REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS

Han Wu and Max Kontorovich
Advisor: Professor Alexander Barvinok

June 10th, 2016

1 Introduction

Let $A = (a_{ij})$ be an $n \times n$ real or complex matrix. The *permanent* of A is defined as

$$\text{per}A = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)},$$

where S_n is the symmetric group of permutations σ of $\{1, ..., n\}$. The permanent is difficult to compute; the quickest known exact algorithm of roughly 2^n complexity belongs to Ryser and goes as follows.

Let $t_1, ..., t_n$ be independent commuting variables and let

$$r_A(t_1, ..., t_n) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} t_j \right)$$

For a subset $I \subseteq \{1, ..., n\}$ let $t_I = (t_1, ..., t_n)$ be a vector where

$$t_i = \begin{cases} 0 \text{ if } i \in I \\ 1 \text{ if } i \notin I \end{cases}$$

Then we have

$$\text{per}A = \sum_{I} (-1)^{|I|} r_A(t_I)$$

2 Approximating ln(per)

As can be seen, computing permanent directly is difficult, so we explore a method presented by Prof. Barvinok to approximate ln(per). The basic idea is as follows: let J_n be the $n \times n$ matrix filled with 1s. Following in the footsteps of lemma (3.5) of [1], consider the univariate function

$$f_A(z) = \ln \text{per}(J_n + z(A - J_n))$$

in a neighborhood of $z = 0$ and let

$$T_{m,A}(z) = f_A(0) + \sum_{k=1}^{m} f_A^{(k)}(0) \frac{z^k}{k!}$$

be the Taylor polynomial of $f_A(z)$ of degree m at $z = 0$. Then define

$$p_{m,n}(A) = T_{m,A}(1) = f_A(0) + \sum_{k=1}^{m} \frac{f_A^{(k)}(0)}{k!}$$

Then for complex $n \times n$ matrices $A, p_{m,n}(A)$ approximates $\ln \operatorname{per} A$ reasonably well so long as

$$|1 - a_{ij}| \leq \delta \tag{2.1}$$

for all $1 \leq i, j \leq n$ and $\delta \in (0, 1)$. Much of our work went into implementing this approximation. Naively one could just compute the functions as written, but that requires computing a permanent, which defeats the purpose, so we need to improve the speed.

Let $g_A(z) = \operatorname{per}(J_n + z(A - J_n))$. It is shown in [1] that computing the first m derivatives $f^{(1)}(0), ..., f^{(m)}(0)$ reduces to computing the first m derivatives of $g^{(1)}(0), ..., g^{(m)}(0)$. Retrieving the values of the derivatives of f from those of g is equivalent to solving the linear system in section 3.6 of [1] and can be done in $O(m^2)$ time. It can also be shown that

$$g^{(k)}(0) = (n-k)! \sum_{\substack{(i_1,..,i_k) \\ (j_1,...,j_k)}} (a_{i_1 j_1} - 1) \cdots (a_{i_k j_k} - 1)$$

where the sum is taken over all ordered sets $\{i_1, ..., i_k\}, \{j_1, ..., j_k\}$ satisfying $1 \leq i_1 < ... < i_k \leq n$ and $1 \leq j_1 < ... < j_k \leq n$. This can be further re-written as

$$g^{(k)}(0) = (n-k)! k! w_k(A - J_n)$$

where $w_k B$ is the sum of the permanents of all $k \times k$ submatrices of a matrix B; we introduce the $k!$ term to account for the fact that the permanent of those submatrices is exactly the sum, only taken over unordered sets. The approach for computing $w_k A$ is given in [2] as follows:

Let $p(\mathbf{x})$ be a polynomial of degree k in $n \geq k$ variables with $p(0) = 0$. Define

$$s_i = \sum_{\substack{b_j \in \{0,1\} \\ \sum_{j=1}^{n} b_j = i}} p(b_1, ..., b_n) \tag{2.2}$$

where the sum is taken over all 0-1 vectors of length n with exactly i 1s.

Define an $k - 1 \times k - 1$ lower triangular matrix $A = (a_{ij})$ by

$$a_{ij} = \binom{n-j}{i-j} \quad \text{if } i \geq j \quad \text{and} \quad a_{ij} = 0 \quad \text{otherwise}$$

Let $\mathbf{d}_n = (d_{n,1}, ..., d_{n,k-1})$ be the unique solutions of the system of linear equations $\mathbf{d}_n A = (-1, ..., -1)$. Then we have

$$\sum_{1 \leq i_1 < ... < i_k \leq n} \frac{\partial^k p}{\partial x_{i_1} ... \partial x_{i_k}}(\mathbf{0}) = p(1, ..., 1) + \sum_{1 \leq j \leq m-1} s_j d_{n,j} \tag{2.3}$$

For $\mathbf{x} := (x_1, ..., x_n)^\top \in \mathbb{C}^n$ let

$$S_k(\mathbf{x}) = \sum_{1 \leq i_1 < ... < i_k \leq n} x_{i_1} ... x_{i_k}$$

Let A be an $n \times n$ complex matrix and define $p_{k,A}(\mathbf{x}) := S_k(A\mathbf{x})$. Then we have

$$w_k A = p_{k,A}(1, ..., 1) + \sum_{1 \leq j \leq k-1} s_j d_{n,j} \tag{2.4}$$

where the s_i is defined by 2.2 for $p = p_{k,A}$. Everything in this approach can be implemented in a straightforward manner with the exception of the computing of $S_k(\mathbf{x})$, which can be done very quickly using Newton's identities.

Using a third degree approximation (which is motivated by the incredible accuracy of this approximation, see Table 1), this method allows us to approximate 100×100 matrices in less than 10 seconds, while the naive method (which computed permanents using Rysers method) was limited to about 15×15 in a similar amount of time. For a more complete idea of the speed of this approximation, see Table 3, which shows the average time (taken over 100 trials) to compute the 3rd degree approximations of matrices whose entries were selected independently randomly from the interval $(0, 2)$. For accuracy of approximation, see Table 1. Note that we only give the accuracy of the approximations up to $n = 15$ since we need to compute the real value to get the error, and we cannot handle larger matrices quickly. Table 1 shows the average percent errors for 100 matrices who's entries were picked independently randomly subject to 2.1 and $\delta \in \{0.1, 0.2, ..., 0.9\}$ and $n \in \{3, 4, ..., 15\}$ when approximated to degree 3. For results using complex matrices, see 9, which shows the average error (taken over 100 trials) of matrices whose complex entries were selected independently randomly from the set of a_{ij} satisfying $|1 - a_{ij}| < \delta$ for various δ. Again we use a third degree approximation. Here we stop at $n = 14$ because computing the real value of the permanent for complex matrices is about twice as slow as for real matrices, and we cannot handle 15×15 in a reasonable amount of time. Note that all error terms (in all tables presented in this paper) are errors approximating the actual permanent, not the natural log of the permanent.

2.1 0-1 Matrices

There is great interest in computing the permanent of 0-1 matrices because of their relation to graph theory, and the approach above gives good numerical results for such matrices. Table 2 gives average percent errors of matrices constructed randomly (entries chosen independently) with probability p of a given entry being 0 (taken over 100 trials).

3 Doubly stochastic matrices

As is shown in [6], computing the permanent of a non-negative matrix reduces by scaling to computing the permanent of a doubly stochastic matrix. This motivates the need for similar approximation schemes for doubly stochastic matrices.

3.1 Defining $q_A(z)$

In an attempt to create such a scheme we answer a question posed in [1]. Define the univariate polynomial

$$q_A(z) = \text{per}\left(\frac{1}{n}J_n + z(A - \frac{1}{n}J_n)\right) \tag{3.1}$$

where J_n is the $n \times n$ matrix filled with 1s. Does there exist an absolute constant $\gamma > 0$ such that

$$q_A(z) = 0 \quad \text{for} \quad z \in \mathbb{C} \implies \text{dist}(z, [0, 1]) \geq \gamma$$

for any doubly-stochastic A? If the answer is yes, then a modification of Section 4.2 of [1] would produce a similar approximation scheme as the one presented in Section 2.

Unfortunately, we prove that no such constant exists. To do this, define

$$w_k(A) := \sum_{\substack{I,J\subseteq\{1,\dots,n\}\\ |I|=|J|=k}} \mathrm{per}A[I,J] \tag{3.2}$$

where $A[I,J]$ is the $k\times k$ matrix with entries in rows from I and columns from J. Then by the work of professor Barvinok we have the following reformulation:

$$\begin{aligned}
q_A(z) &= \mathrm{per}\left(zA + (1-z)\frac{1}{n}J_n\right) && (3.3)\\
&= \sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} w_k(A) z^k (1-z)^{n-k} && (3.4)\\
&= (1-z)^n \sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} w_k(A) \frac{z^k}{(1-z)^k}\\
&= (1-z)^n \sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} w_k(A) y^k \quad \text{where} \quad y = \frac{z}{1-z} && (3.5)
\end{aligned}$$

The jump from 3.3 to 3.4 is a direct consequence of Theorem 1.4 of [7]. Because of 3.3 we know $q_A(1) \neq 0$ so we are interested in the roots of

$$\sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} w_k(A) y^k$$

3.2 Special Case: $A = I_n$

Note that

$$w_k(I_n) = \binom{n}{k}$$

Then by substituting in 3.5 we have the further reformulation, also given by professor Barvinok:

$$\begin{aligned}
q_{I_n}(z) &= (1-z)^n \sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} \binom{n}{k} y^k\\
&= (1-z)^n \sum_{k=0}^{n} \frac{n!}{k!n^{n-k}} y^k \quad \text{where} \quad y = \frac{z}{1-z}\\
&= (1-z)^n \frac{n!}{n^n} \sum_{k=0}^{n} \frac{x^k}{k!} \quad \text{where} \quad x = \frac{nz}{1-z}
\end{aligned}$$

Thus we are interested in the roots of

$$r(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$$

Luckily this is the truncated exponential series, and thanks to [5] we know the normalized complex roots of this series converge to the Szegő curve given by the set of complex solutions of the equation

$$\{|ze^{1-z}| = 1\}$$

We know that the Szegő curve is the asymptotic distribution of the roots of $r(nx)$. Thus if x_0 is a root of $r(nx)$ we substitute to get that $nx_0 = \frac{nz}{1-z} \implies z = \frac{x_0}{1+x_0}$. As $n \to \infty$ we know that the values of x_0 become the Szegő curve. In particular, since the Szegő curve passes through $(1,0)$, our transformed Szegő curve passes through $(1/2, 0)$. Numerical evidence of this fact can be found in figure 1.

