also be bounded by $\mathcal{O}(n^2 K_N) = \mathcal{O}(n^2 e^{\pi\sqrt{2N/3}})$.

References

1. G.J. Byers and F. Takawira, *Spatially and temporally correlated MIMO channels: modeling and capacity analysis*, IEEE Transactions on Vehicular Technology **53** (2004), 634–643.
2. Michael Clausen and Ulrich Baum, *Fast Fourier transforms for symmetric groups: theory and implementation*, Math. Comp. **61** (1993), no. 204, 833–847. MR MR1192969 (94a:20028)

3. James W. Cooley and John W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp. **19** (1965), 297–301. MR MR0178586 (31 #2843)
4. J. Demmel and P. Koev, *Accurate and efficient evaluation of Schur and Jack functions*, Math. Comp. **75** (2006), 223–239.
5. Persi Diaconis and Daniel Rockmore, *Efficient computation of the Fourier transform on finite groups*, J. Amer. Math. Soc. **3** (1990), no. 2, 297–332. MR MR1030655 (92g:20024)
6. H. Gao, P.J. Smith, and M.V. Clark, *Theoretical reliability of MMSE linear diversity combining in Rayleigh-fading additive interference channels*, IEEE Transactions on Communications **46** (1998), 666–672.
7. A.J. Grant, *Performance analysis of transmit beamforming*, IEEE Transactions on Communications **53** (2005), 738–744.
8. R. Gutiérrez, J. Rodriguez, 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.
9. G. H. Hardy, *Ramanujan: Twelve lectures on subjects suggested by his life and work.*, AMS Chelsea, New York, 1999.
10. M. Kang and M.-S. Alouini, *Largest eigenvalue of complex Wishart matrices and performance analysis of MIMO MRC systems*, IEEE Journal on Selected Areas in Communications **21** (2003), no. 3, 418–431.
11. Plamen Koev and Alan Edelman, *The efficient evaluation of the hypergeometric function of a matrix argument*, Math. Comp. **75** (2006), no. 254, 833–846 (electronic). MR MR2196994 (2006k:33007)
12. I. G. Macdonald, *Symmetric functions and Hall polynomials*, Second ed., Oxford University Press, New York, 1995.
13. M.R. McKay and I.B. Collings, *Capacity bounds for correlated rician MIMO channels*, 2005 IEEE International Conference on Communications. ICC 2005., vol. 2, 16-20 May 2005, pp. 772–776.
14. ———, *General capacity bounds for spatially correlated Rician MIMO channels*, IEEE Transactions on Information Theory **51** (2005), 3121–3145.
15. R. J. Muirhead, *Aspects of multivariate statistical theory*, John Wiley & Sons Inc., New York, 1982.
16. A. Ozyildirim and Y. Tanik, *Outage probability analysis of a CDMA system with antenna arrays in a correlated Rayleigh environment*, IEEE Military Communications Conference Proceedings, 1999. MILCOM 1999, vol. 2, 31 Oct.–3 Nov. 1999, pp. 939–943.
17. ———, *SIR statistics in antenna arrays in the presence of correlated Rayleigh fading*, IEEE VTS 50th Vehicular Technology Conference, 1999. VTC 1999 - Fall, vol. 1, 19-22 September 1999, pp. 67–71.
18. Markus Püschel, *Cooley-Tukey FFT like algorithms for the DCT*, Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 2, 2003, pp. 501–504.
19. Markus Püschel and José M. F. Moura, *The algebraic approach to the discrete cosine and sine transforms and their fast algorithms*, SIAM J. Comput. **32** (2003), no. 5, 1280–1316. MR MR2001274 (2004j:42023)
20. Hyundong Shin and Jae Hong Lee, *Capacity of multiple-antenna fading channels: spatial fading correlation, double scattering, and keyhole*, IEEE Transactions on Information Theory **49** (2003), 2636–2647.
21. V. Smidl and A. Quinn, *Fast variational PCA for functional analysis of dynamic image sequences*, Proceedings of the 3rd International Symposium on Image and Signal Processing and Analysis, 2003. ISPA 2003., vol. 1, 18-20 September 2003, pp. 555–560.

Department of Mathematics, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, U.S.A.

E-mail address: cychan@mit.edu

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

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

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