Commun. Comput. Phys.
doi: 10.4208/cicp.260514.231214a

Vol. **18**, No. 1, pp. 104-124
July 2015

Computing the Smallest Eigenvalue of Large Ill-Conditioned Hankel Matrices

Niall Emmart[1,*], Yang Chen[2] and Charles C. Weems[1]

[1] *College of Information and Computer Science, University of Massachusetts, Amherst, MA 01002, USA.*
[2] *Department of Mathematics, University of Macau, Macau, China.*

Received 26 May 2014; Accepted (in revised version) 23 December 2014

Abstract. This paper presents a parallel algorithm for finding the smallest eigenvalue of a family of Hankel matrices that are ill-conditioned. Such matrices arise in random matrix theory and require the use of extremely high precision arithmetic. Surprisingly, we find that a group of commonly-used approaches that are designed for high efficiency are actually less efficient than a direct approach for this class of matrices. We then develop a parallel implementation of the algorithm that takes into account the unusually high cost of individual arithmetic operations. Our approach combines message passing and shared memory, achieving near-perfect scalability and high tolerance for network latency. We are thus able to find solutions for much larger matrices than previously possible, with the potential for extending this work to systems with greater levels of parallelism. The contributions of this work are in three areas: determination that a direct algorithm based on the secant method is more effective when extreme fixed-point precision is required than are the algorithms more typically used in parallel floating-point computations; the particular mix of optimizations required for extreme precision large matrix operations on a modern multi-core cluster, and the numerical results themselves.

AMS subject classifications: 15B52, 15B57, 42C05, 65F15, 65G30, 65Y05

Key words: Parallel eigensolver, Hankel matrices, extremely ill-conditioned matrices.

1 Introduction

In the majority of standard matrix computations, the matrix elements come from measurements of one kind or another. Accurate measurements might have 6-15 significant

*Corresponding author. *Email addresses:* nemmart@yrrid.com (N. Emmart), yayangchen@umac.mo (Y. Chen), weems@cs.umass.edu (C. C. Weems)

 ©2015 Global-Science Press

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

digits. In most cases, computations with double precision arithmetic and the right pivoting strategy produce results that are valid to the precision allowed by the initial measurements. The round-off errors introduced during the computation are generally much smaller than the uncertainty in the original measurements. In this paper, we examine a class of ill-conditioned matrix problems from random matrix theory, whose elements come not from measurements but are given by an explicit formula that can be evaluated to arbitrary precision. Unlike standard problems, to perform accurate computations, the initial matrix must be evaluated to thousands of digits of precision and the intermediate computations must be performed with tens of thousands of digits of precision. The extreme precision requires enormous computing resources, and presents novel challenges and trade offs.

In particular, we study the problem of quickly and efficiently computing the smallest eigenvalue of a family of ill-conditioned Hankel matrices. Recall that Hankel or moments matrices, denoted as $\mathtt{A}$, are obtained through (positive) weight functions $w(x)$ supported on $\mathbb{R}$ or subsets of $\mathbb{R}$, and are defined by

$$\mathtt{A}_{i,j}=\mu_{i+j}:=\int_a^b x^{i+j}w(x)dx,\quad (i,j=0,1,2,\cdots).$$

These matrices are symmetric, and generate a positive definite quadratic form. For the problem at hand, we consider

$$w(x):=\mathrm{e}^{-x^\beta},\quad 0\le x<\infty,\quad \beta>0,$$

and the moments are given by the formula:

$$\mu_j=\int_0^\infty x^j\mathrm{e}^{-x^\beta}dx=\frac{1}{\beta}\Gamma\Big(\frac{1+j}{\beta}\Big),\quad (j=0,1,2,\cdots).$$

We denote the order N Hankel matrix by $\mathtt{A}_N=(A_{i,j})_{0\le i,j\le N-1}$. $\Gamma(z)$ is the gamma function and for complex numbers with a positive real part, gamma is defined by

$$\Gamma(z)=\int_0^\infty t^{z-1}\mathrm{e}^{-t}dt.$$

The following are two example matrixes for $N=4$, $\beta=1$ and $N=4$, $\beta=\frac{7}{4}$:

$$\begin{bmatrix} 0! & 1! & 2! & 3! \\ 1! & 2! & 3! & 4! \\ 2! & 3! & 4! & 5! \\ 3! & 4! & 5! & 6! \end{bmatrix}\quad\text{and}\quad \begin{bmatrix} \frac{4}{7}\Gamma(\frac{4}{7}) & \frac{4}{7}\Gamma(\frac{8}{7}) & \frac{4}{7}\Gamma(\frac{12}{7}) & \frac{4}{7}\Gamma(\frac{16}{7}) \\ \frac{4}{7}\Gamma(\frac{8}{7}) & \frac{4}{7}\Gamma(\frac{12}{7}) & \frac{4}{7}\Gamma(\frac{16}{7}) & \frac{4}{7}\Gamma(\frac{20}{7}) \\ \frac{4}{7}\Gamma(\frac{12}{7}) & \frac{4}{7}\Gamma(\frac{16}{7}) & \frac{4}{7}\Gamma(\frac{20}{7}) & \frac{4}{7}\Gamma(\frac{24}{7}) \\ \frac{4}{7}\Gamma(\frac{16}{7}) & \frac{4}{7}\Gamma(\frac{20}{7}) & \frac{4}{7}\Gamma(\frac{24}{7}) & \frac{4}{7}\Gamma(\frac{28}{7}) \end{bmatrix}.$$

In the numerical section of [9], Chen and Lawrence used the Jacobi rotation method to compute the smallest eigenvalues for matrices up to size 300 by 300. For the 300 by

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

300, their algorithm required weeks of CPU time on a single core machine. Yet some of the problems involving Hankel matrices only become interesting when the matrices are at least 5000 by 5000. Even though computing capabilities have increased substantially since their initial research in 1999, it is apparent that to handle such large matrices requires an efficient, scalable, parallel solution.

In this paper, we explore this problem in depth. What makes it so hard? Why does it require a supercomputer even for moderate matrix sizes? We examine several serial eigensolver algorithms, comparing their speed and accuracy. Surprisingly, we find that the best amongst these is a direct approach using the Secant method to find the first zero of the characteristic polynomial. We then explore parallelizing this algorithm and show that a simple *LDLT* determinant algorithm that assigns columns to nodes using a round-robin technique can successfully overlap the computation and communication. We demonstrate that this algorithm scales nearly linearly, achieving utilization rates of over 90% on a range of cluster sizes.