4 Negative matrices

Lastly we work to expand our method to real matrices with negative entries. Consider a matrix A such that the entries are non-negative and the row and column sums do not exceed 1. Define the polynomial

$$r_A(z) = \operatorname{per}\left(\frac{1}{n}J_n + zA\right) \quad \text{for} \quad z \in \mathbb{C}.$$

If $\rho > 0$ such that

$$r_A(z) \neq 0 \quad \text{for} \quad |z| < \rho$$

then for any $0 < \delta < \rho$ we can efficiently approximate

$$r_A(-\delta) = \operatorname{per}\left(\frac{1}{n}J_n - \delta A\right)$$

which gives a method to approximate permanents of real matrices $\frac{1}{n}J_n - \delta A$ that allow positive, negative, and zero entries. As before we can write

$$r_A(z) = \sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} w_k(A) z^k$$

From [3] we know that the roots of

$$p_A(z) = \sum_{k=0}^{n} w_k(A) z^k$$

are negative real and satisfy $z \leq -\frac{1}{4}$. From [4] we can find a real $\rho_0 > 0$ such that all roots of the polynomial

$$e_n(z) = \sum_{k=0}^{n} \frac{(nz)^k}{k!}$$

satisfy $|z| > \rho_0$. In fact the largest value of ρ_0 is given as the distance of the point on the Szegő which is closest to the origin, and is the solution to $xe^{1-x} = -1$, which is ≈ 0.278. Then by a theorem of Szegő given in [8] we know that the roots of the Schur composition

$$\begin{aligned}(p_A \circ e_n)(z) &= \sum_{k=0}^{n} \frac{n^k w_k(A)}{k!\binom{n}{k}} z^k \\ &= \frac{n^n}{n!} \sum_{k=0}^{n} \frac{(n-k)!}{n^{n-k}} w_k(A) z^k \\ &= \frac{n^n}{n!} r_A(z)\end{aligned}$$

satisfy $|z| > \frac{\rho_0}{4}$, so we can choose

$$\rho = \frac{\rho_0}{4}$$

In other words we can use the approximation method of section 2 to approximate $r_A(-\delta)$ for $0 < \delta < \frac{0.278}{4}$. For numerical results, see Tables 4, 5, and 6. The first of these tables shows the average percent error of matrices constructed randomly as follows. A matrix A would be made by choosing each entry independently randomly from $(0, 1/n)$, and then we would approximate $\frac{1}{n}J_n - \delta A$ with the various δ shown. Each spot in the table represents average percent error for 100 trials. The second and third tables shows the case when A is the $n \times n$ identity matrix, and again we approximate $\frac{1}{n}J_n - \delta A$ for various δ. We give two tables, one shows the data for a large spread of δ, and the other zooms in around the values of δ where the approximation begins to break down. In all cases we used a third degree approximation. While we can only guarantee that these methods work for $0 < \delta < \frac{0.278}{4} \approx 0.07$, the data suggests that we can reasonably use this approach for $0 < \delta < 1/3$, and if the matrix is unstructured for $0 < \delta < 0.8$. Lastly this method can be used to approximate 0-1 matrices as follows:

If B is an $n \times n$ matrix of 0s and 1s, we may ask how many 0s can we afford in each row and column so that per$B$ can be approximated using the same approach. In other words we can write

$$\frac{1}{n}B = \frac{1}{n}J_n - tA$$

where A is an $n \times n$ non-negative matrix with row and column sums not exceeding 1 and $0 \leq t < \rho$. Fix any row (or column) i and let k_i be the number of 0's in the the ith row of B. Then we have

$$tA = \frac{1}{n}(J_n - B)$$

Since $\sum_{j=1}^{n} a_{ij} \leq 1$ the above implies that $t \geq \frac{k_i}{n} \implies \rho \geq \frac{k_i}{n} \implies \approx \frac{0.278}{4} \geq \frac{k_i}{n}$ meaning we can guarantee only $\frac{0.278}{4} \approx 7\%$ of zeros. However, numerical trials suggest that we can realistically allow more zeros. For example, see tables 7 and 8, which give 3rd degree approximations for random $0-1$ matrices B with exactly $\lfloor \frac{\delta}{n} \rfloor$ 0s in each row and column. Here the entries are not independently random, because we require an exact number of 0s in each row and column - however we do randomly sample from the set of matrices with exactly $\lfloor \frac{\delta}{n} \rfloor$ 0s in each row and column. We take 100 trials for various n and δ, and the second table zooms in around where the approximation begins to break down.

5 Complex Roots

Lastly we address and unrelated question posed by professor Barvinok: Theorem 1.4 of [1] states that if $Z = (z_{ij})$ is an $n \times n$ complex matrix such that

$$|1 - z_{ij}| \leq \frac{1}{2} \quad \forall i, j$$

then per$Z \neq 0$. What is the largest possible $\delta > 0$ such that if $Z = (z_{ij})$ is an $n \times n$ complex matrix satisfying

$$|1 - z_{ij}| \leq \delta \quad \forall i, j$$

then per$Z \neq 0$?

While we cannot answer the question definitively, we note that the proof of Theorem 1.4 relies on a geometric lemma about complex vectors and a chain of inequalities. Consider

$$M = \begin{pmatrix} \frac{7}{6} + \frac{\sqrt{2}}{3}i & \frac{3}{4} + \frac{\sqrt{3}}{4}i \\ 1 & \frac{3}{2} \end{pmatrix}$$

This matrix achieves every equality in the proof and the permanent is non-zero, which suggests that if $1/2$ is a sharp bound, it can only be achieved asymptotically.

6 Tables and figures

Table 1: percent error of random matrices for various n and δ

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
3	4.46E-04	5.56E-03	3.11E-02	1.05E-01	2.98E-01	6.52E-01	1.29E+00	2.50E+00	3.71E+00
4	3.80E-04	5.37E-03	3.03E-02	9.59E-02	2.37E-01	5.52E-01	1.11E+00	1.83E+00	3.13E+00
5	3.03E-04	4.65E-03	1.93E-02	6.26E-02	1.94E-01	3.13E-01	6.88E-01	1.58E+00	2.38E+00
6	2.30E-04	3.67E-03	1.50E-02	6.39E-02	1.32E-01	3.26E-01	4.94E-01	1.00E+00	1.84E+00
7	1.64E-04	3.01E-03	1.50E-02	4.95E-02	1.47E-01	3.09E-01	4.98E-01	1.09E+00	1.29E+00
8	1.59E-04	2.22E-03	1.04E-02	3.57E-02	9.56E-02	1.79E-01	4.18E-01	7.85E-01	1.15E+00
9	1.44E-04	2.06E-03	1.06E-02	3.43E-02	7.96E-02	1.81E-01	3.84E-01	5.39E-01	1.03E+00
10	1.24E-04	1.77E-03	1.02E-02	2.99E-02	8.92E-02	1.63E-01	2.69E-01	5.41E-01	7.93E-01
11	9.11E-05	1.63E-03	7.77E-03	2.39E-02	6.89E-02	1.53E-01	2.49E-01	4.84E-01	7.39E-01
12	1.01E-04	1.66E-03	8.82E-03	2.46E-02	5.87E-02	1.36E-01	2.95E-01	4.62E-01	7.74E-01
13	8.28E-05	1.28E-03	6.69E-03	2.16E-02	5.30E-02	9.47E-02	2.08E-01	4.12E-01	7.25E-01
14	8.90E-05	1.45E-03	6.76E-03	2.15E-02	6.01E-02	9.85E-02	2.19E-01	3.23E-01	5.97E-01
15	8.18E-05	1.43E-03	6.39E-03	2.14E-02	4.42E-02	1.03E-01	1.91E-01	3.71E-01	5.61E-01

Table 2: percent error of random 0-1 matrices for various n and probability p of zeros

	0.1	0.2	0.3	0.4	0.5
3	1.60E+00	6.97E+00	9.33E+00	1.51E+01	1.55E+01
4	1.12E+00	7.17E+00	1.69E+01	1.88E+01	2.97E+01
5	1.09E+00	1.02E+01	2.12E+01	2.98E+01	4.05E+01
6	6.96E-01	5.65E+00	1.37E+01	5.06E+01	6.50E+01
7	7.94E-01	6.23E+00	2.11E+01	3.65E+01	9.45E+01
8	5.25E-01	3.24E+00	1.37E+01	3.59E+01	1.81E+02
9	4.87E-01	3.28E+00	1.18E+01	4.18E+01	1.23E+02
10	4.25E-01	3.03E+00	1.26E+01	5.22E+01	1.15E+02
11	4.38E-01	2.75E+00	9.77E+00	3.06E+01	1.36E+02
12	4.16E-01	2.49E+00	1.06E+01	3.92E+01	1.34E+02
13	5.14E-01	2.78E+00	1.11E+01	3.36E+01	1.25E+02
14	3.61E-01	2.65E+00	1.10E+01	3.88E+01	1.14E+02
15	3.74E-01	2.96E+00	1.19E+01	3.99E+01	1.20E+02

Table 3: average time (of 100 trials) to compute $p_3, n(A)$ for various n and A, where each a_{ij} is selected independently randomly from $(0, 2)$

n	time (sec)
20	0.0995
40	0.4636
60	1.1916
80	4.8871
100	8.4683
120	14.8043
140	23.3279
160	30.187
180	42.0778
200	61.2965

Table 4: Average percent error (taken over 100 trials) of the 3rd degree approximation of $J_n - \delta A$ for various δ and n, where A is constructed by randomly independently selecting a_{ij} from the interval $(0, 1/n)$.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
3	9.00E-04	1.53E-02	8.34E-02	2.81E-01	6.34E-01	1.93E+00	2.82E+00	6.99E+00	1.45E+01
4	1.00E-03	1.69E-02	9.16E-02	3.37E-01	8.43E-01	2.03E+00	3.25E+00	7.17E+00	1.55E+01
5	1.20E-03	1.85E-02	1.11E-01	3.91E-01	1.01E+00	1.96E+00	4.13E+00	8.87E+00	1.47E+01
6	1.30E-03	2.24E-02	1.13E-01	4.01E-01	1.02E+00	2.25E+00	4.62E+00	9.08E+00	1.61E+01
7	1.50E-03	2.43E-02	1.26E-01	4.40E-01	1.14E+00	2.83E+00	5.01E+00	1.04E+01	1.83E+01
8	1.70E-03	2.73E-02	1.47E-01	5.07E-01	1.33E+00	2.82E+00	6.10E+00	1.12E+01	2.17E+01
9	1.80E-03	3.07E-02	1.57E-01	5.35E-01	1.42E+00	3.18E+00	6.43E+00	1.24E+01	2.40E+01
10	2.00E-03	3.37E-02	1.81E-01	6.14E-01	1.50E+00	3.57E+00	6.92E+00	1.36E+01	2.45E+01
11	2.10E-03	3.58E-02	1.85E-01	6.34E-01	1.66E+00	3.84E+00	7.50E+00	1.43E+01	2.66E+01
12	2.30E-03	3.77E-02	2.04E-01	6.97E-01	1.75E+00	4.07E+00	8.13E+00	1.54E+01	2.73E+01
13	2.50E-03	4.02E-02	2.25E-01	7.19E-01	1.87E+00	4.45E+00	8.37E+00	1.64E+01	3.01E+01
14	2.60E-03	4.43E-02	2.43E-01	7.97E-01	1.98E+00	4.55E+00	9.17E+00	1.78E+01	3.25E+01
15	2.90E-03	4.76E-02	2.44E-01	8.30E-01	2.16E+00	4.75E+00	9.59E+00	1.87E+01	3.40E+01

Table 5: Percent error of 3rd degree approximation of $J_n - \delta I_n$ for various δ and n. Note: The ERROR terms in the table indicate that the permanent was negative or zero, and our approximation is meaningless in these cases.

