Randomized methods for computing low-rank approximations of matrices

by

Nathan P. Halko

B.S., University of California, Santa Barbara, 2007

M.S., University of Colorado, Boulder, 2010

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Applied Mathematics

2012

This thesis entitled:
Randomized methods for computing low-rank approximations of matrices
written by Nathan P. Halko
has been approved for the Department of Applied Mathematics

Per-Gunnar Martinsson

Keith Julian

David M. Bortz

Francois G. Meyer

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Halko, Nathan P. (Ph. D., Applied Mathematics)

Randomized methods for computing low-rank approximations of matrices

Thesis directed by Professor Per-Gunnar Martinsson

Randomized sampling techniques have recently proved capable of efficiently solving many standard problems in linear algebra, and enabling computations at scales far larger than what was previously possible. The new algorithms are designed from the bottom up to perform well in modern computing environments where the expense of communication is the primary constraint. In extreme cases, the algorithms can even be made to work in a *streaming* environment where the matrix is not stored at all, and each element can be seen only once.

The dissertation describes a set of randomized techniques for rapidly constructing a low-rank approximation to a matrix. The algorithms are presented in a modular framework that first computes an approximation to the range of the matrix via randomized sampling. Secondly, the matrix is projected to the approximate range, and a factorization (SVD, QR, LU, etc.) of the resulting low-rank matrix is computed via variations of classical deterministic methods. Theoretical performance bounds are provided.

Particular attention is given to very large scale computations where the matrix does not fit in RAM on a single workstation. Algorithms are developed for the case where the original matrix must be stored *out-of-core* but where the factors of the approximation fit in RAM. Numerical examples are provided that perform Principal Component Analysis of a data set that is so large that less than one hundredth of it can fit in the RAM of a standard laptop computer. Furthermore, the dissertation presents a parallelized randomized scheme for computing a reduced rank Singular Value Decomposition. By parallelizing and distributing both the randomized sampling stage and the processing of the factors in the approximate factorization, the method requires an amount

of memory per node which is independent of both dimensions of the input matrix. Numerical experiments are performed on Hadoop clusters of computers in Amazon's Elastic Compute Cloud with up to 64 total cores. Finally, we directly compare the performance and accuracy of the randomized algorithm with the classical Lanczos method on extremely large, sparse matrices and substantiate the claim that randomized methods are superior in this environment.

Dedication

Acknowledgements

Contents

Chapter

Chapter 1

Introduction

Techniques based on randomized sampling have recently proved capable of solving many standard problems in linear algebra far more efficiently than classical techniques. This dissertation describes a set of such randomized techniques for rapidly constructing low-rank approximations to matrices.

The primary focus of the work presented has been to develop methods for numerical linear algebra that perform well in a modern computing environment where floating point operations are becoming ever cheaper, and *communication costs* are emerging as the real bottleneck. The dissertation demonstrates that randomized methods can dramatically reduce the amount of communication required to complete a computation when compared to classical deterministic methods. Particular attention is paid to *out-of-core* computing, and computing in a distributed environment.

1.1 Approximation by low rank matrices

A standard task in scientific computing is to determine for a given $m \times n$ matrix A, an approximate factorization

$$\underset{m \times n}{\mathsf{A}} \quad \approx \quad \underset{m \times k}{\mathsf{B}} \quad \underset{k \times n}{\mathsf{C}} \tag{1.1}$$

where the inner dimension k is called the *numerical rank* of the matrix. When the numerical rank is much smaller than either m or n, a factorization such as (2.1) allows the matrix to be stored inexpensively, and to be multiplied to vectors or other matrices rapidly. Factorizations such as (2.1) can also be used to analyze and synthesize the data.

Matrices for which the numerical rank k is much smaller than either m or n abound in applications. Examples include:

- Principal Component Analysis (PCA) is a basic tool in statistics and data mining. By projecting the data onto the orthogonal directions of maximal variance (principal components) we can visualize or explain the data in far fewer degrees of freedom than the ambient dimension. This amounts to computing a truncated Singular Value Decomposition [19].

- A particular application of PCA is a technique in facial recognition called Eigenfaces [33]. Representing faces as 2-D vectors, the eigenfaces are the significant features (principal components) among the known faces in the database. Just a small number of eigenfaces are needed to span the significant variations so faces can accurately be described by a weighted sum of each eigenface. The task of matching a face against the entire database reduces to comparing only the hand full of weights which greatly reduces the complexity of the problem.

- PCA is popular in the analysis of population genetic variation [26]. Though problems exist with its interpretation and more advanced nonlinear methods have been developed, PCA still remains an indispensable data analysis tool in the sciences.

- Estimating parameters via least squares in biology, engineering, and physics often leads to large over determined linear systems. Low rank factorization of the coefficient matrix leads to efficient solutions of the problem [29].

- The fast multipole method for rapidly evaluating potential fields relies on low rank approximations of continuum operators with exponentially decaying spectra [16].

- Laplacian Eigenmaps arise in image processing. A few eigenvectors of a graph derived from the image provide a non-linear embedding [5]. This is also an example of how linear approximations are used to solve non-linear problems.

- Low cost sensor networks provide a wealth of data. Their low cost enables many sensors to be densely deployed while limiting their on board data processing capability [4]. This inevitably results in lots of redundant data to be processed by the user.

As the list indicates, in most applications, the task that needs to be performed is not just to compute any factorization satisfying (2.1), but also to enforce additional constraints on the factors B and C. We will describe techniques for constructing several different specific factorizations, including:

The pivoted QR factorization: Each $m \times n$ matrix A of rank k admits a decomposition

$$\mathsf{A} = \mathsf{Q}\mathsf{R}, \tag{1.2}$$

where Q is an $m \times k$ orthonormal matrix, and R is an $k \times n$ *weakly upper-triangular* matrix. That is, there exists a permutation J of the numbers $\{1, 2, \ldots, n\}$ such that $\mathsf{R}(:, J)$ is upper triangular. Moreover, the diagonal entries of $\mathsf{R}(:, J)$ are weakly decreasing. See [15, §5.4.1] for details.

The singular value decomposition (SVD): Each $m \times n$ matrix A of rank k admits a factorization

$$\mathsf{A} = \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*, \tag{1.3}$$

where U is an $m \times k$ orthonormal matrix, V is an $n \times k$ orthonormal matrix, and $\mathsf{\Sigma}$ is a $k \times k$ nonnegative, diagonal matrix

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_k \end{bmatrix} \tag{1.4}$$

The numbers σ_j are called the *singular values* of A. They are arranged in weakly decreasing order:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq 0. \tag{1.5}$$

The columns of U and V are called *left singular vectors* and *right singular vectors*, respectively.

The interpolative decomposition (ID): The ID identifies a collection of k columns from a rank-k matrix A that span the range of A. To be precise, there exists an index set $J = [j_1, \ldots, j_k]$ such that

$$\mathsf{A} = \mathsf{A}(:, J)\, \mathsf{X}, \tag{1.6}$$

where X is a $k \times n$ matrix that contains the $k \times k$ identity matrix I_k. Specifically, $\mathsf{X}(:, J) = \mathsf{I}_k$. Furthermore, no entry of X has magnitude larger than one.

1.2 Existing methods for computing low-rank approximations

The best method to choose for computing factorizations depends on several factors and is not the same for every problem. Current state of the art procedures are very well suited and optimized for matrices that fit in RAM. That is, they achieve excellent accuracy and robustness at the cost of much random access to the elements of, or many passes over, the matrix A.

1.2.1 Truncated Factorizations

The singular value decomposition, when truncated to the leading k terms, provides the optimal approximation in terms of ℓ^2 error. This often is accomplished by bidiagonalizing the matrix via orthogonal transformations from the left and the right, and then using a quickly converging iterative process to compute the SVD of the resulting bidiagonal matrix [15]. It is not possible to extract the first k components without completing the entire factorization.

$$\underset{m \times n}{\mathsf{A}} = \underset{m \times m}{\mathsf{U}} \quad \underset{m \times n}{\Sigma} \quad \underset{n \times n}{\mathsf{V}^T} \approx \underset{m \times k}{\mathsf{U}(:, 1:k)} \quad \underset{k \times k}{\Sigma(1:k, 1:k)} \quad \underset{k \times n}{\mathsf{V}(:, 1:k)^T} \tag{1.7}$$

At a cost of $O(mn \cdot \min\{m, n\})$ this method provides the minimal error $\|\mathsf{A} - \mathsf{A}_{\text{approx}}\| = \sigma_{k+1}$ and has an advantage when the matrix is not too large and most of the singular values/vectors are needed.

1.2.2 Direct Access

The thin QR factorization is an $O(kmn)$ method that uses a subset of columns of A to build a basis for the column space or range of A. Variations differ based on how columns are selected as described below. These methods require extensive random access to the matrix elements but work very well when the matrix is small enough to fit into RAM.

The classic Golub-Businger algorithm [7] computes a pivoted QR decomposition using Householder reflections. Pivoting is done greedily by choosing the column with maximal norm at each iteration. In other words, the column least similar to the columns already selected is added to the basis. This approach is fast and easy to implement, but can fail in some pathological cases.

Gu and Eisenstat's strong rank-revealing QR [17] seeks to maximize the determinant of the upper left k block of R by interchanging columns of A. The j^{th} diagonal entry of R represents the norm of the j^{th} basis vector projected away from basis vectors $1, \ldots, j-1$. Maximizing the determinant, or sum of R's diagonal entries, seeks to minimize the similarity among basis vectors. This approach is more robust and accurate than Golub-Businger, but can fail to work quickly costing as much as a full QR decomposition in worst case scenarios.

1.2.3 Iterative methods

Iterative methods are appealing when random access to the matrix is unavailable or there is a fast matrix vector multiply as when A is sparse. They operate by only requiring the action of the matrix on a vector rather than random access to its elements. In large sparse systems, these methods avoid *fill-in* common to the methods of §1.2.2. Typically $O(k)$ passes over the matrix are needed.

Lanczos method [21] uses the Krylov subspace generated by initial starting vector ω:

$$\mathcal{K}(\mathsf{A}, \omega) = [\omega, \mathsf{A}\omega, \mathsf{A}^2\omega, \ldots] \tag{1.8}$$

for a symmetric $n \times n$ matrix. The coordinates of this basis map A to a tridiagonal matrix whose eigenvalues can be found very efficiently. The method can also be used on non-symmetric matrices by substituting $\mathsf{A}^T\mathsf{A}$ or $\mathsf{A}\mathsf{A}^T$. Additionally, Arnoldi and non-symmetric Lanczos methods generalize to working directly with a non-symmetric matrix A. In practice, stability issues are a huge concern and the method is sensitive and difficult to implement robustly. The method is also a serial process requiring many passes over the matrix A.

Simultaneous iteration, also orthogonal iteration, is a blocked generalization of the power method [30]. For a symmetric matrix A, an $O(k)$ dimensional subspace is constructed by repeatedly multiplying an initial subspace by A. Each iteration amplifies the components of the leading k eigenspace and continues until the iterated subspace converges to the leading k eigenvectors of A. To avoid collapse of all vectors onto the leading eigenvector (or eigenspace associated with the eigenvalue of largest magnitude) the subspace is orthogonalized between iterations. In addition,

solution of an intermediate eigenproblem is done as an acceleration step [32] to improve the error analysis.

These type of iterative methods are similar in spirit to the randomized techniques. The main contribution of this research was to consider carefully the properties of the initial starting subspace. This approach generated tight error bounds that required only a constant number of passes over the data improving greatly upon previous $O(k)$ pass methods. In some situations, only a *single* pass is required for high accuracy.

1.3 Changes in computer architecture, and the need for new algorithms

With the current shift towards a heterogeneous computing environment, the algorithms of §1.2.2-3 leave room for improvement. They are optimized to minimize flops and are not always well suited for computer architecture that is constrained by communication and data flow. In this environment reducing complexity and reorganizing computation become very important. For example:

Multi-core and multi-processor CPU speeds are no longer improving at Moore's pace leading to changes in architecture to keep up. Multi-core chips are standard in nearly all new computers. Access to parallel computing clusters are widely available to utilize parallelized data analysis techniques.

Communication Data transfer between processors, from disk, or through networks is slow and typically dominates the cost of parallel computation.

Amount of data Disk storage is very cheap leading to massive amounts of stored data. Data acquisition out paces the speed up in computing power. An additional problem with large data sets is noise and propagation of rounding errors.

It is necessary to develop new algorithms optimized for this changing environment.

1.4 Computational framework and problem formulation

The dissertation describes a set of techniques for computing approximate low rank factorizations to a given matrix A. In order to obtain a flexible set of algorithms that perform well in a broad range of different environments, we suggest that it is convenient to split the computational task into two stages:

Stage A Construct a low dimensional subspace that approximates the range of A. In other words, given an $\epsilon > 0$ compute a matrix Q with orthonormal columns so that $\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| < \epsilon$.

Stage B Given an approximate basis Q, use Q to help compute a factorization of A.

For example, consider computing an SVD via the two stage procedure. Stage A computes an approximate basis and Stage B utilizes the basis as follows:

- Form $\mathsf{B} = \mathsf{Q}^*\mathsf{A}$.
- Compute SVD of the rectangular matrix $\mathsf{B} = \tilde{\mathsf{U}}\Sigma\mathsf{V}^*$.
- Set $\mathsf{U} = \mathsf{Q}\tilde{\mathsf{U}}$ so that $\mathsf{A} \approx \mathsf{U}\Sigma\mathsf{V}^*$.

The randomized methods that form the focus of the current work are aimed primarily at solving Stage A above. It turns out that randomization allows this task to be executed in a very robust manner that permits minimal interaction with the matrix A.

Stage B is well suited for deterministic techniques whenever the matrices B and Q fit in RAM. We demonstrate in Section 5 of Chapter 2 that with slight modifications, the classical methods of numerical linear algebra can produce the QR and SVD factorizations (see equations (1.2) and (1.3)) identified as targets in Section 1.1. For the interpolative decomposition (1.6) which is important in data mining, Section 5 of Chapter 2 provides some substantially new techniques.

Chapter 3 develops methods for cases where matrix A is too large to fit in RAM of a single workstation. In this situation, interaction with A must be minimized since reading A from disk is prohibitively slow. The matrices B and Q are still assumed to fit in memory. This is often the case with large dense matrices. Chapter 4 extends the methods even further to accommodate cases where not only A cannot fit in RAM, but also matrices B and Q are too large for RAM. We develop a distributed variant of the algorithm that utilizes multiple processors and out-of-core techniques to construct B and Q. The SVD for example, is constructed via

- $\mathsf{B}\mathsf{B}^* = \tilde{\mathsf{U}}\Sigma^2\tilde{\mathsf{U}}^*$
- $\mathsf{U} = \mathsf{Q}\tilde{\mathsf{U}}$
- $\mathsf{V}^* = \Sigma^{-1}\tilde{\mathsf{U}}^*\mathsf{B}$

The small square matrix $\mathsf{B}\mathsf{B}^*$ fits comfortably in RAM and the factorization is computed with classical methods. The factors U and V are the same size as Q and B respectively and are computed out-of-core.

Remark 1. *The division of the computation in to two stages is convenient in many ways: It allows us to present the proposed algorithms in a clear manner, it allows for a unified error analysis, and — perhaps most importantly — it allows for the development of a set of different software modules that can be put together in different configurations to suit any particular computational environment. However, it should be noted that it is occasionally beneficial to deviate from the strict 2-stage template. For instance, we demonstrate in Section 5.5 of Chapter 2 an algorithm that executes Stages A and B simultaneously in order to obtain an algorithm that works in a streaming environment where you get to look at each element of the matrix only once (we call such algorithms* ***"single-pass"*** *in the text).*

1.5 Randomized algorithms for approximating the range of a matrix

In this section, we describe a set of randomized methods for executing the task we introduced as "Stage A" in Section 2.1.2, namely how to construct a basis for an approximation to the range of a given matrix. Early versions of these techniques were described in [24, 34], and are based on randomized sampling. As a consequence, they have in theory a non-zero risk of "failing" in the sense that the factorizations computed may not satisfy the requested tolerance. However, this risk can be controlled by the user, and can for a very low cost be rendered negligible. Failure risks of less that 10^{-15} are entirely standard.

1.5.1 Intuition

To get a feel for the randomized sampling technique let us consider an $n \times n$ matrix A of exact rank $k < n$. For simplicity assume that A is symmetric so we can easily discuss a set of real, orthogonal eigenvectors. Note that these restrictions are purely for convenience and the randomized sampling technique is effective on any matrix regardless of dimension or distribution of entries. It is most insightful to view the range of matrix A as a linear combination of its columns.

$$ran\,(\mathsf{A}) = \left\{ \sum_{j=1}^{n} \alpha_j \mathsf{A}(:\,, j) \colon \alpha_j \in \mathbb{R} \right\} \tag{1.9}$$

For any vector $\boldsymbol{\omega} \in \mathbb{R}^n$, the vector formed by multiplying by A is a vector in the range of A,

$$\mathbf{y} = \mathsf{A}\boldsymbol{\omega} = \textstyle\sum_j \omega_j \mathsf{A}(:, j). \tag{1.10}$$

We call $\mathbf{y}$ a *sample* vector of $ran(\mathsf{A})$. If we can collect k sample vectors that are linearly independent, then we can find a basis for $ran(\mathsf{A})$ by orthogonalizing the k samples. If $\boldsymbol{\omega}$ is a random vector in $\mathbb{R}^n$ then we call $\mathbf{y} = \mathsf{A}\boldsymbol{\omega}$ a *random sample* of $ran(\mathsf{A})$. For the purpose of illustration and analysis it is convenient to choose each entry of $\boldsymbol{\omega}$ from the standard normal distribution,

$$\boldsymbol{\omega} \in \mathbb{R}^n : \omega_i \sim \mathcal{N}\,(0, 1)\,. \tag{1.11}$$

Vectors chosen in this manner, known as *Gaussian* vectors, have important properties that help clarify analysis of the randomized sampling technique. First, k independently drawn Gaussian vectors $\{\boldsymbol{\omega}^{(i)}\}_{i=1}^{k}$, are almost surely in general position. This means that no linear combination of $\boldsymbol{\omega}$'s will fall in the null space of A and implies that the sample vectors $\{\mathbf{y}^{(i)} = A\boldsymbol{\omega}^{(i)}\}_{i=1}^{k}$ are linearly independent. Below we use this fact to show that k sample vectors span the range of A. Secondly, Gaussian vectors are *rotationally invariant* meaning that for any orthogonal matrix U and Gaussian vector $\boldsymbol{\omega}$, the vector $\mathsf{U}\boldsymbol{\omega}$ is also Gaussian.

Consider the eigenexpansion of sample vector $\mathbf{y}$

$$\mathbf{y} = \sum_j \lambda_j \langle \boldsymbol{\omega}, \mathbf{x}_j \rangle \mathbf{x}_j \tag{1.12}$$

where $\{\mathbf{x}_j\}_{j=1}^n$ are the eigenvectors of A. Pick an eigenvector $\mathbf{x}$ and then pick an orthogonal rotation matrix U such that $\mathsf{U}\mathbf{x} = \mathbf{e}_1$. The quantity $\langle \boldsymbol{\omega}, \mathbf{x}_j \rangle$ in equation (1.12) is itself a Gaussian random variable

$$\begin{aligned} \langle \boldsymbol{\omega}, \mathbf{x}_j \rangle &= \langle \mathsf{U}\boldsymbol{\omega}, \mathsf{U}\mathbf{x}_j \rangle \\ &= \langle \tilde{\boldsymbol{\omega}}, \mathbf{e}_1 \rangle \\ &= \tilde{\omega}_1 \qquad \text{the first element of } \tilde{\boldsymbol{\omega}} \\ &\sim \mathcal{N}(0,1) \end{aligned} \tag{1.13}$$

The probability of this quantity being indentically zero is non-existent and we get a non-zero contribution of the eigenvector in the sample vector $\mathbf{y}$. We can easily repeat this argument for each eigenvector to show that the sample vector has a non-zero contribution from *every* eigendirection. Therefore we do not miss any of the range in our sampling process. Putting this all together we have k linearly independent vectors in the k dimensional space $ran(\mathsf{A})$ and we can orthonormalize the samples to create a basis for $ran\,(\mathsf{A})$.

The situation is not so clear if A is not exactly rank k. It is possible that the last $n-k$ singular values reflect noise in our data or we simply want a low rank approximation of the matrix. Consider A to be the sum of an exact rank k matrix and a perturbation.

$$\begin{aligned} \mathsf{A} &= \sum_{j=1}^{k} \lambda_j \mathbf{x}_j \mathbf{x}_j^T + \sum_{j=k+1}^{n} \lambda_j \mathbf{x}_j \mathbf{x}_j^T \\ &= \mathsf{B} + \mathsf{E} \end{aligned} \tag{1.14}$$

We have reasoned that if $\mathsf{E} \equiv \mathsf{0}$, as in the exact rank case, then k sample vectors will capture every direction in $ran(\mathsf{A})$. For rank of A greater than k, $\mathsf{E} \neq \mathsf{0}$ and the perturbation shifts some of the direction of each sample vector outside the range of A. We already saw how the quantity $\langle \boldsymbol{\omega}, \mathbf{x}_j \rangle$ was uniform over j, implying that we are actually more likely to sample the range of E as we are the range of B (assuming $k \ll n$). However, the quantity

$$\lambda_j \langle \boldsymbol{\omega}, \mathbf{x}_j \rangle \tag{1.15}$$

is weighted to favor the components of B, the dominant rank k subspace we are after. Since the eigenvalues of E are generally much smaller than those of B, the sample vectors are naturally more heavily weighted with components of $ran(\mathsf{B})$ and the perturbation only adds a slight shift outside the desired range. In the event of an unlucky (and unlikely) sample, though the perturbation is small, it can also be the case that $\lambda_j \langle \boldsymbol{\omega}, \mathbf{x}_j \rangle$ is even smaller for $j = 1, \ldots, k$ and we miss the target range space. We can view this as a wasted sample and simply take another one to augment the sample pool. Since it is not known whether or not the sample is good at this point, the remedy is to assume that some samples are unlucky and take an extra p samples to account for them. The number of extra samples p is small and adds a great deal of accuracy for a small computational cost. Typically $p = 10$ is sufficient though we can take up to $p = k$ and not change the asymptotic qualities of the algorithm. Further analysis will provide insight to choosing p and its effect on accuracy of the method.

1.5.2 Basic Algorithm

Given an $m \times n$ matrix A, a target rank k, and an oversampling parameter p, the following algorithm computes an $m \times (k+p)$ matrix Q whose columns are orthonormal and form an approximate basis for the range of A.

INPUT $\mathsf{A}^{m\times n}$, k , p

(1) Draw a Gaussian $n \times (k+p)$ test matrix Ω.

(2) Form the product $\mathsf{Y} = \mathsf{A}\Omega$.

(3) Orthogonalize the columns of $\mathsf{Y} \mapsto \mathsf{Q}$

OUTPUT Q

This algorithm can also easily be summarized in Matlab notation

```
Q = orth(A*randn(n,k+p))
```
(1.16)

and we see that this is the first step of an orthogonal iteration. The novelty of the algorithm lies in Theorem 1.5.1 of Chapter 2 that depending on properties of A, this is the *only* step of the iteration we need to take.

Theorem 1.5.1. *Let* A *be a real valued* $m \times n$ *matrix. Choose a target rank* k *and an oversampling parameter* p *so that* $p \geq 4$ *and* $k+p \leq \min\{m,n\}$. *Execute the algorithm with a standard Gaussian test matrix* Ω. *Then the approximate basis* $\mathsf{Q}^{m\times(k+p)}$ *satisfies*

$$\begin{aligned}\mathbb{E}\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^{\mathsf{T}}\mathsf{A}\|_2 &\leq \left(1 + \sqrt{\frac{k}{p-1}}\right)\sigma_{k+1} + \frac{e\sqrt{k+p}}{p}\cdot\left(\sum_{j>k}\sigma_j^2\right)^{1/2} \\ &\leq \left[1 + \frac{4\sqrt{k+p}}{p-1}\cdot\sqrt{\min\{m,n\}}\right]\sigma_{k+1} \\ &= C\cdot\sigma_{k+1}.\end{aligned}$$

Traditional analysis of orthogonal iteration [15], [32] relies on a parameter q which defines the number of times the method needs to access the elements of A. The analysis shows that by increasing q, the approximate basis Q converges to the dominant, invariant subspace of A. With $q = 0$ as in the basic algorithm, the existing theory would not provide any certainty of an accurate basis after one step. Notice that with a small oversampling parameter, say $p = 10$, Theorem 1.5.1 puts us within a small polynomial factor of the minimum possible error. The error bound is also slightly sharper than bounds for the rank revealing QR algorithms [17].

1.5.3 Probabilistic error bounds

Theoretical results that guarantee the effectiveness of randomized methods such as the one described in Section 1.5.2 must first establish that in the "typical" outcome, they provide close to optimal results. In the context described in Section 1.5.2, the best possible basis for the column space is given by the left singular vectors of A [15]. To be precise, if Q consists of the first k left singular vectors, then

$$||\mathsf{A} - \mathsf{Q}\,\mathsf{Q}^*\,\mathsf{A}|| = \inf_{\mathrm{rank}(\mathsf{B})=k} ||\mathsf{A} - \mathsf{B}|| = \sigma_{k+1},$$

where σ_j denotes the j^{th} singular value of A. We consequently seek to prove that for a matrix Q constructed via randomized sampling, the quantity $||\mathsf{A} - \mathsf{Q}\,\mathsf{Q}^*\,\mathsf{A}||$ is typically close to σ_{k+1}. An upper bound on the expected value of the error as in Theorem 1.5.1 is an example of a result of this kind.

Once the behavior in the typical case has been established, the second task of the analysis is to assert that the probability of a non-typical outcome is small. The probability is controlled by the oversampling parameter p where by moderately increasing p the probability of a larger than typical error is vanishingly small (less than 10^{-15} is entirely typical).

To illustrate this type of result, we give here a simplified version of a result that first appeared in [24] and that was sharpened and reproved in Chapter 2. For details, see Section 2.10 of Chapter 2.

Theorem 1.5.2. *Let m, n, and k be positive integers such that $m \geq n \geq k$. Let p be a positive integer. Let A be an $m \times n$ matrix with singular values $\{\sigma_j\}_{j=1}^n$. Let Ω be an $n \times (k+p)$ Gaussian random matrix, and let Q have $k+p$ orthonormal columns such that $\mathsf{QR} = \mathsf{A}\,\Omega$. Then the inequality*

$$||\mathsf{A} - \mathsf{Q}\mathsf{Q}^{\mathsf{T}}\,\mathsf{A}|| \leq 10\,\sqrt{(k+p)\,(n-k)}\,\sigma_{k+1}. \tag{1.17}$$

holds with probability at least

$$1 - \varphi(p)$$

where φ is a universal function that decreases monotonically, and satisfies, e.g.

$$\varphi(5) < 3 \cdot 10^{-6}, \qquad \varphi(10) < 3 \cdot 10^{-13}, \qquad \varphi(15) < 8 \cdot 10^{-21}, \qquad \varphi(20) < 6 \cdot 10^{-27}.$$

The very rapid decay of the "failure probability" is a key feature of randomized methods. For the method to go wrong, it is necessary for a large number of random variables to be drawn not only from their individual tail probabilities, but in a way that conspires to produce the error.

Theorem 1.5.2 shows that slightly loosening the upper bound dramatically improves the probability of success. Setting $p = 5$ renders the probability of failure extremely negligible. Theorem 1.5.1 gives a different picture of the oversampling parameter. Here we can see that choosing an increasingly larger p in fact diminishes the error. In practice, a value of $p = 10$ offers a good balance between accuracy, assurance of success, and computational efficiency.

There are two other quantities of interest in the error bounds: the singular values of A and the factor $\sqrt{n\,k}$. If the singular values decay rapidly the factor $\sqrt{n\,k}$ becomes irrelevant. In the case of Theorem 1.5.1, the $\sqrt{n}$ factor is a worse case approximation to the Frobenius factor

$$\sigma_{k+1} \quad \leq \quad \left(\sum_{j>k} \sigma_j^2\right)^{1/2} \quad \leq \quad \sqrt{n-k} \cdot \sigma_{k+1}, \tag{1.18}$$

where equality holds on the right only if the singular values exhibit no decay at all. If the singular values decay sufficiently the quantity is closer to the lower bound on the left. For matrices with slowly decaying singular values, the factor $\sqrt{n\,k}$ can be improved upon via power iteration type methods discussed in §1.6.2.

