THE EFFICIENT EVALUATION
OF THE HYPERGEOMETRIC FUNCTION
OF A MATRIX ARGUMENT

PLAMEN KOEV AND ALAN EDELMAN

Abstract. We present new algorithms that efficiently approximate the hypergeometric function of a matrix argument through its expansion as a series of Jack functions. Our algorithms exploit the combinatorial properties of the Jack function, and have complexity that is only linear in the size of the matrix.

1. Introduction

The hypergeometric function of a matrix argument has a wide area of applications in multivariate statistical analysis [17], random matrix theory [7], wireless communications [8], [12], etc. Except in a few special cases, it can be expressed only as a series of multivariate homogeneous polynomials, called Jack functions. This series often converges very slowly [16, p. 390], [17], and the cost of the straightforward evaluation of a single Jack function is exponential [3]. The hypergeometric function of a matrix argument has thus acquired a reputation of being notoriously difficult to approximate even in the simplest cases [2], [10].

In this paper we present new algorithms for approximating the value of the hypergeometric function of a matrix argument. We exploit recursive combinatorial relationships between the Jack functions, which allow us to only update the value of a Jack function from other Jack functions computed earlier in the series. The savings in computational time are enormous; the resulting algorithm has complexity that is only linear in the size of the matrix argument. In the special case when the matrix argument is a multiple of the identity, the evaluation becomes even faster.

We have made a MATLAB [15] implementation of our algorithms available [13]. This implementation is very efficient (see performance results in Section 6), and has lead to new results [1], [6].

The hypergeometric function of a matrix argument is defined as follows. Let \( p \geq 0 \) and \( q \geq 0 \) be integers, and let \( X \) be an \( n \times n \) complex symmetric matrix with eigenvalues \( x_1, x_2, \ldots, x_n \). Then

\[
\left. \begin{array}{c}
pF_q^\alpha(a_1, \ldots, a_p; b_1, \ldots, b_q; X) = \sum_{\kappa} \sum_{\kappa} \binom{a_1}{\kappa} \cdots \binom{a_p}{\kappa} \binom{b_1}{\kappa} \cdots \binom{b_q}{\kappa} \cdot C_\kappa(X),
\end{array} \right\}
\]

Received by the editor September 16, 2004 and, in revised form, February 26, 2005.

2000 Mathematics Subject Classification. Primary 33C20, 65B10; Secondary 05A99.

Key words and phrases. Hypergeometric function of a matrix argument, Jack function, zonal polynomial, eigenvalues of random matrices.

This work was supported in part by NSF Grant DMS-0314286.

©2006 American Mathematical Society
where \( \alpha > 0 \) is a parameter, \( \kappa \vdash k \) means \( \kappa = (\kappa_1, \kappa_2, \ldots) \) is a partition of \( k \) (i.e., \( \kappa_1 \geq \kappa_2 \geq \cdots \geq 0 \) are integers such that \( |\kappa| = \kappa_1 + \kappa_2 + \cdots = k \)),

\[
(1.2) \quad (a)_k^{(\alpha)} \equiv \prod_{(i,j) \in \kappa} \left( a - \frac{i-1}{\alpha} + j - 1 \right)
\]

is the generalized Pochhammer symbol, and \( C_k^{(\alpha)}(X) \) is the Jack function.

The Jack function \( C_k^{(\alpha)}(X) = C_k^{(\alpha)}(x_1, x_2, \ldots, x_n) \) is a symmetric, homogeneous polynomial of degree \( |\kappa| \) in the eigenvalues \( x_1, x_2, \ldots, x_n \) of \( X \) \cite[Rem. 2, p. 228]{17}, \cite{20}. For example, when \( \alpha = 1 \), \( C_k^{(1)}(X) \) becomes the (normalized) Schur function, and for \( \alpha = 2 \), the zonal polynomial. There are several normalizations of the Jack function which are scalar multiples of one another: \( C_k^{(\alpha)}(X) \) is normalized so that \( \sum_{\kappa \vdash k} C_k^{(\alpha)}(X) = (\text{tr } X)^k \); in Section 3 we express (1.1) in terms of the Jack function \( J_k^{(\alpha)}(X) \), which is normalized so that the coefficient of \( x_1 x_2 \cdots x_{|\kappa|} \) is \(|\kappa|!\). The functions \( C_k^{(\alpha)}(X) \) and \( J_k^{(\alpha)}(X) \) can be defined recursively, e.g.,

\[
(1.3) \quad J_k^{(\alpha)}(x_1, x_2, \ldots, x_n) = \sum_{\mu \leq \kappa} J_{\mu}^{(\alpha)}(x_1, x_2, \ldots, x_{n-1}) \cdot x_n^{(\kappa/\mu)} \cdot \beta_{\alpha \mu},
\]

where \( \beta_{\alpha \mu} \) is a rational function of \( \alpha \) (see Section 3 for details). The relationship (1.3) becomes key in achieving efficiency in our algorithms.

The Jack functions \( C_k^{(\alpha)}(X) \) and \( J_k^{(\alpha)}(X) \), and in turn the hypergeometric function of a matrix argument, depend only on the eigenvalues \( x_1, x_2, \ldots, x_n \) of \( X \). Many authors, however, have found the matrix notation in (1.1) and the use of a matrix argument to be more convenient. We follow the same practice.

