AN ALGORITHM FOR THE PRINCIPAL COMPONENT ANALYSIS OF LARGE DATA SETS

NATHAN HALKO*, PER-GUNNAR MARTINSSON†, YOEL SHKOLNISKY‡, AND MARK TYGERT§

Abstract. Recently popularized randomized methods for principal component analysis (PCA) efficiently and reliably produce nearly optimal accuracy — even on parallel processors — unlike the classical (deterministic) alternatives. We adapt one of these randomized methods for use with data sets that are too large to be stored in random-access memory (RAM). (The traditional terminology is that our procedure works efficiently *out-of-core*.) We illustrate the performance of the algorithm via several numerical examples. For example, we report on the PCA of a data set stored on disk that is so large that less than a hundredth of it can fit in our computer's RAM.

Key words. algorithm, principal component analysis, PCA, SVD, singular value decomposition, low rank

AMS subject classifications. 65F15, 65C60, 68W20

1. Introduction. Principal component analysis (PCA) is among the most popular tools in machine learning, statistics, and data analysis more generally. PCA is the basis of many techniques in data mining and information retrieval, including the latent semantic analysis of large databases of text and HTML documents described in [1]. In this paper, we compute PCAs of very large data sets via a randomized version of the block Lanczos method, summarized in Section 2 below. The proofs in [4] and [9] show that this method requires only a couple of iterations to produce nearly optimal accuracy, with overwhelmingly high probability (the probability is independent of the data being analyzed, and is typically $1 - 10^{-15}$ or greater). The randomized algorithm has many advantages, as shown in [4] and [9]; the present article adapts the algorithm for use with data sets that are too large to be stored in the RAM of a typical computer system.

Computing a PCA of a data set amounts to constructing a singular value decomposition (SVD) that accurately approximates the matrix A containing the data being analyzed (possibly after suitably "normalizing" A, for example, by subtracting from each column its mean). That is, if A is $m \times n$, then we must find a positive integer $k < \min(m, n)$ and construct matrices U, Σ, and V such that

$$A \approx U \, \Sigma \, V^*, \tag{1.1}$$

with U being an $m \times k$ matrix whose columns are orthonormal, V being an $n \times k$ matrix whose columns are orthonormal, and Σ being a diagonal $k \times k$ matrix whose entries are all nonnegative. Most often, the relevant measure of the quality of the approximation in (1.1) is the spectral norm of the discrepancy $A - U \, \Sigma \, V^*$; see, for example, Section 2 below. The present article focuses on the spectral norm, though

*Department of Applied Mathematics, University of Colorado at Boulder, 526 UCB, Boulder, CO 80309-0526 (nathan.halko@colorado.edu)

†Department of Applied Mathematics, University of Colorado at Boulder, 526 UCB, Boulder, CO 80309-0526 (martinss@colorado.edu)

‡Department of Applied Mathematics, School of Mathematical Sciences, Tel Aviv University, Ramat Aviv, Tel Aviv, 69978, Israel (yoelsh@post.tau.ac.il)

§Courant Institute of Mathematical Sciences, NYU, 251 Mercer St., New York, NY 10012 (tygert@aya.yale.edu)

our methods produce similar accuracy in the Frobenius/Hilbert-Schmidt norm (see, for example, [4]).

In the present paper, the entries of all matrices are real valued; our techniques extend trivially to matrices whose entries are complex valued. The remainder of the article has the following structure: Section 2 outlines the algorithm. Section 3 details the implementation for very large matrices. Section 4 quantifies the main factors influencing the running-time of the algorithm. Section 5 illustrates the performance of the algorithm via several numerical examples. Section 6 draws some conclusions and proposes directions for further research.

2. Summary of the algorithm. In this section, we will construct a low-rank (say, rank k) approximation $U\,\Sigma\,V^*$ to any given real matrix A, such that

$$\|A - U\,\Sigma\,V^*\|_2 \le (Ckn)^{1/(4i+2)}\,\sigma_{k+1} \tag{2.1}$$

with very high probability (typically $1-10^{-15}$, independent of A), where m and n are the dimensions of the given $m \times n$ matrix A, U is a real $m \times k$ matrix whose columns are orthonormal, V is a real $n \times k$ matrix whose columns are orthonormal, Σ is a real diagonal $k \times k$ matrix whose entries are all nonnegative, σ_{k+1} is the $(k+1)^{\text{st}}$ greatest singular value of A, and C is a constant determining the probability of failure (the probability of failure is small when $C = 10$, negligible when $C = 100$). In (2.1), i is any nonnegative integer such that $(i+2)k \le n$ (for most applications, $i = 1$ or $i = 2$ is sufficient; the algorithm becomes less efficient as i increases), and $\|A - U\,\Sigma\,V^*\|_2$ is the spectral (l^2-operator) norm of $A - U\,\Sigma\,V^*$, that is,