Finally, the a priori error bounds of Theorems 1.5.1 and 1.5.2 are not the only assurance we have of constructing an accurate basis Q. A posteriori estimates provide a computationally cheap and effective way of assessing the quantity $\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\|$. In addition, the techniques provide a way to adaptively determine Q for cases where the rank is not known in advance but rather a tolerance is given as in the fixed precision problem. In other words, given a tolerance ϵ, construct a matrix Q of *minimal* rank such that

$$\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| < \epsilon. \tag{1.19}$$

It is unlikely that the randomized methods will initially produce a minimal rank matrix Q, however, post-processing will reveal the true rank. For example, from an $m \times (k+p)$ matrix Q satisfying equation (1.28) we can compute a rank $k+p$ approximate singular value decomposition satisfying the same tolerance ϵ. Truncating the factorization to the leading k terms produces rank k factors that satisfy

$$\|\mathsf{A} - \mathsf{U}\Sigma\mathsf{V}^*\| < \sigma_{k+1} + \epsilon. \tag{1.20}$$

Since k is not know in this situation, we can determine it as accurate as we like by examining the entries of Σ. See Theorem 2.9.3 page 91 of Chapter 2 for details.

1.5.4 Computational cost

The algorithm described in Section 1.5.2 is particularly efficient in situations where the matrix A can rapidly be applied to a vector. In this situation, its computational cost T_{total} satisfies

$$T_{\text{total}} \sim k\,n \times T_{\text{rand}} + k \times T_{\text{mult}} + m\,k^2 \times T_{\text{flop}}, \tag{1.21}$$

where T_{rand} is the cost of generating a pseudo-random number, T_{mult} is the cost of applying A to a vector, and T_{flop} is the cost of a floating point operation.

For a dense matrix, $T_{\text{mult}} = m\,n$, and the estimate (1.21) reduces to the $O(m\,n\,k)$ complexity of standard dense matrix techniques. However, it was shown in [34] that the scheme can be modified by using a random matrix Ω that has some internal structure that allows the matrix-vector product $\mathsf{Y} = \mathsf{A}\,\Omega$ to be evaluated in $O(m\,n\,\log(k))$ operations. It turns out that the constants involved are sufficiently small that this algorithm beats standard techniques for very moderate problem sizes (say $m, n \sim 10^3$ and $k \sim 10^2$).

In order to implement the algorithms described in this section, a number of practical issues must be addressed: What random matrix Ω should be used? How is the matrix Q computed from the columns of Y? How do you solve the "fixed precision" problem where the computational tolerance is given and the numerical rank is to be determined? How do you compute the singular value decomposition once you have constructed the basis Q? Answering questions of this type was a principal motivation behind the paper [18] that forms Chapter 2 of this dissertation.

1.6 Variations of the basic theme

The basic algorithm leaves room for modifications depending on the computational setting and performance requirements. We describe a technique to speed up the matrix product $\mathsf{A}\Omega$ that lowers the complexity of the algorithm below that of Krylov methods. We also present techniques that achieve arbitrarily high accuracy and ensure numerical stability.

1.6.1 Structured Random Matrices

The bottleneck in the basic algorithm, when applied to dense matrices stored in fast memory, is the matrix product $\mathsf{A}\Omega$ costing $O(mn\ell)$ flops for $\ell = k + p$. Using a dense Gaussian test matrix Ω is appropriate in certain situations and tighten analysis of the methods, but are expensive to compute with. We can lift the restriction of test matrix Ω being Gaussian and instead use a random test matrix that is structured, reducing the complexity of the matrix product to $O(mn\ \log(\ell))$. One such matrix, the *Fast Johnson Lindenstrauss Transform* (FJLT) is described in [3] to solve the approximate nearest neighbor problem. The FJLT uses a sparse projection matrix preconditioned with a randomized Fourier transform to achieve a low distortion embedding at reduced cost. Adaptation of this idea to the problem of randomized sampling is given in [34] known as the *subsampled randomized Fourier transform* (SRFT):

$$\Omega^{n\times\ell} = \sqrt{\frac{n}{\ell}}\mathsf{DFR} \tag{1.22}$$

where

- D is an $n \times n$ diagonal matrix with d_{ii} drawn randomly from the complex cirle.
- F is the $n \times n$ unitary discrete Fourier transform.
- R is an operator that randomly selects ℓ of the n columns.

For a real variant of the SRFT, consider $d_{ii} = \pm 1$ and F a Hadamard matrix. Other matrices that exhibit structure and randomness have been proposed in [22] and [1]. While the analysis of using these structured matrices call for more oversampling in the worst case scenerios, in practice they often require no more oversampling than Gaussian test matrices.

1.6.2 Increasing accuracy

As we alluded to in §1.5.3, the decay of the singular values plays a large role in the accuracy of the randomized methods. If we have sufficient decay in the singular values then the basic algorithm will produce excellent results. However, as indicated by inequality (1.18), if the tail singular values $\{\sigma_j\}_{j>k+1}$ are significant, we can be penalized with a $\sqrt{n}$ factor in the error. In order to reduce the contribution of these singular values we can sample a similar matrix $\mathsf{B} = \left(\mathsf{AA}^{\mathsf{T}}\right)^q \mathsf{A}$ that has the same singular vectors as A and related singular values $\mu(\mathsf{B}) = \sigma(\mathsf{A})^{2q+1}$. The adaptation exponentially shrinks the constant in the error bound [28]:

$$\begin{aligned}
\mathbb{E}\|(\mathsf{I} - \mathsf{P}_{\mathsf{Z}})\,\mathsf{A}\|_2 &\leq (\mathbb{E}\|(\mathsf{I} - \mathsf{P}_{\mathsf{Z}})\,\mathsf{B}\|_2)^{1/(2q+1)} \\
&\leq \left[1 + \frac{4\sqrt{k+p}}{p-1} \cdot \sqrt{\min\{m,n\}}\right]^{1/(2q+1)} \sigma_{k+1} \\
&= C^{1/(2q+1)}\sigma_{k+1}
\end{aligned}$$

where P_Z is the orthogonal projector onto the space spanned by $\mathsf{Z} = \mathsf{B}\Omega$. With a small q we can get a nearly optimal approximation since $C^{1/(2q+1)} \to 1$ rapidly as q moderately increases. Arbitrarily high precision can be achieved by increasing the parameter q. Numerical experiments confirm that $q \leq 3$ in almost all cases produces excellent results. In fact, often results from $q = 2$ are indistinguishable from $q = 3$ which implies that the computed spectrum has fully converged after just two iterations.

Sampling matrix B is more expensive than the basic scheme and requires multiplication by A and A^T at each iteration. However, the cost is controlled by the fact that q does not depend on k, and that the method is able to compute any size k factorization in a *constant* number of passes. Contrast this with iterative Krylov methods that require $O(k)$ passes over the input matrix. Of course, for matrices that admit a fast matrix multiply, such as sparse matrices, the cost is even further reduced. In addition, the matrix product $\mathsf{A}^\mathsf{T}\mathsf{A}\mathbf{x}$ can be computed in a single pass through A. Assuming A is stored row-wise so that we have access to A one row at a time, compute the outer product

$$\mathsf{A}^\mathsf{T}\mathsf{A}\mathbf{x} = \sum_{i=1}^{m} \langle \mathsf{A}(i,:), \mathbf{x} \rangle \cdot (\mathsf{A}(i,:))^T . \tag{1.23}$$

If we view the matrix B as $\mathsf{A}\left(\mathsf{A}^\mathsf{T}\mathsf{A}\right)^q$, then the scheme needs only $q+1$ passes over matrix A.

1.6.3 Numerical Stability

The basic randomized sampling scheme is numerically very stable. Both the major components of the algorithm, matrix multiplication and orthogonalization are well understood and can be stably implemented. However, there is a possibility that we can loose some accuracy in the smaller eigenvalues of our approximation. Each sample vector can be decomposed as (assuming A is symmetric)

$$\mathbf{y} = \sum_j \lambda_j^{2q+1} \langle \boldsymbol{\omega}, \mathbf{x}_j \rangle \mathbf{x}_j \tag{1.24}$$

where q is the parameter we introduced in §1.6.2. For $q = 0$, as in the basic scheme, any eigenvalue of interest is larger than machine precision, for example $\lambda_j > \epsilon$. For $q \geq 1$ it can be the case that $\lambda_j^{2q+1} < \epsilon$ and that small eigenvalues are pushed below machine precision and appear as noise. If this happens we are unable to detect the eigenvectors associated with the small eigenvalues. Note however that this is not detrimental. Often it is the case that the largest eigenvalues are of interest in which case the method still retains excellent accuracy for their associated eigenvectors.

One solution to the problem is inspired by a block Lanczos procedure and explored in Chapter 3. We keep intermediate iterates in the sample matrix as follows

$$\mathsf{Y} = \begin{bmatrix} \mathsf{Y}_{(0)} & \mathsf{Y}_{(1)} & \dots & \mathsf{Y}_{(q)} \end{bmatrix} \tag{1.25}$$

where the matrices $\mathsf{Y}_{(i)}$ are defined recursively as

$$\begin{aligned} \mathsf{Y}_{(0)} &= \mathsf{A}\Omega \\ \mathsf{Y}_{(i)} &= \mathsf{A}\mathsf{A}^\mathsf{T}\mathsf{Y}_{(i-1)} \quad \text{for } i = 1, 2, \dots, q. \end{aligned} \tag{1.26}$$

Test matrix Ω has ℓ columns but typically ℓ is very close to k, that is $\ell = k+2$ for example. The need for oversampling is greatly reduced with this method. Small eigenvalues are preserved in Y_0 and thus do not get buried by exponentiation with q. This technique results in a larger $m \times (q+1)\ell$ size basis. There is extra work in orthogonalizing the larger basis and increased memory requirements. However, it can be effective when there is sufficient RAM to hold the basis.

A more space efficient solution is to orthogonalize the basis between iterations [25].

$$\begin{array}{rcll} \mathsf{Q}_{(0)}\mathsf{R}_{(0)} & = & \mathsf{A}\Omega & \\ \mathsf{Q}_{(i)} & = & \mathsf{A}\mathsf{Q}_{(i-1)} & \text{for } i = 1, 2, \ldots, q. \end{array} \tag{1.27}$$

The orthogonalization preserves linear independence among the sample vectors.

Remark 2. *Both methods are slightly paranoid in the sense that for most cases, not every intermediate sample matrix needs to be kept nor does orthogonalization need to be done between every iteration.*

1.6.4 Adaptive methods

Thus far we have assumed that k is specified in advance. In other words, we have been solving the fixed rank problem. If instead a tolerance ϵ is specified and the rank is not provided, we need to solve the fixed precision problem. That is, find a matrix Q with orthonormal columns such that

$$\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^{\mathsf{T}}\mathsf{A}\| < \epsilon \tag{1.28}$$

To assess the quantity in (1.28) the following theorem of [34] and [10] provides a cheap way to estimate an upper bound by examining the action of the matrix on random vectors. Substitute $\mathsf{B} = \left(\mathsf{I} - \mathsf{Q}\mathsf{Q}^{\mathsf{T}}\right)\mathsf{A}$ in the following,

Theorem 1.6.1. *For any matrix* B*, positive integer* r *and normalized i.i.d. Gaussian vectors* $\{\boldsymbol{\omega}^{(i)}\}_{i=1}^{r}$*,*

$$\|\mathsf{B}\| \leq 10\sqrt{\frac{2}{\pi}} \max_{i=1,\ldots,r} \|\mathsf{B}\boldsymbol{\omega}^{(i)}\|$$

except with probability 10^{-r}*.*

The theorem states that if we observe $\| \left(\mathsf{I} - \mathsf{Q}\mathsf{Q}^{\mathsf{T}}\right)\mathsf{A}\| < \epsilon / \left(10\sqrt{\frac{2}{\pi}}\right)$ for r consecutive random vectors, then we can be sure with probability 10^{-r} that (1.28) is satisfied.

Not only does the theorem provide an accessible way to assess the norm of the approximation, it comes at virtually *no* cost when integrated into the randomized sampling scheme. Consider an initial starting block of $\ell > r$ samples and compute the QR factorization of Y. The estimates of Theorem 1.6.1 are contained in the R factor! To see this partition Q and R as follows

$$\mathsf{Y} = \mathsf{Q}\mathsf{R} = \begin{bmatrix} \mathsf{Q}_1 & \mathsf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathsf{R}_1 & \mathsf{R}_2 \\ 0 & \mathsf{R}_3 \end{bmatrix} \tag{1.29}$$

where Q_1 is the first $\ell - r$ basis vectors, Q_2 are the remaining r, and R is partitioned conformally. We want to use Q_1 as the basis and use the last r columns of the sample matrix $\mathsf{Y}(:, \ell - r : \ell) = \mathsf{Q}_1\mathsf{R}_2 + \mathsf{Q}_2\mathsf{R}_3$, to assess the error.

$$\left(\mathsf{I} - \mathsf{Q}_1{\mathsf{Q}^{\mathsf{T}}}_1\right)\mathsf{Y}(:, \ell - r : \ell) = \left(\mathsf{I} - \mathsf{Q}_1{\mathsf{Q}^{\mathsf{T}}}_1\right)(\mathsf{Q}_1\mathsf{R}_2 + \mathsf{Q}_2\mathsf{R}_3) = \mathsf{Q}_2\mathsf{R}_3$$

If each of the columns of R_3 have sufficiently small norms, then Q_1 is a suitable basis. If not, we double the amount of samples and repeat the process adding $\ell, 2\ell, 4\ell, \ldots$ vectors to the basis until the tolerance is satisfied. There is a slight computational cost in that we compute r extra samples, but these can also be kept as part of the basis for free and provide an accuracy boost. On the other extreme, if we significantly overshoot the number of samples, it is a simple matter to work backwards until we find a partial column norm of R that is above the threshold.

1.7 Numerical linear algebra in a distributed computing environment

Data sets and their associated matrices can easily grow beyond the size of random access memory necessitating *out-of-core* algorithms. If the size of the approximation factors are small enough to fit in RAM then we only need to perform the randomized sampling stage out-of-core while the remainder of the calculations can be done in memory. For example, dense matrices can take an enormous amount of space on disk but the dimensions of the matrix are such that a few hundred singular vectors can fit comfortably in memory. Consider a 1,000,000 × 1,000,000 dense matrix which occupies over 7TB of disk, meanwhile the factors of its rank 100 singular value decomposition take only 1.5GB. The situation is much different for sparse matrices whose low rank factorizations do not generally provide savings in storage. Assuming a density of 1%, a 7TB sparse matrix has dimensions 100,000,000 × 100,000,00 and its factors occupy 150GB, 100 times the size of the dense example. For cases where the size of the factors are too large to fit in memory we need to do all computations out-of-core.

1.7.1 Distributed computing environment

MapReduce is a distributed programming model popularized by Google. Algorithms are structured into *map* tasks and *reduce* tasks and communication is restricted to a *shuffle and sort* phase in between. Input data is partitioned into chunks and an identical map task operates on each of the chunks independently of each other. During processing, data that needs to be communicated is marked with a key. The shuffle and sort phase groups the data by key and delivers all data with the same key to a reduce task for further processing. These tasks can be chained together to construct complex algorithms. The restrictive model allows for many of the complexities of distributed computing to be handled by a framework. Hadoop is the scalable distributed computing framework that implements MapReduce algorithms. It handles filesystem level tasks such as load balancing and locality optimization, and details of parallelization such as fault tolerance and execution of the algorithms.

The basic unit of parallelism in a MapReduce algorithm is the *task*. The number of tasks is determined by the size of the input data which is partitioned into chunks of equal size. A *node* in a Hadoop cluster can run multiple tasks concurrently. A Hadoop cluster has the ability to coordinate thousands of nodes harnessing massive computing power. In addition, Hadoop can run on a standard single processor computer attesting to the scalable nature of the framework.

1.7.2 Distributed operations

The two major operations in a randomized scheme are the matrix multiplication $\mathsf{Y} = \mathsf{A}\Omega$ and the orthogonalization of Y to form the projector matrix Q. It is necessary to parallelize and distribute these operations to reduce computation time on large data sets, and as we discussed, the matrix Y (and consequently Q or U, V of a partial svd) is too large to fit in memory.

1.7.2.1 Matrix multiplication

Matrix A is partitioned into blocks of rows and each block is processed by a map task. We form Y row by row via outer products of row of A and Ω

$$\mathsf{Y}(i, :) = \sum_j \mathsf{A}(i, j) \cdot \Omega(j, :) \tag{1.30}$$

where row blocks of Y are formed in each task. Notice this requires a copy of Ω in each task to form a block of Y. Luckily we can avoid this by generating entries of Ω *on the fly* using a seeded random number generator. Equation (1.30) can be done with only a map phase, there is no further communication required after the map tasks have completed and the reducers are unnecessary in this case. Taking each entry of A individually uncouples memory requirements from n. For blocksize s, the multiplication only requires $s\ell$ memory.

The power method of §1.6.2 presented a way to increase accuracy but also presents another challenge. After the first matrix multiplication with Ω we need to multiply A by a dense matrix that cannot be generated on the fly but rather needs to be distributed to each task. Let $\mathsf{B}^T = \mathsf{A}^T\mathsf{A}\Omega$, then the product $\mathsf{Y} = \mathsf{A}\mathsf{B}^T$ is done via blocked outer products. A treatment of this operation is given in Chapter 4.

1.7.2.2 Orthogonalization

Givens rotations are well suited to parallel computations. Operating on a small scale by zeroing entries one by one allows Givens rotations to effect only two rows (or columns) of a matrix at a time. From §1.7.2.1 we see that row wise blocks of Y are available in each task. We can form QR decompositions of the blocks of Y independently and then merge them together using Givens rotations. All that is needed is a communication phase to distribute a copy of the R factor from each task to each of the other partial factorizations. The sequence of rotations that achieves

$$G_{seq}^T \begin{bmatrix} \mathsf{R}_1 \\ \mathsf{R}_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathsf{R} \\ 0 \\ \vdots \end{bmatrix}, \tag{1.31}$$

when applied to the block decomposition of Y, will form a block of the full factorization $\mathsf{QR} = \mathsf{Y}$. The computation of the block factorizations and the blocks of the final factorizations are carried out in parallel with only the communication of the R factors in between. Memory requirements for this method are proportional to the block size and require only $2 \cdot s\ell$ memory.

1.7.3 Stochastic Singular Value Decomposition in MapReduce

The Stochastic Singular Value Decomposition (ssvd) is a combination of algorithms in Chapter 2 and the distributed techniques described in §1.7.2. The algorithm incorporates optional power iterations of §1.6.2 to increase accuracy and orthogonalization between the iterations as described in §1.6.3 to increase numerical stability. Given a user specified rank k, it computes a rank k singular value decomposition of a matrix on Hadoop clusters of computers. Owing to the distributed matrix multiplication and orthogonalization schemes, memory requirements for ssvd are independent of both dimensions of the input matrix.

1.7.4 Lanczos comparison

An important question to answer is how does the ssvd method compare to the Lanczos svd. We examine the following areas:

Execution time The Hadoop environment is ill suited for iterative computations. With each pass through the data an algorithm should extract as much information as possible thus limiting interaction with the data and minimizing set up and data movement costs. The Lanczos algorithm computes only a vector's worth of data each pass through the matrix. This not only requires many passes through the data incurring set up and data movement costs, but also fails to fully utilize processor capability. The ssvd visits the matrix only twice (and two additional times per power iteration). It also does computation in bulk which fully utilizes the processors and minimizes overhead costs of the framework. The ssvd is faster than the Lanczos method in this environment.

Accuracy The Lanczos method provides excellent accuracy. It was designed to do this at the expense of many passes through the data. The ssvd provides comparable accuracy. In particular, for $q = 0$, the ssvd is not as accurate as Lanczos method, however, with just one power iteration we obtain a slightly better approximation.

Scalability This is perhaps the most important feature since if a method does not scale, then it cannot compete on large data sets. The implementation of Lanczos we tested only distributed the matrix vector multiplication, orthogonalization was done in serial on a single machine. We ran into memory (java heap) problems on modest sized data sets and could not use the methods on our larger data sets. The ssvd was able to compute a rank 100 singular value decomposition of a matrix whose dimensions both exceeded 37,000,000. Theoretically, the ssvd's memory usage is not dependent on either dimension of the matrix so we believe that the method will scale to much larger data sets.

1.7.5 Numerical Experiments

Extensive numerical experiments validate the claims of §1.7.4. As discussed in the introduction of this section, sparse matrices provide challenging problems since often the approximation

factors are too large to fit in memory. *Term-frequency document-frequency* (tf-df) matrices are common in vectorizing textual documents. We used a data set of Wikipedia articles and constructed extremely sparse tf-df matrices of various sizes. Crucial to the success of the algorithm are the computational parameters that involve intermediate blocking sizes and Hadoop system settings. A thorough investigation is done on how to tune a cluster to run the ssvd. Experiments are carried out on Hadoop clusters in Amazon's Elastic Compute Cloud with up to 64 total cores.

1.8 Survey of prior work on randomized algorithms in linear algebra

That randomized sampling techniques can advantageously be used to compute partial spectral decompositions of matrices was demonstrated in [24, 34, 23]. These papers describe algorithms that greatly improve upon both the speed and the robustness of the prevailing standard techniques for both sparse and dense matrices. The algorithms of [24, 34, 23] had a shortcoming in that they did not perform optimally for matrices whose singular values decay slowly, making them unsuitable for very large matrices with low "signal-to-noise" ratios. This shortcoming was overcome by Rokhlin, Tygert, and Szlam [28], who demonstrated that with certain modifications, the randomized techniques produce results that are significantly more accurate than the standard methods, while still retaining advantages in speed and robustness. This development was game-changing since standard techniques would have been entirely unable to handle the large noisy matrices described throughout this dissertation.

The techniques of [24, 34, 23] can be viewed as intellectual descendants of earlier work in functional analysis and probability theory by Johnson and Lindenstraus [20], Bourgain [6], and others, who demonstrated the power of randomized projections in reducing the effective dimension of data sets. In fact, the work of [24, 34, 23] can be viewed as an application of these ideas to standard problems of numerical linear algebra in a sense similar to how the celebrated work on compressed sensing [8, 11] could be viewed as an application of these ideas to signal processing. The techniques described here are also indirectly inspired by other recent work on randomized techniques in linear algebra, including, [14, 2, 12, 9, 13, 31], and in particular the work of Papadimitriou [27] and Frieze, Kannan and Vempala [14].

Remark 3. *A more detailed review of the literature on randomized methods can be found in Section 2 of Chapter 2.*

1.9 Structure of dissertation and overview of principal contributions

Structure: Beyond this introductory chapter, the dissertation consists of three independent chapters, two of which have previously appeared as independent articles in refereed journals. Each chapter is self-contained and can be read by itself. While improving readability and accessibility, this organization necessitates some repetition of material across the different chapters. Notation is consistent within chapters, but not necessarily across the entire dissertation.

In the remainder of this section, we will briefly summarize the key contributions in each chapter.

Chapter 2: The material in this chapter has appeared as:

N. Halko, P. G. Martinsson, and J. A. Tropp.*Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, SIAM Review **53**, pp. 217–288, 2011.

This paper presents several new algorithms, provides a new theoretical analysis of randomized methods for low-rank approximation, and also serves as a review of recent work on randomized methods for matrix approximation. Principal contributions include:

- A unified framework for computing low-rank factorizations via randomized methods is presented. The idea is to organize a computation into two distinct "stages" — one that constructs an approximate basis for the range of a matrix, and one that factorizes the matrix within the computed range. The first stage is executed via any of several randomized methods presented, and the second stage is executed via deterministic methods. It is demonstrated that by organizing the computation in this way, theoretical analysis of the methods is substantially simplified. It also leads to a very flexible set of algorithms that can readily be adapted to different computing environment: sparse or dense matrices, storage in RAM or out-of-core, single processor or parallel computing, etc.

- A new set of techniques for adaptively computing the rank of a matrix is presented. Earlier randomized algorithms tended to rely on the numerical rank (or at least an upper bound for the numerical rank) to be provided as an input to the computation.

- Earlier versions of randomized methods focussed on computing an approximate *singular value decomposition*. The paper demonstrates how these algorithms can be modified to compute other factorizations such as QR, LU, or ULV-factorizations. In particular, it is demonstrated that randomized methods are particularly well suited for the so called *skeletonization problem* which is the task of finding a subset of columns of the matrix that form an optimal basis for the column space. (To be precise, the paper demonstrates that a very slightly relaxed version of the skeletonization problem can be solved efficiently.)

- The paper provides a new theoretical framework for analyzing randomized methods for computing low-rank approximations. It is demonstrated that the task of proving probabilistic error bounds naturally splits into two distinct components: one consisting of mostly classical linear algebra, and one that relies on precise estimates for the probability distribution of certain classes of random matrices. The paper furnishes the needed estimates and provides new bounds both on the expected error and on the tail probabilities for the risk of incurring an atypical large error. Two different classes of sampling are considered: Gaussian random matrices, and the Fast Johnson-Lindenstrauss transform.

Chapter 3: The material in this chapter has appeared as:

N. Halko, P.G. Martinsson, Y. Shkolnisky, and M. Tygert. *An algorithm for the principal component analysis of large data sets.* SIAM Journal on Scientific Computing, **33**(5):2580–2594, 2011.

This paper exploits techniques described in Chapter 2, but focuses on the implementation of the methods in a situation where the data is stored *out-of-core.* The purpose is to demonstrate that randomized algorithms involve a sufficiently small amount of communication that very large dense matrices (of size $n, m \sim 10^5$ or more) can be factored even on a modest office laptop. Specific contributions include:

- The algorithms described in Chapter 2 are customized for executing out-of-core. That is, the input matrix is too large to fit in RAM and each entry must be read from slow memory when needed. The algorithms seek to minimize interaction with the input matrix as the cost of data movement dominates execution time.
- A technique for improving the accuracy of the computed factorization is developed. The observation here is that in the out-of-core environment, the only relevant cost is moving data between slow and fast memory. This allows us to perform extra computations on the data in fast memory to reduce approximation errors.
- A key contribution of this paper are the extensive numerical experiments. The algorithms are tested on several very large data sets including standard test sets, numerical simulations, and physical measurements. The tests are carried out on both a modest office laptop and a high performance workstation to show the scheme succeeds in a variety of environments.

Chapter 4: This chapter describes modifications to the basic algorithms developed in Chapter 2 that allows them to run efficiently in a distributed computing environment. While Chapter 3 focusses on the case where the matrix is stored outside the fast memory for a single processor, we now involve very large numbers of processors with distributed storage (that could be out-of-core as well). Specific contributions include:

- A parallel versions of algorithms in Chapter 2 are developed that include distributed matrix multiplication and a distributed orthogonalization scheme. Givens rotations are used to compute blocks of orthonormal matrix `Q` independently of each other. The scheme distributes and parallelizes computation to as many nodes as are available greatly reducing the execution time of the orthogonalization phase. With this modification we can efficiently orthogonalize the basis in between power iterations thus increasing stability. In addition, the parallel adaptations uncouple memory requirements from both dimensions of the input matrix allowing great scalability in both the size of the input matrix and the size of the reduced rank factorization.
- The algorithm is implemented in Hadoop, a scalable distributed computing framework. MapReduce, a programming model based on restricted communication, is well suited to the bulk and infrequent communication patterns of the algorithm. The algorithm, termed Stochastic Singular Value Decomposion (ssvd), was incorporated into Mahout, a scalable machine learning library that uses MapReduce algorithms to leverage the power of Hadoop. Mahout's popularity has grown as the need to extract knowledge from massive data sets continues to rise.

- This paper provides thorough numerical experiments that both study the mathematical and computational parameters of the algorithm, and showcase its scalability. Experiments are carried out on clusters of computers from Amazon Elastic Compute Cloud. We discuss parameters and settings necessary to run the algorithm in this environment. Experiments are done on matrices whose dimensions are millions of times larger than in previous works.
- This paper also provides an analysis and comparison of the ssvd with the classical Lanczos SVD method. Both methods are available in Mahout. An important contribution of this paper is to show that ssvd out scales and out performs Lanczos in this environment, making data analysis feasible on matrices much larger than previously possible with classical methods.

The material in this chapter has not yet been published.

Bibliography

[1] D. Achlioptas, Database-friendly random projections: Johnson–Lindenstrauss with binary coins, J. Comput. System Sci., 66 (2003), pp. 671–687.