One aspect of this work that is particularly interesting is the unusual combination of computation and communication. The computations are performed using fixed point integer arithmetic where each value requires kilobytes of storage. Because of the large size of the values, the effective use of level 1 and level 2 caches are determined by the implementation of the arbitrary precision integer routines rather than the blocking of the original matrix. Further, communication of these very large values across partitions in a parallel implementation represents an unusual range of message granularities. Our approach of combining MPI with OpenMP is not in itself novel, but we believe that the way in which we balance these to obtain scalable performance in this context is informative for future research in this area.

As a result of our exploration of this corner of the parallel implementation design space, we are able to obtain completely new numerical results for much larger matrices in just a few hours of wall-clock time, using a fairly modest cluster of up to 440 cores. The approach does not appear to have any limitations that would prevent it from efficiently scaling to larger problem sizes, running at the highest levels of parallelism that are feasible with current technology. Finally, we present some of our numerical results and describe future research.

2 Background

Random matrix theory which originated in multi-variate statistics in the 1920's, and independently in nuclear physics in the 1950's is an active and on-going area of research. It has applications in many diverse fields: multi-variate statistics, quantum physics, traffic and communications networks, and stock movements in financial markets. For a variety of applications of random matrix theory, some of which are conjectural, ranging from the zero distributions of the zeta-function to the statistics of the energy levels in chaotic cavity see Mehta [19]. The recent monographs of Anderson, Guionnet and Zeitouni [2] and

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

of Blower [5] approach random matrices from a probabilistic point of view. In [23] one finds statistical linkage analysis on chronic HCV infection. For the application of random matrices to information theory of wireless networks, see for example [11], [17], and [8], and the references therein. A collection of review articles on the application of random matrices can be found in [3]. For a recent application of random matrix theory to a class of problems in multi-variate statistics see [12], and the references therein.

Random matrix theory considers the properties (determinants, eigenvalues, eigenvalue distributions, eigenvectors, spectra, inverses, etc) of matrices whose elements are random variables chosen from a given distribution. The properties of certain random matrices in turn depend on the properties of matrices whose elements are exact mathematical functions, such as the Hankel matrices studied in this paper. Kerov [15], for example, presents a monograph that examines the connections between representation theory, random matrices and moment problems.

Hankel matrices occur naturally in moment problems - for a given sequence of moments (mean, variance, skewness, kurtosis, etc), they can be used to determine a probability distribution/measure that reproduces the moment sequence. In-depth studies on this can be found in the monographs by Akhiezer [1] and by Krein [16]. Moment problems are classified according to the support of their distribution/measure. If the support is a closed interval (the Hausdorff moment problem), there will always be a unique solution. If the support is the whole number line (the Hamburger moment problem), then one can encounter a situation where the problem is said to be indeterminate, that is there are infinitely many probability distributions/measures with the same sequence of moments.

The classic papers by Szegö [24], by Widom and Wilf [26] and by Widom [25] showed that for those Hankel matrices of order N, generated by a probability density supported on a closed interval, the corresponding smallest eigenvalue tends to zero as N tends to infinity.

Berg, Chen and Ismail [4] showed that in the infinite interval case, the moment problem will be indeterminate if and only if the smallest eigenvalue of the Hankel matrix tends to a strictly positive number as N tends to infinity. This is a new criteria for the determinacy of the Hamburger moment problem and is part of our motivation for finding fast algorithms to compute the smallest eigenvalues of large Hankel matrices. For recent accounts of this and related problem see for example, the papers of Chen and Lubinsky [10], Lubinsky [18], and Berg and Szwarc [6].

3 The scale of the problem

The condition number is often a good indicator of the amount of precision required in order to compute the smallest eigenvalue. Here we show that $\mathtt{A}_N$ is extremely ill conditioned.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

The condition number is defined to be:

$$\text{cond}(\mathtt{A}_N) = \frac{\lambda_N[\mathtt{A}_N]}{\lambda_1[\mathtt{A}_N]}, \quad \text{where } \lambda_N \text{ is the largest eigenvalue and } \lambda_1 \text{ is the smallest.}$$

The condition numbers can be estimated using the computation of λ_1 and by bounding λ_N as follows. The Raleigh quotient, $\rho(u;\mathtt{A}_N)$ ranges over the interval $[\lambda_1,\lambda_N]$ for non-zero u, therefore:

$$\rho(u;\mathtt{A}_N) = \frac{\langle \mathtt{A}_N u,u\rangle}{\langle u,u\rangle} \leq \lambda_N \quad \text{for all non-zero } u.$$

Choosing u to be the column vector $(0,0,0,\cdots,1)$:

$$\rho(u;\mathtt{A}_N) = \frac{\langle \mathtt{A}_N u,u\rangle}{\langle u,u\rangle} = \frac{1}{\beta}\Gamma\Big(\frac{2N-1}{\beta}\Big) \leq \lambda_N.$$

Further, $\lambda_N \leq \text{trace}(\mathtt{A}_N)$, therefore:

$$\frac{1}{\beta}\Gamma\Big(\frac{2N-1}{\beta}\Big) \leq \lambda_N \leq \text{trace}(\mathtt{A}_N) = \frac{1}{\beta}\Gamma\Big(\frac{2N-1}{\beta}\Big) + \sum_{k=1}^{N-1}\frac{1}{\beta}\Gamma\Big(\frac{2k-1}{\beta}\Big).$$

For large N, $\frac{1}{\beta}\Gamma\big(\frac{2N-1}{\beta}\big)$ is a good estimate for λ_N because $\sum_{k=1}^{N-1}\frac{1}{\beta}\Gamma\big(\frac{2k-1}{\beta}\big)$ is very small compared to $\frac{1}{\beta}\Gamma\big(\frac{2N-1}{\beta}\big)$.

In Table 1, condition numbers has been calculated using the lower bounds for λ_N and the results of the λ_1 computations. As can be seen from this table, in each case the condition number is growing faster than N^N.

Table 1: Lower bound on COND($\mathtt{A}_N$).

N	β=1/3	β=1/2	β=1	β=7/4	N^N
100	8.52x10^1396	7.36x10^861	9.40x10^384	1.94x10^228	1.00x10^200
300	5.17x10^5066	4.65x10^3167	6.38x10^1429	3.55x10^819	1.37x10^743
500	4.56x10^9116	6.89x10^5726	3.62x10^2597	4.11x10^1472	3.05x10^1349
1000	1.85x10^20050	6.97x10^12663	7.62x10^5780	6.80x10^3240	1.00x10^3000
1500	1.11x10^31666	3.81x10^20055	8.45x10^9187	3.32x10^5125	1.37x10^4764

4 Properties of $\mathtt{A}_N$

The algorithms we explore in this paper rely on three properties of $\mathtt{A}_N$:

- $\mathtt{A}_N$ is real and symmetric. $\mathtt{A}_N$ has N positive eigenvalues and the smallest eigenvalue is simple.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

- The eigenvalues of $\mathtt{A}_N$ are the zeros of the characteristic polynomial $P(x)=\det(\mathtt{A}_N - xI)$.