$$\|A - U\,\Sigma\,V^*\|_2 = \max_{x \in \mathbb{R}^n : \|x\|_2 \ne 0} \frac{\|(A - U\,\Sigma\,V^*)x\|_2}{\|x\|_2}, \tag{2.2}$$

$$\|x\|_2 = \sqrt{\sum_{j=1}^{n} (x_j)^2}. \tag{2.3}$$

To simplify the presentation, we will be assuming that $n \le m$ (if $n > m$, then the user can apply the algorithm to A^*). In this section, we summarize the algorithm; see [4] and [9] for a detailed discussion, including proofs of analogues of (2.1).

The minimal value of the spectral norm $\|A - B\|_2$, minimized over all rank-k matrices B, is σ_{k+1}. Hence, (2.1) guarantees that the algorithm summarized below produces approximations of nearly optimal accuracy.

To construct a rank-k approximation to A, we could apply A to about k random vectors, in order to identify the part of its range corresponding to the larger singular values. To enhance the decay of the singular values, we apply $A\,(A^*\,A)^i$ instead. Once we have identified most of the range of A, we perform some linear-algebraic manipulations in order to recover an approximation satisfying (2.1).

A numerically stable realization of the scheme outlined in the preceding paragraph is the following. We choose an integer $l \ge k$ such that $(i+1)l \le n - k$ (it is generally sufficient to choose $l = k + 2$; increasing l can improve the accuracy marginally, but increases computational costs), and make the following six steps:

1. Using a random number generator, form a real $n \times l$ matrix G whose entries are independent, identically distributed Gaussian random variables of zero mean and unit variance, and compute the $m \times l$ matrices $H^{(0)}$, $H^{(1)}$, ..., $H^{(i-1)}$, $H^{(i)}$ defined via the formulae

$$H^{(0)} = A\,G, \tag{2.4}$$

$$H^{(1)} = A\,(A^{\top} H^{(0)}), \tag{2.5}$$

$$H^{(2)} = A\,(A^{\top} H^{(1)}), \tag{2.6}$$

$$\vdots$$

$$H^{(i)} = A\,(A^{\top} H^{(i-1)}). \tag{2.7}$$

   Form the $m \times ((i+1)l)$ matrix

$$H = \left(\begin{array}{c|c|c|c|c} H^{(0)} & H^{(1)} & \ldots & H^{(i-1)} & H^{(i)} \end{array} \right). \tag{2.8}$$

2. Using a pivoted QR-decomposition, form a real $m \times ((i+1)l)$ matrix Q whose columns are orthonormal, such that there exists a real $((i+1)l) \times ((i+1)l)$ matrix R for which

$$H = Q\,R. \tag{2.9}$$

   (See, for example, Chapter 5 in [3] for details concerning the construction of such a matrix Q.)
3. Compute the $n \times ((i+1)l)$ product matrix

$$T = A^{\top}\,Q. \tag{2.10}$$

4. Form an SVD of T,

$$T = \tilde{V}\,\tilde{\Sigma}\,W^{\top}, \tag{2.11}$$

   where $\tilde{V}$ is a real $n \times ((i+1)l)$ matrix whose columns are orthonormal, W is a real $((i+1)l) \times ((i+1)l)$ matrix whose columns are orthonormal, and $\tilde{\Sigma}$ is a real diagonal $((i+1)l) \times ((i+1)l)$ matrix such that $\tilde{\Sigma}_{1,1} \geq \tilde{\Sigma}_{2,2} \geq \cdots \geq \tilde{\Sigma}_{(i+1)l-1,(i+1)l-1} \geq \tilde{\Sigma}_{(i+1)l,(i+1)l} \geq 0$. (See, for example, Chapter 8 in [3] for details concerning the construction of such an SVD.)
5. Compute the $m \times ((i+1)l)$ product matrix

$$\tilde{U} = Q\,W. \tag{2.12}$$

6. Retrieve the leftmost $m \times k$ block U of $\tilde{U}$, the leftmost $n \times k$ block V of $\tilde{V}$, and the leftmost uppermost $k \times k$ block Σ of $\tilde{\Sigma}$. The product $U\,\Sigma\,V^{\top}$ then approximates A as in (2.1) (we omit the proof; see [4] for proofs of similar bounds).