[2] D. Achlioptas and F. McSherry, Fast computation of low-rank matrix approximations, J. Assoc. Comput. Mach., 54 (2007), pp. Art. 9, 19 pp. (electronic).

[3] N. Ailon and B. Chazelle, Approximate nearest neighbors and the fast Johnson–Lindenstrauss transform, in STOC '06: Proc. 38th Ann. ACM Symp. Theory of Computing, 2006, pp. 557–563.

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, Wireless sensor networks: a survey, Computer Networks, 38 (2002), pp. 393–422.

[5] M. Belkin and P. Niyogi, Laplacian eigenmaps for dimensionality reduction and data representation, Neural Computation, 15 (2003), pp. 1373–1396.

[6] J. Bourgain, On Lipschitz embedding of finite metric spaces in Hilbert space, Israel J. Math., 52 (1985), pp. 46–52.

[7] P. Businger and G. H. Golub, Linear least squares solutions by householder transformations, Numerische Mathematik, 7 (1965), pp. 269–276. 10.1007/BF01436084.

[8] E. J. Candès, Compressive sampling, in Proc. 2006 Intl. Cong. Mathematicians, Madrid, 2006.

[9] A. Deshpande and S. Vempala, Adaptive sampling and fast low-rank matrix approximation, in Approximation, randomization and combinatorial optimization, vol. 4110 of LNCS, Springer, Berlin, 2006, pp. 292–303.

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

[11] D. L. Donoho, Compressed sensing, IEEE Trans. Inform. Theory, 52 (2006), pp. 1289–1306.

[12] P. Drineas, R. Kannan, and M. W. Mahoney, Fast Monte Carlo algorithms for matrices. II. Computing a low-rank approximation to a matrix, SIAM J. Comput., 36 (2006), pp. 158–183 (electronic).

[13] S. Friedland, A. Niknejad, M. Kaveh, and H. Zare, Fast monte carlo low rank approximations for matrices, in Proc. IEEE Conf. Systems-of-Systems Engineering, Los Angeles, CA, 2006, IEEE Computer Society, pp. 218–223.

[14] A. Frieze, R. Kannan, and S. Vempala, Fast Monte Carlo algorithms for finding low-rank approximations, J. Assoc. Comput. Mach., 51 (2004), pp. 1025–1041. (electronic).

[15] G. H. Golub and C. F. van Loan, Matrix Computations, Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins Univ. Press, Baltimore, MD, 3rd ed., 1996.

[16] L. Greengard and V. Rokhlin, A new version of the fast multipole method for the Laplace equation in three dimensions, Acta Numer., 17 (1997), pp. 229–269.

[17] M. Gu and S. C. Eisenstat, Efficient algorithms for computing a strong rank-revealing QR factorization, SIAM J. Sci. Comput., 17 (1996), pp. 848–869.

[18] N. Halko, P.-G. Martinsson, and J. Tropp, Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions., SIAM Review, 53(2) (2011), pp. 217–288.

[19] T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer, Berlin, 2nd ed., 2008.

[20] W. B. Johnson and J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space, Contemp. Math., 26 (1984), pp. 189–206.

[21] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, research of the national bureau of standards, 45 (1950).

[22] E. Liberty, Accelerated dense random projections, Ph.D. thesis, Computer Science Dept., Yale University, New Haven, CT, 2009.

[23] E. Liberty, F. 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.

[24] P.-G. Martinsson, V. Rokhlin, and M. Tygert, A randomized algorithm for the approximation of matrices, Computer Science Dept. Tech. Report 1361, Yale Univ., New Haven, CT, 2006.

[25] P.-G. Martinsson, A. Szlam, and M. Tygert, Normalized power iterations for the computation of svd, 2010.

[26] J. Novembre and M. Stephens, Interpreting principal component analyses of spatial population genetic variation, Nat Genet, 40 (2008), pp. 646–649.

[27] C. H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala, Latent semantic indexing: A probabilistic analysis, J. Comput. System Sci., 61 (2000), pp. 217–235.

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

[29] V. Rokhlin and M. Tygert, A fast randomized algorithm for overdetermined linear least-squares regression, Proc. Natl. Acad. Sci. USA, 105 (2008), pp. 13212–13217.

[30] H. Rutishauser, Computational aspects of f. l. bauer's simultaneous iteration method, Numerische Mathematik, 13 (1969), pp. 4–13. 10.1007/BF02165269.

[31] T. Sarlós, Improved approximation algorithms for large matrices via random projections, in Proc. 47th Ann. IEEE Symp. Foundations of Computer Science (FOCS), 2006, pp. 143–152.

[32] G. Stewart, Accelerating the orthogonal iteration for the eigenvectors of a hermitian matrix, Numerische Mathematik, 13 (1969), pp. 362–376. 10.1007/BF02165413.

[33] Turk, M. A. and Pentland, A. P., Eigenfaces for Recognition, Journal of Cognitive Neuroscience, 3 (1991), pp. 71–86.

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

Chapter 2

Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions

Nathan Halko, Per-Gunnar Martinsson, Joel A. Tropp

Note: *The work described in this chapter was carried out in collaboration with Professors Joel Tropp of Caltech and Per-Gunnar Martinsson of the University of Colorado. It appeared in the June 2011 issue of SIAM Review (volume 53, pages 217–288) under the title: "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions"*

Abstract: *Low-rank matrix approximations, such as the truncated singular value decomposition and the rank-revealing QR decomposition, play a central role in data analysis and scientific computing. This work surveys and extends recent research which demonstrates that* ***randomization*** *offers a powerful tool for performing low-rank matrix approximation. These techniques exploit modern computational architectures more fully than classical methods and open the possibility of dealing with truly massive data sets.*

This paper presents a modular framework for constructing randomized algorithms that compute partial matrix decompositions. These methods use random sampling to identify a subspace that captures most of the action of a matrix. The input matrix is then compressed—either explicitly or implicitly—to this subspace, and the reduced matrix is manipulated deterministically to obtain the desired low-rank factorization. In many cases, this approach beats its classical competitors in terms of accuracy, speed, and robustness. These claims are supported by extensive numerical experiments and a detailed error analysis.

The specific benefits of randomized techniques depend on the computational environment. Consider the model problem of finding the k dominant components of the singular value decomposition of an $m \times n$ matrix. (i) For a dense input matrix, randomized algorithms require $\mathrm{O}(mn \log(k))$ floating-point operations (flops) in contrast with $\mathrm{O}(mnk)$ for classical algorithms. (ii) For a sparse input matrix, the flop count matches classical Krylov subspace methods, but the randomized approach is more robust and can easily be reorganized to exploit multi-processor architectures. (iii) For a matrix that is too large to fit in fast memory, the randomized techniques require only a constant number of passes over the data, as opposed to $\mathrm{O}(k)$ passes for classical algorithms. In fact, it is sometimes possible to perform matrix approximation with a ***single pass*** *over the data.*

Keywords: Dimension reduction, eigenvalue decomposition, interpolative decomposition, Johnson–Lindenstrauss lemma, matrix approximation, parallel algorithm, pass-efficient algorithm, principal component analysis, randomized algorithm, random matrix, rank-revealing QR factorization, singular value decomposition, streaming algorithm.

AMS classification: Primary: 65F30. Secondary: 68W20, 60B20.

Part I: Introduction

2.1 Overview

On a well-known list of the "Top 10 Algorithms" that have influenced the practice of science and engineering during the 20th century [40], we find an entry that is not really an algorithm: the **idea** of using matrix factorizations to accomplish basic tasks in numerical linear algebra. In the accompanying article [127], Stewart explains that

> The underlying principle of the decompositional approach to matrix computation is that it is not the business of the matrix algorithmicists to solve particular problems but to construct computational platforms from which a variety of problems can be solved.

Stewart goes on to argue that this point of view has had many fruitful consequences, including the development of robust software for performing these factorizations in a highly accurate and provably correct manner.

The decompositional approach to matrix computation remains fundamental, but developments in computer hardware and the emergence of new applications in the information sciences have rendered the classical algorithms for this task inadequate in many situations:

- A salient feature of modern applications, especially in data mining, is that the matrices are stupendously big. Classical algorithms are not always well adapted to solving the type of large-scale problems that now arise.
- In the information sciences, it is common that data are missing or inaccurate. Classical algorithms are designed to produce highly accurate matrix decompositions, but it seems profligate to spend extra computational resources when the imprecision of the data inherently limits the resolution of the output.
- Data transfer now plays a major role in the computational cost of numerical algorithms. Techniques that require few passes over the data may be substantially faster in practice, even if they require as many—or more—floating-point operations.
- As the structure of computer processors continues to evolve, it becomes increasingly important for numerical algorithms to adapt to a range of novel architectures, such as graphics processing units.

The purpose of this paper is make the case that **randomized** algorithms provide a powerful tool for constructing approximate matrix factorizations. These techniques are simple and effective, sometimes impressively so. Compared with standard deterministic algorithms, the randomized methods are often faster and—perhaps surprisingly—more robust. Furthermore, they can produce factorizations that are accurate to any specified tolerance above machine precision, which allows the user to trade accuracy for speed if desired. We present numerical evidence that these algorithms succeed for real computational problems.

In short, our goal is to demonstrate how randomized methods interact with classical techniques to yield effective, modern algorithms supported by detailed theoretical guarantees. We have made a special effort to help practitioners identify situations where randomized techniques may outperform established methods.

Throughout this article, we provide detailed citations to previous work on randomized techniques for computing low-rank approximations. The primary sources that inform our presentation include [17, 46, 58, 91, 105, 112, 113, 118, 137].

Remark 4. Our experience suggests that many practitioners of scientific computing view randomized algorithms as a desperate and final resort. Let us address this concern immediately. Classical Monte Carlo methods are highly sensitive to the random number generator and typically produce output with low and uncertain accuracy. In contrast, the algorithms discussed herein are relatively insensitive to the quality of randomness and produce highly accurate results. The probability of failure is a user-specified parameter that can be rendered negligible (say, less than 10^{-15}) with a nominal impact on the computational resources required.

2.1.1 Approximation by low-rank matrices

The roster of standard matrix decompositions includes the pivoted QR factorization, the eigenvalue decomposition, and the singular value decomposition (SVD), all of which expose the (numerical) range of a matrix. Truncated versions of these factorizations are often used to express a **low-rank approximation** of a given matrix:

$$\underset{m \times n}{\mathsf{A}} \quad \approx \quad \underset{m \times k}{\mathsf{B}} \quad \underset{k \times n.}{\mathsf{C}}, \tag{2.1}$$

The inner dimension k is sometimes called the **numerical rank** of the matrix. When the numerical rank is much smaller than either dimension m or n, a factorization such as (2.1) allows the matrix to be stored inexpensively and to be multiplied rapidly with vectors or other matrices. The factorizations can also be used for data interpretation or to solve computational problems, such as least squares.

Matrices with low numerical rank appear in a wide variety of scientific applications. We list only a few:

- A basic method in statistics and data mining is to compute the directions of maximal variance in vector-valued data by performing **principal component analysis** (PCA) on the data matrix. PCA is nothing other than a low-rank matrix approximation [71, §14.5].

- Another standard technique in data analysis is to perform low-dimensional embedding of data under the assumption that there are fewer degrees of freedom than the ambient dimension would suggest. In many cases, the method reduces to computing a partial SVD of a matrix derived from the data. See [71, §§14.8–14.9] or [30].

- The problem of estimating parameters from measured data via least-squares fitting often leads to very large systems of linear equations that are close to linearly dependent. Effective techniques for factoring the coefficient matrix lead to efficient techniques for solving the least-squares problem, [113].

- Many fast numerical algorithms for solving PDEs and for rapidly evaluating potential fields such as the fast multipole method [66] and $\mathcal{H}$-matrices [65], rely on low-rank approximations of continuum operators.

- Models of multiscale physical phenomena often involve PDEs with rapidly oscillating coefficients. Techniques for **model reduction** or **coarse graining** in such environments are often based on the observation that the linear transform that maps the input data to the requested output data can be approximated by an operator of low rank [56].

2.1.2 Matrix approximation framework

The task of computing a low-rank approximation to a given matrix can be split naturally into two computational stages. The first is to construct a low-dimensional subspace that captures the action of the matrix. The second is to restrict the matrix to the subspace and then compute a standard factorization (QR, SVD, etc.) of the reduced matrix. To be slightly more formal, we subdivide the computation as follows.

Stage A: Compute an approximate basis for the range of the input matrix A. In other words, we require a matrix Q for which

$$\mathsf{Q} \text{ has orthonormal columns and } \mathsf{A} \approx \mathsf{Q}\mathsf{Q}^*\mathsf{A}. \tag{2.2}$$

We would like the basis matrix Q to contain as few columns as possible, but it is even more important to have an accurate approximation of the input matrix.

Stage B: Given a matrix Q that satisfies (2.2), we use Q to help compute a standard factorization (QR, SVD, etc.) of A.

The task in Stage A can be executed very efficiently with random sampling methods, and these methods are the primary subject of this work. In the next subsection, we offer an overview of these ideas. The body of the paper provides details of the algorithms (§2.4) and a theoretical analysis of their performance (§§2.8–2.11).

Stage B can be completed with well-established deterministic methods. Section 2.3.3.3 contains an introduction to these techniques, and §2.5 shows how we apply them to produce low-rank factorizations.

At this point in the development, it may not be clear why the output from Stage A facilitates our job in Stage B. Let us illustrate by describing how to obtain an approximate SVD of the input matrix A given a matrix Q that satisfies (2.2). More precisely, we wish to compute matrices U and V with orthonormal columns and a nonnegative, diagonal matrix $\mathsf{\Sigma}$ such that $\mathsf{A} \approx \mathsf{U\Sigma V}^*$. This goal is achieved after three simple steps:

(1) Form $\mathsf{B} = \mathsf{Q}^*\mathsf{A}$, which yields the low-rank factorization $\mathsf{A} \approx \mathsf{QB}$.

(2) Compute an SVD of the small matrix: $\mathsf{B} = \widetilde{\mathsf{U}}\mathsf{\Sigma V}^*$.

(3) Set $\mathsf{U} = \mathsf{Q}\widetilde{\mathsf{U}}$.

When Q has few columns, this procedure is efficient because we can easily construct the reduced matrix B and rapidly compute its SVD. In practice, we can often avoid forming B explicitly by means of subtler techniques. In some cases, it is not even necessary to revisit the input matrix A during Stage B. This observation allows us to develop *single-pass algorithms*, which look at each entry of A only once.

Similar manipulations readily yield other standard factorizations, such as the pivoted QR factorization, the eigenvalue decomposition, etc.

2.1.3 Randomized algorithms

This paper describes a class of randomized algorithms for completing Stage A of the matrix approximation framework set forth in §2.1.2. We begin with some details about the approximation problem these algorithms target (§2.1.3.1). Afterward, we motivate the random sampling technique with a heuristic explanation (§2.1.3.2) that leads to a prototype algorithm (§2.1.3.3).

2.1.3.1 Problem formulations

The basic challenge in producing low-rank matrix approximations is a primitive question that we call the *fixed-precision approximation problem*. Suppose we are given a matrix A and a positive error tolerance ε. We seek a matrix Q with $k = k(\varepsilon)$ orthonormal columns such that

$$\|\mathsf{A} - \mathsf{QQ}^*\mathsf{A}\| \leq \varepsilon, \tag{2.3}$$

where $\|\cdot\|$ denotes the ℓ_2 operator norm. The range of Q is a k-dimensional subspace that captures most of the action of A, and we would like k to be as small as possible.

The singular value decomposition furnishes an optimal answer to the fixed-precision problem [97]. Let σ_j denote the jth largest singular value of A. For each $j \geq 0$,

$$\min_{\mathrm{rank}(\mathsf{X}) \leq j} \|\mathsf{A} - \mathsf{X}\| = \sigma_{j+1}. \tag{2.4}$$

One way to construct a minimizer is to choose $\mathsf{X} = \mathsf{Q}\mathsf{Q}^*\mathsf{A}$, where the columns of Q are k dominant left singular vectors of A. Consequently, the minimal rank k where (2.3) holds equals the number of singular values of A that exceed the tolerance ε.

To simplify the development of algorithms, it is convenient to assume that the desired rank k is specified in advance. We call the resulting problem the *fixed-rank approximation problem*. Given a matrix A, a target rank k, and an oversampling parameter p, we seek to construct a matrix Q with $k + p$ orthonormal columns such that

$$\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| \approx \min_{\mathrm{rank}(\mathsf{X}) \leq k} \|\mathsf{A} - \mathsf{X}\|. \tag{2.5}$$

Although there exists a minimizer Q that solves the fixed rank problem for $p = 0$, the opportunity to use a small number of additional columns provides a flexibility that is crucial for the effectiveness of the computational methods we discuss.

We will demonstrate that algorithms for the fixed-rank problem can be adapted to solve the fixed-precision problem. The connection is based on the observation that we can build the basis matrix Q incrementally and, at any point in the computation, we can inexpensively estimate the residual error $\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\|$. Refer to §2.4.4 for the details of this reduction.

2.1.3.2 Intuition

To understand how randomness helps us solve the fixed-rank problem, it is helpful to consider some motivating examples.

First, suppose that we seek a basis for the range of a matrix A with *exact* rank k. Draw a random vector $\boldsymbol{\omega}$, and form the product $\mathbf{y} = \mathsf{A}\boldsymbol{\omega}$. For now, the precise distribution of the random vector is unimportant; just think of $\mathbf{y}$ as a random sample from the range of A. Let us repeat this sampling process k times:

$$\mathbf{y}^{(i)} = \mathsf{A}\boldsymbol{\omega}^{(i)}, \quad i = 1, 2, \ldots, k. \tag{2.6}$$

Owing to the randomness, the set $\{\boldsymbol{\omega}^{(i)} : i = 1, 2, \ldots, k\}$ of random vectors is likely to be in general linear position. In particular, the random vectors form a linearly independent set and no linear combination falls in the null space of A. As a result, the set $\{\mathbf{y}^{(i)} : i = 1, 2, \ldots, k\}$ of sample vectors is also linearly independent, so it spans the range of A. Therefore, to produce an orthonormal basis for the range of A, we just need to orthonormalize the sample vectors.

Now, imagine that $\mathsf{A} = \mathsf{B} + \mathsf{E}$ where B is a rank-k matrix containing the information we seek and E is a small perturbation. Our priority is to obtain a basis that covers as much of the range of B as possible, rather than to minimize the number of basis vectors. Therefore, we fix a small number p, and we generate $k + p$ samples

$$\mathbf{y}^{(i)} = \mathsf{A}\boldsymbol{\omega}^{(i)} = \mathsf{B}\boldsymbol{\omega}^{(i)} + \mathsf{E}\boldsymbol{\omega}^{(i)}, \quad i = 1, 2, \ldots, k + p. \tag{2.7}$$

PROTO-ALGORITHM: SOLVING THE FIXED-RANK PROBLEM

Given an $m \times n$ matrix A, a target rank k, and an oversampling parameter p, this procedure computes an $m \times (k+p)$ matrix Q whose columns are orthonormal and whose range approximates the range of A.

1 Draw a random $n \times (k+p)$ test matrix Ω.
2 Form the matrix product $\mathsf{Y} = \mathsf{A}\Omega$.
3 Construct a matrix Q whose columns form an orthonormal basis for the range of Y.

The perturbation E shifts the direction of each sample vector outside the range of B, which can prevent the span of $\{\mathbf{y}^{(i)} : i = 1, 2, \ldots, k\}$ from covering the entire range of B. In contrast, the enriched set $\{\mathbf{y}^{(i)} : i = 1, 2, \ldots, k+p\}$ of samples has a much better chance of spanning the required subspace.

Just how many extra samples do we need? Remarkably, for certain types of random sampling schemes, the failure probability decreases superexponentially with the oversampling parameter p; see (2.9). As a practical matter, setting $p = 5$ or $p = 10$ often gives superb results. This observation is one of the principal facts supporting the randomized approach to numerical linear algebra.

2.1.3.3 A prototype algorithm

The intuitive approach of §2.1.3.2 can be applied to general matrices. Omitting computational details for now, we formalize the procedure in the figure labeled Proto-Algorithm.

This simple algorithm is by no means new. It is essentially the first step of a subspace iteration with a random initial subspace [61, §7.3.2]. The novelty comes from the additional observation that the initial subspace should have a slightly higher dimension than the invariant subspace we are trying to approximate. With this revision, it is often the case that **no further iteration is required** to obtain a high-quality solution to (2.5). We believe this idea can be traced to [118, 91, 105].

In order to invoke the proto-algorithm with confidence, we must address several practical and theoretical issues:

- What random matrix Ω should we use? How much oversampling do we need?
- The matrix Y is likely to be ill-conditioned. How do we orthonormalize its columns to form the matrix Q?
- What are the computational costs?

- How can we solve the fixed-precision problem (2.3) when the numerical rank of the matrix is not known in advance?
- How can we use the basis Q to compute other matrix factorizations?
- Does the randomized method work for problems of practical interest? How does its speed/accuracy/robustness compare with standard techniques?
- What error bounds can we expect? With what probability?

The next few sections provide a summary of the answers to these questions. We describe several problem regimes where the proto-algorithm can be implemented efficiently, and we present a theorem that describes the performance of the most important instantiation. Finally, we elaborate on how these ideas can be applied to approximate the truncated SVD of a large data matrix. The rest of the paper contains a more exhaustive treatment—including pseudocode, numerical experiments, and a detailed theory.

2.1.4 A comparison between randomized and traditional techniques

To select an appropriate computational method for finding a low-rank approximation to a matrix, the practitioner must take into account the properties of the matrix. Is it dense or sparse? Does it fit in fast memory or is it stored out of core? Does the singular spectrum decay quickly or slowly? The behavior of a numerical linear algebra algorithm may depend on all these factors [13, 61, 132]. To facilitate a comparison between classical and randomized techniques, we summarize their relative performance in each of three representative environments. Section 2.6 contains a more in-depth treatment.

We focus on the task of computing an approximate SVD of an $m \times n$ matrix A with numerical rank k. For randomized schemes, Stage A generally dominates the cost of Stage B in our matrix approximation framework (§2.1.2). Within Stage A, the computational bottleneck is usually the matrix–matrix product $\mathsf{A}\Omega$ in Step 2 of the proto-algorithm (§2.1.3.3). The power of randomized algorithms stems from the fact that we can reorganize this matrix multiplication for maximum efficiency in a variety of computational architectures.

2.1.4.1 A general dense matrix that fits in fast memory

A standard deterministic technique for computing an approximate SVD is to perform a rank-revealing QR factorization of the matrix, and then to manipulate the factors to obtain the final decomposition. The cost of this approach is typically $\mathrm{O}(kmn)$ floating-point operations, or *flops*, although these methods require slightly longer running times in rare cases [68].

In contrast, randomized schemes can produce an approximate SVD using only $\mathrm{O}(mn \log(k) + (m+n)k^2)$ flops. The gain in asymptotic complexity is achieved by using a random matrix Ω that has some internal structure, which allows us to evaluate the product $\mathsf{A}\Omega$ rapidly. For example, randomizing and subsampling the discrete Fourier transform works well. Sections 2.4.6 and 2.11 contain more information on this approach.

2.1.4.2 A matrix for which matrix–vector products can be evaluated rapidly

When the matrix A is sparse or structured, we may be able to apply it rapidly to a vector. In this case, the classical prescription for computing a partial SVD is to invoke a Krylov subspace method, such as the Lanczos or Arnoldi algorithm. It is difficult to summarize the computational cost of these methods because their performance depends heavily on properties of the input matrix and on the amount of effort spent to stabilize the algorithm. (Inherently, the Lanczos and Arnoldi methods are numerically unstable.) For the same reasons, the error analysis of such schemes is unsatisfactory in many important environments.

At the risk of being overly simplistic, we claim that the typical cost of a Krylov method for approximating the k leading singular vectors of the input matrix is proportional to $k\,T_{\text{mult}} + (m+n)k^2$, where T_{mult} denotes the cost of a matrix–vector multiplication with the input matrix and the constant of proportionality is small. We can also apply randomized methods using a Gaussian test matrix Ω to complete the factorization at the same cost, $\mathrm{O}(k\,T_{\text{mult}} + (m+n)k^2)$ flops.

With a given budget of floating-point operations, Krylov methods sometimes deliver a more accurate approximation than randomized algorithms. Nevertheless, the methods described in this survey have at least two powerful advantages over Krylov methods. First, the randomized schemes are inherently stable, and they come with very strong performance guarantees that do not depend on subtle spectral properties of the input matrix. Second, the matrix–vector multiplies required to form $\mathsf{A}\Omega$ can be performed **in parallel**. This fact allows us to restructure the calculations to take full advantage of the computational platform, which can lead to dramatic accelerations in practice, especially for parallel and distributed machines.

A more detailed comparison or randomized schemes and Krylov subspace methods is given in §2.6.2.

2.1.4.3 A general dense matrix stored in slow memory or streamed

When the input matrix is too large to fit in core memory, the cost of transferring the matrix from slow memory typically dominates the cost of performing the arithmetic. The standard techniques for low-rank approximation described in §2.1.4.1 require $\mathrm{O}(k)$ passes over the matrix, which can be prohibitively expensive.

In contrast, the proto-algorithm of §2.1.3.3 requires only one pass over the data to produce the approximate basis Q for Stage A of the approximation framework. This straightforward approach, unfortunately, is not accurate enough for matrices whose singular spectrum decays slowly, but we can address this problem using very few (say, 2 to 4) additional passes over the data [112]. See §2.1.6 or §2.4.5 for more discussion.

Typically, Stage B uses one additional pass over the matrix to construct the approximate SVD. With slight modifications, however, the two-stage randomized scheme can be revised so that it only makes a single pass over the data. Refer to §2.5.5 for information.

2.1.5 Performance analysis

A principal goal of this paper is to provide a detailed analysis of the performance of the proto-algorithm described in §2.1.3.3. This investigation produces precise error bounds, expressed in terms of the singular values of the input matrix. Furthermore, we determine how several choices of the random matrix Ω impact the behavior of the algorithm.

Let us offer a taste of this theory. The following theorem describes the average-case behavior of the proto-algorithm with a Gaussian test matrix, assuming we perform the computation in exact arithmetic. This result is a simplified version of Theorem 2.10.6.

Theorem 2.1.1. *Suppose that* A *is a real* $m \times n$ *matrix. Select a target rank* $k \geq 2$ *and an oversampling parameter* $p \geq 2$*, where* $k + p \leq \min\{m, n\}$*. Execute the proto-algorithm with a standard Gaussian test matrix to obtain an* $m \times (k+p)$ *matrix* Q *with orthonormal columns. Then*

$$\mathbb{E}\,\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| \leq \left[1 + \frac{4\sqrt{k+p}}{p-1} \cdot \sqrt{\min\{m,n\}}\right] \sigma_{k+1}, \tag{2.8}$$

where $\mathbb{E}$ *denotes expectation with respect to the random test matrix and* σ_{k+1} *is the* $(k+1)$*th singular value of* A*.*

We recall that the term σ_{k+1} appearing in (2.8) is the smallest possible error (2.4) achievable with any basis matrix Q. The theorem asserts that, on average, the algorithm produces a basis whose error lies within a small polynomial factor of the theoretical minimum. Moreover, the error bound (2.8) in the randomized algorithm is slightly sharper than comparable bounds for deterministic techniques based on rank-revealing QR algorithms [68].

The reader might be worried about whether the expectation provides a useful account of the approximation error. Fear not: the actual outcome of the algorithm is **almost always** very close to the typical outcome because of measure concentration effects. As we discuss in §2.10.3, the probability that the error satisfies

$$\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| \leq \left[1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}\right] \sigma_{k+1} \tag{2.9}$$

is at least $1 - 6 \cdot p^{-p}$ under very mild assumptions on p. This fact justifies the use of an oversampling term as small as $p = 5$. This simplified estimate is very similar to the major results in [91].

The theory developed in this paper provides much more detailed information about the performance of the proto-algorithm.

- When the singular values of A decay slightly, the error $\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\|$ does not depend on the dimensions of the matrix (§§2.10.2–2.10.3).
- We can reduce the size of the bracket in the error bound (2.8) by combining the proto-algorithm with a power iteration (§2.10.4). For an example, see §2.1.6 below.
- For the structured random matrices we mentioned in §2.1.4.1, related error bounds are in force (§2.11).
- We can obtain inexpensive **a posteriori** error estimates to verify the quality of the approximation (§2.4.3).

2.1.6 Example: Randomized SVD

We conclude this introduction with a short discussion of how these ideas allow us to perform an approximate SVD of a large data matrix, which is a compelling application of randomized matrix approximation [112].