The hypergeometric function of a matrix argument is scalar-valued, which is a major distinction from other functions of a matrix argument (e.g., the matrix exponential), which are matrix-valued. The hypergeometric function of a matrix argument generalizes the classical hypergeometric function to which it reduces for \( n = 1 \). In general, however, there is no explicit relationship between these two functions for \( n \geq 2 \).

We approximate the series (1.1) by computing its truncation for \( |\kappa| \leq m \):

\[
(1.4) \quad \sum_{k=0}^{m} \sum_{k \vdash \kappa: k \leq q} \left( \frac{(a_1)_k^{(\alpha)}}{k!} \cdots \frac{(a_p)_k^{(\alpha)}}{k!} \right) \cdot C_k^{(\alpha)}(X).
\]

The series (1.1) converges for any \( X \) when \( p \leq q \); it converges if \( \max |x_i| < 1 \) when \( p = q + 1 \), and diverges when \( p > q + 1 \), unless it terminates \cite[p. 258]{17}. When it converges, its \( \kappa \)-term converges to zero as \( |\kappa| \to \infty \). In these cases (1.4) is a good approximation to (1.1) for a large enough \( m \).

The computational difficulties in evaluating (1.4) are:

- (A) the series (1.4) converges slowly in many cases \cite{16}; thus a rather large \( m \) may be needed before (1.4) becomes a good approximation to (1.1);
- (B) the number of terms in (1.4) (i.e., the number of partitions \( |\kappa| \leq m \)) grows, roughly, as \( O(e^\sqrt{m}) \) (see Section 5);
- (C) the straightforward evaluation of a single Jack function, \( C_k^{(\alpha)}(X) \) for \( |\kappa| = m \), has complexity that grows as \( O(n^m) \) \cite{3}.

\[\text{Some authors define the hypergeometric function of a matrix argument through the series (4.1) for } \alpha = 1 \text{ or } \alpha = 2 \text{ only. There is no reason for us to treat the different } \alpha \text{'s separately (see also \cite{4}, \cite{5}, \cite{7} for the uniform treatment of the different } \alpha \text{'s in other settings).}\]
While there is little we can do about (A) (which is also a major problem even in the univariate \( n = 1 \) case [18]) or (B), our major contribution is in improving (C), the cost of evaluating the Jack function. We exploit the combinatorial properties of the Pochhammer symbol and the Jack function to only update the \( \kappa \)-term in (1.4) from the \( \mu \)-terms, \( \mu \leq \kappa \). As a result the complexity of our main algorithm for computing (1.4), Algorithm 4.2, is only linear in the size \( n \) of the matrix argument \( X \), exponentially faster than the previous best algorithm [10] (see Sections 2, 5 and 6.3 for details). In the special case when \( X \) is a multiple of the identity, we present an even faster algorithm, Algorithm 4.1, whose complexity is independent of \( n \).

A number of interesting problems remain open. Among these are:

- detecting convergence;
- selecting the optimal value of \( m \) in (1.4) for a desired accuracy;
- selecting the optimal truncation of the series (1.1).

We do not believe that a uniform answer to these problems exists for every \( \alpha \) and every \( p \) and \( q \). Therefore, we leave the choice of \( m \) and an appropriate truncation to the user. We elaborate more on these open problems in Section 7.

With minimal changes our algorithms can approximate the hypergeometric function of two matrix arguments

\[
_pF_q^{(a)}(a_1:p; b_1:q; X, Y) \equiv \sum_{k=0}^{\infty} \sum_{\kappa \vdash k} \frac{(a_1)_\kappa \cdots (a_p)_\kappa}{k!(b_1)_\kappa \cdots (b_q)_\kappa} \frac{C^{(\alpha)}_\kappa(X) \cdot C^{(\alpha)}_\kappa(Y)}{C^{(\alpha)}_\kappa(I)},
\]

and more generally functions of the form

\[
G(X) = \sum_{k=0}^{\infty} \sum_{\kappa \vdash k} a_\kappa C^{(\alpha)}_\kappa(X),
\]

for arbitrary coefficients \( a_\kappa \) at a similar computational cost (see, e.g., (6.5) in sub-section 6.2).

In (1.5) and throughout this paper, we denote a vector \((z_1, \ldots, z_t)\) as \( z_1:t \).

This paper is organized as follows. We survey previous algorithms for computing the hypergeometric function of a matrix argument in Section 2. In Section 3 we describe our approach in computing the truncation (1.4). We present our new algorithms in Section 4, and analyze their complexity in Section 5. We present numerical experiments in Section 6. Finally, we draw conclusions and present open problems in Section 7.

2. Previous algorithms

Butler and Wood [2] used Laplace approximations to compute the integral representations [17] Thm. 7.4.2, p. 264:

\[
1F_1^{(2)}(a; c; X) = \frac{\Gamma_n^{(2)}(c)}{\Gamma_n^{(2)}(c-a)} \int_{0 < Y < I} e^{\text{tr}(XY)}(\det Y)^{\frac{c-a}{2}} \times \det(I - Y)^{\frac{c-a}{2}}(dY),
\]
valid for real symmetric $X$, $\Re(a) > \frac{n-1}{2}$, $\Re(c) > \frac{n-1}{2}$, and $\Re(c - a) > \frac{n-1}{2}$; and
\[ 2F_1^{(2)}(a, b; c; X) = \frac{\Gamma_n^{(2)}(c)}{\Gamma_n^{(2)}(a)\Gamma_n^{(2)}(c - a)} \int_{0 < Y < I} \det(I - XY)^{-b} \times (\det Y)^{a - \frac{n+1}{2}} \det(I - Y)^{c - a - \frac{n+1}{2}} (dY), \]
valid for $\Re(X) < I$, $\Re(a) > \frac{n-1}{2}$, and $\Re(c - a) > \frac{n-1}{2}$.