REMARK 2.1. In the present paper, we assume that the user specifies the rank k of the approximation $U\,\Sigma\,V^{\top}$ being constructed. See [4] for techniques for determining the rank k adaptively, such that the accuracy $\|A - U\,\Sigma\,V^{\top}\|_2$ meets a user-specified threshold.

REMARK 2.2. Variants of the fast Fourier transform (FFT) permit additional accelerations; see [4], [6], and [10]. However, these accelerations have negligible effect on the algorithm running out-of-core. For out-of-core computations, the simpler techniques of the present paper are preferable.

3. Out-of-core computations. With suitably large matrices, some steps in Section 2 above require either storage on disk, or on-the-fly computations obviating the need for storing all the entries of the $m \times n$ matrix A being approximated. Conveniently, Steps 2, 4, 5, and 6 involve only matrices having $\mathcal{O}((i+1)\,l\,(m+n))$ entries; we perform these steps using only storage in RAM. However, Steps 1 and 3 involve A, which has mn entries; we perform Steps 1 and 3 differently depending on how A is provided, as detailed below in Subsections 3.1 and 3.2.

3.1. Computations with on-the-fly evaluation of matrix entries. If A does not fit in memory, but we have access to a computational routine that can evaluate each entry (or row or column) of A individually, then obviously we can perform Steps 1 and 3 using only storage in RAM. Every time we evaluate an entry (or row or column) of A in order to compute part of a matrix product involving A or A^*, we immediately perform all computations associated with this particular entry (or row or column) that contribute to the matrix product.

3.2. Computations with storage on disk. If A does not fit in memory, but is provided as a file on disk, then Steps 1 and 3 require access to the disk. We assume for definiteness that A is provided in row-major format on disk (if A is provided in column-major format, then we apply the algorithm to A^* instead). To construct the matrix product in (2.4), we retrieve as many rows of A from disk as will fit in memory, form their inner products with the appropriate columns of G, store the results in $H^{(0)}$, and then repeat with the remaining rows of A. To construct the matrix product in (2.10), we initialize all entries of T to zeros, retrieve as many rows of A from disk as will fit in memory, add to T the transposes of these rows, weighted by the appropriate entries of Q, and then repeat with the remaining rows of A. We construct the matrix product in (2.5) similarly, forming $F = A^* H^{(0)}$ first, and $H^{(1)} = A\,F$ second. Constructing the matrix products in (2.6)–(2.7) is analogous.

4. Computational costs. In this section, we tabulate the computational costs of the algorithm described in Section 2, for the particular out-of-core implementations described in Subsections 3.1 and 3.2. We will be using the notation from Section 2, including the integers i, k, l, m, and n, and the $m \times n$ matrix A.

REMARK 4.1. For most applications, $i \leq 2$ suffices. In contrast, the classical Lanczos algorithm generally requires many iterations in order to yield adequate accuracy, making the computational costs of the classical algorithm prohibitive for out-of-core (or parallel) computations (see, for example, Chapter 9 in [3]).

4.1. Costs with on-the-fly evaluation of matrix entries. We denote by C_A the number of floating-point operations (flops) required to evaluate all nonzero entries in A. We denote by N_A the number of nonzero entries in A. With on-the-fly evaluation of the entries of A, the six steps of the algorithm described in Section 2 have the following costs:

1. Forming $H^{(0)}$ in (2.4) costs $C_A + \mathcal{O}(l\,N_A)$ flops. Forming any of the matrix products in (2.5)–(2.7) costs $2C_A + \mathcal{O}(l\,N_A)$ flops. Forming H in (2.8) costs $\mathcal{O}(ilm)$ flops. All together, Step 1 costs $(2i+1)\,C_A + \mathcal{O}(il(m+N_A))$ flops.
2. Forming Q in (2.9) costs $\mathcal{O}(i^2 l^2 m)$ flops.
3. Forming T in (2.10) costs $C_A + \mathcal{O}(il\,N_A)$ flops.
4. Forming the SVD of T in (2.11) costs $\mathcal{O}(i^2 l^2 n)$ flops.
5. Forming $\tilde{U}$ in (2.12) costs $\mathcal{O}(i^2 l^2 m)$ flops.
6. Forming U, Σ, and V in Step 6 costs $\mathcal{O}(k(m+n))$ flops.

Summing up the costs for the six steps above, and using the fact that $k \leq l \leq n \leq m$, we see that the full algorithm requires

$$C_{\text{on-the-fly}} = 2(i+1)\,C_A + \mathcal{O}(il\,N_A + i^2 l^2 m) \tag{4.1}$$

flops, where C_A is the number of flops required to evaluate all nonzero entries in A, and N_A is the number of nonzero entries in A.