The two-stage randomized method offers a natural approach to SVD computations. Unfortunately, the simplest version of this scheme is inadequate in many applications because the singular spectrum of the input matrix may decay slowly. To address this difficulty, we incorporate q steps of a power iteration, where $q = 1$ or $q = 2$ usually suffices in practice. The complete scheme appears in the box labeled Prototype for Randomized SVD. For most applications, it is important to incorporate additional refinements, as we discuss in §§2.4–2.5.

The Randomized SVD procedure requires only $2(q+1)$ passes over the matrix, so it is efficient even for matrices stored out-of-core. The flop count satisfies

$$T_{\text{randSVD}} = (2q + 2)\, k\, T_{\text{mult}} + \mathrm{O}(k^2(m + n)),$$

where T_{mult} is the flop count of a matrix–vector multiply with A or A^*. We have the following theorem on the performance of this method in exact arithmetic, which is a consequence of Corollary 2.10.10.

Theorem 2.1.2. *Suppose that A is a real $m \times n$ matrix. Select an exponent q and a target number k of singular vectors, where $2 \le k \le 0.5 \min\{m, n\}$. Execute the Randomized SVD algorithm to obtain a rank-$2k$ factorization $\mathsf{U}\Sigma\mathsf{V}^*$. Then*

$$\mathbb{E}\, \|\mathsf{A} - \mathsf{U}\Sigma\mathsf{V}^*\| \le \left[1 + 4\sqrt{\frac{2 \min\{m, n\}}{k - 1}}\right]^{1/(2q+1)} \sigma_{k+1}, \tag{2.10}$$

where $\mathbb{E}$ denotes expectation with respect to the random test matrix and σ_{k+1} is the $(k+1)$th singular value of A.

This result is new. Observe that the bracket in (2.10) is essentially the same as the bracket in the basic error bound (2.8). We find that the power iteration drives the leading constant to one exponentially fast as the power q increases. The rank-k approximation of A can never achieve an error smaller than σ_{k+1}, so the randomized procedure computes $2k$ approximate singular vectors that capture as much of the matrix as the first k actual singular vectors.

In practice, we can truncate the approximate SVD, retaining only the first k singular values and vectors. Equivalently, we replace the diagonal factor Σ by the matrix $\Sigma_{(k)}$ formed by zeroing out all but the largest k entries of Σ. For this truncated SVD, we have the error bound

$$\mathbb{E}\, \left\|\mathsf{A} - \mathsf{U}\Sigma_{(k)}\mathsf{V}^*\right\| \le \sigma_{k+1} + \left[1 + 4\sqrt{\frac{2 \min\{m, n\}}{k - 1}}\right]^{1/(2q+1)} \sigma_{k+1}. \tag{2.11}$$

In words, we pay no more than an additive term σ_{k+1} when we perform the truncation step. Our numerical experience suggests that the error bound (2.11) is pessimistic. See Remark 13 and §2.9.4 for some discussion of truncation.

PROTOTYPE FOR RANDOMIZED SVD

Given an $m \times n$ matrix A, a target number k of singular vectors, and an exponent q (say $q = 1$ or $q = 2$), this procedure computes an approximate rank-$2k$ factorization $\mathsf{U\Sigma V}^*$, where U and V are orthonormal, and $\mathsf{\Sigma}$ is nonnegative and diagonal.

Stage A:

1 Generate an $n \times 2k$ Gaussian test matrix Ω.
2 Form $\mathsf{Y} = (\mathsf{AA}^*)^q\mathsf{A}\Omega$ by multiplying alternately with A and A^*.
3 Construct a matrix Q whose columns form an orthonormal basis for the range of Y.

Stage B:

4 Form $\mathsf{B} = \mathsf{Q}^*\mathsf{A}$.
5 Compute an SVD of the small matrix: $\mathsf{B} = \widetilde{\mathsf{U}}\mathsf{\Sigma V}^*$.
6 Set $\mathsf{U} = \mathsf{Q}\widetilde{\mathsf{U}}$.

Note: The computation of Y in Step 2 is vulnerable to round-off errors. When high accuracy is required, we must incorporate an orthonormalization step between each application of A and A^*; see Algorithm 4.4.

2.1.7 Outline of paper

The paper is organized into three parts: an introduction (§§2.1–2.3), a description of the algorithms (§§2.4–2.7), and a theoretical performance analysis (§§2.8–2.11). The two latter parts commence with a short internal outline. Each part is more or less self-contained, and after a brief review of our notation in §§2.3.1–2.3.2, the reader can proceed to either the algorithms or the theory part.

2.2 Related work and historical context

Randomness has occasionally surfaced in the numerical linear algebra literature; in particular, it is quite standard to initialize iterative algorithms for constructing invariant subspaces with a randomly chosen point. Nevertheless, we believe that sophisticated ideas from random matrix theory have not been incorporated into classical matrix factorization algorithms until very recently. We can trace this development to earlier work in computer science and—especially—to probabilistic methods in geometric analysis. This section presents an overview of the relevant work. We begin with a survey of randomized methods for matrix approximation; then we attempt to trace some of the ideas backward to their sources.

2.2.1 Randomized matrix approximation

Matrices of low numerical rank contain little information relative to their apparent dimension owing to the linear dependency in their columns (or rows). As a result, it is reasonable to expect that these matrices can be approximated with far fewer degrees of freedom. A less obvious fact is that randomized schemes can be used to produce these approximations efficiently.

Several types of approximation techniques build on this idea. These methods all follow the same basic pattern:

(1) Preprocess the matrix, usually to calculate sampling probabilities.

(2) Take random samples from the matrix, where the term *sample* refers generically to a linear function of the matrix.

(3) Postprocess the samples to compute a final approximation, typically with classical techniques from numerical linear algebra. This step may require another look at the matrix.

We continue with a description of the most common approximation schemes.

2.2.1.1 Sparsification

The simplest approach to matrix approximation is the method of *sparsification* or the related technique of *quantization*. The goal of sparsification is to replace the matrix by a surrogate that contains far fewer nonzero entries. Quantization produces an approximation whose components are drawn from a (small) discrete set of values. These methods can be used to limit storage requirements or to accelerate computations by reducing the cost of matrix–vector and matrix–matrix multiplies [94, Ch. 6]. The manuscript [33] describes applications in optimization.

Sparsification typically involves very simple elementwise calculations. Each entry in the approximation is drawn independently at random from a distribution determined from the corresponding entry of the input matrix. The expected value of the random approximation equals the original matrix, but the distribution is designed so that a typical realization is much sparser.

The first method of this form was devised by Achlioptas and McSherry [2], who built on earlier work on graph sparsification due to Karger [75, 76]. Arora–Hazan–Kale presented a different sampling method in [7]. See [123, 60] for some recent work on sparsification.

2.2.1.2 Column selection methods

A second approach to matrix approximation is based on the idea that a small set of columns describes most of the action of a numerically low-rank matrix. Indeed, classical existential results [117] demonstrate that every $m \times n$ matrix A contains a k-column submatrix C for which

$$\left\| \mathsf{A} - \mathsf{C}\mathsf{C}^{\dagger}\mathsf{A} \right\| \leq \sqrt{1 + k(n-k)} \cdot \left\| \mathsf{A} - \mathsf{A}_{(k)} \right\|, \tag{2.12}$$

where k is a parameter, the dagger $\dagger$ denotes the pseudoinverse, and $\mathsf{A}_{(k)}$ is a best rank-k approximation of A. It is NP-hard to perform column selection by optimizing natural objective functions, such as the condition number of the submatrix [27]. Nevertheless, there are efficient deterministic algorithms, such as the rank-revealing QR method of [68], that can nearly achieve the error bound (2.12).

There is a class of randomized algorithms that approach the fixed-rank approximation problem (2.5) using this intuition. These methods first compute a sampling probability for each column, either using the squared Euclidean norms of the columns or their **leverage scores**. (Leverage scores reflect the relative importance of the columns to the action of the matrix; they can be calculated easily from the dominant k right singular vectors of the matrix.) Columns are then selected randomly according to this distribution. Afterward, a postprocessing step is invoked to produce a more refined approximation of the matrix.

We believe that the earliest method of this form appeared in a 1998 paper of Frieze–Kannan–Vempala [57, 58]. This work was refined substantially in the papers [44, 43, 46]. The basic algorithm samples columns from a distribution related to the squared ℓ_2 norms of the columns. This sampling step produces a small column submatrix whose range is aligned with the range of the input matrix. The final approximation is obtained from a truncated SVD of the submatrix. Given a target rank

k and a parameter $\varepsilon > 0$, this approach samples $\ell = \ell(k, \varepsilon)$ columns of the matrix to produce a rank-k approximation B that satisfies

$$\|\mathsf{A} - \mathsf{B}\|_{\mathrm{F}} \leq \left\|\mathsf{A} - \mathsf{A}_{(k)}\right\|_{\mathrm{F}} + \varepsilon \|\mathsf{A}\|_{\mathrm{F}}, \tag{2.13}$$

where $\|\cdot\|_{\mathrm{F}}$ denotes the Frobenius norm. We note that the algorithm of [46] requires only a constant number of passes over the data.

Rudelson and Vershynin later showed that the same type of column sampling method also yields spectral-norm error bounds [116]. The techniques in their paper have been very influential; their work has found other applications in randomized regression [52], sparse approximation [133], and compressive sampling [19].

Deshpande et al. [37, 38] demonstrated that the error in the column sampling approach can be improved by iteration and adaptive volume sampling. They showed that it is possible to produce a rank-k matrix B that satisfies

$$\|\mathsf{A} - \mathsf{B}\|_{\mathrm{F}} \leq (1 + \varepsilon) \left\|\mathsf{A} - \mathsf{A}_{(k)}\right\|_{\mathrm{F}} \tag{2.14}$$

using a k-pass algorithm. Around the same time, Har-Peled [70] independently developed a recursive algorithm that offers the same approximation guarantees. Very recently, Desphande and Rademacher have improved the running time of volume-based sampling methods [36].

Drineas et al. and Boutsidis et al. have also developed randomized algorithms for the *column subset selection problem*, which requests a column submatrix C that achieves a bound of the form (2.12). Via the methods of Rudelson and Vershynin [116], they showed that sampling columns according to their leverage scores is likely to produce the required submatrix [50, 51]. Subsequent work [17, 18] showed that postprocessing the sampled columns with a rank-revealing QR algorithm can reduce the number of output columns required (2.12). The argument in [17] explicitly decouples the linear algebraic part of the analysis from the random matrix theory. The theoretical analysis in the present work involves a very similar technique.

2.2.1.3 Approximation by dimension reduction

A third approach to matrix approximation is based on the concept of **dimension reduction**. Since the rows of a low-rank matrix are linearly dependent, they can be embedded into a low-dimensional space without altering their geometric properties substantially. A random linear map provides an efficient, nonadaptive way to perform this embedding. (Column sampling can also be viewed as an adaptive form of dimension reduction.)

The proto-algorithm we set forth in §2.1.3.3 is simply a dual description of the dimension reduction approach: collecting random samples from the column space of the matrix is equivalent to reducing the dimension of the rows. No precomputation is required to obtain the sampling distribution, but the sample itself takes some work to collect. Afterward, we orthogonalize the samples as preparation for constructing various matrix approximations.

We believe that the idea of using dimension reduction for algorithmic matrix approximation first appeared in a 1998 paper of Papadimitriou et al. [104, 105], who described an application to latent semantic indexing (LSI). They suggested projecting the input matrix onto a random subspace and compressing the original matrix to (a subspace of) the range of the projected matrix. They established error bounds that echo the result (2.13) of Frieze et al. [58]. Although the Euclidean column selection method is a more computationally efficient way to obtain this type of error bound, dimension reduction has other advantages, e.g., in terms of accuracy.

Sarlós argued in [118] that the computational costs of dimension reduction can be reduced substantially by means of the structured random maps proposed by Ailon–Chazelle [3]. Sarlós used these ideas to develop efficient randomized algorithms for least-squares problems; he also studied approximate matrix multiplication and low-rank matrix approximation. The recent paper [102] analyzes a very similar matrix approximation algorithm using Rudelson and Vershynin's methods [116].

The initial work of Sarlós on structured dimension reduction did not immediately yield algorithms for low-rank matrix approximation that were superior to classical techniques. Woolfe et al. showed how to obtain an improvement in asymptotic computational cost, and they applied these techniques to problems in scientific computing [137]. Related work includes [86, 88].

Martinsson–Rokhlin–Tygert have studied dimension reduction using a Gaussian transform matrix, and they demonstrated that this approach performs much better than earlier analyses had suggested [91]. Their work highlights the importance of oversampling, and their error bounds are very similar to the estimate (2.9) we presented in the introduction. They also demonstrated that dimension reduction can be used to compute an interpolative decomposition of the input matrix, which is essentially equivalent to performing column subset selection.

Rokhlin–Szlam–Tygert have shown that combining dimension reduction with a power iteration is an effective way to improve its performance [112]. These ideas lead to very efficient randomized methods for large-scale PCA [69]. An efficient, numerically stable version of the power iteration is discussed in §2.4.5, as well as [92]. Related ideas appear in a paper of Roweis [114].

Very recently, Clarkson and Woodruff [29] have developed one-pass algorithms for performing low-rank matrix approximation, and they have established lower bounds which prove that many of their algorithms have optimal or near-optimal resource guarantees, modulo constants.

2.2.1.4 Approximation by submatrices

The matrix approximation literature contains a subgenre that discusses methods for building an approximation from a submatrix and computed coefficient matrices. For example, we can construct an approximation using a subcollection of columns (the interpolative decomposition), a subcollection of rows and a subcollection of columns (the CUR decomposition), or a square submatrix (the matrix skeleton). This type of decomposition was developed and studied in several papers, including [26, 64, 126]. For data analysis applications, see the recent paper [89].

A number of works develop randomized algorithms for this class of matrix approximations. Drineas et al. have developed techniques for computing CUR decompositions, which express $\mathsf{A} \approx$

CUR, where C and R denote small column and row submatrices of A and where U is a small linkage matrix. These methods identify columns (rows) that approximate the range (corange) of the matrix; the linkage matrix is then computed by solving a small least-squares problem. A randomized algorithm for CUR approximation with controlled absolute error appears in [47]; a relative error algorithm appears in [51]. We also mention a paper on computing a closely related factorization called the *compact matrix decomposition* [129].

It is also possible to produce interpolative decompositions and matrix skeletons using randomized methods, as discussed in [91, 112] and §2.5.2 of the present work.

2.2.1.5 Other numerical problems

The literature contains a variety of other randomized algorithms for solving standard problems in and around numerical linear algebra. We list some of the basic references.

Tensor skeletons. Randomized column selection methods can be used to produce CUR-type decompositions of higher-order tensors [49].

Matrix multiplication. Column selection and dimension reduction techniques can be used to accelerate the multiplication of rank-deficient matrices [45, 118]. See also [10].

Overdetermined linear systems. The randomized Kaczmarz algorithm is a linearly convergent iterative method that can be used to solve overdetermined linear systems [101, 128].

Overdetermined least squares. Fast dimension-reduction maps can sometimes accelerate the solution of overdetermined least-squares problems [52, 118].

Nonnegative least squares. Fast dimension reduction maps can be used to reduce the size of nonnegative least-squares problems [16].

Preconditioned least squares. Randomized matrix approximations can be used to precondition conjugate gradient to solve least-squares problems [113].

Other regression problems. Randomized algorithms for ℓ_1 regression are described in [28]. Regression in ℓ_p for $p \in [1, \infty)$ has also been considered [31].

Facility location. The Fermat–Weber facility location problem can be viewed as matrix approximation with respect to a different discrepancy measure. Randomized algorithms for this type of problem appear in [121].

2.2.1.6 Compressive sampling

Although randomized matrix approximation and compressive sampling are based on some common intuitions, it is facile to consider either one as a subspecies of the other. We offer a short

overview of the field of compressive sampling—especially the part connected with matrices—so we can highlight some of the differences.

The theory of compressive sampling starts with the observation that many types of vector-space data are **compressible**. That is, the data are approximated well using a short linear combination of basis functions drawn from a fixed collection [42]. For example, natural images are well approximated in a wavelet basis; numerically low-rank matrices are well approximated as a sum of rank-one matrices. The idea behind compressive sampling is that suitably chosen random samples from this type of compressible object carry a large amount of information. Furthermore, it is possible to reconstruct the compressible object from a small set of these random samples, often by solving a convex optimization problem. The initial discovery works of Candès–Romberg–Tao [20] and Donoho [41] were written in 2004.

The earliest work in compressive sampling focused on vector-valued data; soon after, researchers began to study compressive sampling for matrices. In 2007, Recht–Fazel–Parillo demonstrated that it is possible to reconstruct a rank-deficient matrix from Gaussian measurements [111]. More recently, Candès–Recht [22] and Candès–Tao [23] considered the problem of completing a low-rank matrix from a random sample of its entries.

The usual goals of compressive sampling are (i) to design a method for collecting informative, nonadaptive data about a compressible object and (ii) to reconstruct a compressible object given some measured data. In both cases, there is an implicit assumption that we have limited—if any—access to the underlying data.

In the problem of matrix approximation, we typically have a complete representation of the matrix at our disposal. The point is to compute a simpler representation as efficiently as possible under some operational constraints. In particular, we would like to perform as little computation as we can, but we are usually allowed to revisit the input matrix. Because of the different focus, randomized matrix approximation algorithms require fewer random samples from the matrix and use fewer computational resources than compressive sampling reconstruction algorithms.

2.2.2 Origins

This section attempts to identify some of the major threads of research that ultimately led to the development of the randomized techniques we discuss in this paper.

2.2.2.1 Random embeddings

The field of random embeddings is a major precursor to randomized matrix approximation. In a celebrated 1984 paper [74], Johnson and Lindenstrauss showed that the pairwise distances among a collection of N points in a Euclidean space are approximately maintained when the points are mapped randomly to a Euclidean space of dimension $\mathrm{O}(\log N)$. In other words, random embeddings preserve Euclidean geometry. Shortly afterward, Bourgain showed that appropriate

random low-dimensional embeddings preserve the geometry of point sets in finite-dimensional ℓ_1 spaces [15].

These observations suggest that we might be able to solve some computational problems of a geometric nature more efficiently by translating them into a lower-dimensional space and solving them there. This idea was cultivated by the theoretical computer science community beginning in the late 1980s, with research flowering in the late 1990s. In particular, nearest-neighbor search can benefit from dimension-reduction techniques [73, 78, 80]. The papers [57, 104] were apparently the first to apply this approach to linear algebra.

Around the same time, researchers became interested in simplifying the form of dimension reduction maps and improving the computational cost of applying the map. Several researchers developed refined results on the performance of a Gaussian matrix as a linear dimension reduction map [32, 73, 93]. Achlioptas demonstrated that discrete random matrices would serve nearly as well [1]. In 2006, Ailon and Chazelle proposed the *fast Johnson–Lindenstrauss transform* [3], which combines the speed of the FFT with the favorable embedding properties of a Gaussian matrix. Subsequent refinements appear in [4, 87]. Sarlós then imported these techniques to study several problems in numerical linear algebra, which has led to some of the fastest algorithms currently available [88, 137].

2.2.2.2 Data streams

Muthukrishnan argues that a distinguishing feature of modern data is the manner in which it is **presented** to us. The sheer volume of information and the speed at which it must be processed tax our ability to **transmit** the data elsewhere, to **compute** complicated functions on the data, or to **store** a substantial part of the data [100, §3]. As a result, computer scientists have started to develop algorithms that can address familiar computational problems under these novel constraints. The data stream phenomenon is one of the primary justifications cited by [45] for developing pass-efficient methods for numerical linear algebra problems, and it is also the focus of the recent treatment [29].

One of the methods for dealing with massive data sets is to maintain **sketches**, which are small summaries that allow functions of interest to be calculated. In the simplest case, a sketch is simply a random projection of the data, but it might be a more sophisticated object [100, §5.1]. The idea of sketching can be traced to the work of Alon et al. [6, 5].

2.2.2.3 Numerical linear algebra

Classically, the field of numerical linear algebra has focused on developing deterministic algorithms that produce highly accurate matrix approximations with provable guarantees. Nevertheless, randomized techniques have appeared in several environments.

One of the original examples is the use of random models for arithmetical errors, which was pioneered by von Neumann and Goldstine. Their papers [135, 136] stand among the first works

to study the properties of random matrices. The earliest numerical linear algebra algorithm that depends essentially on randomized techniques is probably Dixon's method for estimating norms and condition numbers [39].

Another situation where randomness commonly arises is the initialization of iterative methods for computing invariant subspaces. For example, most numerical linear algebra texts advocate random selection of the starting vector for the power method because it ensures that the vector has a nonzero component in the direction of a dominant eigenvector. Woźniakowski and coauthors have analyzed the performance of the power method and the Lanczos iteration given a random starting vector [79, 85].

Among other interesting applications of randomness, we mention the work by Parker and Pierce, which applies a randomized FFT to eliminate pivoting in Gaussian elimination [106], work by Demmel et al. who have studied randomization in connection with the stability of fast methods for linear algebra [35], and work by Le and Parker utilizing randomized methods for stabilizing fast linear algebraic computations based on recursive algorithms, such as Strassen's matrix multiplication [81].

2.2.2.4 Scientific computing

One of the first algorithmic applications of randomness is the method of Monte Carlo integration introduced by Von Neumann and Ulam [95], and its extensions, such as the Metropolis algorithm for simulations in statistical physics. (See [9] for an introduction.) The most basic technique is to estimate an integral by sampling m points from the measure and computing an empirical mean of the integrand evaluated at the sample locations:

$$\int f(x)\,\mathrm{d}\mu(x) \approx \frac{1}{m}\sum_{i=1}^{m} f(X_i),$$

where X_i are independent and identically distributed according to the probability measure μ. The law of large numbers (usually) ensures that this approach produces the correct result in the limit as $m \to \infty$. Unfortunately, the approximation error typically has a standard deviation of $m^{-1/2}$, and the method provides no certificate of success.

The disappointing computational profile of Monte Carlo integration seems to have inspired a distaste for randomized approaches within the scientific computing community. Fortunately, there are many other types of randomized algorithms—such as the ones in this paper—that do not suffer from the same shortcomings.

2.2.2.5 Geometric functional analysis

There is one more character that plays a central role in our story: the probabilistic method in geometric analysis. Many of the algorithms and proof techniques ultimately come from work in this beautiful but recondite corner of mathematics.

Dvoretsky's theorem [53] states (roughly) that every infinite-dimensional Banach space contains an n-dimensional subspace whose geometry is essentially the same as an n-dimensional Hilbert space, where n is an arbitrary natural number. In 1971, V. D. Milman developed a striking proof of this result by showing that a **random** n-dimensional subspace of an N-dimensional Banach space has this property with exceedingly high probability, provided that N is large enough [96]. Milman's article debuted the **concentration of measure phenomenon**, which is a geometric interpretation of the classical idea that regular functions of independent random variables rarely deviate far from their mean. This work opened a new era in geometric analysis where the probabilistic method became a basic instrument.

Another prominent example of measure concentration is Kashin's computation of the Gel'fand widths of the ℓ_1 ball [77], subsequently refined in [59]. This work showed that a **random** $(N-n)$-dimensional projection of the N-dimensional ℓ_1 ball has an astonishingly small Euclidean diameter: approximately $\sqrt{(1+\log(N/n))/n}$. In contrast, a nonzero projection of the ℓ_2 ball always has Euclidean diameter one. This basic geometric fact undergirds recent developments in compressive sampling [21].

We have already described a third class of examples: the randomized embeddings of Johnson–Lindenstrauss [74] and of Bourgain [15].

Finally, we mention Maurey's technique of empirical approximation. The original work was unpublished; one of the earliest applications appears in [24, §1]. Although Maurey's idea has not received as much press as the examples above, it can lead to simple and efficient algorithms for sparse approximation. For some examples in machine learning, consider [8, 84, 119, 110]

The importance of random constructions in the geometric analysis community has led to the development of powerful techniques for studying random matrices. Classical random matrix theory focuses on a detailed asymptotic analysis of the spectral properties of special classes of random matrices. In contrast, geometric analysts know methods for determining the approximate behavior of rather complicated finite-dimensional random matrices. See [34] for a fairly current survey article. We also mention the works of Rudelson [115] and Rudelson–Vershynin [116], which describe powerful tools for studying random matrices drawn from certain discrete distributions. Their papers are rooted deeply in the field of geometric functional analysis, but they reach out toward computational applications.

2.3 Linear algebraic preliminaries

This section summarizes the background we need for the detailed description of randomized algorithms in §§2.4–2.6 and the analysis in §§2.8–2.11. We introduce notation in §2.3.1, describe some standard matrix decompositions in §2.3.2, and briefly review standard techniques for computing matrix factorizations in §2.3.3.

2.3.1 Basic definitions

The standard Hermitian geometry for $\mathbb{C}^n$ is induced by the inner product

$$\langle \mathbf{x},\ \mathbf{y} \rangle = \mathbf{x} \cdot \mathbf{y} = \sum\nolimits_j x_j\, \overline{y}_j.$$

The associated norm is

$$\|\mathbf{x}\|^2 = \langle \mathbf{x},\ \mathbf{x} \rangle = \sum\nolimits_j |x_j|^2.$$

We usually measure the magnitude of a matrix A with the operator norm

$$\|\mathsf{A}\| = \max_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathsf{A}\mathbf{x}\|}{\|\mathbf{x}\|},$$

which is often referred to as the *spectral norm*. The Frobenius norm is given by

$$\|\mathsf{A}\|_{\mathrm{F}} = \left[\sum\nolimits_{jk} |a_{jk}|^2 \right]^{1/2}.$$

The conjugate transpose, or *adjoint*, of a matrix A is denoted A^*. The important identities

$$\|\mathsf{A}\|^2 = \|\mathsf{A}^*\mathsf{A}\| = \|\mathsf{A}\mathsf{A}^*\|$$

hold for each matrix A.

We say that a matrix U is *orthonormal* if its columns form an orthonormal set with respect to the Hermitian inner product. An orthonormal matrix U preserves geometry in the sense that $\|\mathsf{U}\mathbf{x}\| = \|\mathbf{x}\|$ for every vector $\mathbf{x}$. A **unitary** matrix is a square orthonormal matrix, and an **orthogonal** matrix is a real unitary matrix. Unitary matrices satisfy the relations $\mathsf{U}\mathsf{U}^* = \mathsf{U}^*\mathsf{U} = \mathsf{I}$. Both the operator norm and the Frobenius norm are *unitarily invariant*, which means that

$$\|\mathsf{U}\mathsf{A}\mathsf{V}^*\| = \|\mathsf{A}\| \qquad \text{and} \qquad \|\mathsf{U}\mathsf{A}\mathsf{V}^*\|_{\mathrm{F}} = \|\mathsf{A}\|_{\mathrm{F}}$$

for every matrix A and all orthonormal matrices U and V

We use the notation of [61] to denote submatrices. If A is a matrix with entries a_{ij}, and if $I = [i_1, i_2, \ldots, i_p]$ and $J = [j_1, j_2, \ldots, j_q]$ are two index vectors, then the associated $p \times q$ submatrix is expressed as

$$\mathsf{A}_{(I,J)} = \begin{bmatrix} a_{i_1,j_1} & \cdots & a_{i_1,j_q} \\ \vdots & & \vdots \\ a_{i_p,j_1} & \cdots & a_{i_p,j_q} \end{bmatrix}.$$

For column- and row-submatrices, we use the standard abbreviations

$$\mathsf{A}_{(:\,,J)} = \mathsf{A}_{([1,2,\ldots,m],J)}, \qquad \text{and} \qquad \mathsf{A}_{(I,:\,)} = \mathsf{A}_{(I,[1,2,\ldots,n])}.$$

2.3.2 Standard matrix factorizations

This section defines three basic matrix decompositions. Methods for computing them are described in §2.3.3.

2.3.2.1 The pivoted QR factorization

Each $m \times n$ matrix A of rank k admits a decomposition

$$\mathsf{A} = \mathsf{Q}\mathsf{R},$$

where Q is an $m \times k$ orthonormal matrix, and R is a $k \times n$ *weakly upper-triangular* matrix. That is, there exists a permutation J of the numbers $\{1, 2, \ldots, n\}$ such that $\mathsf{R}_{(:\,,J)}$ is upper triangular. Moreover, the diagonal entries of $\mathsf{R}_{(:\,,J)}$ are weakly decreasing. See [61, §5.4.1] for details.