This approach, however, is restricted to the cases $p = 1$ or $2$, $q = 1$, and $\alpha = 2$.

Gutiérrez, Rodríguez, and Sáez presented in [10] (see also [19] for the implementation) an algorithm for computing the truncation (1.4) for $\alpha$ where the product (3.2) is normalized so that the coefficient of $\kappa$ transition matrix (14, p. 99) $K$ (indexed by all partitions of $k$) between the monomial symmetric functions $(m_\kappa)_\kappa$ and the zonal polynomials $(C_\kappa)_\kappa$. Then for every partition $\kappa = (\kappa_1, \kappa_2, \ldots) = k$ they compute
\[ m_\kappa = \sum_{\mu} x_1^{\mu_1} x_2^{\mu_2} \cdots \]
(wheres $\mu$ ranges over all distinct permutations of $(\kappa_1, \kappa_2, \ldots)$ [22, p. 289]) and form the product $(C_\kappa)_\kappa = K \cdot (m_\kappa)_\kappa$. Computing the vector $(m_\kappa)_\kappa$ alone costs $m^{(n+m-1)} = O(n^m)$ since every term in every $m_\kappa$ is of degree $m$, and for every nonstrictly increasing sequence of $m$ numbers from the set $\{1, 2, \ldots, n\}$ we obtain a distinct term in some $m_\kappa$. The overall cost is thus at least exponential ($O(n^m)$), which explains the authors’ observation:

We spent about 8 days to obtain the 627 zonal polynomials of degree 20 with a 350 MHz Pentium II processor.

In contrast, our Algorithms [4, 12] takes less than a hundredth of a second to do the same. Its complexity is only linear in $n$ and subexponential in $m$ (see Section 5).

3. Our approach

We make the evaluation of $m_{p}^\alpha F_{q}^{(\alpha)}(a_{1}; b_{1}; q_{i}; X)$ efficient by only updating the $\kappa$-term from the $\mu$-terms, $\mu \leq \kappa$, instead of computing it from scratch.

We first express $m_{p}^\alpha F_{q}^{(\alpha)}(a_{1}; b_{1}; q_{i}; X)$ in terms of the Jack function $J_{\kappa}^{(\alpha)}(X)$, which is normalized so that the coefficient of $x_1 x_2 \cdots x_\kappa$ in $J_{\kappa}^{(\alpha)}(X) = J_{\kappa}^{(\alpha)}(x_1, x_2, \ldots, x_\kappa)$ equals $(|\kappa|)!$ [29] Thm. 1.1. The Jack functions $C_{\kappa}^{(\alpha)}(X)$ and $J_{\kappa}^{(\alpha)}(X)$ are related as
\[ C_{\kappa}^{(\alpha)}(X) = \frac{a^{(|\kappa|)} \cdot (|\kappa|)!}{j_{\kappa}} J_{\kappa}^{(\alpha)}(X), \]
where
\[ j_{\kappa} = \prod_{(i, j) \in \kappa} h_{\alpha}(i, j) h_{\alpha}(i, j), \]
and $h_{\alpha}(i, j) \equiv \kappa' - i + \alpha(\kappa_j - j + 1)$ and $h_{\alpha}(i, j) \equiv \kappa' - i + 1 + \alpha(\kappa_i - j)$ are the upper and lower hook lengths at $(i, j) \in \kappa$, respectively.

Denote
\[ Q_{\kappa} = \frac{a^{(|\kappa|)} (a_{1})^{(\alpha)} \cdots (a_{p})^{(\alpha)}}{j_{\kappa}(b_{1})^{(\alpha)} \cdots (b_{q})^{(\alpha)}}. \]
Since $J_{κ}^{(α)}(X) = 0$ when $κ_{n+1} > 0$, we need to sum only over partitions $κ$ with at most $n$ parts:

$$m \cdot \frac{E^{(α)}_{q}(a_1:p; b_1:q; X)}{E^{(α)}_{q}(a_1:p; b_1:q; X)} = \sum_{κ \leq m, κ_{n+1} = 0} Q_{κ} J_{κ}^{(α)}(X).$$

When computing (3.4), we recursively generate all partitions $|κ| \leq m$ in such a way that consecutively generated partitions differ in only one part. Therefore it is convenient to introduce the notation

$$κ_{(i)} \equiv (κ_1, \ldots, κ_{i-1}, κ_i = 1, κ_{i+1}, \ldots)$$

for any partition $κ$ such that $κ_i > κ_{i+1}$.

In the following subsections we derive formulas for updating the $κ$-term in (3.4) from the $µ$-terms, $µ \leq κ$.

### 3.1. Updating the coefficients $Q_κ$.

We update $Q_κ$ from $Q_{κ_{(i)}}$ using the following lemma.

**Lemma 3.1.**

$$Q_{κ} = \prod_{j=1}^{p} (a_j + c) \prod_{j=1}^{q} (b_j + c) \prod_{j=1}^{κ_{(i)}} (g_j - α)e_j \prod_{j=1}^{κ_{(i)}} (g_j + α) \prod_{j=1}^{κ_{(i)}} l_j - f_j,$$

where $c = -\frac{κ}{α} - 1$, $d = κ_{α} - i$, $e_j = d - jα + κ_j'$, $g_j = e_j + 1$, $f_j = κ_jα - j - d$, $h_j = f_j + α$, and $l_j = h_j f_j$.