- $\mathtt{A}_N$ is the Hankel moment matrix for the weight function $w(x)=\mathrm{e}^{-x^\beta}$, $x>0$, $\beta>0$.

The first point is important for rapid convergence of the Secant algorithm which we argue as follows. Since $w(x)$=exp$(-x^\beta)$, as we have previously seen, the moments μ_j exist for $j=0,1,2,\cdots$. Let $Q_N(x)$ be any polynomials of degree N with real coefficients $\{c_j, j=0,1,\cdots,N\}$, namely,

$$Q_N(x)=\sum_{j=0}^{N} c_j x^j.$$

A computation shows that

$$\int_0^\infty [Q_N(x)]^2 w(x)dx=\sum_{j=0}^{N}\sum_{k=0}^{N} c_j c_k \mu_{j+k},$$

where $\mu_j=\beta^{-1}\Gamma((j+1)/\beta), j=0,1,2,\cdots$. Since

$$\int_0^\infty [Q_N(x)]^2 w(x)dx>0,$$

for $Q_N(x)$ not vanishing identically in x, the quadratic form is positive definite,

$$\sum_{j=0}^{N}\sum_{k=0}^{N} c_j c_k \mu_{j+k}>0.$$

Szego's variational characterization [24] of the quadratic form, shows that the smallest eigenvalue is simple. Such a technique has been adopted in [9] for the least eigenvalue of our Hankel matrix $(\beta^{-1}\Gamma((j+k)/\beta))$, where $\beta>1/2$.

5 Algorithm selection

There are several standard techniques for finding eigenvalues, see e.g., [21]. For this paper we evaluated four that are widely used, based on three criteria: (a) how much precision does the algorithm require to meet a desired level of precision in the output; (b) how fast is the calculation relative to the other algorithms; (c) how effectively can the algorithm be parallelized. To evaluate these criteria in the context of the precision required by the problem, the four algorithms were implemented using the GNU Multiple Precision (GMP) library [14]. The specific algorithms are:

- Lanczos Algorithm as described by Stoer and Bulirsch [22, p. 401].

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

- Householder's Algorithm as described by Stoer and Bulirsch [22, p. 391] (with minor modification).
- The Jacobi Method with Rutishauser's enhancements, as described by Parlett [21, pp. 189-196]. The Jacobi Method is known to perform well on graded matrices such as ours.
- A direct approach with an *LDLT* determinant algorithm and a Secant root finder, as described below (hereafter referred to as the Secant algorithm).

The algorithms are tested by varying the number of bits of precision (K) of the inputs and the computation required to achieve a specific level of precision in the result. In Table 2, *Accuracy* is the number of correct significant decimal digits in the result. *Run Time* is in seconds. *Factor* is the slowdown factor for this algorithm relative to the fastest for a given N. The algorithm with the most accuracy for the least run time will be the best choice for computing the smallest eigenvalue.

Table 2:

Algorithm	N	β	K	Accuracy	Run Time	Factor
Secant	100	1	800	60	0.96	1.0
Householder	100	1	2490	9	3.80	3.96
Jacobi	100	1	700	58	5.91	6.15
Lanczos	100	1	206250	36	361.46	376.50
Secant	200	1	1400	90	23.67	1.0
Householder	200	1	5940	59	134.32	5.67
Jacobi	200	1	1500	24	216.55	9.15
Secant	300	1	1600	60	144.02	1.0
Householder	300	1	10000	53	990.26	6.88
Jacobi	300	1	2400	8	2027.72	14.08
Secant	400	1	2100	51	603.84	1.0
Householder	400	1	13250	41	4036.11	6.68
Jacobi	400	1	3375	48	9888.59	16.38

It is well known that graded matrices such as $\mathtt{A}_N$ present special challenges for eigensolvers, but one unexpected result was the very poor performance of the Lanczos algorithm, which failed for $N=100$ with anything less than 205000 bits of precision. There are a number of variants of the basic Lanczos algorithm, some orthogonalize the new vector after each iteration and some stop and restart the process. However we do not believe the variants would improve the Lanczos results enough to make it competitive with the other algorithms.

As can be seen from the table, for each N, the Secant algorithm is both the fastest and the most accurate of the algorithms. Therefore, it's valid to compare the run times. For

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

example, we can conclude that for $N=100$, the Secant algorithm is at least 3.96 times faster than the Householder method.

In the remainder of the paper, we thus focus on the Secant algorithm and show how it can be effectively parallelized in the context of extreme precision arithmetic.

6 The Secant algorithm

The Secant algorithm can be used to find the smallest root of the characteristic polynomial, $P(x)$. $P(x)$ can be defined as either $\det(xI-\mathtt{A}_N)$ or $\det(\mathtt{A}_N-xI)$. We use the latter because it guarantees $P(0)$ will be positive. The Secant algorithm starts with two initial points x_1 and x_2 which must be less than λ_1, the smallest eigenvalue. We choose x_1 to be a small negative value and x_2 to be zero. The Secant algorithm then computes a sequence of x_j's using the following recurrence relation:

$$x_{j+1}=x_j-\frac{x_j-x_{j-1}}{P(x_j)-P(x_{j-1})}P(x_j).$$

The computation is done when the most significant digits (15 decimal digits in our case) of x_j have stabilized. Since the eigenvalues of $\mathtt{A}_N$ are distinct, the roots of $P(x)$ will all be simple and the Secant algorithm is guaranteed to converge rapidly to λ_1, see [7]. Also note, since $P(x)$ has no inflection points for x less than λ_1, the slope of the Secant at x_j will always be less than $P'(x)$ when $x_j<x\leq\lambda_1$ and therefore the Secant is guaranteed not to overshoot λ_1.

Finally, there is no need to solve for $P(x)$. Instead, we can evaluate $P(x)$ directly by computing $\det(\mathtt{A}_N-xI)$. Thus, for our matrices, the problem of finding the smallest eigenvalue reduces to that of solving a sequence of determinants.

7 Verification using interval analysis

After the secant method has converged to some x^*, the next step is to prove that x^* is in fact an eigenvalue of $\mathtt{A}_N$. To do this, we need show that

$$\det\Big(\mathtt{A}_N-(x^*-\epsilon)I\Big)>0 \quad \text{and} \quad \det\Big(\mathtt{A}_N-(x^*+\epsilon)I\Big)<0$$

and prove that round off errors (both in the initial matrix and during the determinant computation) have not poisoned the result. We do this using interval arithmetic. Interval arithmetic replaces each value in the computation with a small interval containing the exact value. Computations on intervals always round the lower end point down and round the upper end point up, thus guaranteeing that the resulting interval contains the correct result computed with infinite precision. For further reading see *Interval Analysis,* by R.E. Moore [20].