2.3.2.2 The singular value decomposition (SVD)

Each $m \times n$ matrix A of rank k admits a factorization

$$\mathsf{A} = \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*,$$

where U is an $m \times k$ orthonormal matrix, V is an $n \times k$ orthonormal matrix, and $\mathsf{\Sigma}$ is a $k \times k$ nonnegative, diagonal matrix

$$\mathsf{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_k \end{bmatrix}.$$

The numbers σ_j are called the *singular values* of A. They are arranged in weakly decreasing order:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq 0.$$

The columns of U and V are called *left singular vectors* and *right singular vectors*, respectively.

Singular values are connected with the approximability of matrices. For each j, the number σ_{j+1} equals the spectral-norm discrepancy between A and an optimal rank-j approximation [97]. That is,

$$\sigma_{j+1} = \min\{\|\mathsf{A} - \mathsf{B}\| \,:\, \mathsf{B} \text{ has rank } j\}. \tag{2.15}$$

In particular, $\sigma_1 = \|\mathsf{A}\|$. See [61, §2.5.3 and §5.4.5] for additional details.

2.3.2.3 The interpolative decomposition (ID)

Our final factorization identifies a collection of k columns from a rank-k matrix A that span the range of A. To be precise, we can compute an index set $J = [j_1, \ldots, j_k]$ such that

$$\mathsf{A} = \mathsf{A}_{(:\,,J)}\,\mathsf{X},$$

where X is a $k \times n$ matrix that satisfies $\mathsf{X}_{(:\,,J)} = \mathsf{I}_k$. Furthermore, no entry of X has magnitude larger than two. In other words, this decomposition expresses each column of A using a linear

combination of k fixed columns with **bounded** coefficients. Stable and efficient algorithms for computing the ID appear in the papers [26, 68].

It is also possible to compute a two-sided ID

$$\mathsf{A} = \mathsf{W}\,\mathsf{A}_{(J',J)}\,\mathsf{X},$$

where J' is an index set identifying k of the rows of A, and W is an $m \times k$ matrix that satisfies $\mathsf{W}_{(J',:)} = \mathsf{I}_k$ and whose entries are all bounded by two.

Remark 5. There always exists an ID where the entries in the factor X have magnitude bounded by one. Known proofs of this fact are constructive, e.g., [103, Lem. 3.3], but they require us to find a collection of k columns that has "maximum volume." It is NP-hard to identify a subset of columns with this type of extremal property [27]. We find it remarkable that ID computations are possible as soon as the bound on X is relaxed.

2.3.3 Techniques for computing standard factorizations

This section discusses some established deterministic techniques for computing the factorizations presented in §2.3.2. The material on pivoted QR and SVD can be located in any major text on numerical linear algebra, such as [61, 132]. References for the ID include [68, 26].

2.3.3.1 Computing the full decomposition

It is possible to compute the full QR factorization or the full SVD of an $m \times n$ matrix to double-precision accuracy with $\mathrm{O}(mn \min\{m, n\})$ flops. Techniques for computing the SVD are iterative by necessity, but they converge so fast that we can treat them as finite for practical purposes.

2.3.3.2 Computing partial decompositions

Suppose that an $m \times n$ matrix has numerical rank k, where k is substantially smaller than m and n. In this case, it is possible to produce a structured low-rank decomposition that approximates the matrix well. Sections 2.4 and 2.5 describe a set of randomized techniques for obtaining these partial decompositions. This section briefly reviews the classical techniques, which also play a role in developing randomized methods.

To compute a partial QR decomposition, the classical device is the Businger–Golub algorithm, which performs successive orthogonalization with pivoting on the columns of the matrix.

The procedure halts when the Frobenius norm of the remaining columns is less than a computational tolerance ε. Letting ℓ denote the number of steps required, the process results in a partial factorization

$$\mathsf{A} = \mathsf{Q}\mathsf{R} + \mathsf{E}, \tag{2.16}$$

where Q is an $m \times \ell$ orthonormal matrix, R is a $\ell \times n$ weakly upper-triangular matrix, and E is a residual that satisfies $\|\mathsf{E}\|_{\mathrm{F}} \leq \varepsilon$. The computational cost is $\mathrm{O}(\ell mn)$, and the number ℓ of steps taken is typically close to the minimal rank k for which precision ε (in the Frobenius norm) is achievable. The Businger–Golub algorithm can in principle significantly overpredict the rank, but in practice this problem is very rare provided that orthonormality is maintained scrupulously.

Subsequent research has led to strong rank-revealing QR algorithms that succeed for all matrices. For example, the Gu–Eisenstat algorithm [68] (setting their parameter $f = 2$) produces an QR decomposition of the form (2.16), where

$$\|\mathsf{E}\| \leq \sqrt{1 + 4k(n-k)} \cdot \sigma_{k+1}.$$

Recall that σ_{k+1} is the minimal error possible in a rank-k approximation [97]. The cost of the Gu–Eisenstat algorithm is typically $O(kmn)$, but it can be slightly higher in rare cases. The algorithm can also be used to obtain an approximate ID [26].

To compute an approximate SVD of a general $m \times n$ matrix, the most straightforward technique is to compute the full SVD and truncate it. This procedure is stable and accurate, but it requires $\mathrm{O}(mn \min\{m, n\})$ flops. A more efficient approach is to compute a partial QR factorization and postprocess the factors to obtain a partial SVD using the methods described below in §2.3.3.3. This scheme takes only $\mathrm{O}(kmn)$ flops. Krylov subspace methods can also compute partial SVDs at a comparable cost of $\mathrm{O}(kmn)$, but they are less robust.

Note that all the techniques described in this section require extensive random access to the matrix, and they can be very slow when the matrix is stored out-of-core.

2.3.3.3 Converting from one partial factorization to another

Suppose that we have obtained a partial decomposition of a matrix A by some means:

$$\|\mathsf{A} - \mathsf{C}\mathsf{B}\| \leq \varepsilon,$$

where B and C have rank k. Given this information, we can efficiently compute any of the basic factorizations.

We construct a partial QR factorization using the following three steps:

(1) Compute a QR factorization of C so that $\mathsf{C} = \mathsf{Q}_1\mathsf{R}_1$.

(2) Form the product $\mathsf{D} = \mathsf{R}_1\mathsf{B}$, and compute a QR factorization: $\mathsf{D} = \mathsf{Q}_2\mathsf{R}$.

(3) Form the product $\mathsf{Q} = \mathsf{Q}_1\mathsf{Q}_2$.

The result is an orthonormal matrix Q and a weakly upper-triangular matrix R such that $\|\mathsf{A} - \mathsf{QR}\| \leq \varepsilon$.

An analogous technique yields a partial SVD:

(1) Compute a QR factorization of C so that $\mathsf{C} = \mathsf{Q}_1\mathsf{R}_1$.

(2) Form the product $\mathsf{D} = \mathsf{R}_1\mathsf{B}$, and compute an SVD: $\mathsf{D} = \mathsf{U}_2\mathsf{\Sigma}\mathsf{V}^*$.

(3) Form the product $\mathsf{U} = \mathsf{Q}_1\mathsf{U}_2$.

The result is a diagonal matrix $\mathsf{\Sigma}$ and orthonormal matrices U and V such that $\|\mathsf{A} - \mathsf{U\Sigma V}^*\| \leq \varepsilon$.

Converting B and C into a partial ID is a one-step process:

(1) Compute J and X such that $\mathsf{B} = \mathsf{B}_{(:\,,J)}\mathsf{X}$.

Then $\mathsf{A} \approx \mathsf{A}_{(:\,,J)}\mathsf{X}$, but the approximation error may deteriorate from the initial estimate. For example, if we compute the ID using the Gu–Eisenstat algorithm [68] with the parameter $f = 2$, then the error $\left\|\mathsf{A} - \mathsf{A}_{(:\,,J)}\mathsf{X}\right\| \leq (1 + \sqrt{1 + 4k(n-k)}) \cdot \varepsilon$. Compare this bound with Lemma 2.5.1 below.

2.3.3.4 Krylov-subspace methods

Suppose that the matrix A can be applied rapidly to vectors, as happens when A is sparse or structured. Then Krylov subspace techniques can very effectively and accurately compute partial spectral decompositions. For concreteness, assume that A is Hermitian. The idea of these techniques is to fix a starting vector $\boldsymbol{\omega}$ and to seek approximations to the eigenvectors within the corresponding *Krylov subspace*

$$\mathcal{V}_q(\boldsymbol{\omega}) = \operatorname{span}\{\boldsymbol{\omega}, \mathsf{A}\boldsymbol{\omega}, \mathsf{A}^2\boldsymbol{\omega}, \dots, \mathsf{A}^{q-1}\boldsymbol{\omega}\}.$$

Krylov methods also come in blocked versions, in which the starting *vector* $\boldsymbol{\omega}$ is replaced by a starting *matrix* Ω. A common recommendation is to draw a starting vector $\boldsymbol{\omega}$ (or starting matrix Ω) from a standardized Gaussian distribution, which indicates a significant overlap between Krylov methods and the methods in this paper.

The most basic versions of Krylov methods for computing spectral decompositions are numerically unstable. High-quality implementations require that we incorporate restarting strategies, techniques for maintaining high-quality bases for the Krylov subspaces, etc. The diversity and complexity of such methods make it hard to state a precise computational cost, but in the environment we consider in this paper, a typical cost for a fully stable implementation would be

$$T_{\text{Krylov}} \sim k\, T_{\text{mult}} + k^2(m+n), \tag{2.17}$$

where T_{mult} is the cost of a matrix–vector multiplication.

Part II: Algorithms

This part of the paper, §§2.4–2.7, provides detailed descriptions of randomized algorithms for constructing low-rank approximations to matrices. As discussed in §2.1.2, we split the problem into two stages. In Stage A, we construct a subspace that captures the action of the input matrix. In Stage B, we use this subspace to obtain an approximate factorization of the matrix.

Section 2.4 develops randomized methods for completing Stage A, and §2.5 describes deterministic methods for Stage B. Section 2.6 compares the computational costs of the resulting two-stage algorithm with the classical approaches outlined in §2.3. Finally, §2.7 illustrates the performance of the randomized schemes via numerical examples.

2.4 Stage A: Randomized schemes for approximating the range

This section outlines techniques for constructing a subspace that captures most of the action of a matrix. We begin with a recapitulation of the proto-algorithm that we introduced in §2.1.3. We discuss how it can be implemented in practice (§2.4.1) and then consider the question of how many random samples to acquire (§2.4.2). Afterward, we present several ways in which the basic scheme can be improved. Sections 2.4.3 and 2.4.4 explain how to address the situation where the numerical rank of the input matrix is not known in advance. Section 2.4.5 shows how to modify the scheme to improve its accuracy when the singular spectrum of the input matrix decays slowly. Finally, §2.4.6 describes how the scheme can be accelerated by using a structured random matrix.

2.4.1 The proto-algorithm revisited

The most natural way to implement the proto-algorithm from §2.1.3 is to draw a random test matrix Ω from the standard Gaussian distribution. That is, each entry of Ω is an independent Gaussian random variable with mean zero and variance one. For reference, we formulate the resulting scheme as Algorithm 4.1.

The number T_{basic} of flops required by Algorithm 4.1 satisfies

$$T_{\text{basic}} \sim \ell n\, T_{\text{rand}} + \ell\, T_{\text{mult}} + \ell^2 m \tag{2.18}$$

where T_{rand} is the cost of generating a Gaussian random number and T_{mult} is the cost of multiplying A by a vector. The three terms in (2.18) correspond directly with the three steps of Algorithm 4.1.

Empirically, we have found that the performance of Algorithm 4.1 depends very little on the quality of the random number generator used in Step 1.

The actual cost of Step 2 depends substantially on the matrix A and the computational environment that we are working in. The estimate (2.18) suggests that Algorithm 4.1 is especially efficient when the matrix–vector product $\mathbf{x} \mapsto \mathsf{A}\mathbf{x}$ can be evaluated rapidly. In particular, the

Algorithm 4.1: Randomized Range Finder

Given an $m \times n$ matrix A, and an integer ℓ, this scheme computes an $m \times \ell$ orthonormal matrix Q whose range approximates the range of A.

1 Draw an $n \times \ell$ Gaussian random matrix Ω.
2 Form the $m \times \ell$ matrix $\mathsf{Y} = \mathsf{A}\Omega$.
3 Construct an $m \times \ell$ matrix Q whose columns form an orthonormal basis for the range of Y, e.g., using the QR factorization $\mathsf{Y} = \mathsf{QR}$.

scheme is appropriate for approximating sparse or structured matrices. Turn to §2.6 for more details.

The most important implementation issue arises when performing the basis calculation in Step 3. Typically, the columns of the sample matrix Y are almost linearly dependent, so it is imperative to use stable methods for performing the orthonormalization. We have found that the Gram–Schmidt procedure, augmented with the *double orthogonalization* described in [12], is both convenient and reliable. Methods based on Householder reflectors or Givens rotations also work very well. Note that very little is gained by pivoting because the columns of the random matrix Y are independent samples drawn from the same distribution.

2.4.2 The number of samples required

The goal of Algorithm 4.1 is to produce an orthonormal matrix Q with few columns that achieves

$$\|(\mathsf{I} - \mathsf{QQ}^*)\mathsf{A}\| \leq \varepsilon, \tag{2.19}$$

where ε is a specified tolerance. The number of columns ℓ that the algorithm needs to reach this threshold is usually slightly larger than the minimal rank k of the smallest basis that verifies (2.19). We refer to this discrepancy $p = \ell - k$ as the *oversampling parameter*. The size of the oversampling parameter depends on several factors:

The matrix dimensions. Very large matrices may require more oversampling.

The singular spectrum. The more rapid the decay of the singular values, the less oversampling is needed. In the extreme case that the matrix has exact rank k, it is not necessary to oversample.

The random test matrix. Gaussian matrices succeed with very little oversampling, but are not always the most cost-effective option. The structured random matrices discussed in §2.4.6 may require substantial oversampling, but they still yield computational gains in certain settings.

The theoretical results in Part III provide detailed information about how the behavior of randomized schemes depends on these factors. For the moment, we limit ourselves to some general remarks on implementation issues.

For Gaussian test matrices, it is adequate to choose the oversampling parameter to be a small constant, such as $p = 5$ or $p = 10$. There is rarely any advantage to select $p > k$. This observation, first presented in [91], demonstrates that a Gaussian test matrix results in a negligible amount of extra computation.

In practice, the target rank k is rarely known in advance. Randomized algorithms are usually implemented in an adaptive fashion where the number of samples is increased until the error satisfies the desired tolerance. In other words, the user never **chooses** the oversampling parameter. Theoretical results that bound the amount of oversampling are valuable primarily as aids for designing algorithms. We develop an adaptive approach in §§2.4.3–2.4.4.

The computational bottleneck in Algorithm 4.1 is usually the formation of the product $\mathsf{A}\Omega$. As a result, it often pays to draw a larger number ℓ of samples than necessary because the user can minimize the cost of the matrix multiplication with tools such as blocking of operations, high-level linear algebra subroutines, parallel processors, etc. This approach may lead to an ill-conditioned sample matrix Y, but the orthogonalization in Step 3 of Algorithm 4.1 can easily identify the numerical rank of the sample matrix and ignore the excess samples. Furthermore, Stage B of the matrix approximation process succeeds even when the basis matrix Q has a larger dimension than necessary.

2.4.3 A posteriori error estimation

Algorithm 4.1 is designed for solving the fixed-rank problem, where the target rank of the input matrix is specified in advance. To handle the fixed-precision problem, where the parameter is the computational tolerance, we need a scheme for estimating how well a putative basis matrix Q captures the action of the matrix A. To do so, we develop a probabilistic error estimator. These methods are inspired by work of Dixon [39]; our treatment follows [88, 137].

The exact approximation error is $\|(\mathsf{I} - \mathsf{Q}\mathsf{Q}^*)\mathsf{A}\|$. It is intuitively plausible that we can obtain some information about this quantity by computing $\|(\mathsf{I} - \mathsf{Q}\mathsf{Q}^*)\mathsf{A}\boldsymbol{\omega}\|$, where $\boldsymbol{\omega}$ is a standard Gaussian vector. This notion leads to the following method. Draw a sequence $\{\boldsymbol{\omega}^{(i)} : i = 1, 2, \dots, r\}$ of standard Gaussian vectors, where r is a small integer that balances computational cost and reliability. Then

$$\|(\mathsf{I} - \mathsf{Q}\mathsf{Q}^*)\mathsf{A}\| \le 10\sqrt{\frac{2}{\pi}} \max_{i=1,\dots,r} \left\|(\mathsf{I} - \mathsf{Q}\mathsf{Q}^*)\mathsf{A}\boldsymbol{\omega}^{(i)}\right\| \tag{2.20}$$

with probability at least $1 - 10^{-r}$. This statement follows by setting $\mathsf{B} = (\mathsf{I} - \mathsf{Q}\mathsf{Q}^*)\mathsf{A}$ and $\alpha = 10$ in the following lemma, whose proof appears in [137, §3.4].

Lemma 2.4.1. *Let B be a real $m \times n$ matrix. Fix a positive integer r and a real number $\alpha > 1$. Draw an independent family $\{\boldsymbol{\omega}^{(i)} : i = 1, 2, \dots, r\}$ of standard Gaussian vectors. Then*

$$\|\mathsf{B}\| \le \alpha\sqrt{\frac{2}{\pi}} \max_{i=1,\dots,r} \left\|\mathsf{B}\boldsymbol{\omega}^{(i)}\right\|$$

except with probability α^{-r}.

The critical point is that the error estimate (2.20) is computationally inexpensive because it requires only a small number of matrix–vector products. Therefore, we can make a lowball guess for the numerical rank of A and add more samples if the error estimate is too large. The asymptotic cost of Algorithm 4.1 is preserved if we double our guess for the rank at each step. For example, we can start with 32 samples, compute another 32, then another 64, etc.

Remark 6. The estimate (2.20) is actually somewhat crude. We can obtain a better estimate at a similar computational cost by initializing a power iteration with a random vector and repeating the process several times [88].

2.4.4 Error estimation (almost) for free

The error estimate described in §2.4.3 can be combined with any method for constructing an approximate basis for the range of a matrix. In this section, we explain how the error estimator can be incorporated into Algorithm 4.1 at almost no additional cost.

To be precise, let us suppose that A is an $m \times n$ matrix and ε is a computational tolerance. We seek an integer ℓ and an $m \times \ell$ orthonormal matrix $\mathsf{Q}^{(\ell)}$ such that

$$\left\| \left(\mathsf{I} - \mathsf{Q}^{(\ell)}(\mathsf{Q}^{(\ell)})^*\right)\mathsf{A} \right\| \le \varepsilon. \tag{2.21}$$

The size ℓ of the basis will typically be slightly larger than the size k of the smallest basis that achieves this error.

The basic observation behind the adaptive scheme is that we can generate the basis in Step 3 of Algorithm 4.1 incrementally. Starting with an empty basis matrix $\mathsf{Q}^{(0)}$, the following scheme generates an orthonormal matrix whose range captures the action of A:

for $i = 1, 2, 3, \ldots$
  Draw an $n \times 1$ Gaussian random vector $\boldsymbol{\omega}^{(i)}$, and set $\mathbf{y}^{(i)} = \mathsf{A}\boldsymbol{\omega}^{(i)}$.
  Compute $\tilde{\mathbf{q}}^{(i)} = \left(\mathsf{I} - \mathsf{Q}^{(i-1)}(\mathsf{Q}^{(i-1)})^*\right)\mathbf{y}^{(i)}$.
  Normalize $\mathbf{q}^{(i)} = \tilde{\mathbf{q}}^{(i)} / \left\|\tilde{\mathbf{q}}^{(i)}\right\|$, and form $\mathsf{Q}^{(i)} = [\mathsf{Q}^{(i-1)}\ \mathbf{q}^{(i)}]$.
end for

How do we know when we have reached a basis $\mathsf{Q}^{(\ell)}$ that verifies (2.21)? The answer becomes apparent once we observe that the vectors $\tilde{\mathbf{q}}^{(i)}$ are precisely the vectors that appear in the error bound (2.20). The resulting rule is that we break the loop once we observe r consecutive vectors $\tilde{\mathbf{q}}^{(i)}$ whose norms are smaller than $\varepsilon/(10\sqrt{2/\pi})$.

A formal description of the resulting algorithm appears as Algorithm 4.2. A potential complication of the method is that the vectors $\tilde{\mathbf{q}}^{(i)}$ become small as the basis starts to capture most of the action of A. In finite-precision arithmetic, their direction is extremely unreliable. To address

this problem, we simply reproject the normalized vector $\mathbf{q}^{(i)}$ onto range$(\mathsf{Q}^{(i-1)})^{\perp}$ in steps 7 and 8 of Algorithm 4.2.

The CPU time requirements of Algorithms 4.2 and 4.1 are essentially identical. Although Algorithm 4.2 computes the last few samples purely to obtain the error estimate, this apparent extra cost is offset by the fact that Algorithm 4.1 always includes an oversampling factor. The failure probability stated for Algorithm 4.2 is pessimistic because it is derived from a simple union bound argument. In practice, the error estimator is reliable in a range of circumstances when we take $r = 10$.

Remark 7. The calculations in Algorithm 4.2 can be organized so that each iteration processes a block of samples simultaneously. This revision can lead to dramatic improvements in speed because it allows us to exploit higher-level linear algebra subroutines (e.g., BLAS3) or parallel processors. Although blocking can lead to the generation of unnecessary samples, this outcome is generally harmless, as noted in §2.4.2.

2.4.5 A modified scheme for matrices whose singular values decay slowly

The techniques described in §2.4.1 and §2.4.4 work well for matrices whose singular values exhibit some decay, but they may produce a poor basis when the input matrix has a flat singular spectrum or when the input matrix is very large. In this section, we describe techniques, originally proposed in [67, 112], for improving the accuracy of randomized algorithms in these situations. Related earlier work includes [114] and the literature on classical orthogonal iteration methods [61, p. 332].

The intuition behind these techniques is that the singular vectors associated with small singular values interfere with the calculation, so we reduce their weight relative to the dominant singular vectors by taking powers of the matrix to be analyzed. More precisely, we wish to apply the randomized sampling scheme to the matrix $\mathsf{B} = (\mathsf{A}\mathsf{A}^*)^q\mathsf{A}$, where q is a small integer. The matrix B has the same singular vectors as the input matrix A, but its singular values decay much more quickly:

$$\sigma_j(\mathsf{B}) = \sigma_j(\mathsf{A})^{2q+1}, \quad j = 1, 2, 3, \dots. \tag{2.22}$$

We modify Algorithm 4.1 by replacing the formula $\mathsf{Y} = \mathsf{A}\Omega$ in Step 2 by the formula $\mathsf{Y} = \mathsf{B}\Omega = \big(\mathsf{A}\mathsf{A}^*\big)^q\mathsf{A}\Omega$, and we obtain Algorithm 4.3.

Algorithm 4.3 requires $2q + 1$ times as many matrix–vector multiplies as Algorithm 4.1, but is far more accurate in situations where the singular values of A decay slowly. A good heuristic is that when the original scheme produces a basis whose approximation error is within a factor C of the optimum, the power scheme produces an approximation error within $C^{1/(2q+1)}$ of the optimum. In other words, the power iteration drives the approximation gap to one exponentially fast. See Theorem 2.9.2 and §2.10.4 for the details.

Algorithm 4.3 targets the fixed-rank problem. To address the fixed-precision problem, we can incorporate the error estimators described in §2.4.3 to obtain an adaptive scheme analogous with Algorithm 4.2. In situations where it is critical to achieve near-optimal approximation errors,

one can increase the oversampling beyond our standard recommendation $\ell = k + 5$ all the way to $\ell = 2k$ without changing the scaling of the asymptotic computational cost. A supporting analysis appears in Corollary 2.10.10.

Remark 8. Unfortunately, when Algorithm 4.3 is executed in floating point arithmetic, rounding errors will extinguish all information pertaining to singular modes associated with singular values that are small compared with $\|\mathsf{A}\|$. (Roughly, if machine precision is μ, then all information associated with singular values smaller than $\mu^{1/(2q+1)}\,\|\mathsf{A}\|$ is lost.) This problem can easily be remedied by orthonormalizing the columns of the sample matrix between each application of A and A^*. The resulting scheme, summarized as Algorithm 4.4, is algebraically equivalent to Algorithm 4.3 when executed in exact arithmetic [124, 92]. We recommend Algorithm 4.4 because its computational costs are similar to Algorithm 4.3, even though the former is substantially more accurate in floating-point arithmetic.

2.4.6 An accelerated technique for general dense matrices

This section describes a set of techniques that allow us to compute an approximate rank-ℓ factorization of a general dense $m \times n$ matrix in roughly $\mathrm{O}(mn \log(\ell))$ flops, in contrast to the asymptotic cost $\mathrm{O}(mn\ell)$ required by earlier methods. We can tailor this scheme for the real or complex case, but we focus on the conceptually simpler complex case. These algorithms were introduced in [137]; similar techniques were proposed in [118].

The first step toward this accelerated technique is to observe that the bottleneck in Algorithm 4.1 is the computation of the matrix product $\mathsf{A}\Omega$. When the test matrix Ω is standard Gaussian, the cost of this multiplication is $\mathrm{O}(mn\ell)$, the same as a rank-revealing QR algorithm [68]. The key idea is to use a **structured** random matrix that allows us to compute the product in $\mathrm{O}(mn \log(\ell))$ flops.

The *subsampled random Fourier transform*, or SRFT, is perhaps the simplest example of a structured random matrix that meets our goals. An SRFT is an $n \times \ell$ matrix of the form

$$\Omega = \sqrt{\frac{n}{\ell}}\,\mathsf{DFR}, \tag{2.23}$$

where

- D is an $n \times n$ diagonal matrix whose entries are independent random variables uniformly distributed on the complex unit circle,
- F is the $n \times n$ unitary discrete Fourier transform (DFT), whose entries take the values $f_{pq} = n^{-1/2}\,\mathrm{e}^{-2\pi\mathrm{i}(p-1)(q-1)/n}$ for $p, q = 1, 2, \ldots, n$, and
- R is an $n \times \ell$ matrix that samples ℓ coordinates from n uniformly at random, i.e., its ℓ columns are drawn randomly without replacement from the columns of the $n \times n$ identity matrix.

When Ω is defined by (2.23), we can compute the sample matrix $\mathsf{Y} = \mathsf{A}\Omega$ using $\mathrm{O}(mn\log(\ell))$ flops via a subsampled FFT [137]. Then we form the basis Q by orthonormalizing the columns of Y, as described in §2.4.1. This scheme appears as Algorithm 4.5. The total number T_{struct} of flops required by this procedure is

$$T_{\text{struct}} \sim mn\log(\ell) + \ell^2 n \tag{2.24}$$

Note that if ℓ is substantially larger than the numerical rank k of the input matrix, we can perform the orthogonalization with $\mathrm{O}(k\ell n)$ flops because the columns of the sample matrix are almost linearly dependent.

The test matrix (2.23) is just one choice among many possibilities. Other suggestions that appear in the literature include subsampled Hadamard transforms, chains of Givens rotations acting on randomly chosen coordinates, and many more. See [86] and its bibliography. Empirically, we have found that the transform summarized in Remark 11 below performs very well in a variety of environments [113].

At this point, it is not well understood how to quantify and compare the behavior of structured random transforms. One reason for this uncertainty is that it has been difficult to analyze the amount of oversampling that various transforms require. Section 2.11 establishes that the random matrix (2.23) can be used to identify a near-optimal basis for a rank-k matrix using $\ell \sim (k + \log(n))\log(k)$ samples. In practice, the transforms (2.23) and (2.25) typically require no more oversampling than a Gaussian test matrix requires. (For a numerical example, see §2.7.4.) As a consequence, setting $\ell = k + 10$ or $\ell = k + 20$ is typically more than adequate. Further research on these questions would be valuable.

Remark 9. The structured random matrices discussed in this section do not adapt readily to the fixed-precision problem, where the computational tolerance is specified, because the samples from the range are usually computed in bulk. Fortunately, these schemes are sufficiently inexpensive that we can progressively increase the number of samples computed starting with $\ell = 32$, say, and then proceeding to $\ell = 64, 128, 256, \ldots$ until we achieve the desired tolerance.