4.2. Costs with storage on disk. We denote by j the number of floating-point words of random-access memory (RAM) available to the algorithm. With A stored on disk, the six steps of the algorithm described in Section 2 have the following costs (assuming for convenience that $j > 2\,(i+1)\,l\,(m+n)$):

1. Forming $H^{(0)}$ in (2.4) requires at most $\mathcal{O}(lmn)$ floating-point operations (flops), $\mathcal{O}(mn/j)$ disk accesses/seeks, and a total data transfer of $\mathcal{O}(mn)$ floating-point words. Forming any of the matrix products in (2.5)–(2.7) also requires $\mathcal{O}(lmn)$ flops, $\mathcal{O}(mn/j)$ disk accesses/seeks, and a total data transfer of $\mathcal{O}(mn)$ floating-point words. Forming H in (2.8) costs $\mathcal{O}(ilm)$ flops. All together, Step 1 requires $\mathcal{O}(ilmn)$ flops, $\mathcal{O}(imn/j)$ disk accesses/seeks, and a total data transfer of $\mathcal{O}(imn)$ floating-point words.
2. Forming Q in (2.9) costs $\mathcal{O}(i^2 l^2 m)$ flops.
3. Forming T in (2.10) requires $\mathcal{O}(ilmn)$ floating-point operations, $\mathcal{O}(mn/j)$ disk accesses/seeks, and a total data transfer of $\mathcal{O}(mn)$ floating-point words.
4. Forming the SVD of T in (2.11) costs $\mathcal{O}(i^2 l^2 n)$ flops.
5. Forming $\tilde{U}$ in (2.12) costs $\mathcal{O}(i^2 l^2 m)$ flops.
6. Forming U, Σ, and V in Step 6 costs $\mathcal{O}(k(m+n))$ flops.

Summing up the costs for the six steps above, and using the fact that $k \leq l \leq n \leq m$, we see that the full algorithm requires

$$C_{\text{flops}} = \mathcal{O}(ilmn + i^2 l^2 m) \tag{4.2}$$

flops,

$$C_{\text{accesses}} = \mathcal{O}(imn/j) \tag{4.3}$$

disk accesses/seeks (where j is the number of floating-point words of RAM available to the algorithm), and a total data transfer of

$$C_{\text{words}} = \mathcal{O}(imn) \tag{4.4}$$

floating-point words (more specifically, $C_{\text{words}} \approx (2i+1)mn$).

5. Numerical examples. In this section, we describe the results of several numerical tests of the algorithm of the present paper.

We set $l = k+2$ for all examples, setting $i = 3$ for the first two examples, and $i = 1$ for the last two, where i, k, and l are the parameters from Section 2 above. We ran all examples on a laptop with 1.5 GB of random-access memory (RAM), connected to an external hard drive via USB 2.0. The processor was a single-core 32-bit 2-GHz Intel Pentium M, with 2 MB of L2 cache. We ran all examples in Matlab 7.4.0, storing floating-point numbers in RAM using IEEE standard double-precision variables (requiring 8 bytes per real number), and on disk using IEEE standard single-precision variables (requiring 4 bytes per real number).

All our numerical experiments indicate that the quality and distribution of the pseudorandom numbers have little effect on the accuracy of the algorithm of the present paper. We used Matlab's built-in pseudorandom number generator for all results reported below.

5.1. Synthetic data. In this subsection, we illustrate the performance of the algorithm with the principal component analysis of three examples, including a computational simulation.

For the first example, we apply the algorithm to the $m \times n$ matrix

$$A = F\,S\,G, \tag{5.1}$$

where F and G are $m \times m$ and $n \times n$ unitary discrete cosine transforms of the second type (DCT-II), and S is an $m \times n$ matrix whose entries are zero off the main diagonal, with

$$S_{j,j} = \begin{cases} 10^{-4(j-1)/19}, & j = 1, 2, \ldots, 19, \text{ or } 20 \\ 10^{-4}/(j-20)^{1/10}, & j = 21, 22, \ldots, n-1, \text{ or } n. \end{cases} \tag{5.2}$$

Clearly, $S_{1,1}$, $S_{2,2}$, $\ldots$, $S_{n-1,n-1}$, $S_{n,n}$ are the singular values of A.