**Proof.** From (3.2), $(a)_{α}^{(κ)} = (a)_{κ_{(i)}}^{(α)} \cdot (a - (i - 1)/α + κ_i - 1)$, and from (3.2),

$$\frac{J_{κ_{(i)}}}{J_{κ}} = \frac{1}{α} \prod_{j=1}^{κ_{(i)}} \frac{h_{κ_{(i)}}^{(α)}(i, j)}{h_{κ_{(i)}}^{(α)}(i, j)} \cdot \frac{1}{h_{κ_{(i)}}^{(α)}(j, κ_i)} \cdot \frac{1}{h_{κ_{(i)}}^{(α)}(j, κ_i)}

which along with (3.3) imply (3.3).

The seemingly complicated notation of Lemma 3.1 is needed in order to minimize the number of arithmetic operations needed to update $Q_κ$. A straightforward evaluation of (3.3) costs $6|κ|(2 + p + q)$; in contrast, (3.3) costs only $2(p + q) + 10κ + 9i - 11$ arithmetic operations.

### 3.2. Updating the Jack function.

When $κ = (0)$, $J_{κ}^{(α)}(x_1, \ldots, x_n) = 1$. For $κ > (0)$, we update $J_{κ}^{(α)}(x_1, \ldots, x_n)$ from $J_{µ}^{(α)}(x_1, \ldots, x_r)$, $µ \leq κ$, $r \leq n$.

When $X$ is a multiple of an identity we have an easy special case [20, Thm. 5.4]:

$$J_{κ}^{(α)}(xI) = x^{|κ|} \prod_{(i,j) \in κ} (n - (i - 1) + α(j - 1)).$$

Therefore we can update $J_{κ}^{(α)}(xI)$ from $J_{κ_{(i)}}^{(α)}(xI)$ as

$$J_{κ}^{(α)}(xI) = J_{κ_{(i)}}^{(α)}(xI) \cdot x \cdot (n - i + 1 + α(κ_i - 1)).$$

In the general case, we update the Jack function using the identity (see, e.g., [20, Prop. 4.2]),

$$J_{κ}^{(α)}(x_1, x_2, \ldots, x_n) = \sum_{µ \leq κ} J_{µ}^{(α)}(x_1, x_2, \ldots, x_n) x_1|κ/µ| \beta_{κµ},$$
where the summation is over all \( \mu \leq \kappa \) such that \( \kappa/\mu \) is a horizontal strip, and

\[
\beta_{\kappa\mu} = \prod_{(i,j) \in \kappa} B_{\kappa\mu}^\kappa(i,j) \prod_{(i,j) \in \mu} B_{\kappa\mu}^\mu(i,j), \quad \text{where} \quad B_{\kappa\mu}^\nu(i,j) = \begin{cases} h_{\nu}^\nu(i,j), & \text{if } \kappa_j' = \mu_j', \\ h_{\nu}^\nu(i,j), & \text{otherwise}. \end{cases}
\]

The skew partition \( \kappa/\mu \) is a horizontal strip when \( \kappa_1 \geq \mu_1 \geq \kappa_2 \geq \mu_2 \geq \cdots \) [22, p. 339].

We borrow the idea for updating the Jack function from [3], but make two important improvements. We only update the coefficients \( \beta_{\kappa\mu} \), and store the precomputed Jack functions much more efficiently than in [3].

The coefficients \( \beta_{\kappa\mu} \) are readily computable using (3.9) at the cost of \( 6(|\kappa| + |\mu|) \) arithmetic operations. The following lemma allows us to start with \( \beta_{\kappa\kappa} = 1 \) and update \( \beta_{\kappa\mu(k)} \) from \( \beta_{\kappa\mu} \) at the cost of only \( 12k + 6\mu_k - 7 \) arithmetic operations.

**Lemma 3.2.** Let \( \kappa, \mu, \) and \( \nu = \mu(k) \) be partitions such that \( \kappa/\mu \) and \( \kappa/\nu \) are horizontal strips, and \( \kappa_j' = \mu_j' \) for \( 0 \leq r \leq k - 1 \). Then

\[
\beta_{\kappa\nu} = \alpha \prod_{r=1}^{k} u_r \prod_{r=1}^{k-1} v_r + \alpha \prod_{r=1}^{k-1} w_r + \alpha \prod_{r=1}^{\mu_k-1} w_r,
\]

where \( \alpha' = \alpha - 1, t = k - \alpha \mu_k, q = t + 1, u_r = q - r + \alpha \kappa_r, v_r = t - r + \alpha \mu_r, \) and \( w_r = \mu_r' - t - \alpha \).