Remark 10. When using the SRFT (2.23) for matrix approximation, we have a choice whether to use a subsampled FFT or a full FFT. The complete FFT is so inexpensive that it often pays to construct an extended sample matrix $\mathsf{Y}_{\text{large}} = \mathsf{ADF}$ and then generate the actual samples by drawing columns at random from $\mathsf{Y}_{\text{large}}$ and rescaling as needed. The asymptotic cost increases to $\mathrm{O}(mn\log(n))$ flops, but the full FFT is actually faster for moderate problem sizes because the constant suppressed by the big-O notation is so small. Adaptive rank determination is easy because we just examine extra samples as needed.

Remark 11. Among the structured random matrices that we have tried, one of the strongest candidates involves sequences of random Givens rotations [113]. This matrix takes the form

$$\Omega = \mathsf{D}''\,\Theta'\,\mathsf{D}'\,\Theta\,\mathsf{D}\,\mathsf{F}\,\mathsf{R}, \tag{2.25}$$

where the prime symbol $'$ indicates an independent realization of a random matrix. The matrices R, F, and D are defined after (2.23). The matrix Θ is a chain of random Givens rotations:

$$\Theta = \Pi\,\mathsf{G}(1, 2; \theta_1)\,\mathsf{G}(2, 3; \theta_2)\,\cdots\,\mathsf{G}(n-1, n; \theta_{n-1})$$

where $\mathsf{\Pi}$ is a random $n \times n$ permutation matrix; where $\theta_1, \dots, \theta_{n-1}$ are independent random variables uniformly distributed on the interval $[0, 2\pi]$; and where $\mathsf{G}(i, j; \theta)$ denotes a rotation on $\mathbb{C}^n$ by the angle θ in the (i, j) coordinate plane [61, §5.1.8].

Remark 12. When the singular values of the input matrix A decay slowly, Algorithm 4.5 may perform poorly in terms of accuracy. When randomized sampling is used with a Gaussian random matrix, the recourse is to take a couple of steps of a power iteration; see Algorithm 4.4. However, it is not currently known whether such an iterative scheme can be accelerated to $\mathrm{O}(mn \log(k))$ complexity using "fast" random transforms such as the SRFT.

2.5 Stage B: Construction of standard factorizations

The algorithms for Stage A described in §2.4 produce an orthonormal matrix Q whose range captures the action of an input matrix A:

$$\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| \leq \varepsilon, \tag{2.26}$$

where ε is a computational tolerance. This section describes methods for approximating standard factorizations of A using the information in the basis Q.

To accomplish this task, we pursue the idea from §2.3.3.3 that any low-rank factorization $\mathsf{A} \approx \mathsf{C}\mathsf{B}$ can be manipulated to produce a standard decomposition. When the bound (2.26) holds, the low-rank factors are simply $\mathsf{C} = \mathsf{Q}$ and $\mathsf{B} = \mathsf{Q}^*\mathsf{A}$. The simplest scheme (§2.5.1) computes the factor B directly with a matrix–matrix product to ensure a minimal error in the final approximation. An alternative approach (§2.5.2) constructs factors B and C without forming any matrix–matrix product. The approach of §2.5.2 is often faster than the approach of §2.5.1 but typically results in larger errors. Both schemes can be streamlined for an Hermitian input matrix (§2.5.3) and a positive semidefinite input matrix (§2.5.4). Finally, we develop single-pass algorithms that exploit other information generated in Stage A to avoid revisiting the input matrix (§2.5.5).

Throughout this section, A denotes an $m \times n$ matrix, and Q is an $m \times k$ orthonormal matrix that verifies (2.26). For purposes of exposition, we concentrate on methods for constructing the partial SVD.

2.5.1 Factorizations based on forming $\mathsf{Q}^*\mathsf{A}$ directly

The relation (2.26) implies that $\|\mathsf{A} - \mathsf{Q}\mathsf{B}\| \leq \varepsilon$, where $\mathsf{B} = \mathsf{Q}^*\mathsf{A}$. Once we have computed B, we can produce any standard factorization using the methods of §2.3.3.3. Algorithm 5.1 illustrates how to build an approximate SVD.

The factors produced by Algorithm 5.1 satisfy

$$\|\mathsf{A} - \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*\| \leq \varepsilon. \tag{2.27}$$

In other words, the approximation error does not degrade.

The cost of Algorithm 5.1 is generally dominated by the cost of the product $\mathsf{Q}^*\mathsf{A}$ in Step 1, which takes $\mathrm{O}(kmn)$ flops for a general dense matrix. Note that this scheme is particularly well suited to environments where we have a fast method for computing the matrix–vector product $\mathbf{x} \mapsto \mathsf{A}^*\mathbf{x}$, for example when A is sparse or structured. This approach retains a strong advantage over Krylov-subspace methods and rank-revealing QR because Step 1 can be accelerated using BLAS3, parallel processors, and so forth. Steps 2 and 3 require $\mathrm{O}(k^2n)$ and $\mathrm{O}(k^2m)$ flops respectively.

Remark 13. Algorithm 5.1 produces an approximate SVD with the same rank as the basis matrix Q. When the size of the basis exceeds the desired rank k of the SVD, it may be preferable to retain only the dominant k singular values and singular vectors. Equivalently, we replace the diagonal matrix $\mathsf{\Sigma}$ of computed singular values with the matrix $\mathsf{\Sigma}_{(k)}$ formed by zeroing out all but the largest k entries of $\mathsf{\Sigma}$. In the worst case, this truncation step can increase the approximation error by σ_{k+1}; see §2.9.4 for an analysis. Our numerical experience suggests that this error analysis is pessimistic, and the term σ_{k+1} often does not appear in practice.

2.5.2 Postprocessing via row extraction

Given a matrix Q such that (2.26) holds, we can obtain a rank-k factorization

$$\mathsf{A} \approx \mathsf{X}\mathsf{B}, \tag{2.28}$$

where B is a $k \times n$ matrix consisting of k rows extracted from A. The approximation (2.28) can be produced without computing any matrix–matrix products, which makes this approach to postprocessing very fast. The drawback comes because the error $\|\mathsf{A} - \mathsf{X}\mathsf{B}\|$ is usually larger than the initial error $\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\|$, especially when the dimensions of A are large. See Remark 15 for more discussion.

To obtain the factorization (2.28), we simply construct the interpolative decomposition (§2.3.2.3) of the matrix Q:

$$\mathsf{Q} = \mathsf{X}\mathsf{Q}_{(J,:)}. \tag{2.29}$$

The index set J marks k rows of Q that span the row space of Q, and X is an $m \times k$ matrix whose entries are bounded in magnitude by two and contains the $k \times k$ identity as a submatrix: $\mathsf{X}_{(J,:)} = \mathsf{I}_k$. Combining (2.29) and (2.26), we reach

$$\mathsf{A} \approx \mathsf{Q}\mathsf{Q}^*\mathsf{A} = \mathsf{X}\mathsf{Q}_{(J,:)}\mathsf{Q}^*\mathsf{A}. \tag{2.30}$$

Since $\mathsf{X}_{(J,:)} = \mathsf{I}_k$, equation (2.30) implies that $\mathsf{A}_{(J,:)} \approx \mathsf{Q}_{(J,:)}\mathsf{Q}^*\mathsf{A}$. Therefore, (2.28) follows when we put $\mathsf{B} = \mathsf{A}_{(J,:)}$.

Provided with the factorization (2.28), we can obtain any standard factorization using the techniques of §2.3.3.3. Algorithm 5.2 illustrates an SVD calculation. This procedure requires $\mathrm{O}(k^2(m+n))$ flops. The following lemma guarantees the accuracy of the computed factors.

Lemma 2.5.1. *Let* A *be an* $m \times n$ *matrix and let* Q *be an* $m \times k$ *matrix that satisfy* (2.26)*. Suppose that* U*,* $\mathsf{\Sigma}$*, and* V *are the matrices constructed by Algorithm 5.2. Then*

$$\|\mathsf{A} - \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*\| \leq \left[1 + \sqrt{1 + 4k(n-k)}\right] \varepsilon. \tag{2.31}$$

Proof. The factors U, $\mathsf{\Sigma}$, V constructed by the algorithm satisfy

$$\mathsf{U}\mathsf{\Sigma}\mathsf{V}^* = \mathsf{U}\mathsf{\Sigma}\widetilde{\mathsf{V}}^*\mathsf{W}^* = \mathsf{Z}\mathsf{W}^* = \mathsf{X}\mathsf{R}^*\mathsf{W}^* = \mathsf{X}\mathsf{A}_{(J,:)}.$$

Define the approximation

$$\widehat{\mathsf{A}} = \mathsf{Q}\mathsf{Q}^*\mathsf{A}. \tag{2.32}$$

Since $\widehat{\mathsf{A}} = \mathsf{X}\mathsf{Q}_{(J,:)}\mathsf{Q}^*\mathsf{A}$ and since $\mathsf{X}_{(J,:)} = \mathsf{I}_k$, it must be that $\widehat{\mathsf{A}}_{(J,:)} = \mathsf{Q}_{(J,:)}\mathsf{Q}^*\mathsf{A}$. Consequently,

$$\widehat{\mathsf{A}} = \mathsf{X}\widehat{\mathsf{A}}_{(J,:)}.$$

We have the chain of relations

$$\begin{aligned}
\|\mathsf{A} - \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*\| &= \left\|\mathsf{A} - \mathsf{X}\mathsf{A}_{(J,:)}\right\| \\
&= \left\|\left(\mathsf{A} - \mathsf{X}\widehat{\mathsf{A}}_{(J,:)}\right) + \left(\mathsf{X}\widehat{\mathsf{A}}_{(J,:)} - \mathsf{X}\mathsf{A}_{(J,:)}\right)\right\| \\
&\leq \left\|\mathsf{A} - \widehat{\mathsf{A}}\right\| + \left\|\mathsf{X}\widehat{\mathsf{A}}_{(J,:)} - \mathsf{X}\mathsf{A}_{(J,:)}\right\| \\
&\leq \left\|\mathsf{A} - \widehat{\mathsf{A}}\right\| + \|\mathsf{X}\| \left\|\mathsf{A}_{(J,:)} - \widehat{\mathsf{A}}_{(J,:)}\right\|.
\end{aligned} \tag{2.33}$$

Inequality (2.26) ensures that $\left\|\mathsf{A} - \widehat{\mathsf{A}}\right\| \leq \varepsilon$. Since $\mathsf{A}_{(J,:)} - \widehat{\mathsf{A}}_{(J,:)}$ is a submatrix of $\mathsf{A} - \widehat{\mathsf{A}}$, we must also have $\left\|\mathsf{A}_{(J,:)} - \widehat{\mathsf{A}}_{(J,:)}\right\| \leq \varepsilon$. Thus, (2.33) reduces to

$$\|\mathsf{A} - \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*\| \leq (1 + \|\mathsf{X}\|)\, \varepsilon. \tag{2.34}$$

The bound (2.31) follows from (2.34) after we observe that X contains a $k \times k$ identity matrix and that the entries of the remaining $(n-k) \times k$ submatrix are bounded in magnitude by two. □

Remark 14. To maintain a unified presentation, we have formulated all the postprocessing techniques so they take an orthonormal matrix Q as input. Recall that, in Stage A of our framework, we construct the matrix Q by orthonormalizing the columns of the sample matrix Y. With finite-precision arithmetic, it is preferable to adapt Algorithm 5.2 to start directly from the sample matrix Y. To be precise, we modify Step 1 to compute X and J so that $\mathsf{Y} = \mathsf{X}\mathsf{Y}_{(J,:)}$. This revision is recommended even when Q is available from the adaptive rank determination of Algorithm 4.2.

Remark 15. As the inequality (2.31) suggests, the factorization produced by Algorithm 5.2 is potentially less accurate than the basis that it uses as input. This loss of accuracy is problematic when ε is not so small or when kn is large. In such cases, we recommend Algorithm 5.1 over Algorithm 5.2; the former is more costly, but it does not amplify the error, as shown in (2.27).

2.5.3 Postprocessing an Hermitian matrix

When A is Hermitian, the postprocessing becomes particularly elegant. In this case, the columns of Q form a good basis for both the column space **and** the row space of A so that we have $\mathsf{A} \approx \mathsf{Q}\mathsf{Q}^*\mathsf{A}\mathsf{Q}\mathsf{Q}^*$. More precisely, when (2.26) is in force, we have

$$\begin{aligned}\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\mathsf{Q}\mathsf{Q}^*\| &= \|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A} + \mathsf{Q}\mathsf{Q}^*\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\mathsf{Q}\mathsf{Q}^*\| \\ &\le \|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\| + \big\|\mathsf{Q}\mathsf{Q}^*\big(\mathsf{A} - \mathsf{A}\mathsf{Q}\mathsf{Q}^*\big)\big\| \le 2\varepsilon. \quad (2.35)\end{aligned}$$

The last inequality relies on the facts that $\|\mathsf{Q}\mathsf{Q}^*\| = 1$ and that

$$\|\mathsf{A} - \mathsf{A}\mathsf{Q}\mathsf{Q}^*\| = \big\|(\mathsf{A} - \mathsf{A}\mathsf{Q}\mathsf{Q}^*)^*\big\| = \|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\|\,.$$

Since $\mathsf{A} \approx \mathsf{Q}\big(\mathsf{Q}^*\mathsf{A}\mathsf{Q}\big)\mathsf{Q}^*$ is a low-rank approximation of A, we can form any standard factorization using the techniques from §2.3.3.3.

For Hermitian A, it is more common to compute an eigenvalue decomposition than an SVD. We can accomplish this goal using Algorithm 5.3, which adapts the scheme from §2.5.1. This procedure delivers a factorization that satisfies the error bound $\|\mathsf{A} - \mathsf{U}\mathsf{\Lambda}\mathsf{U}^*\| \le 2\varepsilon$. The calculation requires $\mathrm{O}(kn^2)$ flops.

We can also pursue the row extraction approach from §2.5.2, which is faster but less accurate. See Algorithm 5.4 for the details. The total cost is $\mathrm{O}(k^2 n)$ flops.

2.5.4 Postprocessing a positive semidefinite matrix

When the input matrix A is positive semidefinite, the *Nyström method* can be used to improve the quality of standard factorizations at almost no additional cost; see [48] and its bibliography. To describe the main idea, we first recall that the direct method presented in §2.5.3 manipulates the approximate rank-k factorization

$$\mathsf{A} \approx \mathsf{Q}\left(\mathsf{Q}^*\mathsf{A}\mathsf{Q}\right)\mathsf{Q}^*. \quad (2.36)$$

In contrast, the Nyström scheme builds a more sophisticated rank-k approximation, namely

$$\begin{aligned}\mathsf{A} &\approx (\mathsf{A}\mathsf{Q})\left(\mathsf{Q}^*\mathsf{A}\mathsf{Q}\right)^{-1}(\mathsf{A}\mathsf{Q})^* \\ &= \left[(\mathsf{A}\mathsf{Q})\left(\mathsf{Q}^*\mathsf{A}\mathsf{Q}\right)^{-1/2}\right]\left[(\mathsf{A}\mathsf{Q})\left(\mathsf{Q}^*\mathsf{A}\mathsf{Q}\right)^{-1/2}\right]^* = \mathsf{F}\mathsf{F}^*, \quad (2.37)\end{aligned}$$

where F is an approximate Cholesky factor of A with dimension $n \times k$. To compute the factor F numerically, first form the matrices $\mathsf{B}_1 = \mathsf{A}\mathsf{Q}$ and $\mathsf{B}_2 = \mathsf{Q}^*\mathsf{B}_1$. Then decompose the psd matrix $\mathsf{B}_2 = \mathsf{C}^*\mathsf{C}$ into its Cholesky factors. Finally compute the factor $\mathsf{F} = \mathsf{B}_1\mathsf{C}^{-1}$ by performing a triangular solve. The low-rank factorization (2.37) can be converted to a standard decomposition using the techniques from §2.3.3.3.

The literature contains an explicit expression [48, Lem. 4] for the approximation error in (2.37). This result implies that, in the spectral norm, the Nyström approximation error never exceeds $\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}\|$, and it is often substantially smaller. We omit a detailed discussion.

For an example of the Nyström technique, consider Algorithm 5.5, which computes an approximate eigenvalue decomposition of a positive semidefinite matrix. This method should be compared with the scheme for Hermitian matrices, Algorithm 5.3. In both cases, the dominant cost occurs when we form AQ, so the two procedures have roughly the same running time. On the other hand, Algorithm 5.5 is typically much more accurate than Algorithm 5.3. In a sense, we are exploiting the fact that A is positive semidefinite to take one step of subspace iteration (Algorithm 4.4) for free.

2.5.5 Single-pass algorithms

The techniques described in §§2.5.1–2.5.4 all require us to revisit the input matrix. This may not be feasible in environments where the matrix is too large to be stored. In this section, we develop a method that requires just one pass over the matrix to construct not only an approximate basis but also a complete factorization. Similar techniques appear in [137] and [29].

For motivation, we begin with the case where A is Hermitian. Let us recall the proto-algorithm from §2.1.3.3: Draw a random test matrix Ω; form the sample matrix $\mathsf{Y} = \mathsf{A}\Omega$; then construct a basis Q for the range of Y. It turns out that the matrices Ω, Y, and Q contain all the information we need to approximate A.

To see why, define the (currently unknown) matrix B via $\mathsf{B} = \mathsf{Q}^*\mathsf{AQ}$. Postmultiplying the definition by $\mathsf{Q}^*\Omega$, we obtain the identity $\mathsf{BQ}^*\Omega = \mathsf{Q}^*\mathsf{AQQ}^*\Omega$. The relationships $\mathsf{AQQ}^* \approx \mathsf{A}$ and $\mathsf{A}\Omega = \mathsf{Y}$ show that B must satisfy

$$\mathsf{BQ}^*\Omega \approx \mathsf{Q}^*\mathsf{Y}. \tag{2.38}$$

All three matrices Ω, Y, and Q are available, so we can solve (2.38) to obtain the matrix B. Then the low-rank factorization $\mathsf{A} \approx \mathsf{QBQ}^*$ can be converted to an eigenvalue decomposition via familiar techniques. The entire procedure requires $\mathrm{O}(k^2 n)$ flops, and it is summarized as Algorithm 5.6.

When A is not Hermitian, it is still possible to devise single-pass algorithms, but we must modify the initial Stage A of the approximation framework to simultaneously construct bases for the ranges of A and A^*:

(1) Generate random matrices Ω and $\widetilde{\Omega}$.

(2) Compute $\mathsf{Y} = \mathsf{A}\Omega$ and $\widetilde{\mathsf{Y}} = \mathsf{A}^*\widetilde{\Omega}$ in a single pass over A.

(3) Compute QR factorizations $\mathsf{Y} = \mathsf{QR}$ and $\widetilde{\mathsf{Y}} = \widetilde{\mathsf{Q}}\widetilde{\mathsf{R}}$.

This procedure results in matrices Q and $\widetilde{\mathsf{Q}}$ such that $\mathsf{A} \approx \mathsf{QQ}^*\mathsf{A}\widetilde{\mathsf{Q}}\widetilde{\mathsf{Q}}^*$. The reduced matrix we must approximate is $\mathsf{B} = \mathsf{Q}^*\mathsf{A}\widetilde{\mathsf{Q}}$. In analogy with (2.38), we find that

$$\mathsf{Q}^*\mathsf{Y} = \mathsf{Q}^*\mathsf{A}\Omega \approx \mathsf{Q}^*\mathsf{A}\widetilde{\mathsf{Q}}\widetilde{\mathsf{Q}}^*\Omega = \mathsf{B}\widetilde{\mathsf{Q}}^*\Omega. \tag{2.39}$$

An analogous calculation shows that B should also satisfy

$$\widetilde{\mathsf{Q}}^*\widetilde{\mathsf{Y}} \approx \mathsf{B}^*\mathsf{Q}^*\widetilde{\Omega}. \tag{2.40}$$

Now, the reduced matrix $\mathsf{B}_{\text{approx}}$ can be determined by finding a minimum-residual solution to the system of relations (2.39) and (2.40).

Remark 16. The single-pass approaches described in this section can degrade the approximation error in the final decomposition significantly. To explain the issue, we focus on the Hermitian case. It turns out that the coefficient matrix $\mathsf{Q}^*\Omega$ in the linear system (2.38) is usually ill-conditioned. In a worst-case scenario, the error $\|\mathsf{A} - \mathsf{U}\Lambda\mathsf{U}^*\|$ in the factorization produced by Algorithm 5.6 could be larger than the error resulting from the two-pass method of Section 2.5.3 by a factor of $1/\tau_{\min}$, where $\tau_{\min}$ is the minimal singular value of the matrix $\mathsf{Q}^*\Omega$.

The situation can be improved by oversampling. Suppose that we seek a rank-k approximate eigenvalue decomposition. Pick a small oversampling parameter p. Draw an $n \times (k+p)$ random matrix Ω, and form the sample matrix $\mathsf{Y} = \mathsf{A}\Omega$. Let Q denote the $n \times k$ matrix formed by the k leading left singular vectors of Y. Now, the linear system (2.38) has a coefficient matrix $\mathsf{Q}^*\Omega$ of size $k \times (k+p)$, so it is overdetermined. An approximate solution of this system yields a $k \times k$ matrix B.

2.6 Computational costs

So far, we have postponed a detailed discussion of the computational cost of randomized matrix approximation algorithms because it is necessary to account for both the first stage, where we compute an approximate basis for the range (§2.4), and the second stage, where we postprocess the basis to complete the factorization (§2.5). We are now prepared to compare the cost of the two-stage scheme with the cost of traditional techniques.

Choosing an appropriate algorithm, whether classical or randomized, requires us to consider the properties of the input matrix. To draw a nuanced picture, we discuss three representative computational environments in §2.6.1–2.6.3. We close with some comments on parallel implementations in §2.6.4.

For concreteness, we focus on the problem of computing an approximate SVD of an $m \times n$ matrix A with numerical rank k. The costs for other factorizations are similar.

2.6.1 General matrices that fit in core memory

Suppose that A is a general matrix presented as an array of numbers that fits in core memory. In this case, the appropriate method for Stage A is to use a structured random matrix (§2.4.6), which allows us to find a basis that captures the action of the matrix using $\mathrm{O}(mn\log(k) + k^2m)$

flops. For Stage B, we apply the row-extraction technique (§2.5.2), which costs an additional $O(k^2(m+n))$ flops. The total number of operations T_{random} for this approach satisfies

$$T_{\text{random}} \sim mn\log(k) + k^2(m+n).$$

As a rule of thumb, the approximation error of this procedure satisfies

$$\|\mathsf{A} - \mathsf{U}\Sigma\mathsf{V}^*\| \lesssim n \cdot \sigma_{k+1}, \tag{2.41}$$

where σ_{k+1} is the $(k+1)$th singular value of A. The estimate (2.41), which follows from Theorem 2.11.2 and Lemma 2.5.1, reflects the worst-case scenario; actual errors are usually smaller.

This algorithm should be compared with modern deterministic techniques, such as rank-revealing QR followed by postprocessing (§2.3.3.2) which typically require

$$T_{\text{RRQR}} \sim kmn$$

operations to achieve a comparable error.

In this setting, the randomized algorithm can be several times faster than classical techniques even for problems of moderate size, say $m, n \sim 10^3$ and $k \sim 10^2$. See §2.7.4 for numerical evidence.

Remark 17. In case row extraction is impractical, there is an alternative $O(mn\log(k))$ technique described in [137, §5.2]. When the error (2.41) is unacceptably large, we can use the direct method (§2.5.1) for Stage B, which brings the total cost to $O(kmn)$ flops.

2.6.2 Matrices for which matrix–vector products can be rapidly evaluated

In many problems in data mining and scientific computing, the cost T_{mult} of performing the matrix–vector multiplication $\mathbf{x} \mapsto \mathsf{A}\mathbf{x}$ is substantially smaller than the nominal cost $O(mn)$ for the dense case. It is not uncommon that $O(m+n)$ flops suffice. Standard examples include (i) very sparse matrices; (ii) structured matrices, such as Töplitz operators, that can be applied using the FFT or other means; and (iii) matrices that arise from physical problems, such as discretized integral operators, that can be applied via, e.g., the fast multipole method [66].

Suppose that both A and A^* admit fast multiplies. The appropriate randomized approach for this scenario completes Stage A using Algorithm 4.1 with p constant (for the fixed-rank problem) or Algorithm 4.2 (for the fixed-precision problem) at a cost of $(k+p)\,T_{\text{mult}} + O(k^2m)$ flops. For Stage B, we invoke Algorithm 5.1, which requires $(k+p)\,T_{\text{mult}} + O(k^2(m+n))$ flops. The total cost T_{sparse} satisfies

$$T_{\text{sparse}} = 2\,(k+p)\,T_{\text{mult}} + O(k^2(m+n)). \tag{2.42}$$

As a rule of thumb, the approximation error of this procedure satisfies

$$\|\mathsf{A} - \mathsf{U}\Sigma\mathsf{V}^*\| \lesssim \sqrt{kn} \cdot \sigma_{k+1}. \tag{2.43}$$

The estimate (2.43) follows from Corollary 2.10.9 and the discussion in §2.5.1. Actual errors are usually smaller.

When the singular spectrum of A decays slowly, we can incorporate q iterations of the power method (Algorithm 4.3) to obtain superior solutions to the fixed-rank problem. The computational cost increases to, cf. (2.42),

$$T_{\text{sparse}} = (2q + 2)\,(k + p)\,T_{\text{mult}} + \mathrm{O}(k^2(m + n)), \tag{2.44}$$

while the error (2.43) improves to

$$\|\mathsf{A} - \mathsf{U}\Sigma\mathsf{V}^*\| \lesssim (kn)^{1/2(2q+1)} \cdot \sigma_{k+1}. \tag{2.45}$$

The estimate (2.45) takes into account the discussion in §2.10.4. The power scheme can also be adapted for the fixed-precision problem (§2.4.5).

In this setting, the classical prescription for obtaining a partial SVD is some variation of a Krylov-subspace method; see §2.3.3.4. These methods exhibit great diversity, so it is hard to specify a "typical" computational cost. To a first approximation, it is fair to say that in order to obtain an approximate SVD of rank k, the cost of a numerically stable implementation of a Krylov method is no less than the cost (2.42) with p set to zero. At this price, the Krylov method often obtains better accuracy than the basic randomized method obtained by combining Algorithms 4.1 and 5.1, especially for matrices whose singular values decay slowly. On the other hand, the randomized schemes are inherently more robust and allow much more freedom in organizing the computation to suit a particular application or a particular hardware architecture. The latter point is in practice of crucial importance because it is usually much faster to apply a matrix to k vectors simultaneously than it is to execute k matrix–vector multiplications consecutively. In practice, blocking and parallelism can lead to enough gain that a few steps of the power method (Algorithm 4.3) can be performed more quickly than k steps of a Krylov method.

Remark 18. Any comparison between randomized sampling schemes and Krylov variants becomes complicated because of the fact that "basic" Krylov schemes such as Lanczos [61, p. 473] or Arnoldi [61, p. 499] are inherently unstable. To obtain numerical robustness, we must incorporate sophisticated modifications such as restarts, reorthogonalization procedures, etc. Constructing a high-quality implementation is sufficiently hard that the authors of a popular book on "numerical recipes" qualify their treatment of spectral computations as follows [109, p. 567]:

> You have probably gathered by now that the solution of eigensystems is a fairly complicated business. It is. It is one of the few subjects covered in this book for which we do **not** recommend that you avoid canned routines. On the contrary, the purpose of this chapter is precisely to give you some appreciation of what is going on inside such canned routines, so that you can make intelligent choices about using them, and intelligent diagnoses when something goes wrong.

Randomized sampling does not eliminate the difficulties referred to in this quotation; however it reduces the task of computing a **partial** spectral decomposition of a very large matrix to the task of computing a **full** decomposition of a small dense matrix. (For example, in Algorithm 5.1, the input matrix A is large and B is small.) The latter task is much better understood and is

eminently suitable for using canned routines. Random sampling schemes interact with the large matrix only through matrix–matrix products, which can easily be implemented by a user in a manner appropriate to the application and to the available hardware.

The comparison is further complicated by the fact that there is significant overlap between the two sets of ideas. Algorithm 4.3 is conceptually similar to a "block Lanczos method" [61, p. 485] with a random starting matrix. Indeed, we believe that there are significant opportunities for cross-fertilization in this area. Hybrid schemes that combine the best ideas from both fields may perform very well.

2.6.3 General matrices stored in slow memory or streamed

The traditional metric for numerical algorithms is the number of floating-point operations they require. When the data does not fit in fast memory, however, the computational time is often dominated by the cost of memory access. In this setting, a more appropriate measure of algorithmic performance is *pass-efficiency*, which counts how many times the data needs to be cycled through fast memory. Flop counts become largely irrelevant.