For the second example, we apply the algorithm to the $m \times n$ matrix

$$A = F\,S\,G, \tag{5.3}$$

where F and G are $m \times m$ and $n \times n$ unitary discrete cosine transforms of the second type (DCT-II), and S is an $m \times n$ matrix whose entries are zero off the main diagonal, with

$$S_{j,j} = \begin{cases} 1.00, & j = 1, 2, \text{ or } 3 \\ 0.67, & j = 4, 5, \text{ or } 6 \\ 0.34, & j = 7, 8, \text{ or } 9 \\ 0.01, & j = 10, 11, \text{ or } 12 \\ 0.01 \cdot \frac{n-j}{n-13}, & j = 13, 14, \ldots, n-1, \text{ or } n. \end{cases} \tag{5.4}$$

Clearly, $S_{1,1}$, $S_{2,2}$, $\ldots$, $S_{n-1,n-1}$, $S_{n,n}$ are the singular values of A.

Table 1a summarizes results of applying the algorithm to the first example, storing on disk the matrix being approximated. Table 1b summarizes results of applying the algorithm to the first example, generating on-the-fly the columns of the matrix being approximated.

Table 2a summarizes results of applying the algorithm to the second example, storing on disk the matrix being approximated. Table 2b summarizes results of applying the algorithm to the second example, generating on-the-fly the columns of the matrix being approximated.

The headings of the tables have the following meanings:

- m is the number of rows in the matrix A being approximated.
- n is the number of columns in the matrix A being approximated.
- k is the parameter from Section 2 above; k is the rank of the approximation being constructed.
- t_{gen} is the time in seconds required to generate and store on disk the matrix A being approximated.
- t_{PCA} is the time in seconds required to compute the rank-k approximation (the PCA) provided by the algorithm of the present paper.
- ε_0 is the spectral norm of the difference between the matrix A being approximated and its best rank-k approximation.
- ε is an estimate of the spectral norm of the difference between the matrix A being approximated and the rank-k approximation produced by the algorithm of the present paper. The estimate ε of the error is accurate to within a

TABLE 1A
On-disk storage of the first example.

m	n	k	t_{gen}	t_{PCA}	ε_0	ε
2E5	2E5	16	2.7E4	6.6E4	4.3E-4	4.3E-4
2E5	2E5	20	2.7E4	6.6E4	1.0E-4	1.0E-4
2E5	2E5	24	2.7E4	6.9E4	1.0E-4	1.0E-4

TABLE 1B
On-the-fly generation of the first example.

m	n	k	t_{PCA}	ε_0	ε
2E5	2E5	16	7.7E1	4.3E-4	4.3E-4
2E5	2E5	20	1.0E2	1.0E-4	1.0E-4
2E5	2E5	24	1.3E2	1.0E-4	1.0E-4

factor of two with extraordinarily high probability; the expected accuracy of the estimate ε of the error is about 10%, relative to the best possible error ε_0 (see [5]). The appendix below details the construction of the estimate ε of the spectral norm of $D = A - U\Sigma V^{\top}$, where A is the matrix being approximated, and $U\Sigma V^{\top}$ is the rank-k approximation produced by the algorithm of the present paper.

For the third example, we apply the algorithm with $k = 3$ to an $m \times 1000$ matrix whose rows are independent and identically distributed (i.i.d.) realizations of the random vector

$$\alpha\, w_1 + \beta\, w_2 + \gamma\, w_3 + \delta, \tag{5.5}$$

where w_1, w_2, and w_3 are orthonormal 1×1000 vectors, δ is a 1×1000 vector whose entries are i.i.d. Gaussian random variables of mean zero and standard deviation 0.1, and (α, β, γ) is drawn at random from inside an ellipsoid with axes of lengths $a = 1.5$, $b = 1$, and $c = 0.5$, specifically,

$$\alpha = a\, r\, (\cos\varphi)\, \sin\theta, \tag{5.6}$$

$$\beta = b\, r\, (\sin\varphi)\, \sin\theta, \tag{5.7}$$

$$\gamma = c\, r \cos\theta, \tag{5.8}$$

with r drawn uniformly at random from $[0, 1]$, φ drawn uniformly at random from $[0, 2\pi]$, and θ drawn uniformly at random from $[0, \pi]$. We obtained w_1, w_2, and w_3 by applying the Gram-Schmidt process to three vectors whose entries were i.i.d. centered Gaussian random variables; w_1, w_2, and w_3 are exactly the same in every row, whereas the realizations of α, β, γ, and δ in the various rows are independent. We generated all the random numbers on-the-fly using a high-quality pseudorandom number generator; whenever we had to regenerate exactly the same matrix (as the algorithm requires with $i > 0$), we restarted the pseudorandom number generator with the original seed.