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

In our process, we first compute $\mathtt{A}_N$ to high precision and then construct a matrix of intervals, guaranteeing that the gamma function evaluated to infinite precision resides in the interval. We then truncate x^* to 15 significant digits and compute $\det(\mathtt{A}_N - x^* I)$. The lower end point of the interval must be strictly positive. Next we add 1 to the least significant digit of x^* and compute $\det(\mathtt{A}_N - x^* I)$ again. The upper end point of the interval must be strictly negative. If zero is contained in the interval that results from either determinant computation, then the verification has failed and we must increase the precision and repeat the process.

The verification using interval analysis is quite slow to perform, but the results are very strong. They guarantee that we have found the smallest eigenvalue[†] and that the result is correct to 15 significant digits.

8 The *LDLT* determinant algorithm

There are several standard techniques to compute the determinant of a matrix. The fastest general methods all involve factoring the matrix in some way and then calculating the determinant from the factors. In our case, we can make use of the fact that $\mathtt{A}_N$ is symmetric and perform an *LDLT* factorization. Because $x < \lambda_1$, $\mathtt{A}_N - xI$ will be positive definite and the factorization will be numerically stable without the need for pivoting. This is essentially the same as a Cholesky factorization except that it avoids the square root operations. To make the algorithm easy to parallelize, we use the *sub-matrix* order [13] for the *LDLT* algorithm which applies a column of the matrix to all the remaining columns to its right. See Fig. 1.

Once the matrix has been factorized, the determinant can be computed as the product of the terms along the main diagonal:

$$\det(\mathtt{A}_N) = \Pi_{i=1}^{N} \mathtt{A}_\mathtt{i}[\mathtt{i}].$$

One thing to note, the divisions as shown in Fig. 1 are very expensive. In practice we will introduce a new vector $\mathtt{B}_\mathtt{i}$, where $\mathtt{B}_\mathtt{i}$[j] = $\mathtt{A}_\mathtt{i}$[j] / $\mathtt{A}_\mathtt{i}$[i], and significantly reduce the number of divisions.

Presented below are serial and parallel versions of the algorithm. All the numeric computations are performed with arbitrary precision fixed point arithmetic using the GMP Z functions. Fixed point arithmetic uses a fixed number of bits to the right of the decimal point and a variable number of bits to the left of the decimal point. For each element of the initial matrix $\mathtt{A}_N$ the gamma functions are all evaluated with K bits of precision, i.e., K bits to the right of the decimal point. All intermediate computations during the factorization and final determinant computation are done with $2K$ bits of precision.

[†]Theoretically, it is possible that roundoff errors during the secant method have caused the algorithm to overshoot the smallest eigenvalue and instead converge on the wrong odd eigenvalue, i.e., $\lambda_3, \lambda_5, \lambda_7, \cdots$, however, we believe it is highly improbable. Further, any such overshoot should be highly dependent on the exact precision used during the secant method. A second run with slightly higher precision should eliminate the risk entirely.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

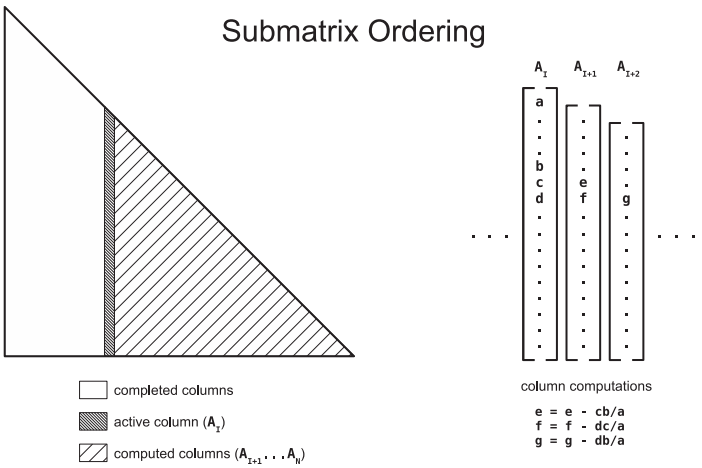


Figure 1: Submatrix order.

```
for (i=1 to N) {
  // this loop computes B_i
  for (j=i+1 to N)
    B_i[j] = A_i[j] / A_i[i];

  // these loops apply i^th column (B_i) to columns A_i+1, A_i+2, ... A_N
  for (j=i+1 to N)
    for (k=j to N)
      A_j[k]=A_j[k] - A_i[j]*B_i[k];
}
```

Algorithm 1: Serial LDLT Code

Serial *LDLT* algorithm

Algorithm 1 shows the serial version of the *LDLT* algorithm. The algorithm loops through the columns of A, computing $B_i[j] = A_i[j] / A_i[i]$ and then applying it the column to the remaining columns to the right.

Parallel *LDLT* algorithm

The parallel version of the *LDLT* algorithm partitions the columns across the nodes. The columns are assigned using a round robin approach that balances the load as described below. We use a combination of MPI and OpenMP to manage the parallelism. MPI is used to distribute data between nodes and OpenMP is used to spread the computation across multiple cores within a node. The inter-core communication overhead of OpenMP is, of course, much less than for MPI, and the majority of the computation time is spent in

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

just a few loops that naturally parallelize with OpenMP. The data distribution between the nodes is done using MPI broadcasts. These are run in a separate thread that allows the communication to be overlapped with the computation.

```
Assign the columns to nodes in a balanced round robin fashion
Broadcast the values needed to construct the initial matrix
Compute B1
for (i=1 to N) {
  if(column i+1 is assigned to this node) {
    Apply the ith column (Bi) to column Ai+1
    Compute Bi+1
    Initiate background transmit of Ai+1 and Bi+1
    Apply the ith column (Bi) to assigned columns from Ai+2 ... AN
    Wait for transmit to complete
  }
  else {
    Initiate background receive of Ai+1 and Bi+1
    Apply the ith column (Bi) to assigned columns from Ai+1 ... AN
    Wait for receive to complete
    Compute any missing Bi+1 elements -- discussed in detail below
  }
  MPI barrier
}
```

Algorithm 2: Parallel LDLT Code

Collection of timing data

For all runs where the total run time is more than a few minutes, more than 99% of the time is taken up by the *LDLT* factorizations. All the other tasks: start up, generating and distributing the initial matrix, the secant and interval verification driver routines, and collection and processing of run times, etc, take less than 1% of the total time.