All the classical matrix factorization techniques that we discuss in §2.3.2—including dense SVD, rank-revealing QR, Krylov methods, and so forth—require at least k passes over the the matrix, which is prohibitively expensive for huge data matrices. A desire to reduce the pass count of matrix approximation algorithms served as one of the early motivations for developing randomized schemes [105, 58, 46]. Detailed recent work appears in [29].

For many matrices, randomized techniques can produce an accurate approximation using just one pass over the data. For Hermitian matrices, we obtain a single-pass algorithm by combining Algorithm 4.1, which constructs an approximate basis, with Algorithm 5.6, which produces an eigenvalue decomposition without any additional access to the matrix. Section 2.5.5 describes the analogous technique for general matrices.

For the huge matrices that arise in applications such as data mining, it is common that the singular spectrum decays slowly. Relevant applications include image processing (see §§2.7.2–2.7.3 for numerical examples), statistical data analysis, and network monitoring. To compute approximate factorizations in these environments, it is crucial to enhance the accuracy of the randomized approach using the power scheme, Algorithm 4.3, or some other device. This approach increases the pass count somewhat, but in our experience it is very rare that more than five passes are required.

2.6.4 Gains from parallelization

As mentioned in §§2.6.2–2.6.3, randomized methods often outperform classical techniques not because they involve fewer floating-point operations but rather because they allow us to reorganize the calculations to exploit the matrix properties and the computer architecture more fully. In addition, these methods are well suited for parallel implementation. For example, in Algorithm 4.1, the computational bottleneck is the evaluation of the matrix product $\mathsf{A}\Omega$, which is embarrassingly parallelizable.

2.7 Numerical examples

By this time, the reader has surely formulated a pointed question: Do these randomized matrix approximation algorithms actually work in practice? In this section, we attempt to address this concern by illustrating how the algorithms perform on a diverse collection of test cases.

Section 2.7.1 starts with two examples from the physical sciences involving discrete approximations to operators with exponentially decaying spectra. Sections 2.7.2 and 2.7.3 continue with two examples of matrices arising in "data mining." These are large matrices whose singular spectra decay slowly; one is sparse and fits in RAM, one is dense and is stored out-of-core. Finally, §2.7.4 investigates the performance of randomized methods based on structured random matrices.

Sections 2.7.1–2.7.3 focus on the algorithms for Stage A that we presented in §2.4 because we wish to isolate the performance of the randomized step.

Computational examples illustrating truly large data matrices have been reported elsewhere, for instance in [69].

2.7.1 Two matrices with rapidly decaying singular values

We first illustrate the behavior of the adaptive range approximation method, Algorithm 4.2. We apply it to two matrices associated with the numerical analysis of differential and integral operators. The matrices in question have rapidly decaying singular values and our intent is to demonstrate that in this environment, the approximation error of a bare-bones randomized method such as Algorithm 4.2 is **very** close to the minimal error achievable by any method. We observe that the approximation error of a randomized method is itself a random variable (it is a function of the random matrix Ω) so what we need to demonstrate is not only that the error is small in a typical realization, but also that it clusters tightly around the mean value.

We first consider a 200×200 matrix A that results from discretizing the following single-layer operator associated with the Laplace equation:

$$[S\sigma](x) = \text{const} \cdot \int_{\Gamma_1} \log|x - y| \; \sigma(y) \, \mathrm{d}A(y), \qquad x \in \Gamma_2, \tag{2.46}$$

where Γ_1 and Γ_2 are the two contours in $\mathbb{R}^2$ illustrated in Figure 2.1(a). We approximate the integral with the trapezoidal rule, which converges superalgebraically because the kernel is smooth. In the absence of floating-point errors, we estimate that the discretization error would be less than 10^{-20} for a smooth source σ. The leading constant is selected so the matrix A has unit operator norm.

We implement Algorithm 4.2 in Matlab v6.5. Gaussian test matrices are generated using the `randn` command. For each number ℓ of samples, we compare the following three quantities:

(1) The minimum rank-ℓ approximation error $\sigma_{\ell+1}$ is determined using `svd`.

(2) The actual error $e_\ell = \left\| \left(\mathsf{I} - \mathsf{Q}^{(\ell)}(\mathsf{Q}^{(\ell)})^*\right)\mathsf{A} \right\|$ is computed with `norm`.

(3) A random estimator f_ℓ for the actual error e_ℓ is obtained from (2.20), with the parameter r set to 5.

Note that any values less than 10^{-15} should be considered numerical artifacts.

Figure 2.2 tracks a characteristic execution of Algorithm 4.2. We make three observations: (i) The error e_ℓ incurred by the algorithm is remarkably close to the theoretical minimum $\sigma_{\ell+1}$. (ii) The error estimate always produces an upper bound for the actual error. Without the built-in $10\times$ safety margin, the estimate would track the actual error almost exactly. (iii) The basis constructed by the algorithm essentially reaches full double-precision accuracy.

How typical is the trial documented in Figure 2.2? To answer this question, we examine the empirical performance of the algorithm over 2000 independent trials. Figure 2.3 charts the error estimate versus the actual error at four points during the course of execution: $\ell = 25, 50, 75, 100$. We offer four observations: (i) The initial run detailed in Figure 2.2 is entirely typical. (ii) Both the actual and estimated error concentrate about their mean value. (iii) The actual error drifts slowly away from the optimal error as the number ℓ of samples increases. (iv) The error estimator is **always** pessimistic by a factor of about ten, which means that the algorithm **never** produces a basis with lower accuracy than requested. The only effect of selecting an unlucky sample matrix Ω is that the algorithm proceeds for a few additional steps.

We next consider a matrix B which is defined implicitly in the sense that we cannot access its elements directly; we can only evaluate the map $\mathbf{x} \mapsto \mathsf{B}\mathbf{x}$ for a given vector $\mathbf{x}$. To be precise, B represents a transfer matrix for a network of resistors like the one shown in Figure 2.1(b). The vector $\mathbf{x}$ represents a set of electric potentials specified on the red nodes in the figure. These potentials induce a unique equilibrium field on the network in which the potential of each black and blue node is the average of the potentials of its three or four neighbors. The vector $\mathsf{B}\mathbf{x}$ is then the restriction of the potential to the blue exterior nodes. Given a vector $\mathbf{x}$, the vector $\mathsf{B}\mathbf{x}$ can be obtained by solving a large sparse linear system whose coefficient matrix is the classical five-point stencil approximating the 2D Laplace operator.

We applied Algorithm 4.2 to the 1596×532 matrix B associated with a lattice in which there were 532 nodes (red) on the "inner ring" and 1596 nodes on the (blue) "outer ring." Each application of B to a vector requires the solution of a sparse linear system of size roughly $140\,000 \times 140\,000$. We implemented the scheme in Matlab using the "backslash" operator for the linear solve. The results of a typical trial appear in Figure 2.4. Qualitatively, the performance matches the results in Figure 2.3.

2.7.2 A large, sparse, noisy matrix arising in image processing

Our next example involves a matrix that arises in image processing. A recent line of work uses information about the local geometry of an image to develop promising new algorithms for standard tasks, such as denoising, inpainting, and so forth. These methods are based on approximating a

graph Laplacian associated with the image. The dominant eigenvectors of this matrix provide "coordinates" that help us smooth out noisy image patches [131, 120].

We begin with a 95×95 pixel grayscale image. The intensity of each pixel is represented as an integer in the range 0 to 4095. We form for each pixel i a vector $\mathbf{x}^{(i)} \in \mathbb{R}^{25}$ by gathering the 25 intensities of the pixels in a 5×5 neighborhood centered at pixel i (with appropriate modifications near the edges). Next, we form the 9025×9025 *weight matrix* $\widetilde{\mathsf{W}}$ that reflects the similarities between patches:

$$\widetilde{w}_{ij} = \exp\big\{ - \big\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\big\|^2 / \sigma^2 \big\},$$

where the parameter $\sigma = 50$ controls the level of sensitivity. We obtain a sparse weight matrix W by zeroing out all entries in $\widetilde{\mathsf{W}}$ except the seven largest ones in each row. The object is then to construct the low frequency eigenvectors of the graph Laplacian matrix

$$\mathsf{L} = \mathsf{I} - \mathsf{D}^{-1/2}\mathsf{W}\mathsf{D}^{-1/2},$$

where D is the diagonal matrix with entries $d_{ii} = \sum_j w_{ij}$. These are the eigenvectors associated with the dominant eigenvalues of the auxiliary matrix $\mathsf{A} = \mathsf{D}^{-1/2}\mathsf{W}\mathsf{D}^{-1/2}$.

The matrix A is large, and its eigenvalues decay slowly, so we use the power scheme summarized in Algorithm 4.3 to approximate it. Figure 2.5[left] illustrates how the approximation error e_ℓ declines as the number ℓ of samples increases. When we set the exponent $q = 0$, which corresponds with the basic Algorithm 4.1, the approximation is rather poor. The graph illustrates that increasing the exponent q slightly results in a tremendous improvement in the accuracy of the power scheme.

Next, we illustrate the results of using the two-stage approach to approximate the eigenvalues of A. In Stage A, we construct a basis for A using Algorithm 4.3 with $\ell = 100$ samples for different values of q. In Stage B, we apply the Hermitian variant of Algorithm 5.1 described in §2.5.3 to compute an approximate eigenvalue decomposition. Figure 2.5[right] shows the approximate eigenvalues and the actual eigenvalues of A. Once again, we see that the minimal exponent $q = 0$ produces miserable results, but the largest eigenvalues are quite accurate even for $q = 1$.

2.7.3 Eigenfaces

Our next example involves a large, dense matrix derived from the FERET databank of face images [107, 108]. A simple method for performing face recognition is to identify the principal directions of the image data, which are called *eigenfaces*. Each of the original photographs can be summarized by its components along these principal directions. To identify the subject in a new picture, we compute its decomposition in this basis and use a classification technique, such as nearest neighbors, to select the closest image in the database [122].

We construct a data matrix A as follows: The FERET database contains 7254 images, and each 384×256 image contains 98 304 pixels. First, we build a $98\,304 \times 7254$ matrix $\widetilde{\mathsf{A}}$ whose columns are the images. We form A by centering each column of $\widetilde{\mathsf{A}}$ and scaling it to unit norm, so that the images are roughly comparable. The eigenfaces are the dominant left singular vectors of this matrix.

Our goal then is to compute an approximate SVD of the matrix A. Represented as an array of double-precision real numbers, A would require 5.4 GB of storage, which does not fit within the fast memory of many machines. It is possible to compress the database down to at 57 MB or less (in JPEG format), but then the data would have to be uncompressed with each sweep over the matrix. Furthermore, the matrix A has slowly decaying singular values, so we need to use the power scheme, Algorithm 4.3, to capture the range of the matrix accurately.

To address these concerns, we implemented the power scheme to run in a pass-efficient manner. An additional difficulty arises because the size of the data makes it expensive to calculate the actual error e_ℓ incurred by the approximation or to determine the minimal error $\sigma_{\ell+1}$. To estimate the errors, we use the technique described in Remark 6.

Figure 2.6 describes the behavior of the power scheme, which is similar to its performance for the graph Laplacian in §2.7.2. When the exponent $q = 0$, the approximation of the data matrix is very poor, but it improves quickly as q increases. Likewise, the estimate for the spectrum of A appears to converge rapidly; the largest singular values are already quite accurate when $q = 1$. We see essentially no improvement in the estimates after the first 3–5 passes over the matrix.

2.7.4 Performance of structured random matrices

Our final set of experiments illustrates that the structured random matrices described in §2.4.6 lead to matrix approximation algorithms that are both fast and accurate.

First, we compare the computational speeds of four methods for computing an approximation to the ℓ dominant terms in the SVD of an $n \times n$ matrix A. For now, we are interested in execution time only (not accuracy), so the choice of matrix is irrelevant and we have selected A to be a Gaussian matrix. The four methods are summarized in the following table; Remark 19 provides more details on the implementation.

Method	**Stage A**	**Stage B**
`direct`	Rank-revealing QR executed using column pivoting and Householder reflectors	Algorithm 5.1
`gauss`	Algorithm 4.1 with a Gaussian random matrix	Algorithm 5.1
`srft`	Algorithm 4.1 with the modified SRFT (2.25)	Algorithm 5.2
`svd`	Full SVD with LAPACK routine `dgesdd`	Truncate to ℓ terms

Table 2.1 lists the measured runtime of a single execution of each algorithm for various choices of the dimension n of the input matrix and the rank ℓ of the approximation. Of course, the cost of the full SVD does not depend on the number ℓ of components required. A more informative way to look at the runtime data is to compare the **relative** cost of the algorithms. The `direct` method is the best deterministic approach for dense matrices, so we calculate the factor by which

the randomized methods improve on this benchmark. Figure 2.7 displays the results. We make two observations: (i) Using an SRFT often leads to a dramatic speed-up over classical techniques, even for moderate problem sizes. (ii) Using a standard Gaussian test matrix typically leads to a moderate speed-up over classical methods, primarily because performing a matrix–matrix multiplication is faster than a QR factorization.

Second, we investigate how the choice of random test matrix influences the error in approximating an input matrix. For these experiments, we return to the 200×200 matrix A defined in Section 2.7.1. Consider variations of Algorithm 4.1 obtained when the random test matrix Ω is drawn from the following four distributions:

Gauss: The standard Gaussian distribution.
Ortho: The uniform distribution on $n \times \ell$ orthonormal matrices.
SRFT: The SRFT distribution defined in (2.23).
GSRFT: The modified SRFT distribution defined in (2.25).

Intuitively, we expect that **Ortho** should provide the best performance.

For each distribution, we perform 100 000 trials of the following experiment. Apply the corresponding version of Algorithm 4.1 to the matrix A, and calculate the approximation error $e_\ell = \|\mathsf{A} - \mathsf{Q}_\ell \mathsf{Q}_\ell^* \mathsf{A}\|$. Figure 2.8 displays the empirical probability density function for the error e_ℓ obtained with each algorithm. We offer three observations: (i) The SRFT actually performs slightly *better* than a Gaussian random matrix for this example. (ii) The standard SRFT and the modified SRFT have essentially identical errors. (iii) There is almost no difference between the Gaussian random matrix and the random orthonormal matrix in the first three plots, while the fourth plot shows that the random orthonormal matrix performs better. This behavior occurs because, with high probability, a tall Gaussian matrix is well conditioned and a (nearly) square Gaussian matrix is not.

Remark 19. The running times reported in Table 2.1 and in Figure 2.7 depend strongly on both the computer hardware and the coding of the algorithms. The experiments reported here were performed on a standard office desktop with a 3.2 GHz Pentium IV processor and 2 GB of RAM. The algorithms were implemented in Fortran 90 and compiled with the Lahey compiler. The Lahey versions of BLAS and LAPACK were used to accelerate all matrix–matrix multiplications, as well as the SVD computations in Algorithms 5.1 and 5.2. We used the code for the modified SRFT (2.25) provided in the publicly available software package `id_dist` [90].

Table 2.1: Computational times for a partial SVD. The time, in seconds, required to compute the ℓ leading components in the SVD of an $n \times n$ matrix using each of the methods from §2.7.4. The last row indicates the time needed to obtain a full SVD.

	$n = 1024$			$n = 2048$			$n = 4096$		
ℓ	`direct`	`gauss`	`srft`	`direct`	`gauss`	`srft`	`direct`	`gauss`	`srft`
10	1.08e-1	5.63e-2	9.06e-2	4.22e-1	2.16e-1	3.56e-1	1.70e 0	8.94e-1	1.45e 0
20	1.97e-1	9.69e-2	1.03e-1	7.67e-1	3.69e-1	3.89e-1	3.07e 0	1.44e 0	1.53e 0
40	3.91e-1	1.84e-1	1.27e-1	1.50e 0	6.69e-1	4.33e-1	6.03e 0	2.64e 0	1.63e 0
80	7.84e-1	4.00e-1	2.19e-1	3.04e 0	1.43e 0	6.64e-1	1.20e 1	5.43e 0	2.08e 0
160	1.70e 0	9.92e-1	6.92e-1	6.36e 0	3.36e 0	1.61e 0	2.46e 1	1.16e 1	3.94e 0
320	3.89e 0	2.65e 0	2.98e 0	1.34e 1	7.45e 0	5.87e 0	5.00e 1	2.41e 1	1.21e 1
640	1.03e 1	8.75e 0	1.81e 1	3.14e 1	2.13e 1	2.99e 1	1.06e 2	5.80e 1	5.35e 1
1280	—	—	—	7.97e 1	6.69e 1	3.13e 2	2.40e 2	1.68e 2	4.03e 2
`svd`	1.19e 1			8.77e 1			6.90e 2		

Algorithm 4.2: Adaptive Randomized Range Finder

Given an $m \times n$ matrix A, a tolerance ε, and an integer r (e.g. $r = 10$), the following scheme computes an orthonormal matrix Q such that (2.19) holds with probability at least $1 - \min\{m, n\}10^{-r}$.

Draw standard Gaussian vectors $\boldsymbol{\omega}^{(1)}, \ldots, \boldsymbol{\omega}^{(r)}$ of length n.
For $i = 1, 2, \ldots, r$, compute $\mathbf{y}^{(i)} = \mathsf{A}\boldsymbol{\omega}^{(i)}$.
$j = 0$.
$\mathsf{Q}^{(0)} = [\,]$, the $m \times 0$ empty matrix.
while $\max\left\{ \left\|\mathbf{y}^{(j+1)}\right\|, \left\|\mathbf{y}^{(j+2)}\right\|, \ldots, \left\|\mathbf{y}^{(j+r)}\right\| \right\} > \varepsilon/(10\sqrt{2/\pi})$,
    $j = j + 1$.
    Overwrite $\mathbf{y}^{(j)}$ by $\left(\mathsf{I} - \mathsf{Q}^{(j-1)}(\mathsf{Q}^{(j-1)})^*\right)\mathbf{y}^{(j)}$.
    $\mathbf{q}^{(j)} = \mathbf{y}^{(j)} / \left\|\mathbf{y}^{(j)}\right\|$.
    $\mathsf{Q}^{(j)} = [\mathsf{Q}^{(j-1)}\ \mathbf{q}^{(j)}]$.
    Draw a standard Gaussian vector $\boldsymbol{\omega}^{(j+r)}$ of length n.
    $\mathbf{y}^{(j+r)} = \left(\mathsf{I} - \mathsf{Q}^{(j)}(\mathsf{Q}^{(j)})^*\right)\mathsf{A}\boldsymbol{\omega}^{(j+r)}$.
    for $i = (j+1), (j+2), \ldots, (j+r-1)$,
        Overwrite $\mathbf{y}^{(i)}$ by $\mathbf{y}^{(i)} - \mathbf{q}^{(j)}\left\langle \mathbf{q}^{(j)},\ \mathbf{y}^{(i)} \right\rangle$.
    end for
end while
$\mathsf{Q} = \mathsf{Q}^{(j)}$.

Algorithm 4.3: Randomized Power Iteration

Given an $m \times n$ matrix A and integers ℓ and q, this algorithm computes an $m \times \ell$ orthonormal matrix Q whose range approximates the range of A.

Draw an $n \times \ell$ Gaussian random matrix Ω.
Form the $m \times \ell$ matrix $\mathsf{Y} = (\mathsf{A}\mathsf{A}^*)^q\mathsf{A}\Omega$ via alternating application of A and A^*.
Construct an $m \times \ell$ matrix Q whose columns form an orthonormal basis for the range of Y, e.g., via the QR factorization $\mathsf{Y} = \mathsf{Q}\mathsf{R}$.

Note: This procedure is vulnerable to round-off errors; see Remark 8. The recommended implementation appears as Algorithm 4.4.

Algorithm 4.4: Randomized Subspace Iteration

Given an $m \times n$ matrix A and integers ℓ and q, this algorithm computes an $m \times \ell$ orthonormal matrix Q whose range approximates the range of A.

Draw an $n \times \ell$ standard Gaussian matrix Ω.
Form $\mathsf{Y}_0 = \mathsf{A}\Omega$ and compute its QR factorization $\mathsf{Y}_0 = \mathsf{Q}_0\mathsf{R}_0$.
for $j = 1, 2, \ldots, q$
    Form $\widetilde{\mathsf{Y}}_j = \mathsf{A}^*\mathsf{Q}_{j-1}$ and compute its QR factorization $\widetilde{\mathsf{Y}}_j = \widetilde{\mathsf{Q}}_j\widetilde{\mathsf{R}}_j$.
    Form $\mathsf{Y}_j = \mathsf{A}\widetilde{\mathsf{Q}}_j$ and compute its QR factorization $\mathsf{Y}_j = \mathsf{Q}_j\mathsf{R}_j$.
end
$\mathsf{Q} = \mathsf{Q}_q$.

Algorithm 4.5: Fast Randomized Range Finder

Given an $m \times n$ matrix A, and an integer ℓ, this scheme computes an $m \times \ell$ orthonormal matrix Q whose range approximates the range of A.

Draw an $n \times \ell$ SRFT test matrix Ω, as defined by (2.23).
Form the $m \times \ell$ matrix $\mathsf{Y} = \mathsf{A}\Omega$ using a (subsampled) FFT.
Construct an $m \times \ell$ matrix Q whose columns form an orthonormal basis for the range of Y, e.g., using the QR factorization $\mathsf{Y} = \mathsf{QR}$.

Algorithm 5.1: Direct SVD

Given matrices A and Q such that (2.26) holds, this procedure computes an approximate factorization $\mathsf{A} \approx \mathsf{U}\Sigma\mathsf{V}^*$, where U and V are orthonormal, and Σ is a nonnegative diagonal matrix.

Form the matrix $\mathsf{B} = \mathsf{Q}^*\mathsf{A}$.
Compute an SVD of the small matrix: $\mathsf{B} = \widetilde{\mathsf{U}}\Sigma\mathsf{V}^*$.
Form the orthonormal matrix $\mathsf{U} = \mathsf{Q}\widetilde{\mathsf{U}}$.

Algorithm 5.2: SVD via Row Extraction

Given matrices A *and* Q *such that* (2.26) *holds, this procedure computes an approximate factorization* $\mathsf{A} \approx \mathsf{U}\mathsf{\Sigma}\mathsf{V}^*$*, where* U *and* V *are orthonormal, and* $\mathsf{\Sigma}$ *is a nonnegative diagonal matrix.*

1 Compute an ID $\mathsf{Q} = \mathsf{X}\mathsf{Q}_{(J,:)}$. (The ID is defined in §2.3.2.3.)
2 Extract $\mathsf{A}_{(J,:)}$, and compute a QR factorization $\mathsf{A}_{(J,:)} = \mathsf{R}^*\mathsf{W}^*$.
3 Form the product $\mathsf{Z} = \mathsf{X}\mathsf{R}^*$.
4 Compute an SVD $\mathsf{Z} = \mathsf{U}\mathsf{\Sigma}\widetilde{\mathsf{V}}^*$.
5 Form the orthonormal matrix $\mathsf{V} = \mathsf{W}\widetilde{\mathsf{V}}$.

Note: Algorithm 5.2 is faster than Algorithm 5.1 but less accurate.
Note: It is advantageous to replace the basis Q by the sample matrix Y produced in Stage A, cf. Remark 14.

Algorithm 5.3: Direct Eigenvalue Decomposition

Given an Hermitian matrix A *and a basis* Q *such that* (2.26) *holds, this procedure computes an approximate eigenvalue decomposition* $\mathsf{A} \approx \mathsf{U}\mathsf{\Lambda}\mathsf{U}^*$*, where* U *is orthonormal, and* $\mathsf{\Lambda}$ *is a real diagonal matrix.*

1 Form the small matrix $\mathsf{B} = \mathsf{Q}^*\mathsf{A}\mathsf{Q}$.
2 Compute an eigenvalue decomposition $\mathsf{B} = \mathsf{V}\mathsf{\Lambda}\mathsf{V}^*$.
3 Form the orthonormal matrix $\mathsf{U} = \mathsf{Q}\mathsf{V}$.

Algorithm 5.4: Eigenvalue Decomposition via Row Extraction

Given an Hermitian matrix A *and a basis* Q *such that* (2.26) *holds, this procedure computes an approximate eigenvalue decomposition* $\mathsf{A} \approx \mathsf{U}\mathsf{\Lambda}\mathsf{U}^*$*, where* U *is orthonormal, and* $\mathsf{\Lambda}$ *is a real diagonal matrix.*

1 Compute an ID $\mathsf{Q} = \mathsf{X}\mathsf{Q}_{(J,:)}$.
2 Perform a QR factorization $\mathsf{X} = \mathsf{V}\mathsf{R}$.
3 Form the product $\mathsf{Z} = \mathsf{R}\mathsf{A}_{(J,J)}\mathsf{R}^*$.
4 Compute an eigenvalue decomposition $\mathsf{Z} = \mathsf{W}\mathsf{\Lambda}\mathsf{W}^*$.
5 Form the orthonormal matrix $\mathsf{U} = \mathsf{V}\mathsf{W}$.

Note: Algorithm 5.4 is faster than Algorithm 5.3 but less accurate.
Note: It is advantageous to replace the basis Q by the sample matrix Y produced in Stage A, cf. Remark 14.

ALGORITHM 5.5: EIGENVALUE DECOMPOSITION VIA NYSTRÖM METHOD

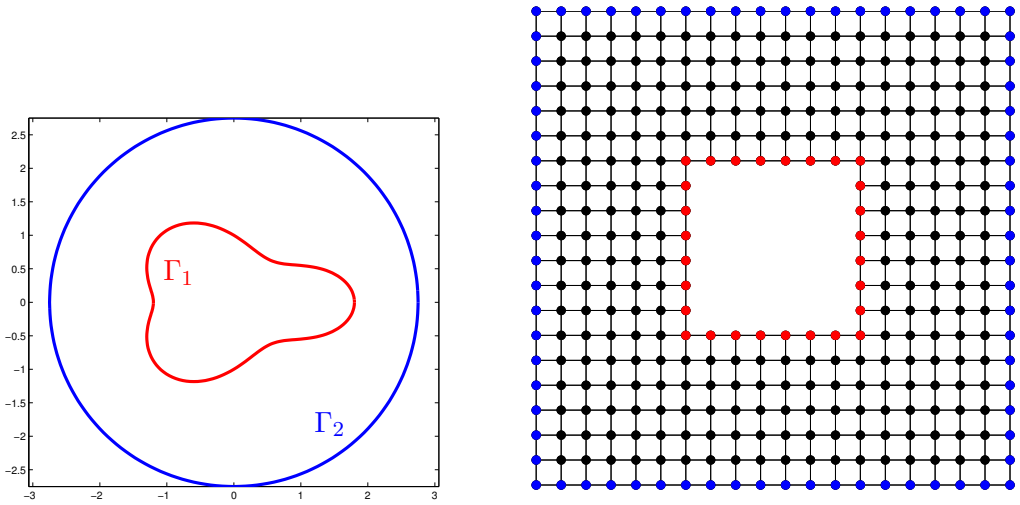
Given a positive semidefinite matrix A *and a basis* Q *such that* (2.26) *holds, this procedure computes an approximate eigenvalue decomposition* $\mathsf{A} \approx \mathsf{U}\Lambda\mathsf{U}^*$*, where* U *is orthonormal, and* Λ *is nonnegative and diagonal.*

1 Form the matrices $\mathsf{B}_1 = \mathsf{AQ}$ and $\mathsf{B}_2 = \mathsf{Q}^*\mathsf{B}_1$.
2 Perform a Cholesky factorization $\mathsf{B}_2 = \mathsf{C}^*\mathsf{C}$.
3 Form $\mathsf{F} = \mathsf{B}_1\mathsf{C}^{-1}$ using a triangular solve.
4 Compute an SVD $\mathsf{F} = \mathsf{U}\Sigma\mathsf{V}^*$ and set $\Lambda = \Sigma^2$.

ALGORITHM 5.6: EIGENVALUE DECOMPOSITION IN ONE PASS

Given an Hermitian matrix A*, a random test matrix* Ω*, a sample matrix* $\mathsf{Y} = \mathsf{A}\Omega$*, and an orthonormal matrix* Q *that verifies* (2.26) *and* $\mathsf{Y} = \mathsf{QQ}^*\mathsf{Y}$*, this algorithm computes an approximate eigenvalue decomposition* $\mathsf{A} \approx \mathsf{U}\Lambda\mathsf{U}^*$*.*

1 Use a standard least-squares solver to find an Hermitian matrix $\mathsf{B}_{\text{approx}}$ that approximately satisfies the equation $\mathsf{B}_{\text{approx}}(\mathsf{Q}^*\Omega) \approx \mathsf{Q}^*\mathsf{Y}$.
2 Compute the eigenvalue decomposition $\mathsf{B}_{\text{approx}} = \mathsf{V}\Lambda\mathsf{V}^*$.
3 Form the product $\mathsf{U} = \mathsf{QV}$.