**Proof.** Denote \( l \equiv \mu_k \). We have \( \nu = \mu \), except for \( \nu_k = \mu_k - 1 \). Since \( \kappa/\mu \) and \( \kappa/\nu \) are horizontal strips, \( \kappa_j' = \mu_j' \) and \( \kappa_j' \neq \nu_j' = \mu_j' - 1 \). Then

\[
\begin{align*}
\beta_{\kappa\nu} &= \prod_{\kappa} B_{\kappa\mu}^\mu \prod_{\nu} B_{\kappa\mu}^\nu \\
&= \prod_{\kappa} B_{\kappa\nu}^\nu \prod_{\nu} B_{\kappa\mu}^\nu \cdot B_{\kappa\mu}^\mu(k, l) \\
&= \prod_{\kappa} B_{\kappa\nu}^\nu \prod_{\nu} B_{\kappa\mu}^\nu \cdot \alpha.
\end{align*}
\]

We transform the first term of (3.11) by observing that \( B_{\kappa\mu}^\nu(i,j) = B_{\kappa\nu}^\nu(i,j) \), except for \( j = l \):

\[
\prod_{r=1}^{k} \frac{B_{\kappa\nu}^\nu(r,l)}{B_{\kappa\mu}^\mu(r,l)} = \prod_{r=1}^{k} \frac{h_{\nu}^\nu(r,l)}{h_{\mu}^\mu(r,l)} = \prod_{r=1}^{k} \frac{k - r + 1 + \alpha(\kappa_r - l)}{k - r + \alpha(\kappa_r - l + 1)}.
\]

To simplify the second term of (3.11), we observe that \( B_{\kappa\mu}^\nu(i,j) = B_{\kappa\nu}^\nu(i,j) \), except for \( i = k \) and \( j = l \). Also \( \kappa_j' = \mu_j' = \nu_j' \) for \( 0 \leq r \leq k - 1 \), and \( \kappa_k' = \mu_k' \neq \nu_k' = \mu_k' - 1 = \nu_k' \).

Therefore

\[
\begin{align*}
\prod_{r=1}^{l-1} & \frac{B_{\kappa\mu}^\mu(r,l)}{B_{\kappa\nu}^\nu(r,l)} \\
&= \prod_{r=1}^{l-1} \frac{h_{\mu}^\mu(r,l)}{h_{\nu}^\nu(r,l)} \\
&= \prod_{r=1}^{l-1} \frac{\mu_r' - k + \alpha(\mu_r - r + 1)}{\mu_r' - k + \alpha(\mu_r - r)} \prod_{r=1}^{k-1} \frac{k - r + \alpha(\mu_r - l + 1)}{k - r + \alpha(\mu_r - l)}.
\end{align*}
\]

which yields (3.10). □
4. Algorithms for efficient evaluation of \( m \| F_q^{(a)}(a_1:p; b_1:q; X) \)

Algorithm 4.1 computes \( m \| F_q^{(a)}(a_1:p; b_1:q; X) \) in the easy special case when \( X \) is a multiple of the identity. Algorithm 4.2 handles the general case.

Both algorithms recursively generate all partitions \(|\kappa| \leq m\) with at most \( n \) parts by allowing \( \kappa_1 \) to take all values \( 1, 2, \ldots, m \) and, independently, \( \kappa_i, i = 1, 2, \ldots, n \), to take all values \( 1, 2, \ldots, \kappa_{i-1} \), subject to the restriction \(|\kappa| \leq m\). The coefficients \( Q_\kappa \) are updated using (3.5). The Jack functions are updated using (3.7) or (3.8) as appropriate.

4.1. The case when \( X \) is a multiple of the identity.

**Algorithm 4.1** (Hypergeometric function, \( X = xI \)). The following algorithm computes \( m \| F_q^{(a)}(a_1:p; b_1:q; xI) \). The variables \( x, n, \alpha, s, a, b, \) and \( \kappa \) are global.

```plaintext
function s = hgi(m, a, b, n, x)
    s = 1
    summation(1, 1, m)
    function summation(i, z, j)
        for \( \kappa_i = 1 : \min(\kappa_{i-1}, j) \) (defaults to \( j \) for \( i = 1 \))
            \( z = z \cdot (n - i + 1 + \alpha(\kappa_i - 1)) \cdot T \) (where \( T \) is the right-hand side of (3.5))
            \( s = s + z \)
            if \( (j > \kappa_i) \) and \( (i < n) \) then
                summation(i + 1, z, j - \kappa_i)
            endif
        endfor
        \( \kappa_i = 0 \)
    endfunction
    return s
endfunction
```

In Algorithm 4.1 the variable \( z \) equals the \( \kappa \)-term in (3.4); it is updated using (3.5) and (3.7). The parameter \( j \) in summation equals \( m - |\kappa| \).

The hypergeometric function of two matrix arguments (1.5) is in this case

\[
m \| F_q^{(a)}(a_1:p; b_1:q; xI, yI) = m \| F_q^{(a)}(a_1:p; b_1:q; xyI).
\]

4.2. The general case. We use the identity (3.8) to update the Jack function, but in order to use it efficiently, we need to store and reuse the Jack functions computed earlier.

Therefore we index all partitions \(|\kappa| \leq m\) with at most \( n \) parts \((\kappa_{n+1} = 0)\) by linearizing the \( m \)-tree that they form (each node \( \kappa = (\kappa_1, \kappa_2, \ldots, \kappa_k) \) has at most \( m \) children \((\kappa_1, \kappa_2, \ldots, \kappa_k, \kappa_{k+1}) \), \(|\kappa_{k+1}| = 1, 2, \ldots, \min(\kappa_k, m - |\kappa|) \), \( k < n \)). In other words, if \( P_{mn} = \#\{\kappa \mid |\kappa| \leq m, \kappa_{n+1} = 0\} \), we assign a distinct integer index \( N_\kappa \in \{0, 1, \ldots, P_{mn}\} \) to every such partition \( \kappa \). We start by assigning the indexes \( 0, 1, \ldots, m \) to partitions with one part: \( N_{(i)} \equiv i, i = 0, 1, 2, \ldots, m \). Then, recursively, once an index \( N_\kappa \) has been assigned to a partition \( \kappa = (\kappa_1, \ldots, \kappa_k) \) with \( k \) parts, we assign \( m - |\kappa| \) consecutive unassigned indexes to the partitions \((\kappa_1, \ldots, \kappa_k, \kappa_{k+1}) \), \(|\kappa_{k+1}| = 1, 2, \ldots, m - |\kappa| \). We record the tree structure in an array \( D \) such that \( D(N_{(\kappa_1, \ldots, \kappa_k)}) = N_{(\kappa_1, \ldots, \kappa_k)} \). Now given \( \kappa \), we can compute \( N_\kappa \) by starting with \( N_{(\kappa_1)} = \kappa_1 \) and using the recurrence