For each *LDLT* factorization, times are gathered for each column on each node. If the next column (column i+1) is assigned to this node, then the following timings are gathered:

- The total time to apply A_i and B_i to column i+1, then compute B_{i+1} and finally to apply A_i and B_i to A_{i+2} to A_N. This is the ***computation time***.
- In the transmit thread, two timers are kept, one times the transmission of the A_{i+1} vector and the other times the transmission of the B_{i+1} vector. The sum is the ***communication time***.
- The main thread records the amount of time the CPU is idle waiting for the transmit to complete. This is the ***idle time***.
- The time spent at the MPI barrier. This is the ***barrier time*** for the column.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

If the next column is not assigned to this node, then the following timings are gathered:

- The time to apply $\mathtt{A}_i$ and $\mathtt{B}_i$ to $\mathtt{A}_{i+1}$ to $\mathtt{A}_N$. This is the ***computation time***.
- In the receive thread, there are two timers, one for the time to receive the $\mathtt{A}_{i+1}$ column and one for the time to receive the $\mathtt{B}_{i+1}$ column. The sum is the ***communication time***.
- If some elements of the B column are not sent, they must be computed locally (this is discussed further in the optimizations section). The time spent computing these values is measured and is the ***division time***.
- The time spent waiting for the receive to complete is the ***idle time***.
- The time spent at the MPI barrier is the ***barrier time***.

At the end of *LDLT* factorization, the timing data from all the nodes and all the columns are collected. The timing values are generated by averaging the values from each node for each column $\mathtt{A}_i$. The averages are then written to a log file. These average times are the basis for the timing graphs presented.

Finally, note that because the solution is threaded, the communication time and the computation time are overlapped.

9 Algorithm optimization

In developing this algorithm, we found that there were four areas in which it was most profitable to focus our optimization efforts: distribution of the columns over the processors, re-blocking and reordering the parallel loop(s), combining partial transmission with local completion of some computations, and removing the remaining synchronization barrier.

Column assignment

Because the matrix is triangular, the amount of processing time required decreases for each successive column. A simple round robin distribution would assign node 1 far more work than the last node in the cluster. To resolve this, we use a balanced round robin approach that alternately assign M columns to nodes 1 up to M and then M columns to nodes M down to 1. This approach does a satisfactory job of balancing load on a homogeneous system.

The column assignment problem is, of course, more complex for a heterogeneous system. We leave distribution over heterogeneous resources, such as a loosely coupled grid, or a CPU/GPU combination, as a topic for future research.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

OpenMP optimizations

Early versions of the algorithm used OpenMP on the innermost loop. The code was structured as follows:

```
for (j=i+1 to N) {
  #pragma omp parallel for
  for (k=j to N)
    Aj[k]=Aj[k] - Ai[j]*Bi[k];
}
```

Since the multiplications in the inner loop are quite slow (each iteration multiplies tens of thousands of bits by tens of thousands of bits), and their computation time varies considerably, depending on their location in the matrix, a significant amount of processing time is wasted at the implicit barrier at the end of the inner for loop. There are a number of ways this might be solved with OpenMP directives, but for our problem it was convenient to lay the column vectors out in a single large array (called V in the example) and iterate with just a single for loop, as shown on the next page.

```
#pragma omp parallel for threadprivate(j, k) schedule(dynamic, chunk_size)
for (index=lastIndex downto firstIndex) {
  j=... decode j from index ...
  k=... decode k from index ...
  Vindex=Vindex - Ai[j]*Bi[k];
}
```

Notice also that this loop counts down through the indices. The reason for going backwards is that the numbers are much smaller toward the upper left of the triangular matrix and therefore the multiplications are faster. This arrangement therefore reduces the time lost at the final barrier. We use a heuristic for the *chunk_size* that balances the thread synchronization overhead against the wasted time at the final barrier.

Sending versus computing B_{i+1}

Another problem with early versions of the algorithm was that as the size of the cluster scaled up, the overall efficiency (percentage of the time spent doing useful computation) dropped significantly. The original approach was to broadcast A_{i+1} and then each node would compute a local copy of B_{i+1}. This seems reasonable, given that the total number of divisions is $\mathcal{O}(N^2)$ versus $\mathcal{O}(N^3)$ multiplications in the inner loop. However, all of the divisions are run on each node, whereas the multiplications are distributed evenly among the nodes. By the point that system size reaches 20 nodes, the duplicated divisions represents a substantial performance hit.

Conveniently, during the early columns of the matrix, there is a significant amount of idle communication bandwidth. We can put that bandwidth to good use by computing B_{i+1} on the node that is assigned column `i+1` and broadcasting the values to the other

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

nodes. Toward the end of the matrix, however, the computation becomes communication bound. At that point, sending the B_{i+1} values takes more time than computing them locally, so we stop transmission and let each node pick up with whatever portion of B_{i+1} that it has already received.

Our final transmission algorithm can be summarized as:

```
serialize the A_{i+1} column
send the A_{i+1} column
break the B_{i+1} column into chunks of 100 values
while (... there are more chunks ... and
       ... at least 8000 more multiplications to perform ...) {
   send the next chunk of B_{i+1}
}
```

The thresholds of 100 values and 8000 multiplications were chosen empirically and seem to work well on a variety of problem sizes and cluster geometries.

Removing the MPI barrier

The final optimization is to remove the MPI barrier. As one might expect, the barrier forces the nodes with less work to wait at the end of each column for all nodes to complete the column, wasting compute resources. This can be seen in Fig. 2. For columns 1 through 1250, the total column time with a barrier is more than the total column time without the barrier. However, after column 1250, when the computation is communication bound, the barrier has almost no effect on the total column time.

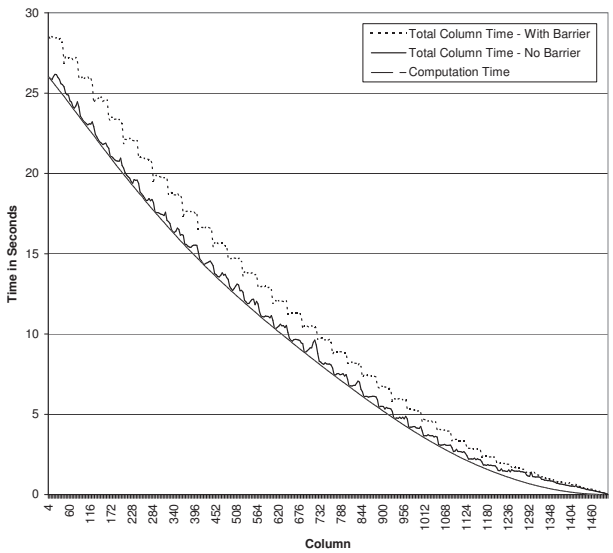


Figure 2: The effect of removing the MPI barrier at the end of the loop. 40 nodes, each with 2 cores.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

10 Performance

The performance testing was done on the University of Massachusetts Swarm cluster, which has 60 compute nodes, each with 8 cores (Xeon 5355 processors at 2.66 GHz) and 16 GB of RAM per node. The nodes are connected via gigabit ethernet. The cluster is normally partitioned such that our smaller tests were required to run on 48 nodes, each with 5 cores for a total of 240 cores. To further evaluate scalability, we were granted a short period in which to run on the full machine, although at that time five of the nodes were inoperative, so we were able to exercise a maximum of 440 cores on the remaining 55 nodes. In addition to scaling, the performance figures thus also serve to demonstrate the flexibility of the approach with respect to cluster geometry.