	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45
3	2.38E-03	4.30E-02	2.46E-01	8.84E-01	2.48E+00	6.02E+00	1.35E+01	2.98E+01	7.15E+01
4	3.15E-04	1.19E-02	1.07E-01	5.32E-01	1.90E+00	5.40E+00	1.28E+01	2.53E+01	4.21E+01
5	4.21E-05	3.34E-03	4.72E-02	3.31E-01	1.59E+00	6.17E+00	2.23E+01	1.03E+02	ERROR
6	5.61E-06	9.41E-04	2.10E-02	2.04E-01	1.26E+00	5.64E+00	1.87E+01	4.34E+01	6.96E+01
7	7.86E-07	2.67E-04	9.37E-03	1.28E-01	1.06E+00	6.60E+00	4.12E+01	ERROR	ERROR
8	1.32E-07	7.62E-05	4.21E-03	8.06E-02	8.59E-01	6.06E+00	2.72E+01	6.42E+01	8.82E+01
9	1.01E-08	2.18E-05	1.90E-03	5.11E-02	7.25E-01	7.24E+00	9.18E+01	ERROR	ERROR
10	1.54E-08	6.30E-06	8.62E-04	3.24E-02	5.97E-01	6.64E+00	3.82E+01	8.11E+01	9.62E+01
11	1.31E-08	1.70E-06	3.93E-04	2.07E-02	5.06E-01	8.12E+00	3.99E+02	ERROR	ERROR
12	6.70E-09	5.28E-07	1.79E-04	1.32E-02	4.21E-01	7.37E+00	5.09E+01	9.12E+01	9.88E+01
13	3.38E-07	5.27E-07	8.22E-05	8.48E-03	3.57E-01	9.23E+00	ERROR	ERROR	ERROR
14	2.06E-07	4.15E-08	3.76E-05	5.45E-03	3.00E-01	8.25E+00	6.38E+01	9.62E+01	9.97E+01
15	2.96E-07	4.28E-07	1.78E-05	3.51E-03	2.55E-01	1.06E+01	ERROR	ERROR	ERROR

Table 6: Percent error of 3rd degree approximation of $J_n - \delta I_n$ for various δ and n, zoomed in around where the approximation begins to break down. Note: The ERROR terms in the table indicate that the permanent was negative or zero, and our approximation is meaningless in these cases.

	0.25	0.26	0.27	0.28	0.29	0.3	0.31	0.32	0.33	0.34	0.35
3	2.48	2.99	3.58	4.27	5.08	6.02	7.10	8.36	9.83	11.50	13.50
4	1.90	2.38	2.95	3.64	4.45	5.40	6.51	7.79	9.25	10.90	12.80
5	1.59	2.11	2.78	3.65	4.76	6.17	7.99	10.30	13.30	17.20	22.30
6	1.26	1.74	2.38	3.21	4.28	5.64	7.35	9.46	12.00	15.10	18.70
7	1.06	1.55	2.25	3.24	4.63	6.60	9.37	13.30	19.10	27.70	41.20
8	0.86	1.31	1.97	2.91	4.23	6.06	8.53	11.80	15.90	21.10	27.20
9	0.73	1.17	1.87	2.95	4.63	7.24	11.30	17.90	29.00	49.20	91.80
10	0.60	1.00	1.66	2.69	4.27	6.64	10.10	14.80	21.20	29.00	38.20
11	0.51	0.90	1.58	2.74	4.72	8.12	14.10	25.00	47.30	104.00	399.00
12	0.42	0.78	1.42	2.52	4.37	7.37	12.00	18.70	27.80	38.90	50.90
13	0.36	0.70	1.35	2.58	4.87	9.23	17.80	36.40	86.50	393.00	ERROR
14	0.30	0.61	1.23	2.39	4.52	8.25	14.40	23.60	35.90	50.00	63.80
15	0.26	0.55	1.17	2.45	5.08	10.60	23.00	55.90	210.00	ERROR	ERROR

Table 7: Average percent error of 3rd degree approximation of $B = J_n - \delta A$, where B is a random $0-1$ matrix with exactly $\lfloor \frac{\delta}{n} \rfloor$ 0s in each row and column, taken for various δ and n, (taken over 100 trials) Note: The 0 terms in the table come from when δ is too small, so $\lfloor \frac{\delta}{n} \rfloor = 0$, and the approximation is unneeded because the matrix has only 1's in it.

	0.1	0.2	0.3	4	0.5	0.6	0.7	0.8
3	0.00E+00	0.00E+00	0.00E+00	1.04E+01	1.04E+01	1.04E+01	5.07E+01	5.07E+01
4	0.00E+00	0.00E+00	1.90E+00	1.90E+00	2.02E+01	2.09E+01	1.97E+01	6.85E+01
5	0.00E+00	3.31E-01	3.31E-01	3.57E+00	3.68E+00	2.63E+01	2.60E+01	7.48E+01
6	0.00E+00	4.79E-02	4.79E-02	5.45E-01	4.25E+00	3.30E+00	3.36E+01	3.39E+01
7	0.00E+00	6.06E-03	2.75E-01	2.97E-01	1.46E+00	4.30E+00	4.34E+00	4.60E+01
8	0.00E+00	6.80E-04	1.80E-01	7.79E-01	1.36E+00	1.19E+00	1.17E+01	7.60E+01
9	0.00E+00	7.00E-05	1.18E-01	5.09E-01	4.85E-01	6.56E+00	2.95E+01	1.51E+02
10	1.00E-05	8.17E-02	3.36E-01	3.01E-01	4.54E+00	1.95E+01	6.80E+01	2.78E+02
11	0.00E+00	5.78E-02	2.28E-01	2.21E-01	3.18E+00	1.35E+01	4.31E+01	1.30E+02
12	0.00E+00	4.25E-02	1.67E-01	1.88E-01	9.54E+00	2.88E+01	7.97E+01	2.45E+02
13	0.00E+00	3.30E-02	1.22E-01	1.82E+00	7.09E+00	2.04E+01	1.43E+02	4.67E+02
14	0.00E+00	2.54E-02	1.14E-01	1.41E+00	1.50E+01	3.75E+01	9.19E+01	8.59E+02
15	0.00E+00	7.38E-02	9.96E-02	4.18E+00	1.14E+01	6.34E+01	1.52E+02	1.62E+03

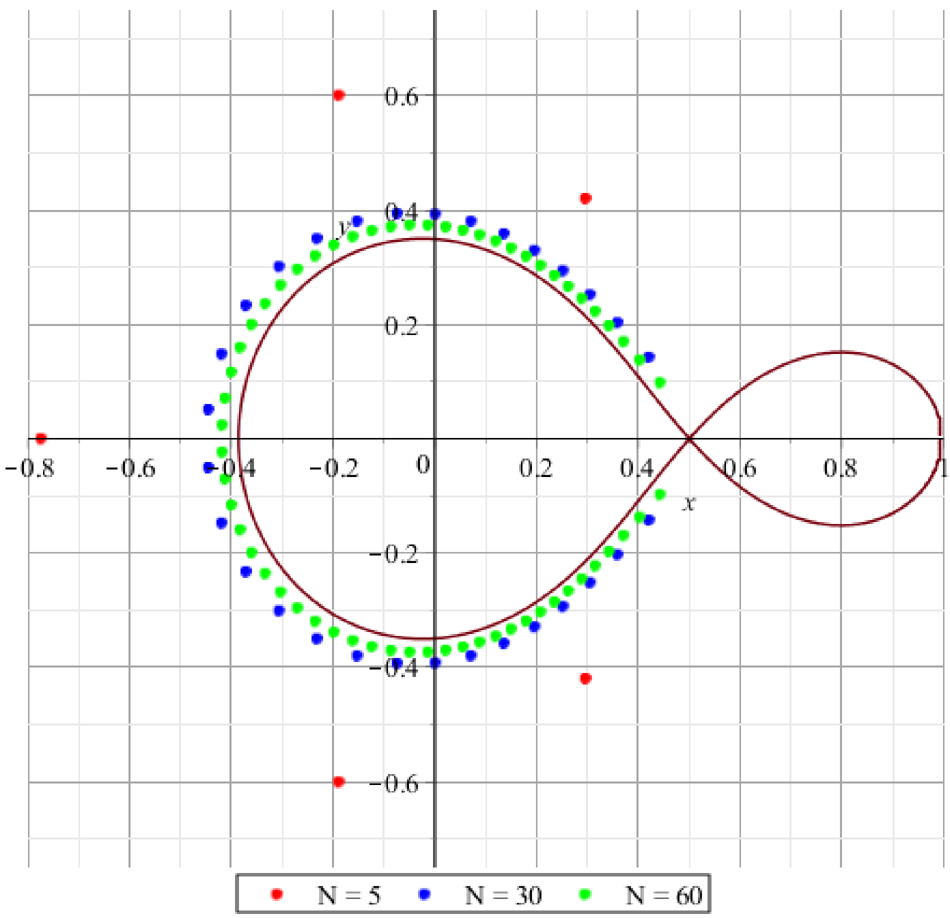
Table 8: Average percent error of 3rd degree approximation of $B = J_n - \delta A$, where B is a random $0-1$ matrix with exactly $\lfloor \frac{\delta}{n} \rfloor$ 0s in each row and column, taken for various δ and n, zoomed in around where the approximation begins to break down (taken over 100 trials) Note: The 0 terms in the table come from when δ is too small, so $\lfloor \frac{\delta}{n} \rfloor = 0$, and the approximation is unneeded because the matrix has only 1's in it.