\[
N_{(\kappa_1, \ldots, \kappa_i)} = D(N_{(\kappa_1, \ldots, \kappa_{i-1})}) + \kappa_i - 1.
\]

We store every computed \( J^{(a)}_\kappa(x_{1:i}) \) \(||\kappa|| \leq m, \kappa_{n+1} = 0, i = 1, 2, \ldots, n\) in the \((N_\kappa, i)\)th entry of an \( P_{mn} \times n \) array, which we call “\( J \)” in Algorithm 4.2 below.
We compute the value of $P_{mn}$ as follows. Let $p_k(i)$ be the number of partitions of $i$ with exactly $k$ parts. The $p_k(i), i = 1, 2, \ldots, m, k = 1, 2, \ldots, \min(m, n)$, are computed using the recurrence $p_k(i) = p_{k-1}(i-1) + p_k(i-k)$ \cite{21} p. 28. Then

$$P_{mn} = \sum_{i=1}^{m} \sum_{k=1}^{\min(m,n)} p_k(i).$$

Next, we present our main algorithm.

\textbf{Algorithm 4.2} (Hypergeometric function). The following algorithm computes $m F_1^{(\alpha)}(a_1 \ldots \alpha; b_1 \ldots \beta; X)$, where $X = \text{diag}(x)$. The variables $m, n, a, b, \alpha, x, J, D, H, \kappa$, $N, \mu, N_{\mu}$, and $s$ are global.

function $s = \text{hg}(m, \alpha, a, b, x)$
  Compute $P_{mn}$ using (4.2)
  $n = \text{length}(x); \ H = m + 1; \ s = 1; \ D = \text{zeros}(P_{mn}, 1)$
  $J = \text{zeros}(P_{mn}, n); \ J(1,:) = 1 \quad (J_{\kappa}^{(\alpha)}(x_{1:1}) \text{ is stored in } J(N_{\kappa}, i))$
  summation(1, 1, 1, $m$)
function summation($i, z, j$)\hspace{1cm} (defaults to $j$ for $i = 1$)
  for $\kappa_i = 1 : r$
    if $(\kappa_i = 1)$ and $(i > 1)$ then $D(N_{\kappa}) = H; \ N_{\kappa} = H; \ H = H + r$
    else $N_{\kappa} = N_{\kappa} + 1$
    endif
    $z = z \cdot T \quad \text{(where } T \text{ is the right-hand side of (3.5))}$
    if $\kappa_i = 1$ then
      $J(N_{\kappa}, 1) = x_1 (1 + \alpha(\kappa_i - 1)) \cdot J(N_{\kappa} - 1, 1)$
    endif
    for $t = 2 : n$ do jack(0, 1, 0, $t$) \hspace{1cm} (computes $J_{\kappa}^{(\alpha)}(x_{1:t})$)
    endfor
    $s = s + z \cdot J(N_{\kappa}, n, 1)$
  if $(j > \kappa_i)$ and $(i < n)$ then
    summation($i + 1, z, j - \kappa_i$)
  endif
endfor
function jack($k, \beta_{\kappa, \mu}, c, t$)
  for $i = k : \mu_i'$
    if $(k > 0)$ and $(\mu_i > \mu_{i+1})$
      $d = N_{\mu}$
      $\mu_i = \mu_i - 1; \quad \text{Compute } N_{\mu}$ using (4.1)
      Update $\beta_{\kappa, \mu}$ using (5.10)
    if $\mu_i > 0$ then jack($i, \beta_{\kappa, \mu}, c + 1, t$)
    else $J(N_{\kappa}, t) = J(N_{\kappa}, t) + \beta_{\kappa, \mu} \cdot J(N_{\mu}, t - 1) \cdot x_t^{c+1}$
    endif
    $\mu_i = \mu_i + 1; \ N_{\mu} = d$
  endif
endfor
if $k = 0$ then $J(N_{\kappa}, t) = J(N_{\kappa}, t) + J(N_{\kappa}, t - 1)$
else $J(N_{\kappa}, t) = J(N_{\kappa}, t) + \beta_{\kappa, \mu} \cdot x_t^c$
endif
Algorithm 4.2 generates all partitions $|\kappa| \leq m$ with at most $n$ parts analogously to Algorithm 4.1. The parameter $z$ in summation is now just $Q_\kappa$.

For every $\kappa$ the function $\text{jack}$ recursively generates all partitions $\mu \leq \kappa$ such that $\kappa/\mu$ is a horizontal strip. The Jack function is computed using (3.8); $\beta_{\kappa \mu}$ is updated using (3.10). The parameter $c$ equals $|\kappa| - |\mu|$ at all times.