The results show the cumulative time spent running the determinant algorithm. The results do not include the startup time, the time to generate the $\mathtt{A}_N$ matrix, nor the time spent running the actual secant computation. The total for these tasks is less than 1% of the time spent computing the determinants.

In the tables and graphs below *Total Time* is the total wall time spent computing determinants. *Computation* is the cumulative time spent executing useful, non-duplicated computation, and *Idle + Divs* is the cumulative time in waiting for the communications and local (duplicated) divisions to complete. We present *Idle + Divs* together because they both represent wasted time due to a lack of network bandwidth and duplicated computation. The values in parenthesis are times in seconds.

The Constant N table shows two trends. First, as β moves away from 1, the computation becomes more efficient. That is, the percentage of time spent doing computations increases and the percentage of time spend waiting for results from another node and divisions (*Idle + Divs*) decreases. Second, if the computation is communication bound, which is the case for $\beta \leq 1$, then increasing the number of cores makes the computation substantially less efficient. However, if it's compute bound, as is the case when $\beta = 7/4$, the computation scales nicely. In fact, the amount of time wasted waiting for results from another node and duplicate division is almost constant, 417 seconds vs. 426 seconds. In other words, the algorithm scales almost perfectly when the problem is large enough to be compute bound. The compute utilization graphs are shown in Figs. 3 and 4.

The constant beta table shows that as N increases, the percentage of time spent doing useful computation increases. Efficiency for large values of N remains well into the 90% range as the number of cores scales up from 80 to 440.

Fig. 5 shows breakdown of the times on a per column basis for a run of $\beta = 7/4$ and $N = 2000$ with 440 cores. For columns up to approximately 1300, the computation time is greater than the communication time, so the system is compute bound. After column 1300, the computation becomes communication bound and starts to do local divisions on each node (which reduces communication time but increases duplicated computation). Fig. 6 shows a graph of CPU utilization (computation time, excluding local division) verses the wall time in seconds for the run. The utilization is roughly 90% for 90% of the time and then drops off rapidly during the final columns.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

Table 3: Constant beta.

		$\beta=7/4$, N=1000		$\beta=7/4$, N=1500		$\beta=7/4$, N=2000		$\beta=7/4$, N=2500	
80 Cores	Total Time	0:43:57	(2,638)	4:04:31	(14,672)	13:57:35	(50,255)		
	Computation	92.2%	(2,432)	97.0%	(14,231)	98.4%	(49,438)	Run not performed	
	Idle + Divs	7.2%	(190)	2.8%	(417)	1.5%	(773)		
240 Cores	Total Time	0:17:33	(1,054)	1:24:53	(5,094)	4:45:25	(17,125)		
	Computation	77.0%	(811)	91.2%	(4,645)	96.2%	(16,466)	Run not performed	
	Idle + Divs	21.5%	(227)	8.4%	(426)	3.7%	(630)		
440 Cores	Total Time	0:13:26	(806)	0:53:06	(3,186)	2:47:48	(10,068)	7:33:36	(27,216)
	Computation	57.1%	(460)	82.5%	(2,629)	91.6%	(9,226)	95.3%	(25,933)
	Idle + Divs	42.0%	(339)	16.8%	(536)	7.9%	(797)	4.4%	(1,202)

Table 4: Constant N.

		$\beta=1/3$, N=1500		$\beta=1/2$, N=1500		$\beta=1$, N=1500		$\beta=7/4$, N=1500	
80 Cores	Total Time	8:29:45	(30,585)	6:38:32	(23,912)	3:59:40	(14,380)	4:04:31	(14,672)
	Computation	88.5%	(27,075)	87.3%	(20,868)	85.4%	(12,277)	97.0%	(14,231)
	Idle + Divs	11.4%	(3,483)	12.6%	(3,013)	14.4%	(2,076)	2.8%	(417)
240 Cores	Total Time	4:21:12	(15,673)	3:38:29	(13,109)	2:23:07	(8,588)	1:24:53	(5,094)
	Computation	54.7%	(8,575)	49.3%	(6,469)	43.9%	(3,767)	91.2%	(4,645)
	Idle + Divs	45.1%	(7,073)	50.4%	(6,609)	55.8%	(4,793)	8.4%	(426)

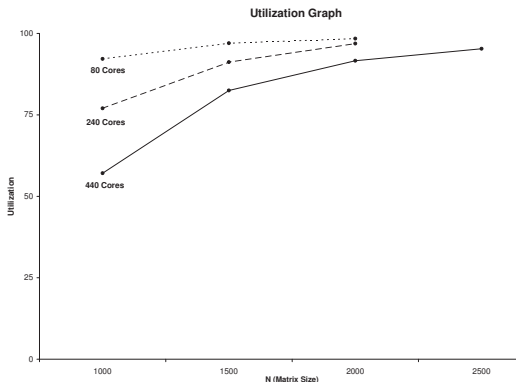


Figure 3: Constant beta.

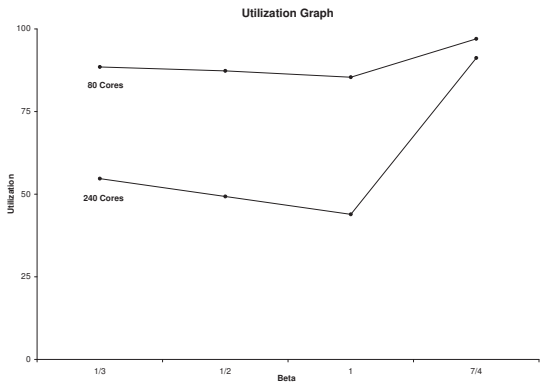


Figure 4: Constant N.