Figure 1a plots the inner product (*i.e.*, correlation) of w_1 in (5.5) and the (normalized) left singular vector associated with the greatest singular value produced by the algorithm of the present article. Figure 1a also plots the inner product of w_2 in (5.5) and the (normalized) left singular vector associated with the second greatest singular value, as well as the inner product of w_3 and the (normalized) left singular vector associated with the third greatest singular value. Needless to say, the inner products

TABLE 2A
On-disk storage of the second example.

m	n	k	t_{gen}	t_{PCA}	ε_0	ε
2E5	2E5	12	2.7E4	6.3E4	1.0E-2	1.0E-2
2E5	2E4	12	1.9E3	6.1E3	1.0E-2	1.0E-2
5E5	8E4	12	2.2E4	6.5E4	1.0E-2	1.0E-2

TABLE 2B
On-the-fly generation of the second example.

m	n	k	t_{PCA}	ε_0	ε
2E5	2E5	12	5.5E1	1.0E-2	1.0E-2
2E5	2E4	12	2.7E1	1.0E-2	1.0E-2
5E5	8E4	12	7.9E1	1.0E-2	1.0E-2

(*i.e.*, correlations) all tend to 1, as m increases — as they should. Figure 1b plots the time required to run the algorithm of the present paper, generating on-the-fly the entries of the matrix being processed. The running-time is roughly proportional to m, in accordance with (4.1).

5.2. Measured data. In this subsection, we illustrate the performance of the algorithm with the principal component analysis of images of faces.

We apply the algorithm with $k = 50$ to the 393,216 $\times$ 102,042 matrix whose columns consist of images from the FERET database of faces described in [7] and [8], with each image duplicated three times. For each duplicate, we set the values of a random choice of 10% of the pixels to numbers chosen uniformly at random from the integers 0, 1, ..., 254, 255; all pixel values are integers from 0, 1, ..., 254, 255. Before processing with the algorithm of the present article, we "normalized" the matrix by subtracting from each column its mean, then dividing the resulting column by its Euclidean norm. The algorithm of the present paper required 12.3 hours to process all 150 GB of this data set stored on disk, using the laptop computer with 1.5 GB of RAM described earlier (at the beginning of Section 5).

Figure 2a plots the computed singular values. Figure 2b displays the computed "eigenfaces" (that is, the left singular vectors) corresponding to the five greatest singular values.

6. Conclusion. The present article describes techniques for the principal component analysis of data sets that are too large to be stored in random-access memory (RAM), and illustrates the performance of the methods on data from various sources, including standard test sets, numerical simulations, and physical measurements. Several of our data sets stored on disk were so large that less than a hundredth of any of them could fit in our computer's RAM; nevertheless, the scheme always succeeded. Theorems, their rigorous proofs, and their numerical validations all demonstrate that the algorithm of the present paper produces nearly optimal spectral-norm accuracy. Moreover, similar results are available for the Frobenius/Hilbert-Schmidt norm. Finally, the core steps of the procedures parallelize easily; with the advent of widespread multicore and distributed processing, exciting opportunities for further development and deployment abound.

Appendix. In this appendix, we describe a method for estimating the spectral norm $\|D\|_2$ of a matrix D. This procedure is particularly useful for checking whether an algorithm has produced a good approximation to a matrix (for this purpose, we choose D to be the difference between the matrix being approximated and its approx-

imation). The procedure is a version of the classic power method, and so requires the application of D and $D^\top$ to vectors, but does not use D in any other way. Though the method is classical, its probabilistic analysis summarized below was introduced fairly recently in [2] and [5] (see also Section 3.4 of [10]).

Suppose that m and n are positive integers, and D is a real $m \times n$ matrix. We define $\omega^{(1)}$, $\omega^{(2)}$, $\omega^{(3)}$, ... to be real $n \times 1$ column vectors with independent and identically distributed entries, each distributed as a Gaussian random variable of zero mean and unit variance. For any positive integers j and k, we define

$$p_{j,k}(D) = \max_{1 \le q \le k} \sqrt{\frac{\|(D^\top D)^j \, \omega^{(q)}\|_2}{\|(D^\top D)^{j-1} \, \omega^{(q)}\|_2}}, \tag{6.1}$$

which is the best estimate of the spectral norm of D produced by j steps of the power method, started with k independent random vectors (see, for example, [5]). Naturally, when computing $p_{j,k}(D)$, we do not form $D^\top D$ explicitly, but instead apply D and $D^\top$ successively to vectors.

Needless to say, $p_{j,k}(D) \le \|D\|_2$ for any positive j and k. A somewhat involved analysis shows that the probability that

$$p_{j,k}(D) \ge \|D\|_2/2 \tag{6.2}$$

is greater than

$$1 - \left(\frac{2n}{(2j-1) \cdot 16^j} \right)^{k/2}. \tag{6.3}$$

The probability in (6.3) tends to 1 very quickly as j increases. Thus, even for fairly small j, the estimate $p_{j,k}(D)$ of the value of $\|D\|_2$ is accurate to within a factor of two, with very high probability; we used $j = 6$ for all numerical examples in this paper. We used the procedure of this appendix to estimate the spectral norm in (2.1), choosing $D = A - U \, \Sigma \, V^\top$, where A, U, Σ, and V are the matrices from (2.1). We set k for $p_{j,k}(D)$ to be equal to the rank of the approximation $U \, \Sigma \, V^\top$ being constructed.