	0.22	0.24	0.26	0.28	0.3	0.32	0.34	0.36	0.38
3	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	1.04E+01	1.04E+01	1.04E+01
4	0.00E+00	0.00E+00	1.90E+00	1.90E+00	1.90E+00	1.90E+00	1.90E+00	1.90E+00	1.90E+00
5	3.31E-01	3.31E-01	3.31E-01	3.31E-01	3.31E-01	3.31E-01	3.31E-01	3.31E-01	3.31E-01
6	4.79E-02	4.79E-02	4.79E-02	4.79E-02	4.79E-02	4.79E-02	6.42E-01	5.69E-01	4.47E-01
7	6.06E-03	6.06E-03	6.06E-03	6.06E-03	2.82E-01	3.08E-01	3.06E-01	2.91E-01	2.91E-01
8	6.80E-04	6.80E-04	1.81E-01	1.78E-01	1.81E-01	1.81E-01	1.80E-01	1.75E-01	8.25E-01
9	7.00E-05	1.16E-01	1.18E-01	1.19E-01	1.18E-01	1.16E-01	4.92E-01	4.86E-01	5.03E-01
10	8.09E-02	7.96E-02	7.93E-02	8.07E-02	3.32E-01	3.32E-01	3.38E-01	3.27E-01	3.32E-01
11	5.88E-02	5.80E-02	5.72E-02	2.31E-01	2.32E-01	2.25E-01	2.34E-01	2.22E-01	2.19E-01
12	4.34E-02	4.34E-02	1.71E-01	1.60E-01	1.69E-01	1.65E-01	1.67E-01	1.77E-01	1.82E-01
13	3.31E-02	1.24E-01	1.24E-01	1.24E-01	1.22E-01	1.50E-01	1.54E-01	1.44E-01	1.48E-01
14	9.37E-02	9.30E-02	9.38E-02	9.27E-02	1.21E-01	1.18E-01	1.20E-01	1.41E+00	1.40E+00
15	7.18E-02	7.44E-02	7.25E-02	9.79E-02	1.01E-01	9.92E-02	1.11E+00	1.11E+00	1.13E+00

Table 9: Average percent error of 3rd degree approximation of random complex matrices A whose entries were picked independently randomly such that $|1 - a_{ij}| < \delta$ for various n and δ.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
3	9.50E-04	1.70E-02	8.27E-02	2.68E-01	6.11E-01	1.41E+00	2.61E+00	4.57E+00	6.18E+00
4	7.82E-04	1.27E-02	5.96E-02	1.94E-01	5.51E-01	9.80E-01	2.22E+00	3.79E+00	6.79E+00
5	5.66E-04	1.03E-02	4.84E-02	1.34E-01	3.92E-01	8.55E-01	1.38E+00	2.44E+00	4.50E+00
6	4.98E-04	7.03E-03	3.93E-02	1.07E-01	2.90E-01	6.53E-01	1.22E+00	1.96E+00	3.52E+00
7	3.55E-04	5.63E-03	2.91E-02	9.74E-02	2.33E-01	5.32E-01	8.15E-01	1.49E+00	2.84E+00
8	2.82E-04	4.89E-03	2.41E-02	7.72E-02	2.02E-01	4.71E-01	7.60E-01	1.18E+00	2.33E+00
9	2.32E-04	4.16E-03	2.04E-02	6.28E-02	1.67E-01	3.00E-01	6.56E-01	1.20E+00	1.82E+00
10	2.06E-04	3.76E-03	1.92E-02	5.91E-02	1.23E-01	2.76E-01	5.66E-01	1.05E+00	1.54E+00
11	2.09E-04	3.40E-03	1.60E-02	5.02E-02	1.16E-01	2.23E-01	4.32E-01	7.37E-01	1.45E+00
12	1.85E-04	2.69E-03	1.24E-02	4.23E-02	1.12E-01	2.06E-01	4.37E-01	6.66E-01	1.31E+00
13	1.74E-04	2.59E-03	1.25E-02	4.47E-02	1.00E-01	1.94E-01	3.60E-01	6.58E-01	1.12E+00
14	1.38E-04	2.29E-03	1.15E-02	3.22E-02	7.84E-02	1.65E-01	3.67E-01	6.07E-01	9.19E-01

Figure 1: Roots of $q_{I_n}(z)$ converging to the transformed Szegő curve.



References

[1] A. Barvinok, *Approximating permanents and hafnians*, preprint, available at http://arxiv.org/abs/1601.07518 or at http://www.math.lsa.umich.edu/~barvinok/nozero.pdf

[2] S. Friedland, L. Gurvits, *Generalized Friedland-Tverberg inequality: applications and extensions*, available at https://arxiv.org/abs/math/0603410

[3] O.J. Heilmann and E.H. Lieb, *Theory of monomer-dimer systems*, Communications in Mathematical Physics **25** (1972), 190-232.

[4] J.D. Buckholtz, *A characterization of the exponential series*, American Mathematical Monthly **73** (1966), no. 4, part II, 121-123

[5] G. Szegő, *über eine eigenschaft der exponetialreihe*, SItzungste (1924).

[6] N. Linial, A. Samorodnitsky, and A. Wigderson, *A deterministic stronly polynomial algorithm for matrix scaling and approximate permanents*, Combinatorica **20** (2000), no. 4, 545-568

[7] H. Mink (1978). *Permanents* London: Addison-Wesley. 15 - 18.

[8] M. Marden, *Geometry of Polynomials. Second edition*, Mathematical Surveys, No. 3, American Mathematical Society, Providence, R.I., 1966.

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Figure 1 of the report

#This plots the roots of the truncated exponential series and transformed Szego curve
#as described in section 3.2 of the report

with(LinearAlgebra) :
with(plots) :

r :=proc(n, z)
local x :
x := n·z/(1 - z) :
sum(x^k/k!, k = 0 .. n)
end proc:

for n from 3 to 60 do
print(N = n) :
Solutions := evalf(solve(r(n, z) = 0)) :
Solutions := [Solutions] :
My_Plot := complexplot(Solutions, style = point) :
My_Curve := implicitplot(abs(eval((z/(1 - z))·exp(1 - (z/(1 - z))), z = x + I·y)) = 1, x
  = -1 .. 1, y = -1 .. 1, gridrefine = 3) :
print(display(My_Plot, My_Curve, gridlines = true, view = [-1..1, -1..1])) :
end do:
```

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Table 2 of the report
#This implements the approach layed out in section 2 to approximate
# 0-1 matrices whose entries are chosen independently randomly from {0,1} with probability
#p that a given entry a_{i,j} is a 0

#a note on indexing:
#everything is indexed such that the first index corresponds to the value for 1
#EXCEPT the symmetric polynomials, where the first index corresponds to e_0
#see the function comments for more details

restart :
with(LinearAlgebra) :

#global variables used to speed up computation of symmetric polynomials
power_sums :
known_polys :
poly_values :

#initializes global lists to be used in computing symmetric polynomials
initialize_global_lists := proc(x, m)
global power_sums, known_polys, poly_values :
power_sums := compute_power_sums(x, m) :
known_polys := [seq(false, i = 1 ..m)] :
poly_values := [seq(0, i = 1 ..m)] :
end proc:

#returns the kth symmetric polynomial of x
#requires 0 <= k <= m
get_symmetric_polynomial := proc(x, k)
local k_e_k, i, e_k_i :
global power_sums, known_polys, poly_values :
if k = 0 then
1
else
 k_e_k := 0 :
for i from 1 to k do
if k - i = 0 then
e_k_i := 1 :
else
if known_polys[k - i] = false then
known_polys[k - i] := true :
poly_values[k - i] := get_symmetric_polynomial(x, k - i) :
end if:
e_k_i := poly_values[k - i] :
end if:
k_e_k := k_e_k + (-1)^(i - 1)·e_k_i·power_sums[i] :
end do:
```

```
 k_e_k
 -----
   k
end if:
end proc:

#returns a list of the first 1 -> m power sums of x
compute_power_sums :=proc(x, m)
local list_of_power_sums, k, i, p_k :
list_of_power_sums := [ ] :
for k from 1 to m do
p_k := 0 :
for i from 1 to numelems(x) do
p_k := p_k + x[i]^k :
end do:
list_of_power_sums := [op(list_of_power_sums), p_k] :
end do:
list_of_power_sums
end proc:

#computes the first 0 -> m symmetric polynomials of the vector x
#IMPORTANT: this returns the list of the 0th through mth symmetric polynomial
#so if using non-computer science indexing, to get the mth symmetric polynomial
#you need to access index m + 1
compute_all_symmetric_polys :=proc(x, m)
local final_poly_values, i :
initialize_global_lists(x, m) :
final_poly_values := [1] :
for i from 1 to m do
final_poly_values := [op(final_poly_values), get_symmetric_polynomial(x, i) ] :
end do:
final_poly_values
end proc:

#computes d_n, which is the unique solutions of the system of linear equations
#d_n·A = (-1,...,-1) (m-1 negative ones), where A is the lower triangular matrix
#defined in the code
d :=proc(n, m)
local A, row, col, negative_one_vector :
A := Matrix(m − 1, m − 1) :
for row from 1 to m − 1 do
for col from 1 to m − 1 do
if row ≥ col then
A[row, col] := binomial(n − col, row − col) :
else
A[row, col] := 0 :
end if:
end do:
end do:
negative_one_vector := Vector[column](m − 1, -1) :
LinearSolve(Transpose(A), negative_one_vector)
end proc:
```

```
#Requires 1 <= i <= 3
#Returns a list of all 0-1 vectors with i ones out of n arguments of the vector
get_valid_vectors := proc(i, n)
local valid_vectors, loop_i, loop_j, loop_k, x :
valid_vectors := [ ] :
if i = 1 then
for loop_i from 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
elif i = 2 then
 for loop_i from 1 to n - 1 do
for loop_j from loop_i + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
else
 for loop_i from 1 to n - 2 do
for loop_j from loop_i + 1 to n - 1 do
for loop_k from loop_j + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
x[loop_k] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
end do:
end if:
valid_vectors
end proc:

s := proc(i, n, A, m)
local valid_vectors, total, arg_of_S_m, j :
valid_vectors := get_valid_vectors(i, n) :
total := 0 :
for j from 1 to numelems(valid_vectors) do
arg_of_S_m := Multiply(A, valid_vectors[j]) :
total := total + compute_all_symmetric_polys(arg_of_S_m, m)[m + 1] :
end do:
total
end proc:

#computes perm_m(A)
#requires m ≤ 4
perm_m := proc(A, m, n)
local ones, total, j, values_of_d_n :
ones := Vector(n, 1) :
```

```
total := compute_all_symmetric_polys(Multiply(A, ones), m)[m + 1] :
values_of_d_n := d(n, m) :
for j from 1 to m - 1 do:
total := total + (s(j, n, A, m)·values_of_d_n[j]) :
end do:
total
end proc:

g := proc(m :: integer, n, A)
local J_n :
J_n := Matrix(n, n, 1) :
evalf((n − m)!· perm_m(A − J_n, m, n)·m!)
end proc:

get_f_solutions := proc(m, n, A)
local g_values, lower_triangular_B, row, col, right_hand_side, i, Solutions :
g_values := [n!] :
for i from 1 to m do
g_values := [op(g_values), g(i, n, A)] :
end do:
lower_triangular_B := Matrix(m, m, 0) :
for row from 1 to m do
for col from 1 to row do
lower_triangular_B[row, col] := g_values[row - col + 1]·binomial(row − 1, row − col) :
end do:
end do:
right_hand_side := Matrix(m, 1, 0) :
for row from 1 to m do
right_hand_side[row] := g_values[row + 1] :
end do:
LinearSolve(lower_triangular_B, right_hand_side)
end proc:

p := proc(m, n, A)
local f_values, sum, i :
f_values := get_f_solutions(m, n, A) :
sum := 0 :
for i from 1 to m do
sum := sum + f_values(i)/i! :
end do:
evalf(ln(n!) + sum)
end proc:

make_matrix := proc(n, prob)
local M, row, col :
M := RandomMatrix(n, n, generator = 0 ..1.0) :
for row from 1 to n do
for col from 1 to n do
if (M[row, col] < prob) then
M[row, col] := 0 :
else
```

```
M[row, col] := 1 :
end if:
end do:
end do:
M
end proc:

#MAIN

n_min := 3 :
n_max := 15 :
n_increment := 1 :
m := 3 :
prob_min := 0.1 :
prob_max := 0.5 :
prob_increment := 0.1 :
num_trial := 100 :
num_col := ceil((prob_max - prob_min)/prob_increment + 1) :
num_row := ceil((n_max - n_min)/n_increment + 1) :
dataMatrix := Matrix(num_row, num_col) :
row := 1 :
for n from n_min by n_increment to n_max do
col := 1 :
for prob from prob_min by prob_increment to prob_max do
total_percent_error := 0 :
successful_trial := 0 :
for i from 1 to num_trial do
A := make_matrix(n, prob) :
perm_A := evalf(Permanent(A)) :
if perm_A ≠ 0  then
successful_trial := successful_trial + 1 :
diff_from_real := evalf(abs(exp(p(m, n, A)) - perm_A)) :
total_percent_error := total_percent_error + evalf(abs(diff_from_real/perm_A)·100) :
end if:
end do:
if successful_trial ≠ 0 then
percent_error := total_percent_error/successful_trial :
dataMatrix[row, col] := parse(sprintf("%.4f", percent_error)) :
else
dataMatrix := Perm_error :
print('``") :
end if:
col := col + 1 :
end do:
row := row + 1 :
end do:
print(dataMatrix) :
```