11 Communication

There are a couple of interesting things to note about the algorithm. First, it works by preparing a column, starting a background broadcast and then applying the column to the remainder of the matrix. As the matrix size grows, or the precision in the numbers

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

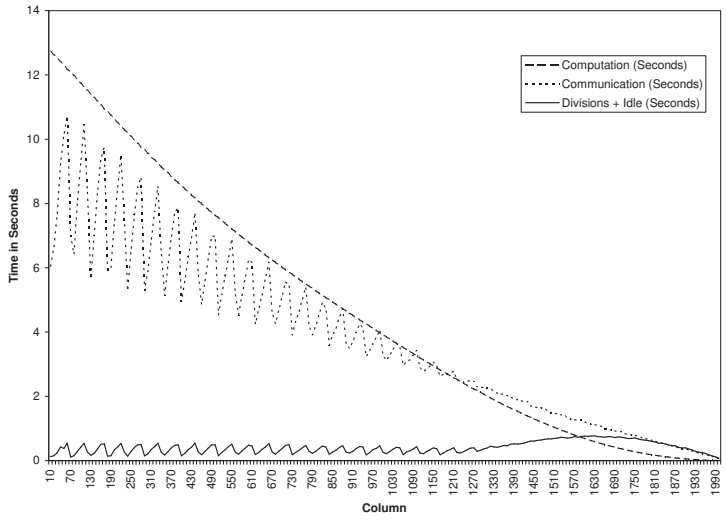


Figure 5: Column Timing Graph, Beta=7/4, N=2000, 440 cores.

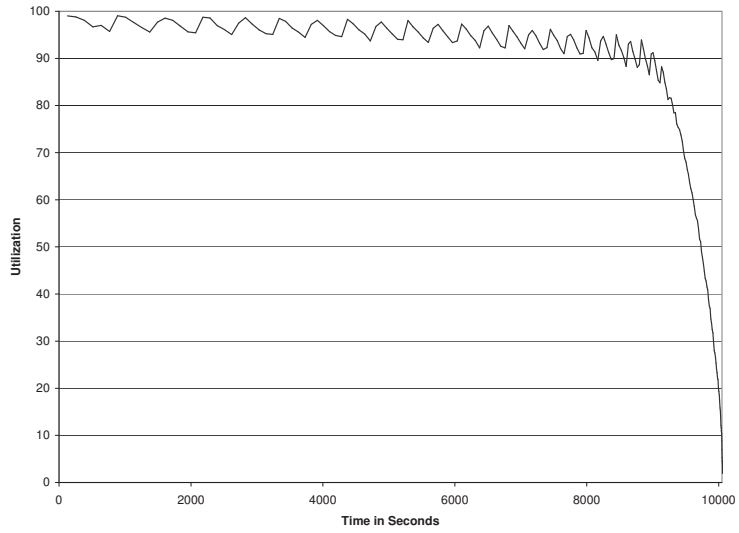


Figure 6: Utilization Graph, Beta=7/4, N=2000, 440 cores.

grows, it becomes easier and easier to fully overlap the computation and communication. This means that, with a large enough problem, network latency has minimal impact on performance, which is dominated by network throughput. Given sufficient network throughput, the algorithm can scale to very large systems.

Second, as N grows, the precision (K) required to perform the computation increases. In the inner loop of the computation two numbers, each with K bits are being multiplied. Increasing K thus increases the processing time by approximately $O(K^2)$, while the com-

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

munication time increases only linearly. Therefore, as N grows, there is a very powerful effect on the efficiency and scalability of the algorithm.

12 Numerical results

Table 5 shows a sampling of results for the smallest eigenvalue computations. It clearly illustrates the wide range of values that the algorithm must handle. These values are entirely new solutions for this problem.

Table 5: The smallest eigenvalue of $\mathtt{A}_N$.

N	β=1/3	β=1/2	β=1	β=7/4
100	3.4720	2.7397×10^{-1}	2.1079×10^{-15}	1.6976×10^{-45}
300	3.3984	1.5837×10^{-1}	5.5215×10^{-28}	1.4844×10^{-102}
500	3.3763	1.2047×10^{-1}	1.1138×10^{-36}	6.7121×10^{-149}
1000	3.3544	8.2087×10^{-2}	1.0892×10^{-52}	3.6209×10^{-246}
1500	3.3447	6.6295×10^{-2}	5.4593×10^{-65}	6.4232×10^{-330}
2000				6.2011×10^{-406}
2500				1.1483×10^{-476}

13 Comparing with theory

When N gets large, there is an analytical approach to provide solution to the smallest eigenvalue problem. The theory based on the ideas of Dyson's Coulomb fluid, gives a large N expression for our orthogonal polynomials, from which we derived an analytic expression for $\beta\neq p+1/2$, $p=1,2,\cdots$ in Chen and Lawrence, eq.(3.18).

Specializing this to $\beta=7/4$, the inverse of the smallest eigenvalue for large N reads

$$\frac{1}{\lambda_1(N)}\sim\frac{1}{8\pi^{5/4}}\frac{c^{1/4}}{\sqrt{A_0}}\mathrm{e}^{\sec(7\pi/4)}N^{-5/7}\exp\left[\frac{2N^{5/7}}{\sqrt{\pi c}}\left(A_0-\frac{A_1}{c}\frac{1}{N^{4/7}}\right)\right],$$

where

$$c=4\left[\frac{(\Gamma(7/4))^2}{\Gamma(7/2)}\right]^{4/7},\quad A_0=\frac{14\sqrt{\pi}}{5},\quad A_1=\frac{7\sqrt{\pi}}{3}.$$

We find, for $\beta=7/4$, the results shown in Table 6, which compares very well with the numerics.

For $\beta=1$, we have,

$$\lambda_1(N)\sim 2^{7/2}\pi^{3/2}\mathrm{e}N^{1/4}\mathrm{e}^{-4\sqrt{N}}.$$

From which we have the results in Table 7, again in very good agreement with numerics.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

Table 6: $\beta=7/4$: numerical vs. theoretical result.

N	Numerical λ_1	theoretical λ_1	error
100	1.6976×10^{-45}	1.7424×10^{-45}	2.64%
300	1.4844×10^{-102}	1.5074×10^{-102}	1.55%
500	6.7121×10^{-149}	6.7929×10^{-149}	1.20%
1000	3.6209×10^{-246}	3.6514×10^{-246}	0.84%
1500	6.4232×10^{-330}	6.4667×10^{-330}	0.68%
2000	6.2011×10^{-406}	6.2369×10^{-406}	0.58%
2500	1.1483×10^{-476}	1.1542×10^{-476}	0.51%

Table 7: $\beta=1$: numerical vs. theoretical result.