4.3. Implementation notes. In our implementation of Algorithms 4.1 and 4.2 in \cite{13} we made a few fairly straightforward, but important improvements:

1. we precompute the values of $x_i^j$ for $i = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, m$;
2. we keep and update the conjugate partitions $\kappa'$ and $\mu'$ along with $\kappa$ and $\mu$, so we never need to recover $\kappa'$ and $\mu'$ when computing (3.5) and (3.10);
3. when two sets of arguments $(x_{1:n}$ and $y_{1:n})$ are passed, the hypergeometric function of two matrix arguments, $^{(1)}_p F^\alpha_q (a_1:p; b_1:q; \tau I)$ for $j = 1, 2, \ldots, r$.

5. Complexity analysis

Algorithm 4.1 (the case $X = xI$) costs $O(P_{mn})$ arithmetic operations (where again, $P_{mn}$ is the number of partitions $\kappa$, $|\kappa| \leq m$ with at most $n$ parts).

To bound the complexity of Algorithm 4.2 (general case) we observe that the formula (3.8) represents the summation of at most $P_{mn}$ terms (in fact a lot less, but we have been unable to obtain a better bound than $P_{mn}$ that is easy to work with). We use (3.8) to compute the Jack function $J^K_{\alpha}(x_{1:t})$ for all $|\kappa| \leq m$ and $t \leq n$, i.e., $P_{mn} \cdot n$ Jack functions, each of which costs at most $O(P_{mn})$ arithmetic operations. The overall cost of Algorithm 4.2 is thus bounded by

$$O(P_{mn}^2 \cdot n).$$

There is no explicit formula for $P_{mn}$, so we use Ramanujan’s asymptotic formula \cite{11} p. 116 for number of partitions of $m$

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

to obtain

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

Therefore the complexity of Algorithm 4.2 is linear in $n$ and subexponential in $m$.

6. Numerical experiments

We performed extensive numerical tests to verify the correctness and complexity of our Algorithms 4.1 and 4.2. We compared the output of our algorithms for $^{(0)}_0 F^\alpha_0$ and $^{(1)}_1 F^\alpha_1$ with explicit expressions (subsection 6.1). We also compared the probability distributions of the eigenvalues of certain random matrices (which are expressed as hypergeometric functions) against the results of Monte Carlo experiments (subsection 6.2). Finally, we present performance results in subsection 6.3.
6.1. **Explicit expressions.** We compared the output of Algorithms [4.1] and [4.2] against the expressions [17] p. 262:

\[(6.1)\]
\[\alpha F_0^{(\alpha)}(X) = e^{\text{tr} X},\]

\[(6.2)\]
\[\beta F_0^{(\beta)}(a; X) = \det(I - X)^{-\alpha},\]

for \(n = 10\) and randomly uniformly distributed values \(x_i \in [0, \frac{1}{2}]\), \(i = 1, 2, \ldots, n\). For \(m = 30\) it took less than one second per test. The results agreed to at least 12 decimal digits with (6.1), and at least 8 decimal digits with (6.2), reflecting the slower convergence of (6.2).

6.2. **Eigenvalue statistics.** We tested Algorithms [4.1] and [4.2] against the eigenvalue statistics of the \(\beta\)-Laguerre and Wishart matrices.

The \(n \times n\) \(\beta\)-Laguerre matrix of parameter \(a\) is defined as \(L = BB^T\), where

\[B = \begin{bmatrix}
\chi_{2a} & \chi_{\beta(n-1)} & \chi_{2a-\beta} \\
\chi_{\beta(n-1)} & \chi_{2a-\beta} & \ddots \\
\vdots & \ddots & \ddots \\
\chi_{\beta} & \chi_{2a-\beta} & \chi_{\beta(n-1)}
\end{bmatrix}, \quad a > \frac{\beta}{2}(n-1).
\]

The \(n \times n\) Wishart matrix with \(l\) degrees of freedom \((l > n)\) and covariance matrix \(\Sigma\), \(A = W_n(l, \Sigma)\), is defined as \(A = \Sigma^{1/2} \cdot Z^T \cdot Z \cdot \Sigma^{1/2}\), where \(Z_{ij} = N(0,1)\) for \(i = 1, 2, \ldots, l; j = 1, 2, \ldots, n\).

The eigenvalue distributions of \(A\) and \(L\) are the same when \(\beta = 1\), \(a = \frac{\alpha}{2}\), and \(\Sigma = I\).

The cumulative distribution functions of the largest eigenvalues, \(\lambda_L\) and \(\lambda_A\), of \(L\) and \(A\), are

\[P(\lambda_L < x) = \frac{\Gamma_n(\frac{n-1}{\alpha} + 1)}{\Gamma_n(a + \frac{n-1}{\alpha} + 1)} \left(\frac{x}{2}\right)^{\frac{n-1}{\alpha}} 1F_1^{(\alpha)}(a; a + \frac{n-1}{\alpha} + 1; -\frac{1}{2}xI),\]

\[P(\lambda_A < x) = \frac{\Gamma_n^{(z)}}{\Gamma_n^{(z)}} \det \left(\frac{1}{2}x\Sigma^{-1}\right)^{1/2} 1F_1^{(2)} \left(\frac{l}{2}; \frac{n+1}{2}; -\frac{1}{2}x\Sigma^{-1}\right),\]

respectively [4] Thm. 10.2.1, p. 147], [17] Thm. 9.7.1, p. 420], where \(\alpha = 2/\beta\), and \(\Gamma_n^{(z)}\) is the multivariate Gamma function of parameter \(\alpha\):

\[(6.3)\]
\[\Gamma_n^{(z)}(c) \equiv \prod_{i=1}^n \Gamma \left( c - \frac{i-1}{\alpha} \right) \text{ for } \Re(c) > \frac{n-1}{\alpha}.
\]