For more information, see [2], [5], or Section 3.4 of [10].

Acknowledgements. We would like to thank the mathematics departments of UCLA and Yale, especially for their support during the development of this paper and its methods. Nathan Halko and Per-Gunnar Martinsson were supported in part by NSF grants DMS0748488 and DMS0610097. Mark Tygert was supported in part by an Alfred P. Sloan Research Fellowship. Portions of the research in this paper use the FERET database of facial images collected under the FERET program, sponsored by the DOD Counterdrug Technology Development Program Office.

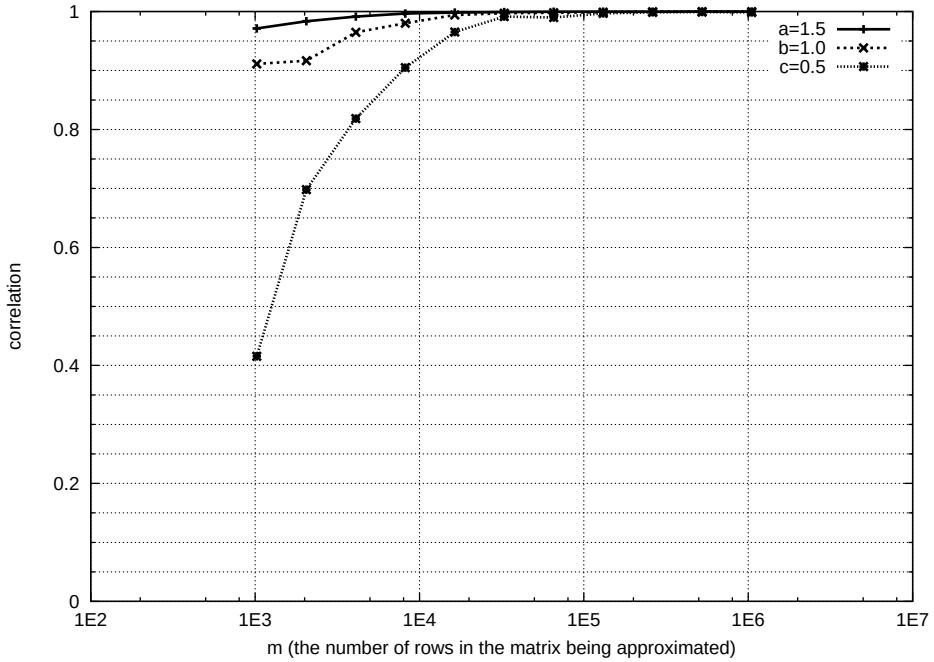

FIG. 1A. *Convergence for the third example (the computational simulation).*

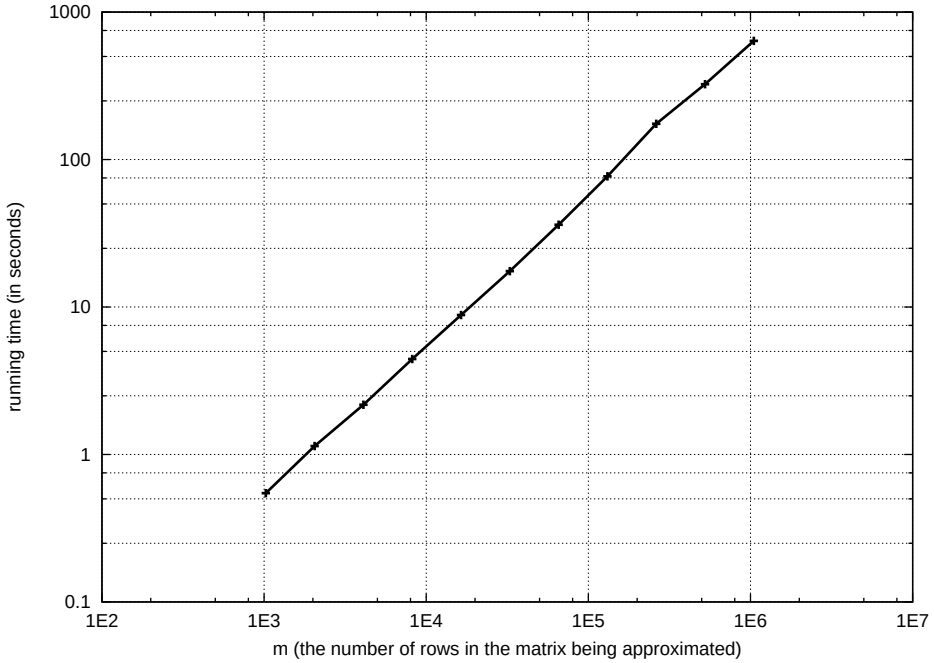

FIG. 1B. *Timing for the third example (the computational simulation).*

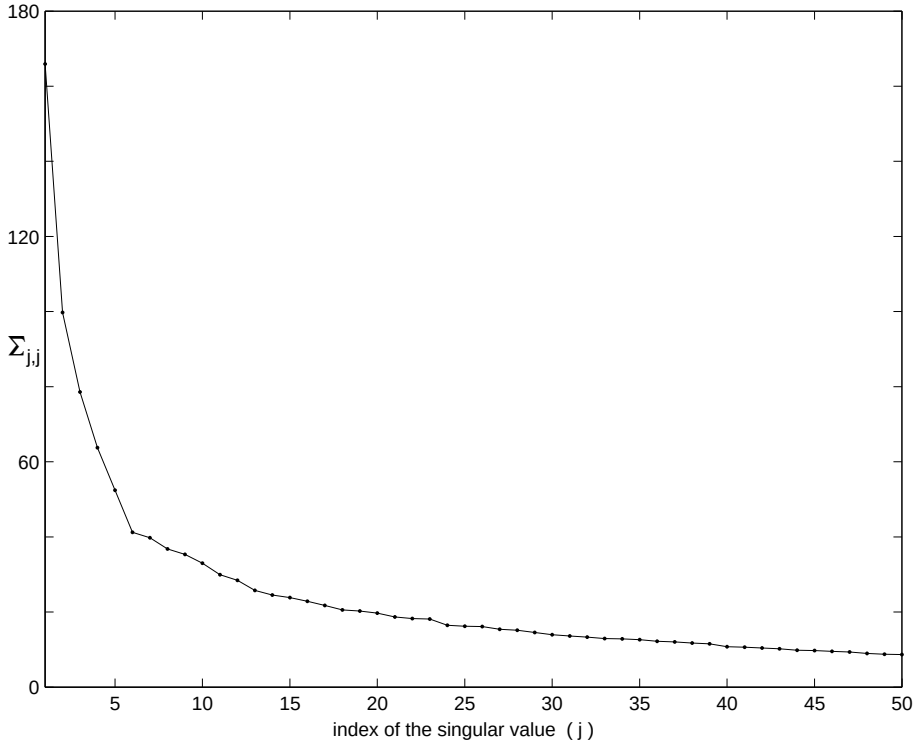

FIG. 2A. *Singular values computed for the fourth example (the database of images).*

FIG. 2B. *Dominant singular vectors computed for the fourth example (the database of images).*

REFERENCES

[1] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, *Indexing by latent semantic analysis*, J. Amer. Soc. Inform. Sci., 41 (1990), pp. 391–407.

[2] J. D. Dixon, *Estimating extremal eigenvalues and condition numbers of matrices*, SIAM J. Numer. Anal., 20 (1983), pp. 812–814.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, Maryland, 1996.

[4] N. Halko, P.-G. Martinsson, and J. Tropp, *Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions*, Technical report 0909.4061, arXiv, 2009. Available at http://arxiv.org.

[5] J. Kuczyński and H. Woźniakowski, *Estimating the largest eigenvalue by the power and Lanczos algorithms with a random start*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 1094–1122.

[6] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert, *Randomized algorithms for the low-rank approximation of matrices*, Proc. Natl. Acad. Sci. USA, 104 (2007), pp. 20167–20172.

[7] P. J. Phillips, H. Moon, S. A. Rizvi, and P. J. Rauss, *The FERET evaluation methodology for face recognition algorithms*, IEEE Trans. Pattern Anal. Machine Intelligence, 22 (2000), pp. 1090–1104.

[8] P. J. Phillips, H. Wechsler, J. Huang, and P. J. Rauss, *The FERET database and evaluation procedure for face recognition algorithms*, J. Image Vision Comput., 16 (1998), pp. 295–306.

[9] V. Rokhlin, A. Szlam, and M. Tygert, *A randomized algorithm for principal component analysis*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 1100–1124.

[10] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert, *A fast randomized algorithm for the approximation of matrices*, Appl. Comput. Harmon. Anal., 25 (2008), pp. 335–366.