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Table 1 of the report
#This implements the approach layed out in section 2 to approximate
# matrices whose entries are chosen independently randomly from (0, 2)


#a note on indexing:
#everything is indexed such that the first index corresponds to the value for 1
#EXCEPT the symmetric polynomials, where the first index corresponds to e_0
#see the function comments for more details

restart :
with(LinearAlgebra) :

#global variables used to speed up computation of symmetric polynomials
power_sums :
known_polys :
poly_values :

#initializes global lists to be used in computing symmetric polynomials
initialize_global_lists := proc(x, m)
global power_sums, known_polys, poly_values :
power_sums := compute_power_sums(x, m) :
known_polys := [seq(false, i = 1 ..m)] :
poly_values := [seq(0, i = 1 ..m)] :
end proc:

#returns the kth symmetric polynomial of x
#requires 0 <= k <= m
get_symmetric_polynomial := proc(x, k)
local k_e_k, i, e_k_i :
global power_sums, known_polys, poly_values :
if k = 0 then
1
else
k_e_k := 0 :
for i from 1 to k do
if k − i = 0 then
e_k_i := 1 :
else
if known_polys[k − i] = false then
known_polys[k − i] := true :
poly_values[k − i] := get_symmetric_polynomial(x, k − i) :
end if:
e_k_i := poly_values[k − i] :
end if:
k_e_k := k_e_k + (-1)^(i − 1)·e_k_i·power_sums[i] :
end do:
```

```
  k_e_k
  -----
    k
end if:
end proc:

#returns a list of the first 1 -> m power sums of x
compute_power_sums := proc(x, m)
local list_of_power_sums, k, i, p_k :
list_of_power_sums := [ ] :
for k from 1 to m do
p_k := 0 :
for i from 1 to numelems(x) do
p_k := p_k + x[i]^k :
end do:
list_of_power_sums := [op(list_of_power_sums), p_k] :
end do:
list_of_power_sums
end proc:

#computes the first 0 -> m symmetric polynomials of the vector x
#IMPORTANT: this returns the list of the 0th through mth symmetric polynomial
#so if using non-computer science indexing, to get the mth symmetric polynomial
#you need to access index m + 1
compute_all_symmetric_polys := proc(x, m)
local final_poly_values, i :
initialize_global_lists(x, m) :
final_poly_values := [1] :
for i from 1 to m do
final_poly_values := [op(final_poly_values), get_symmetric_polynomial(x, i)] :
end do:
final_poly_values
end proc:

#computes d_n, which is the unique solutions of the system of linear equations
#d_n·A = (-1,...,-1) (m-1 negative ones), where A is the lower triangular matrix
#defined in the code
d := proc(n, m)
local A, row, col, negative_one_vector :
A := Matrix(m - 1, m - 1) :
for row from 1 to m - 1 do
for col from 1 to m - 1 do
if row ≥ col then
A[row, col] := binomial(n - col, row - col) :
else
A[row, col] := 0 :
end if:
end do:
end do:
negative_one_vector := Vector[column](m - 1, -1) :
LinearSolve(Transpose(A), negative_one_vector)
end proc:
```

```
#Requires 1 <= i <= 3
#Returns a list of all 0-1 vectors with i ones out of n arguments of the vector
get_valid_vectors := proc(i, n)
local valid_vectors, loop_i, loop_j, loop_k, x :
valid_vectors := [ ] :
if i = 1 then
for loop_i from 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
elif i = 2 then
 for loop_i from 1 to n - 1 do
for loop_j from loop_i + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
else
 for loop_i from 1 to n - 2 do
for loop_j from loop_i + 1 to n - 1 do
for loop_k from loop_j + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
x[loop_k] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
end do:
end if:
valid_vectors
end proc:

s := proc(i, n, A, m)
local valid_vectors, total, arg_of_S_m, j :
valid_vectors := get_valid_vectors(i, n) :
total := 0 :
for j from 1 to numelems(valid_vectors) do
arg_of_S_m := Multiply(A, valid_vectors[j]) :
total := total + compute_all_symmetric_polys(arg_of_S_m, m)[m + 1] :
end do:
total
end proc:

#computes perm_m(A)
#requires m ≤ 4
perm_m := proc(A, m, n)
local ones, total, j, values_of_d_n :
ones := Vector(n, 1) :
```

```
total := compute_all_symmetric_polys(Multiply(A, ones), m)[m + 1] :
values_of_d_n := d(n, m) :
for j from 1 to m - 1 do:
total := total + (s(j, n, A, m)·values_of_d_n[j]) :
end do:
total
end proc:

g := proc(m :: integer, n, A)
local J_n :
J_n := Matrix(n, n, 1) :
evalf((n − m)!· perm_m(A − J_n, m, n)·m!)
end proc:

get_f_solutions := proc(m, n, A)
local g_values, lower_triangular_B, row, col, right_hand_side, i, Solutions :
g_values := [n!] :
for i from 1 to m do
g_values := [op(g_values), g(i, n, A)] :
end do:
lower_triangular_B := Matrix(m, m, 0) :
for row from 1 to m do
for col from 1 to row do
lower_triangular_B[row, col] := g_values[row - col + 1]·binomial(row − 1, row − col) :
end do:
end do:
right_hand_side := Matrix(m, 1, 0) :
for row from 1 to m do
right_hand_side[row] := g_values[row + 1] :
end do:
LinearSolve(lower_triangular_B, right_hand_side)
end proc:

p := proc(m, n, A)
local f_values, sum, i :
f_values := get_f_solutions(m, n, A) :
sum := 0 :
for i from 1 to m do
sum := sum + f_values(i)/i! :
end do:
evalf(ln(n!) + sum)
end proc:

make_matrix := proc(n, delta)
RandomMatrix(n, n, generator = 1 − delta..1 + delta)
end proc:

#MAIN

n_min := 3 :
n_max := 15 :
```

```
n_increment := 1 :
delta_min := 0.1 :
delta_max := 0.9 :
delta_increment := 0.1 :
row_dim := ceil((n_max - n_min)/n_increment + 1) :
col_dim := ceil((delta_max - delta_min)/delta_increment + 1) :
m := 3 :
num_trial := 100 :
data_matrix := Matrix(row_dim, col_dim) :
row := 1 :
for n from n_min by n_increment to n_max do
col := 1 :
for delta from delta_min by delta_increment to delta_max do
total_percent_error := 0 :
successful_trial := 0 :
for i from 1 to num_trial do
A := make_matrix(n, delta) :
perm_A := evalf(Permanent(A)) :
if perm_A > 0 then
successful_trial := successful_trial + 1 :
diff_from_real := evalf(abs(exp(p(m, n, A)) - perm_A)) :
total_percent_error := total_percent_error + evalf(abs(diff_from_real/perm_A)·100) :
end if:
end do:
percent_error := parse(sprintf("%.7f", total_percent_error/successful_trial)) :
data_matrix[row, col] := percent_error :
col := col + 1 :
end do:
row := row + 1 :
end do:
print(data_matrix) :
```

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Table 3 of the report
#This implements the approach layed out in section 2 to approximate
# matrices whose entries are chosen independently randomly from (0, 2)
#this times how long the code takes to run for matrices up to size 200 x 200


#a note on indexing:
#everything is indexed such that the first index corresponds to the value for 1
#EXCEPT the symmetric polynomials, where the first index corresponds to e_0
#see the function comments for more details

restart :
with(LinearAlgebra) :

#global variables used to speed up computation of symmetric polynomials
power_sums :
known_polys :
poly_values :

#initializes global lists to be used in computing symmetric polynomials
initialize_global_lists :=proc(x, m)
global power_sums, known_polys, poly_values :
power_sums := compute_power_sums(x, m) :
known_polys := [seq(false, i = 1..m)] :
poly_values := [seq(0, i = 1..m)] :
end proc:

#returns the kth symmetric polynomial of x
#requires 0 <= k <= m
get_symmetric_polynomial :=proc(x, k)
local k_e_k, i, e_k_i :
global power_sums, known_polys, poly_values :
if k = 0 then
1
else
k_e_k := 0 :
for i from 1 to k do
if k − i = 0 then
e_k_i := 1 :
else
if known_polys[k − i] = false then
known_polys[k − i] := true :
poly_values[k − i] := get_symmetric_polynomial(x, k − i) :
end if:
```

```
e_k_i := poly_values[k - i] :
end if:
k_e_k := k_e_k + (-1)^(i - 1)·e_k_i·power_sums[i] :
end do:
k_e_k / k
end if:
end proc:

#returns a list of the first 1 -> m power sums of x
compute_power_sums :=proc(x, m)
local list_of_power_sums, k, i, p_k :
list_of_power_sums := [ ] :
for k from 1 to m do
p_k := 0 :
for i from 1 to numelems(x) do
p_k := p_k + x[i]^k :
end do:
list_of_power_sums := [op(list_of_power_sums), p_k] :
end do:
list_of_power_sums
end proc:

#computes the first 0 -> m symmetric polynomials of the vector x
#IMPORTANT: this returns the list of the 0th through mth symmetric polynomial
#so if using non-computer science indexing, to get the mth symmetric polynomial
#you need to access index m + 1
compute_all_symmetric_polys :=proc(x, m)
local final_poly_values, i :
initialize_global_lists(x, m) :
final_poly_values := [1] :
for i from 1 to m do
final_poly_values := [op(final_poly_values), get_symmetric_polynomial(x, i)] :
end do:
final_poly_values
end proc:

#computes d_n, which is the unique solutions of the system of linear equations
#d_n·A = (-1,...,-1) (m-1 negative ones), where A is the lower triangular matrix
#defined in the code
d :=proc(n, m)
local A, row, col, negative_one_vector :
A := Matrix(m - 1, m - 1) :
for row from 1 to m - 1 do
for col from 1 to m - 1 do
if row ≥ col then
A[row, col] := binomial(n - col, row - col) :
else
```

```
A[row, col] := 0 :
end if:
end do:
end do:
negative_one_vector := Vector[column](m − 1, -1) :
LinearSolve(Transpose(A), negative_one_vector)
end proc:

#Requires 1 <= i <= 3
#Returns a list of all 0-1 vectors with i ones out of n arguments of the vector
get_valid_vectors :=proc(i, n)
local valid_vectors, loop_i, loop_j, loop_k, x :
valid_vectors := [ ] :
if i = 1 then
for loop_i from 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
elif i = 2 then
 for loop_i from 1 to n − 1 do
for loop_j from loop_i + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
else
 for loop_i from 1 to n − 2 do
for loop_j from loop_i + 1 to n − 1 do
for loop_k from loop_j + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
x[loop_k] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
end do:
end if:
valid_vectors
end proc:

s :=proc(i, n, A, m)
local valid_vectors, total, arg_of_S_m, j :
valid_vectors := get_valid_vectors(i, n) :
total := 0 :
```

```
for j from 1 to numelems(valid_vectors) do
arg_of_S_m := Multiply(A, valid_vectors[j]) :
total := total + compute_all_symmetric_polys(arg_of_S_m, m)[m + 1] :
end do:
total
end proc:

#computes perm_m(A)
#requires m ≤ 4
perm_m :=proc(A, m, n)
local ones, total, j, values_of_d_n :
ones := Vector(n, 1) :
total := compute_all_symmetric_polys(Multiply(A, ones), m)[m + 1] :
values_of_d_n := d(n, m) :
for j from 1 to m - 1 do:
total := total + (s(j, n, A, m)·values_of_d_n[j]) :
end do:
total
end proc:

g := proc(m :: integer, n, A)
local  J_n :
J_n := Matrix(n, n, 1) :
evalf((n − m)!· perm_m(A − J_n, m, n)·m!)
end proc:

get_f_solutions := proc(m, n, A)
local g_values, lower_triangular_B, row, col, right_hand_side, i, Solutions :
g_values := [n!] :
for i from 1 to m do
g_values := [op(g_values), g(i, n, A)] :
end do:
lower_triangular_B := Matrix(m, m, 0) :
for row from 1 to m do
for col from 1 to row do
lower_triangular_B[row, col] := g_values[row - col + 1]·binomial(row − 1, row
   − col) :
end do:
end do:
right_hand_side := Matrix(m, 1, 0) :
for row from 1 to m do
right_hand_side[row] := g_values[row + 1] :
end do:
LinearSolve(lower_triangular_B, right_hand_side)
end proc:

p := proc(m, n, A)
local f_values, sum, i :
```

```
f_values := get_f_solutions(m, n, A) :
sum := 0 :
for i from 1 to m do
sum := sum + f_values(i)/i! :
end do:
evalf(ln(n!) + sum)
end proc:

make_matrix :=proc(n)
local M :
M := RandomMatrix(n, n, generator = 0..2.0) :
M
end proc:
n_min := 100 :
n_max := 200 :
n_increment := 10 :
num_trials := 10 :
dataMatrix := Matrix(1, (n_max - n_min)/n_increment + 1) :
for n from n_min by n_increment to n_max do
total_time := 0 :
for i from 1 to num_trials do
A := make_matrix(n) :
total_time := total_time + time(evalf(p(3, n, A))) :
end do:
dataMatrix[1, (n - n_min)/n_increment + 1] := total_time/num_trials :
end do:
print(dataMatrix) :
```

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Tables 5 and 6 of the report
#This implements the approach layed out in section 4
# to approximate matrices of the form J_n - delta·I_n
#for various delta and where n is the size of the matrix

#a note on indexing:
#everything is indexed such that the first index corresponds to the value for 1
#EXCEPT the symmetric polynomials, where the first index corresponds to e_0
#see the function comments for more details

restart :
with(LinearAlgebra) :

#global variables used to speed up computation of symmetric polynomials
power_sums :
known_polys :
poly_values :

#initializes global lists to be used in computing symmetric polynomials
initialize_global_lists :=proc(x, m)
global power_sums, known_polys, poly_values :
power_sums := compute_power_sums(x, m) :
known_polys := [seq(false, i = 1..m)] :
poly_values := [seq(0, i = 1..m)] :
end proc:

#returns the kth symmetric polynomial of x
#requires 0 <= k <= m
get_symmetric_polynomial :=proc(x, k)
local k_e_k, i, e_k_i :
global power_sums, known_polys, poly_values :
if k = 0 then
1
else
k_e_k := 0 :
for i from 1 to k do
if k − i = 0 then
e_k_i := 1 :
else
if known_polys[k − i] = false then
known_polys[k − i] := true :
poly_values[k − i] := get_symmetric_polynomial(x, k − i) :
end if:
e_k_i := poly_values[k − i] :
```

```
end if:
k_e_k := k_e_k + (-1)^(i - 1)·e_k_i·power_sums[i] :
end do:
k_e_k
-----
  k
end if:
end proc:

#returns a list of the first 1 -> m power sums of x
compute_power_sums :=proc(x, m)
local list_of_power_sums, k, i, p_k :
list_of_power_sums := [ ] :
for k from 1 to m do
p_k := 0 :
for i from 1 to numelems(x) do
p_k := p_k + x[i]^k :
end do:
list_of_power_sums := [op(list_of_power_sums), p_k] :
end do:
list_of_power_sums
end proc:

#computes the first 0 -> m symmetric polynomials of the vector x
#IMPORTANT: this returns the list of the 0th through mth symmetric polynomial
#so if using non-computer science indexing, to get the mth symmetric polynomial
#you need to access index m + 1
compute_all_symmetric_polys :=proc(x, m)
local final_poly_values, i :
initialize_global_lists(x, m) :
final_poly_values := [1] :
for i from 1 to m do
final_poly_values := [op(final_poly_values), get_symmetric_polynomial(x, i)] :
end do:
final_poly_values
end proc:

#computes d_n, which is the unique solutions of the system of linear equations
#d_n·A = (-1,...,-1) (m-1 negative ones), where A is the lower triangular matrix
#defined in the code
d :=proc(n, m)
local A, row, col, negative_one_vector :
A := Matrix(m - 1, m - 1) :
for row from 1 to m - 1 do
for col from 1 to m - 1 do
if row ≥ col then
A[row, col] := binomial(n - col, row - col) :
else
A[row, col] := 0 :
```

```
end if:
end do:
end do:
negative_one_vector := Vector[column](m − 1, -1) :
LinearSolve(Transpose(A), negative_one_vector)
end proc:

#Requires 1 <= i <= 3
#Returns a list of all 0-1 vectors with i ones out of n arguments of the vector
get_valid_vectors :=proc(i, n)
local valid_vectors, loop_i, loop_j, loop_k, x :
valid_vectors := [ ] :
if i = 1 then
for loop_i from 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
elif i = 2 then
 for loop_i from 1 to n − 1 do
for loop_j from loop_i + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
else
 for loop_i from 1 to n − 2 do
for loop_j from loop_i + 1 to n − 1 do
for loop_k from loop_j + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
x[loop_k] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
end do:
end if:
valid_vectors
end proc:

s :=proc(i, n, A, m)
local valid_vectors, total, arg_of_S_m, j :
valid_vectors := get_valid_vectors(i, n) :
total := 0 :
for j from 1 to numelems(valid_vectors) do
```