We use the Kummer relation [17] Thm. 7.4.3, p. 265] \[1F_1^{(\alpha)}(a; c; X) = e^{\text{tr} X} \cdot 1F_1^{(\alpha)}(c-a; c; -X)\]
to obtain the equivalent, but numerically more stable expressions

\[P(\lambda_L < x) = \frac{\Gamma_n(\frac{n-1}{\alpha} + 1)}{\Gamma_n(a + \frac{n-1}{\alpha} + 1)} \left(\frac{x}{2}\right)^{\frac{n-1}{\alpha}} e^{-\frac{nx}{2}} \cdot 1F_1^{(\alpha)} \left( a + \frac{n-1}{\alpha} + 1; -\frac{1}{2}xI \right),\]

\[P(\lambda_A < x) = \frac{\Gamma_n^{(z)}}{\Gamma_n^{(z)}} \left(\frac{l}{2}; \frac{n+1}{2}; -\frac{1}{2}x\Sigma^{-1}\right),\]

which we plot on Figure [4] along with the Monte Carlo results from a sample of 10000 random matrices.
Next we consider the distribution of the smallest eigenvalue of \( L \) and that of \( \text{tr} A \).

If \( c = a - \frac{\beta}{2}(n - 1) - 1 \) is a nonnegative integer, then the probability density function of the smallest eigenvalue of \( L \) (see, e.g., [4, Thm. 10.1.1, p. 146]) is proportional to

\[
(6.4) \quad f(x) = x^c \cdot e^{-x/2} \cdot \frac{\Gamma(2\beta)}{\Gamma(\beta n + 1)} \quad _2F_0^{(2/\beta)}(-c, \beta n + 1; -\frac{x}{2} I_{n-1}).
\]

Since \(-c\) is a nonpositive integer, the series expansion of \(_2F_0^{(2/\beta)}\) in (6.4) terminates, even though it diverges in general.

The probability density function of \( \text{tr} A \) is

\[
(6.5) \quad f(x) = \det(\lambda^{-1} \Sigma)^{-1/2} \sum_{k=0}^{\infty} \frac{g_{r,2\lambda}(x)}{k!} \sum_{\kappa^2-k} (\frac{1}{2})_\kappa J_{\kappa^2}(I - \lambda \Sigma^{-1})
\]

where

\[
g_{r,2\lambda}(x) = \frac{e^{-x/2\lambda} \Gamma(r)}{(2\lambda)^r} \quad (x > 0),
\]
and $\lambda$ is arbitrary [17, p. 341]. We follow the suggestion by Muirhead to use $\lambda = 2\lambda_1\lambda_l/(\lambda_1 + \lambda_l)$, where $\lambda_1$ and $\lambda_l$ are the largest and smallest eigenvalues of $\Sigma$, respectively. Although the expression (6.5) is not a hypergeometric function of a matrix argument, its truncation for $|\kappa| \leq m$ has the form (1.6), and is computed analogously.

We plot (6.4) and (6.5) on Figure 2 and compare the theoretical predictions of these formulas with experimental data.

6.3. Performance results. In Figure 3 we demonstrate the efficiency of Algorithms 4.1 and 4.2 on an 1.8GHz Intel Pentium 4 machine.

In the left plot we present the performance data for Algorithm 4.1 (whose complexity is independent of the size $n$ of the matrix $X = xI$). Its efficiency is evident—we need to sum beyond partitions of size $m = 52$ before this algorithm takes a full second (for reference, the $k!$ in the denominator of the $\kappa$-term in (1.1) then reaches up to $52! \approx 8 \cdot 10^{67}$).

Algorithm 4.2 is also very efficient. The right plot of Figure 3 clearly demonstrates its linear complexity in $n$. It also takes at most a few seconds on matrices of size $n \leq 120$ and partitions of size $m \leq 30$. 
7. Conclusions and open problems

We have presented new algorithms for computing the truncation of the hypergeometric function of a matrix argument. They exploit the combinatorial properties of the Pochhammer symbol and the Jack function to achieve remarkable efficiency, and have lead to new results [1, 6].

Several problems remain open, among them automatic detection of convergence. The $\kappa$-term in (3.4) does approach zero as $|\kappa| \to \infty$, but it need not monotonically decrease. Although we have

$$\prod_{j=1}^{\kappa-1} \left( \frac{g_j - \alpha}{g_j (c_j + \alpha)} \right) \prod_{j=1}^{j-1} \frac{l_j - f_j}{l_j + h_j} < 1$$

in (3.5), it is not always true that $Q_\kappa J_\kappa(\alpha) \leq Q_{\kappa(i)} J_{\kappa(i)}(\alpha)$, and it is unclear how to tell when convergence sets in.

Another open problem is to determine the best way to truncate the series (1.1). Our choice to truncate it for $|\kappa| \leq m$ seems to work well in practice, but one can imagine selecting a partition $\lambda$ and truncating for $\kappa \leq \lambda$ instead of, or in addition to $|\kappa| \leq m$.

Acknowledgments

We thank Brian Sutton and Per-Olof Persson for several discussions that resulted in the simplification of the implementation of our algorithms, as well as the
anonymous referee for the useful comments, which lead to improvements in the exposition.

References


Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139
E-mail address: plamen@math.mit.edu

Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139
E-mail address: edelman@math.mit.edu