N	Numerical λ_1	theoretical λ_1	error
100	2.1079×10^{-15}	2.3006×10^{-15}	9.14%
300	5.5215×10^{-28}	5.8083×10^{-28}	5.19%
500	1.1138×10^{-36}	1.1585×10^{-36}	4.01%
1000	1.0892×10^{-52}	1.1200×10^{-52}	2.83%
1500	5.4593×10^{-65}	5.5852×10^{-65}	2.31%

Table 8: $\beta=1/2$: numerical vs. theoretical result.

N	Numerical λ_1	theoretical λ_1	error
100	0.27397	0.40360	47.32%
300	0.15837	0.21365	34.91%
500	0.12047	0.15855	31.61%
1000	0.082087	0.10555	28.58%
1500	0.066295	0.083130	25.39%

At $\beta=1/2$ the classical moment problem is at the verge of being indeterminate, a conjectured smallest eigenvalue for large N reads

$$\lambda_1(N)\sim 8\pi\,\frac{\sqrt{\ln(4\pi N\mathrm{e})}}{(4\pi N\mathrm{e})^{2/\pi}}$$

from which we find, for $\beta=1/2$, the results shown in Table 8.

14 Conclusion

Large ill-conditioned Hankel matrices present an unusual mix of computation that are not readily solved with traditional approaches. We have explored a space of potential algorithmic solutions to determine which approach provides the greatest efficiency given the special precision requirements of the problem. Surprisingly, we found that a direct

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

method is more effective than the more sophisticated algorithms that are commonly employed. Our parallel implementation of this algorithm uses shared memory and message passing to optimize performance, taking into account the unusual mix of computation and communication involved in working with extreme precision values. Some of the related optimizations involve the distribution and blocking of the data, sending computed columns during the idle communication time that is available during the lengthy local computations, sometimes transmitting partial results for which the computations are completed by the receiver, and eliminating a barrier that counter intuitively improves performance. The result is a parallel implementation that scales nearly perfectly, and that can be made nearly insensitive to network latency while taking maximum advantage of network throughput. The algorithm we presented has thus proved to be an elegant, efficient, fast and scalable solution to the problem, with guaranteed numeric results. As a result, we have been able to considerably extend the known set of solutions for these matrices using a modest 440 node cluster. The apparent continued scalability of the approach at last puts solution of larger Hankel matrices associated with particularly interesting problems within reach of modern hardware.

Acknowledgments

This work is supported in part by the National Science Foundation under Award No. CCF-1217590 and NFS grant #CNS-0619337 and by FDCT 077/2012/A3. Any opinions, findings conclusions or recommendations expressed here are the authors and do not necessarily reflect those of the sponsors.

References

[1] N. I. Akhiezer, The classical moment problem and some related questions in analysis, Oliver and Boyd, Edinburgh, 1965.
[2] G. W. Anderson, A Guionnet and O. Zeitouni, An introduction to random matrices, The University Press, Cambridge, 2010.
[3] Z. Bai, Y. Chen and Y.-C. Liang, Random matrix theory and its applications, Lecture Notes Series. Institute for Mathematical Sciences. National University of Singapore. Vol. 18, 2009.
[4] C. Berg, Y. Chen and M.E.H. Ismail, Small eigenvalues of large Hankel matrices: the indeterminate case, Math. Scand., vol. 91, 67–81, 2002.
[5] G. Blower, Random Matrices: High dimensional phenomena, The Univeristy Press, Cambridge, 2009.
[6] C. Berg and R. Szwarc, The smallest eigenvalue of Hankel matrices, Constructive Approximation, vol. 34, 107–133, 2011.
[7] R. L. Burden, J. D. Faires, Numerical Analysis, 4th edition. PWS-KENT Publishing Company, Boston, Massachusetts, 1989.
[8] Y. Chen, N. Haq and M. McKay, Random matrix models, double-time Painleve equations, and wireless relaying, J. Math. Phys., vol. 54, 063506, 2013.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press

[9] Y. Chen and N. D. Lawrence, Small eigenvalues of large Hankel matrices, J. Phys. A: Math. Gen., vol. 32, 7305–7315, 1999.
[10] Y, Chen and D. S. Lubinsky, Smallest eigenvalues of Hankel matrices for exponential weights, J. Math. Anal. Appl., vol. 293, 476-495, 2004.
[11] Y, Chen and M. McKay, Coulomb fluid, Painlevé Transcendents, and the information theory of MIMO systems, IEEE Trans. Information Theory, vol. 58, 4594–4634, 2012.
[12] P. Dharmawansa, M. McKay and Y. Chen, Distributionsof Demmel and Related Condition Numbers, Siam J. Matrix Anal., vol. 34, 257–279, 2012.
[13] J. W. Demmel, M. T. Heath, H. A. van der Vorst. Parallel Numerical Linear Algebra, ACTA Numerica, 1992.
[14] GNU Open Source Community. The GNU Multiple Precision Arithmetic Library. http://www.gmplib.org/
[15] S. V. Kerov, Asymptotic representation theory of symmetric group and its application in analysis, American Mathematical Society, 2003.
[16] M. G. Krein and A. A. Nudelman. Markov moment problems and extremal problems, American Mathematical Society, 1977.
[17] S. Li, M. Mckay and Y. Chen, On the distribution of MIMO mutual information: An indepth Painlevé based characterization, IEEE Trans. Information Theory, vol. 59, 5271–5296, 2013.
[18] D. S. Lubinsky, Condition of Hankel matrices for exponential weights, J. Math. Anal. Appl., vol. 314, 266285, 2006.
[19] M. L. Mehta, Random Matrices, 3rd edition. Elsevier, Singapore, 2006.
[20] R. E. Moore, Interval Analysis. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1966.
[21] B. N. Parlett, The Symmetric Eigenvalue Problem, Classics in Applies Mathematics; 20. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1997.
[22] J. Stoer, R. Bulirsch, Introduction to Numerical Analysis, 3rd edition. Texts in Applied Mathematics 12. Springer Science and Business Media, LLC, New York, New York, 2002.
[23] A. A. Quadeer, R. H. Louie, K. Shekhar, A. K. Chakraborty, M. Hsing, M. R. Mckay, Statistical linkage analysis of substitutions in patient-derived sequences of genotype 1a Hepatitis C Virius non-structural protein 3 exposes targets for immunogen design. J. Virology, 2014, Apr. 23.
[24] G. Szegö, On some Hermitian forms associated with two given curves of the complex plane, Trans. Amer. Math. Soc., vol. 40, 450–461, 1936.
[25] H. Widom, Rapidly Increasing Kernels, Proc. Amer. Math. Soc. vol., 14, 501–506, 1963.
[26] H. Widom and H. Wilf, Small eigenvalues of large Hankel matrices, Proc. Amer. Math. Soc., vol. 17, 338–344, 1966.

https://doi.org/10.4208/cicp.260514.231214a Published online by Cambridge University Press