```
arg_of_S_m := Multiply(A, valid_vectors[j]) :
total := total + compute_all_symmetric_polys(arg_of_S_m, m)[m + 1] :
end do:
total
end proc:

#computes perm_m(A)
#requires m ≤ 4
perm_m :=proc(A, m, n)
local ones, total, j, values_of_d_n :
ones := Vector(n, 1) :
total := compute_all_symmetric_polys(Multiply(A, ones), m)[m + 1] :
values_of_d_n := d(n, m) :
for j from 1 to m - 1 do:
total := total + (s(j, n, A, m)·values_of_d_n[j]) :
end do:
total
end proc:

g := proc(m :: integer, n, delta, A)
evalf( m!· (-delta)^m·perm_m(A, m, n)·(n − m)! / n^(n − m) )
end proc:

get_f_solutions := proc(m, n, delta, A)
local g_values, lower_triangular_B, row, col, right_hand_side, i, Solutions :
g_values := [ n!/n^n ] :
for i from 1 to m do
g_values := [op(g_values), g(i, n, delta, A)] :
end do:
lower_triangular_B := Matrix(m, m, 0) :
for row from 1 to m do
for col from 1 to row do
lower_triangular_B[row, col] := g_values[row - col + 1]·binomial(row − 1, row
   − col) :
end do:
end do:
right_hand_side := Matrix(m, 1, 0) :
for row from 1 to m do
right_hand_side[row] := g_values[row + 1] :
end do:
LinearSolve(lower_triangular_B, right_hand_side)
end proc:

p := proc(m, n, delta, A)
local f_values, sum, i :
f_values := get_f_solutions(m, n, delta, A) :
```

```
sum := 0 :
for i from 1 to m do
sum := sum + f_values(i)/i! :
end do:
evalf(ln(n!/n^n) + sum)
end proc:

#MAIN

n_min := 3 :
n_max := 15 :
n_increment := 1 :
m := 3 :
delta_min := 0.05 :
delta_max := 0.5 :
delta_increment := 0.05 :
num_col := ceil((delta_max - delta_min)/delta_increment + 1) :
num_row := ceil((n_max - n_min)/n_increment + 1) :
dataMatrix := Matrix(num_row, num_col) :
for n from n_min by n_increment to n_max do
for delta from delta_min by delta_increment to delta_max do
col := ceil((delta - delta_min)/delta_increment + 1) :
row := n - n_min + 1 :
I_n := IdentityMatrix(n, n) :
J_n := Matrix(n, n, 1) :
M := 1/n·J_n - delta·I_n :
perm_M := evalf(Permanent(M)) :
if perm_M > 0 then
diff_from_real := evalf(abs(exp(p(m, n, delta, I_n)) - perm_M)) :
percent_error := evalf(abs(diff_from_real/perm_M)·100) :
dataMatrix[row, col] := parse(sprintf("%.10f", percent_error)) :
else
dataMatrix[row, col] := Perm_error :
end if:
end do:
end do:
print(dataMatrix) :
```

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Table 4 of the report
#This implements the approach layed out in section 4 to approximate
# matrices whose entries are chosen independently randomly from (0, 1/n)
#where n is the size of the matrix


#a note on indexing:
#everything is indexed such that the first index corresponds to the value for 1
#EXCEPT the symmetric polynomials, where the first index corresponds to e_0
#see the function comments for more details

restart :
with(LinearAlgebra) :

#global variables used to speed up computation of symmetric polynomials
power_sums :
known_polys :
poly_values :

#initializes global lists to be used in computing symmetric polynomials
initialize_global_lists := proc(x, m)
global power_sums, known_polys, poly_values :
power_sums := compute_power_sums(x, m) :
known_polys := [seq(false, i = 1 ..m) ] :
poly_values := [seq(0, i = 1 ..m) ] :
end proc:

#returns the kth symmetric polynomial of x
#requires 0 <= k <= m
get_symmetric_polynomial := proc(x, k)
local k_e_k, i, e_k_i :
global power_sums, known_polys, poly_values :
if k = 0 then
1
else
k_e_k := 0 :
for i from 1 to k do
if k − i = 0 then
e_k_i := 1 :
else
if known_polys[k − i] = false then
known_polys[k − i] := true :
poly_values[k − i] := get_symmetric_polynomial(x, k − i) :
end if:
e_k_i := poly_values[k − i] :
end if:
```

```
k_e_k := k_e_k + (-1)^(i - 1)·e_k_i·power_sums[i] :
end do:
k_e_k
-----
  k
end if:
end proc:

#returns a list of the first 1 -> m power sums of x
compute_power_sums :=proc(x, m)
local list_of_power_sums, k, i, p_k :
list_of_power_sums := [ ] :
for k from 1 to m do
p_k := 0 :
for i from 1 to numelems(x) do
p_k := p_k + x[i]^k :
end do:
list_of_power_sums := [op(list_of_power_sums), p_k] :
end do:
list_of_power_sums
end proc:

#computes the first 0 -> m symmetric polynomials of the vector x
#IMPORTANT: this returns the list of the 0th through mth symmetric polynomial
#so if using non-computer science indexing, to get the mth symmetric polynomial
#you need to access index m + 1
compute_all_symmetric_polys :=proc(x, m)
local final_poly_values, i :
initialize_global_lists(x, m) :
final_poly_values := [1] :
for i from 1 to m do
final_poly_values := [op(final_poly_values), get_symmetric_polynomial(x, i) ] :
end do:
final_poly_values
end proc:

#computes d_n, which is the unique solutions of the system of linear equations
#d_n·A = (-1,...,-1) (m-1 negative ones), where A is the lower triangular matrix
#defined in the code
d :=proc(n, m)
local A, row, col, negative_one_vector :
A := Matrix(m - 1, m - 1) :
for row from 1 to m - 1 do
for col from 1 to m - 1 do
if row ≥ col then
A[row, col] := binomial(n - col, row - col) :
else
A[row, col] := 0 :
end if:
end do:
end do:
negative_one_vector := Vector[column](m - 1, -1) :
LinearSolve(Transpose(A), negative_one_vector)
```

```
end proc:

#Requires 1 <= i <= 3
#Returns a list of all 0-1 vectors with i ones out of n arguments of the vector
get_valid_vectors :=proc(i, n)
local valid_vectors, loop_i, loop_j, loop_k, x :
valid_vectors := [ ] :
if i = 1 then
for loop_i from 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
elif i = 2 then
 for loop_i from 1 to n − 1 do
for loop_j from loop_i + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
else
 for loop_i from 1 to n − 2 do
for loop_j from loop_i + 1 to n − 1 do
for loop_k from loop_j + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
x[loop_k] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
end do:
end if:
valid_vectors
end proc:

s :=proc(i, n, A, m)
local valid_vectors, total, arg_of_S_m, j :
valid_vectors := get_valid_vectors(i, n) :
total := 0 :
for j from 1 to numelems(valid_vectors) do
arg_of_S_m := Multiply(A, valid_vectors[j]) :
total := total + compute_all_symmetric_polys(arg_of_S_m, m)[m + 1] :
end do:
total
end proc:

#computes perm_m(A)
#requires m ≤ 4
perm_m :=proc(A, m, n)
```

```
local ones, total, j, values_of_d_n :
ones := Vector(n, 1) :
total := compute_all_symmetric_polys(Multiply(A, ones), m)[m + 1] :
values_of_d_n := d(n, m) :
for j from 1 to m - 1 do:
total := total + (s(j, n, A, m)·values_of_d_n[j]) :
end do:
total
end proc:

g := proc(m :: integer, n, A, delta)
local J_n :
J_n := Matrix(n, n, 1) :
evalf( (-delta)^m · ((n - m)!· perm_m(A, m, n)·m!) / n^(n - m) )
end proc:

get_f_solutions := proc(m, n, A, delta)
local g_values, lower_triangular_B, row, col, right_hand_side, i, Solutions :
g_values := [ n! / n^n ]:
for i from 1 to m do
g_values := [op(g_values), g(i, n, A, delta)] :
end do:
lower_triangular_B := Matrix(m, m, 0) :
for row from 1 to m do
for col from 1 to row do
lower_triangular_B[row, col] := g_values[row - col + 1]·binomial(row - 1, row - col) :
end do:
end do:
right_hand_side := Matrix(m, 1, 0) :
for row from 1 to m do
right_hand_side[row] := g_values[row + 1] :
end do:
LinearSolve(lower_triangular_B, right_hand_side)
end proc:

p := proc(m, n, A, delta)
local f_values, sum, i :
f_values := get_f_solutions(m, n, A, delta) :
sum := 0 :
for i from 1 to m do
sum := sum + f_values(i) / i! :
end do:
evalf( ln( n! / n^n ) + sum )
end proc:

make_matrix := proc(n)
local A :
```

```
RandomMatrix(n, n, generator = 0 .. 1/n)
end proc:

#MAIN

n_min := 3 :
n_max := 15 :
n_increment := 1 :
delta_min := 0.05 :
delta_max := 1 :
delta_increment := 0.05 :
row_dim := ceil((n_max - n_min)/n_increment + 1) :
col_dim := ceil((delta_max - delta_min)/delta_increment + 1) :
m := 3 :
num_trial := 100 :
data_matrix := Matrix(row_dim, col_dim) :
row := 1 :
for n from n_min by n_increment to n_max do
col := 1 :
for delta from delta_min by delta_increment to delta_max do:
total_percent_error := 0 :
successful_trial := 0 :
for i from 1 to num_trial do
A := make_matrix(n) :
perm_M := evalf(Permanent(Matrix(n, n, 1/n) - delta·A)) :
if perm_M > 0 then
successful_trial := successful_trial + 1 :
real_value := evalf(perm_M) :
diff_from_real := evalf(abs(exp(p(m, n, A, delta)) - real_value)) :
total_percent_error := total_percent_error + evalf(abs(diff_from_real/real_value)·100) :
end if:
end do:
percent_error := parse(sprintf("%.4f", total_percent_error/successful_trial)) :
data_matrix[row, col] := percent_error :
col := col + 1 :
end do:
row := row + 1 :
end do:
print(data_matrix) :
```

```
#REU PROJECT ON THE COMPLEX ROOTS AND APPROXIMATION OF PERMANENTS
#Max Kontorovich and Han Wu
#Advisor: Professor Alexander Barvinok

#Code used to produce Tables 7 and 8 of the report
#This implements the approach layed out in the end of section 4 to approximate
# 0-1 matrices with exactly than floor(delta/n) zeros in every row and column
#for various delta and n, where n is the size of the matrix and
#delta is the t defined in the end of section 4


#a note on indexing:
#everything is indexed such that the first index corresponds to the value for 1
#EXCEPT the symmetric polynomials, where the first index corresponds to e_0
#see the function comments for more details

restart :
with(LinearAlgebra) :
with(combinat) :
#global variables used to speed up computation of symmetric polynomials
power_sums :
known_polys :
poly_values :

#initializes global lists to be used in computing symmetric polynomials
initialize_global_lists :=proc(x, m)
global power_sums, known_polys, poly_values :
power_sums := compute_power_sums(x, m) :
known_polys := [seq(false, i = 1..m)] :
poly_values := [seq(0, i = 1..m)] :
end proc:

#returns the kth symmetric polynomial of x
#requires 0 <= k <= m
get_symmetric_polynomial :=proc(x, k)
local k_e_k, i, e_k_i :
global power_sums, known_polys, poly_values :
if k = 0 then
1
else
k_e_k := 0 :
for i from 1 to k do
if k - i = 0 then
e_k_i := 1 :
else
if known_polys[k - i] = false then
known_polys[k - i] := true :
```

```
poly_values[k − i] := get_symmetric_polynomial(x, k − i) :
end if:
e_k_i := poly_values[k − i] :
end if:
k_e_k := k_e_k + (-1)^(i − 1)·e_k_i·power_sums[i] :
end do:
k_e_k / k
end if:
end proc:

#returns a list of the first 1 -> m power sums of x
compute_power_sums :=proc(x, m)
local list_of_power_sums, k, i, p_k :
list_of_power_sums := [ ] :
for k from 1 to m do
p_k := 0 :
for i from 1 to numelems(x) do
p_k := p_k + x[i]^k :
end do:
list_of_power_sums := [op(list_of_power_sums), p_k] :
end do:
list_of_power_sums
end proc:

#computes the first 0 -> m symmetric polynomials of the vector x
#IMPORTANT: this returns the list of the 0th through mth symmetric polynomial
#so if using non-computer science indexing, to get the mth symmetric polynomial
#you need to access index m + 1
compute_all_symmetric_polys :=proc(x, m)
local final_poly_values, i :
initialize_global_lists(x, m) :
final_poly_values := [1] :
for i from 1 to m do
final_poly_values := [op(final_poly_values), get_symmetric_polynomial(x, i)] :
end do:
final_poly_values
end proc:

#computes d_n, which is the unique solutions of the system of linear equations
#d_n·A = (-1,...,-1) (m-1 negative ones), where A is the lower triangular matrix
#defined in the code
d :=proc(n, m)
local A, row, col, negative_one_vector :
A := Matrix(m − 1, m − 1) :
for row from 1 to m − 1 do
for col from 1 to m − 1 do
if row ≥ col then
```

```
A[row, col] := binomial(n - col, row - col) :
else
A[row, col] := 0 :
end if:
end do:
end do:
negative_one_vector := Vector[column](m - 1, -1) :
LinearSolve(Transpose(A), negative_one_vector)
end proc:

#Requires 1 <= i <= 3
#Returns a list of all 0-1 vectors with i ones out of n arguments of the vector
get_valid_vectors :=proc(i, n)
local valid_vectors, loop_i, loop_j, loop_k, x :
valid_vectors := [ ] :
if i = 1 then
for loop_i from 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
elif i = 2 then
 for loop_i from 1 to n - 1 do
for loop_j from loop_i + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
else
 for loop_i from 1 to n - 2 do
for loop_j from loop_i + 1 to n - 1 do
for loop_k from loop_j + 1 to n do
x := Vector(n, 0) :
x[loop_i] := 1 :
x[loop_j] := 1 :
x[loop_k] := 1 :
valid_vectors := [op(valid_vectors), x] :
end do:
end do:
end do:
end if:
valid_vectors
end proc:

s :=proc(i, n, A, m)
local valid_vectors, total, arg_of_S_m, j :
```

```
valid_vectors := get_valid_vectors(i, n) :
total := 0 :
for j from 1 to numelems(valid_vectors) do
arg_of_S_m := Multiply(A, valid_vectors[j]) :
total := total + compute_all_symmetric_polys(arg_of_S_m, m)[m+1] :
end do:
total
end proc:

#computes perm_m(A)
#requires m ≤ 4
perm_m :=proc(A, m, n)
local ones, total, j, values_of_d_n :
ones := Vector(n, 1) :
total := compute_all_symmetric_polys(Multiply(A, ones), m)[m+1] :
values_of_d_n := d(n, m) :
for j from 1 to m - 1 do:
total := total + (s(j, n, A, m)·values_of_d_n[j]) :
end do:
total
end proc:

g := proc(m :: integer, n, delta, A)
```

$$evalf\left(\frac{m!\cdot(-\text{delta})^m\cdot perm\_m(A,\ m,\ n)\cdot(n-m)!}{n^{n-m}}\right)$$

```
end proc:

get_f_solutions := proc(m, n, delta, A)
local g_values, lower_triangular_B, row, col, right_hand_side, i, Solutions :
```

$$g\_values := \left[\frac{n!}{n^n}\right]:$$

```
for i from 1 to m do
g_values := [op(g_values), g(i, n, delta, A)] :
end do:
lower_triangular_B := Matrix(m, m, 0) :
for row from 1 to m do
for col from 1 to row do
lower_triangular_B[row, col] := g_values[row - col + 1]·binomial(row−1, row
   −col) :
end do:
end do:
right_hand_side := Matrix(m, 1, 0) :
for row from 1 to m do
right_hand_side[row] := g_values[row + 1] :
end do:
LinearSolve(lower_triangular_B, right_hand_side)
end proc:
```

```
p := proc(m, n, delta, A)
 local f_values, sum, i :
 f_values := get_f_solutions(m, n, delta, A) :
 sum := 0 :
 for i from 1 to m do
 sum := sum + f_values(i)/i! :
 end do:
 evalf(ln(n!/n^n) + sum)
 end proc:


make_0_1_matrix :=proc(n, delta)
 local M, row, col, col_values, i, threshold, num_zeros_per_col, total,
    too_many_zeros, too_few_zeros :
 local counter, index_in_too_many_zeros, offending_rows, num_successful_swaps :
 local row_index, this_is_a_good_row, potential_target_col_index,
    potential_target_col, target_col :

 threshold := floor(delta·n) :
 M := Matrix(n, n, 1) :
 for row from 1 to n do
 col_values := randcomb(n, threshold) :
 for i from 1 to numelems(col_values) do
 M[row, col_values[i]] := 0 :
 end do:
 end do:

 num_zeros_per_col := Vector(n) :
 for col from 1 to n do
 total := 0 :
 for row from 1 to n do
 if M[row, col] = 0 then
 total := total + 1 :
 end if:
 end do:
 num_zeros_per_col[col] := total :
 end do:

 #makes lists with indexes of offending columns
 too_many_zeros := { } :
 too_few_zeros := { } :
 for counter from 1 to n do
 if num_zeros_per_col[counter] > threshold then
 too_many_zeros := too_many_zeros union {counter} :
 elif num_zeros_per_col[counter] < threshold then
 too_few_zeros := too_few_zeros union{counter} :
```

```
end if:
end do:

for index_in_too_many_zeros from 1 to numelems(too_many_zeros) do
col := too_many_zeros[index_in_too_many_zeros]:

#finds offending rows for this col
offending_rows := [ ]:
for row from 1 to n do
if M[row, col] = 0 then
offending_rows := [op(offending_rows), row]:
end if:
end do:

num_successful_swaps := 0 :
#for a given row in offending_rows, finds the first col to swap into
for row_index from 1 to numelems(offending_rows) do
this_is_a_good_row := true :
row := offending_rows[row_index]:
for potential_target_col_index from 1 to numelems(too_few_zeros) do
potential_target_col := too_few_zeros[potential_target_col_index]:
if M[row, potential_target_col] = 1 then
target_col := potential_target_col :
break:
else

  #if this_is_a_good_row = false then there is a zero in that col, so the swapping
  procedure never runs

  #this means there are no valid columns to swap into, so we have to just go to
  the next row - this is handled
#by the first if statement in the swapping procedure
this_is_a_good_row := false :
end if:
end do:

#swapping procedure
if this_is_a_good_row then
#makes swap
num_successful_swaps := num_successful_swaps + 1 :
M[row, target_col] := 0 :
M[row, col] := 1 :
num_zeros_per_col[target_col] := num_zeros_per_col[target_col] + 1 :

#removes cols from too_few zeros if needed
if num_zeros_per_col[target_col] = threshold then
too_few_zeros := too_few_zeros minus {target_col} :
end if:
```

```
end if:

 #makes sure we don't keep swapping after there no swaps to make
 if num_successful_swaps = numelems(offending_rows) - threshold then
 break:
 end if:

 #ends loop over rows in offending rows
 end do:
#ends loop over cols in too_many_zeros
 end do:
 M
 end proc:

 #MAIN
 n_min := 3 :
 n_max := 15 :
 n_increment := 1 :
 m := 3 :
 delta_min := 0.2 :
 delta_max := 0.4 :
 delta_increment := 0.02 :
 num_col := ceil((delta_max - delta_min)/delta_increment + 1) :
 num_row := ceil((n_max - n_min)/n_increment + 1) :
 num_trials := 100 :
 dataMatrix := Matrix(num_row, num_col) :
 for n from n_min by n_increment to n_max do
 for delta from delta_min by delta_increment to delta_max do
 col := ceil((delta - delta_min)/delta_increment + 1) :
 row := n - n_min + 1 :
 total_error := 0 :
 successful_trials := 0 :
 if n·delta ≥ 1 then
 for i from 1 to num_trials do
B := make_0_1_matrix(n, delta) :
J_n := Matrix(n, n, 1) :
 A := (J_n - B)/(n·delta) :
 perm_B := evalf(Permanent(B)) :
 if perm_B > 0 then
 successful_trials := successful_trials + 1 :
 diff_from_real := evalf(abs(n^n·exp(p(m, n, delta, A)) - perm_B)) :
 total_error := total_error + evalf(abs(diff_from_real/perm_B)·100) :
 end if:
```

```
end do:
percent_error := parse(sprintf("%.5f", total_error/successful_trials)) :
else
percent_error := 0 :
end if:
dataMatrix[row, col] := percent_error :
end do:
end do:
print(dataMatrix) :
```