(a) (b)

Figure 2.1: Configurations for physical problems. (a) The contours Γ_1 (red) and Γ_2 (blue) for the integral operator (2.46). (b) Geometry of the lattice problem associated with matrix B in §2.7.1.

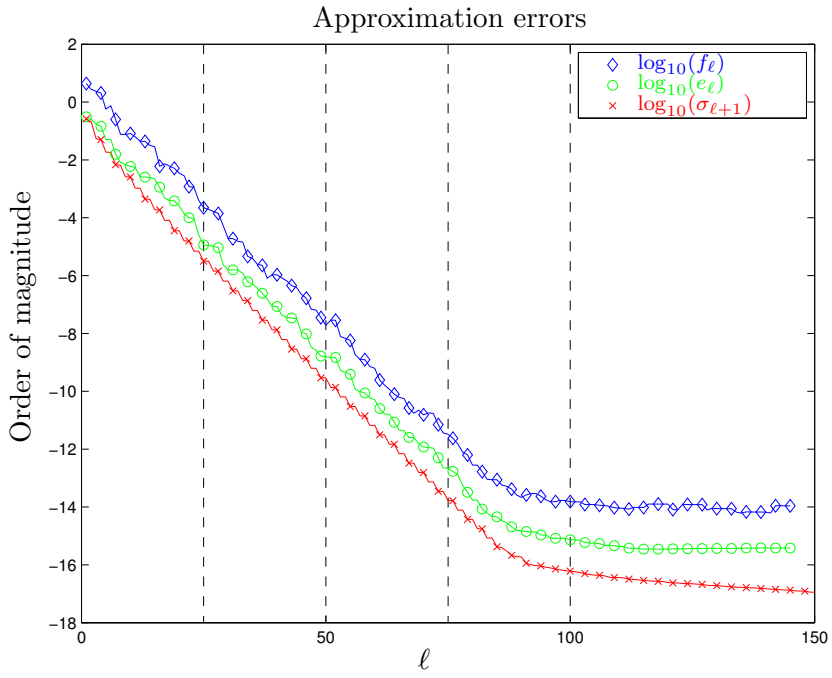


Figure 2.2: Approximating a Laplace integral operator. One execution of Algorithm 4.2 for the 200×200 input matrix A described in §2.7.1. The number ℓ of random samples varies along the horizontal axis; the vertical axis measures the base-10 logarithm of error magnitudes. The dashed vertical lines mark the points during execution at which Figure 2.3 provides additional statistics.

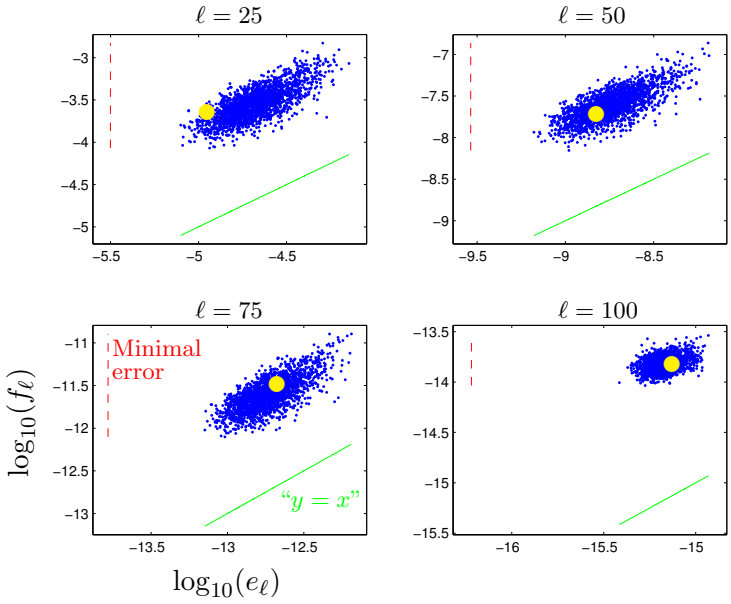


Figure 2.3: Error statistics for approximating a Laplace integral operator. 2,000 trials of Algorithm 4.2 applied to a 200×200 matrix approximating the integral operator (2.46). The panels isolate the moments at which $\ell = 25, 50, 75, 100$ random samples have been drawn. Each solid point compares the estimated error f_ℓ versus the actual error e_ℓ in one trial; the open circle indicates the trial detailed in Figure 2.2. The dashed line identifies the minimal error $\sigma_{\ell+1}$, and the solid line marks the contour where the error estimator would equal the actual error.

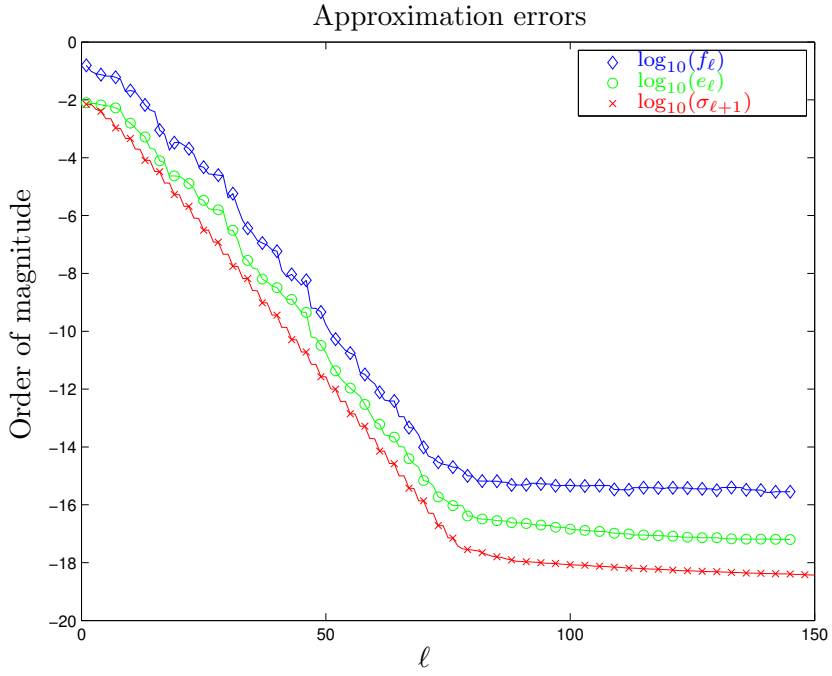


Figure 2.4: Approximating the inverse of a discrete Laplacian. One execution of Algorithm 4.2 for the 1596×532 input matrix B described in §2.7.1. See Figure 2.2 for notations.

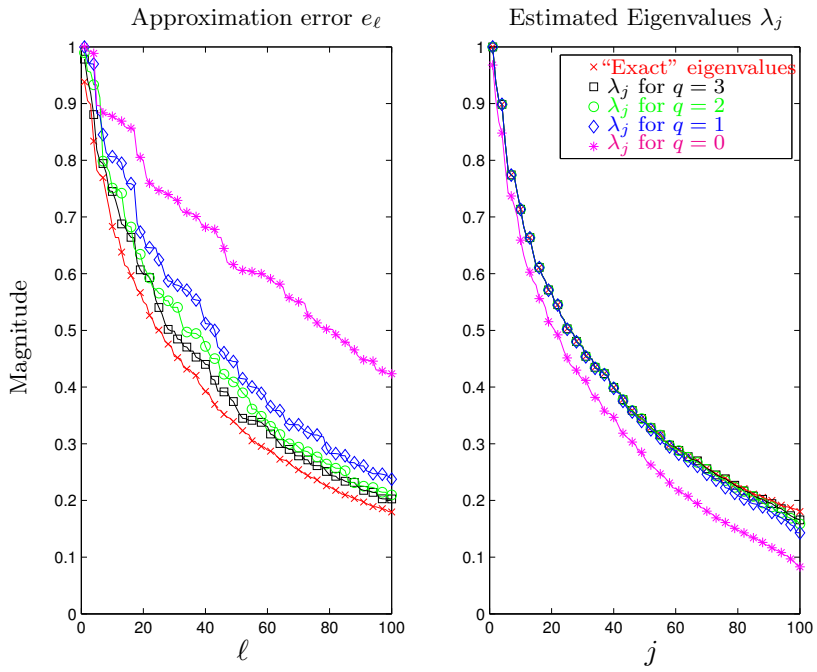


Figure 2.5: Approximating a graph Laplacian. For varying exponent q, one trial of the power scheme, Algorithm 4.3, applied to the 9025×9025 matrix A described in §2.7.2. [Left] Approximation errors as a function of the number ℓ of random samples. [Right] Estimates for the 100 largest eigenvalues given $\ell = 100$ random samples compared with the 100 largest eigenvalues of A.

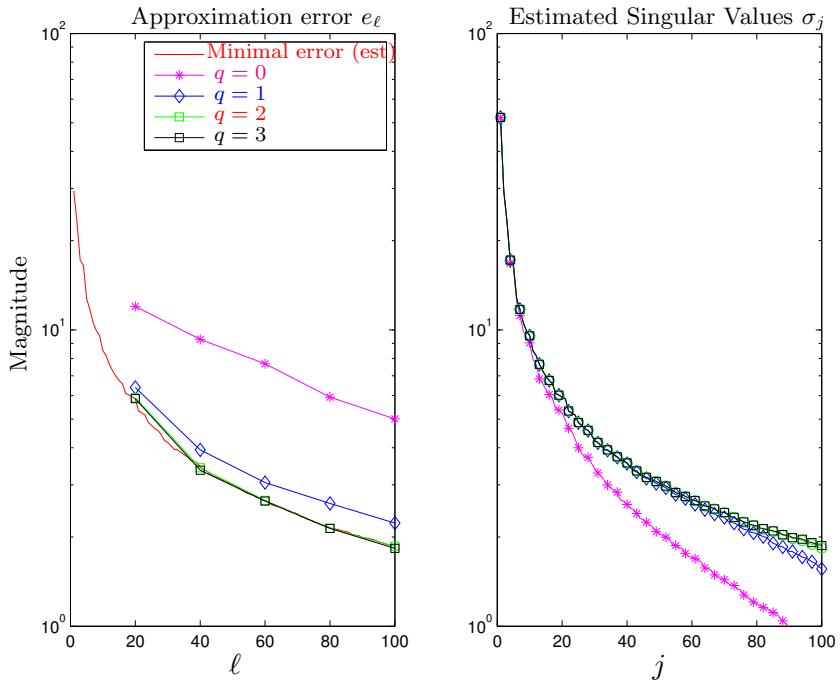


Figure 2.6: Computing eigenfaces. For varying exponent q, one trial of the power scheme, Algorithm 4.3, applied to the $98\,304 \times 7254$ matrix A described in §2.7.3. (Left) Approximation errors as a function of the number ℓ of random samples. The red line indicates the minimal errors as estimated by the singular values computed using $\ell = 100$ and $q = 3$. (Right) Estimates for the 100 largest eigenvalues given $\ell = 100$ random samples.

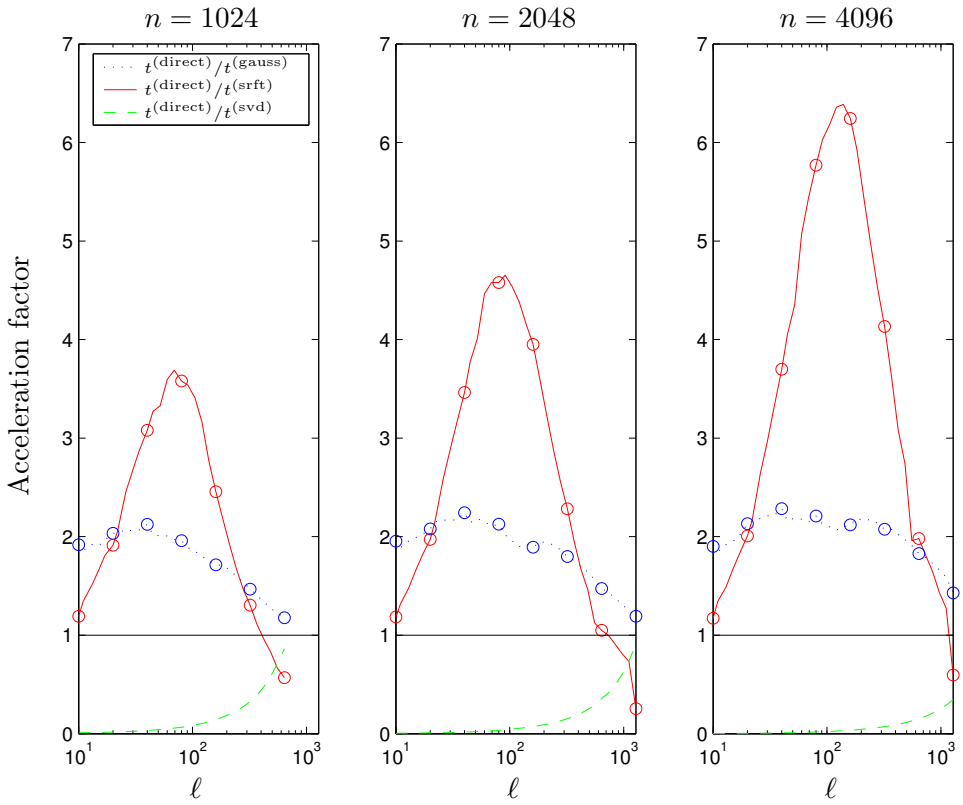


Figure 2.7: Acceleration factor. The relative cost of computing an ℓ-term partial SVD of an $n \times n$ Gaussian matrix using `direct`, a benchmark classical algorithm, versus each of the three competitors described in §2.7.4. The solid red curve shows the speedup using an SRFT test matrix, and the dotted blue curve shows the speedup with a Gaussian test matrix. The dashed green curve indicates that a full SVD computation using classical methods is substantially **slower**. Table 2.1 reports the absolute runtimes that yield the circled data points.

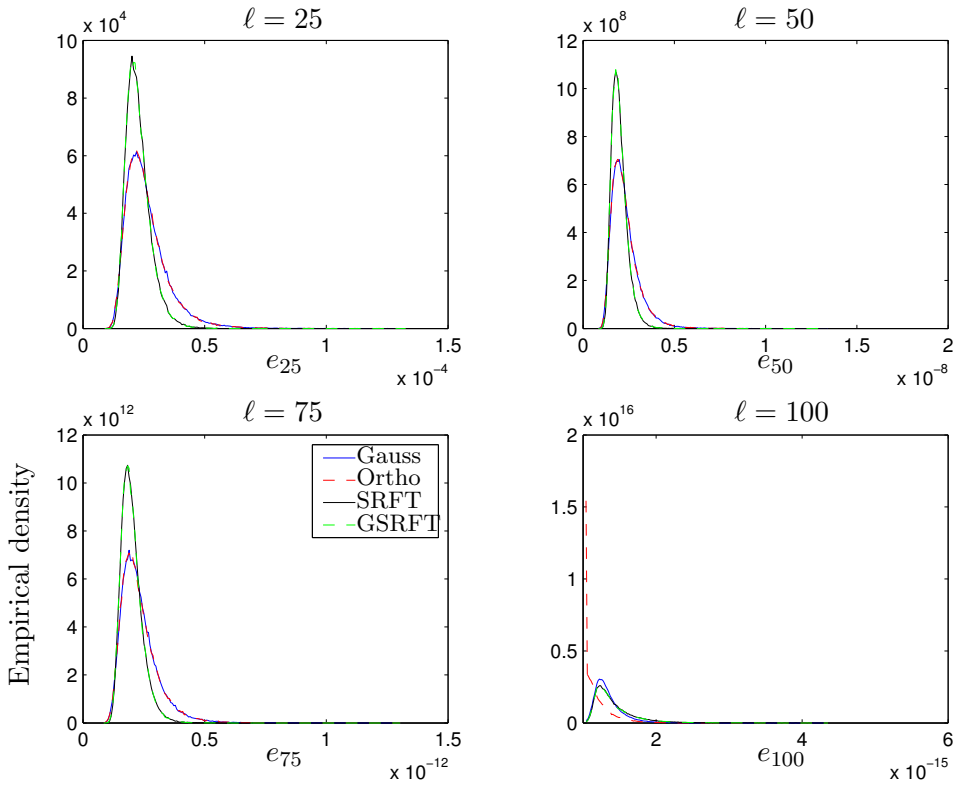


Figure 2.8: Empirical probability density functions for the error in Algorithm 4.1. As described in §2.7.4, the algorithm is implemented with four distributions for the random test matrix and used to approximate the 200×200 input matrix obtained by discretizing the integral operator (2.46). The four panels capture the empirical error distribution for each version of the algorithm at the moment when $\ell = 25, 50, 75, 100$ random samples have been drawn.

Part III: Theory

This part of the paper, §§2.8–2.11, provides a detailed analysis of randomized sampling schemes for constructing an approximate basis for the range of a matrix, the task we refer to as Stage A in the framework of §2.1.2. More precisely, we assess the quality of the basis Q that the proto-algorithm of §2.1.3 produces by establishing rigorous bounds for the approximation error

$$\|\!|\mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A}|\!\|, \tag{2.47}$$

where $\|\!|\cdot|\!\|$ denotes either the spectral norm or the Frobenius norm. The difficulty in developing these bounds is that the matrix Q is random, and its distribution is a complicated nonlinear function of the input matrix A and the random test matrix Ω. Naturally, any estimate for the approximation error must depend on the properties of the input matrix and the distribution of the test matrix.

To address these challenges, we split the argument into two pieces. The first part exploits techniques from linear algebra to deliver a generic error bound that depends on the interaction between the test matrix Ω and the right singular vectors of the input matrix A, as well as the tail singular values of A. In the second part of the argument, we take into account the distribution of the random matrix to estimate the error for specific instantiations of the proto-algorithm. This bipartite proof is common in the literature on randomized linear algebra, but our argument is most similar in spirit to [17].

Section 2.8 surveys the basic linear algebraic tools we need. Section 2.9 uses these methods to derive a generic error bound. Afterward, we specialize these results to the case where the test matrix is Gaussian (§2.10) and the case where the test matrix is a subsampled random Fourier transform (§2.11).

2.8 Theoretical preliminaries

We proceed with some additional background from linear algebra. Section 2.8.1 sets out properties of positive-semidefinite matrices, and §2.8.2 offers some results for orthogonal projectors. Standard references for this material include [11, 72].

2.8.1 Positive semidefinite matrices

An Hermitian matrix M is *positive semidefinite* (briefly, *psd*) when $\mathbf{u}^*\mathsf{M}\mathbf{u} \geq 0$ for all $\mathbf{u} \neq \mathbf{0}$. If the inequalities are strict, M is *positive definite* (briefly, *pd*). The psd matrices form a convex cone, which induces a partial ordering on the linear space of Hermitian matrices: $\mathsf{M} \preccurlyeq \mathsf{N}$ if and only if $\mathsf{N} - \mathsf{M}$ is psd. This ordering allows us to write $\mathsf{M} \succcurlyeq \mathsf{0}$ to indicate that the matrix M is psd.

Alternatively, we can define a psd (resp., pd) matrix as an Hermitian matrix with nonnegative (resp., positive) eigenvalues. In particular, each psd matrix is diagonalizable, and the inverse of a pd matrix is also pd. The spectral norm of a psd matrix M has the variational characterization

$$\|\mathsf{M}\| = \max_{\mathbf{u} \neq \mathbf{0}} \frac{\mathbf{u}^*\mathsf{M}\mathbf{u}}{\mathbf{u}^*\mathbf{u}}, \tag{2.48}$$

according to the Rayleigh–Ritz theorem [72, Thm. 4.2.2]. It follows that

$$\mathsf{M} \preccurlyeq \mathsf{N} \quad \Longrightarrow \quad \|\mathsf{M}\| \le \|\mathsf{N}\|. \tag{2.49}$$

A fundamental fact is that conjugation preserves the psd property.

Proposition 2.8.1 (Conjugation Rule)**.** *Suppose that* $\mathsf{M} \succcurlyeq \mathsf{0}$*. For every* A*, the matrix* $\mathsf{A}^*\mathsf{M}\mathsf{A} \succcurlyeq \mathsf{0}$*. In particular,*

$$\mathsf{M} \preccurlyeq \mathsf{N} \quad \Longrightarrow \quad \mathsf{A}^*\mathsf{M}\mathsf{A} \preccurlyeq \mathsf{A}^*\mathsf{N}\mathsf{A}.$$

Our argument invokes the conjugation rule repeatedly. As a first application, we establish a perturbation bound for the matrix inverse near the identity matrix.

Proposition 2.8.2 (Perturbation of Inverses)**.** *Suppose that* $\mathsf{M} \succcurlyeq \mathsf{0}$*. Then*

$$\mathsf{I} - (\mathsf{I} + \mathsf{M})^{-1} \preccurlyeq \mathsf{M}$$

Proof. Define $\mathsf{R} = \mathsf{M}^{1/2}$, the psd square root of M promised by [72, Thm. 7.2.6]. We have the chain of relations

$$\mathsf{I} - (\mathsf{I} + \mathsf{R}^2)^{-1} = (\mathsf{I} + \mathsf{R}^2)^{-1}\mathsf{R}^2 = \mathsf{R}(\mathsf{I} + \mathsf{R}^2)^{-1}\mathsf{R} \preccurlyeq \mathsf{R}^2.$$

The first equality can be verified algebraically. The second holds because rational functions of a diagonalizable matrix, such as R, commute. The last relation follows from the conjugation rule because $(\mathsf{I} + \mathsf{R}^2)^{-1} \preccurlyeq \mathsf{I}$. □

Next, we present a generalization of the fact that the spectral norm of a psd matrix is controlled by its trace.

Proposition 2.8.3. *We have* $\|\mathsf{M}\| \le \|\mathsf{A}\| + \|\mathsf{C}\|$ *for each partitioned psd matrix*

$$\mathsf{M} = \begin{bmatrix} \mathsf{A} & \mathsf{B} \\ \mathsf{B}^* & \mathsf{C} \end{bmatrix}.$$

Proof. The variational characterization (2.48) of the spectral norm implies that

$$\begin{aligned} \|\mathsf{M}\| &= \sup_{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 = 1} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}^* \begin{bmatrix} \mathsf{A} & \mathsf{B} \\ \mathsf{B}^* & \mathsf{C} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} \\ &\le \sup_{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 = 1} \big(\|\mathsf{A}\| \, \|\mathbf{x}\|^2 + 2 \, \|\mathsf{B}\| \, \|\mathbf{x}\| \, \|\mathbf{y}\| + \|\mathsf{C}\| \, \|\mathbf{y}\|^2 \big). \end{aligned}$$

The block generalization of Hadamard's psd criterion [72, Thm. 7.7.7] states that $\|\mathsf{B}\|^2 \le \|\mathsf{A}\| \, \|\mathsf{C}\|$. Thus,

$$\|\mathsf{M}\| \le \sup_{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 = 1} \big(\|\mathsf{A}\|^{1/2} \, \|\mathbf{x}\| + \|\mathsf{C}\|^{1/2} \, \|\mathbf{y}\| \big)^2 = \|\mathsf{A}\| + \|\mathsf{C}\|.$$

This point completes the argument. □

2.8.2 Orthogonal projectors

An *orthogonal projector* is an Hermitian matrix P that satisfies the polynomial $\mathsf{P}^2 = \mathsf{P}$. This identity implies $\mathsf{0} \preccurlyeq \mathsf{P} \preccurlyeq \mathsf{I}$. An orthogonal projector is completely determined by its range. For a given matrix M, we write P_{M} for the unique orthogonal projector with $\operatorname{range}(\mathsf{P}_{\mathsf{M}}) = \operatorname{range}(\mathsf{M})$. When M has full column rank, we can express this projector explicitly:

$$\mathsf{P}_{\mathsf{M}} = \mathsf{M}(\mathsf{M}^*\mathsf{M})^{-1}\mathsf{M}^*. \tag{2.50}$$

The orthogonal projector onto the complementary subspace, $\operatorname{range}(\mathsf{P})^\perp$, is the matrix $\mathsf{I} - \mathsf{P}$. Our argument hinges on several other facts about orthogonal projectors.

Proposition 2.8.4. *Suppose* U *is unitary. Then* $\mathsf{U}^*\mathsf{P}_{\mathsf{M}}\mathsf{U} = \mathsf{P}_{\mathsf{U}^*\mathsf{M}}$.

Proof. Abbreviate $\mathsf{P} = \mathsf{U}^*\mathsf{P}_{\mathsf{M}}\mathsf{U}$. It is clear that P is an orthogonal projector since it is Hermitian and $\mathsf{P}^2 = \mathsf{P}$. Evidently,

$$\operatorname{range}(\mathsf{P}) = \mathsf{U}^* \operatorname{range}(\mathsf{M}) = \operatorname{range}(\mathsf{U}^*\mathsf{M}).$$

Since the range determines the orthogonal projector, we conclude $\mathsf{P} = \mathsf{P}_{\mathsf{U}^*\mathsf{M}}$. □

Proposition 2.8.5. *Suppose* $\operatorname{range}(\mathsf{N}) \subset \operatorname{range}(\mathsf{M})$. *Then, for each matrix* A, *it holds that* $\|\mathsf{P}_{\mathsf{N}}\mathsf{A}\| \le \|\mathsf{P}_{\mathsf{M}}\mathsf{A}\|$ *and that* $\|(\mathsf{I} - \mathsf{P}_{\mathsf{M}})\mathsf{A}\| \le \|(\mathsf{I} - \mathsf{P}_{\mathsf{N}})\mathsf{A}\|$.

Proof. The projector $\mathsf{P}_{\mathsf{N}} \preccurlyeq \mathsf{I}$, so the conjugation rule yields $\mathsf{P}_{\mathsf{M}}\mathsf{P}_{\mathsf{N}}\mathsf{P}_{\mathsf{M}} \preccurlyeq \mathsf{P}_{\mathsf{M}}$. The hypothesis $\operatorname{range}(\mathsf{N}) \subset \operatorname{range}(\mathsf{M})$ implies that $\mathsf{P}_{\mathsf{M}}\mathsf{P}_{\mathsf{N}} = \mathsf{P}_{\mathsf{N}}$, which results in

$$\mathsf{P}_{\mathsf{M}}\mathsf{P}_{\mathsf{N}}\mathsf{P}_{\mathsf{M}} = \mathsf{P}_{\mathsf{N}}\mathsf{P}_{\mathsf{M}} = (\mathsf{P}_{\mathsf{M}}\mathsf{P}_{\mathsf{N}})^* = \mathsf{P}_{\mathsf{N}}.$$

In summary, $\mathsf{P}_{\mathsf{N}} \preccurlyeq \mathsf{P}_{\mathsf{M}}$. The conjugation rule shows that $\mathsf{A}^*\mathsf{P}_{\mathsf{N}}\mathsf{A} \preccurlyeq \mathsf{A}^*\mathsf{P}_{\mathsf{M}}\mathsf{A}$. We conclude from (2.49) that

$$\|\mathsf{P}_{\mathsf{N}}\mathsf{A}\|^2 = \|\mathsf{A}^*\mathsf{P}_{\mathsf{N}}\mathsf{A}\| \le \|\mathsf{A}^*\mathsf{P}_{\mathsf{M}}\mathsf{A}\| = \|\mathsf{P}_{\mathsf{M}}\mathsf{A}\|^2.$$

The second statement follows from the first by taking orthogonal complements. □

Finally, we need a generalization of the scalar inequality $|px|^q \le |p|\,|x|^q$, which holds when $|p| \le 1$ and $q \ge 1$.

Proposition 2.8.6. *Let* P *be an orthogonal projector, and let* M *be a matrix. For each positive number* q,

$$\|\mathsf{P}\mathsf{M}\| \le \|\mathsf{P}(\mathsf{M}\mathsf{M}^*)^q\mathsf{M}\|^{1/(2q+1)}. \tag{2.51}$$

Proof. Suppose that R is an orthogonal projector, D is a nonnegative diagonal matrix, and $t \geq 1$. We claim that

$$\|\mathsf{RDR}\|^t \leq \left\|\mathsf{RD}^t\mathsf{R}\right\|. \tag{2.52}$$

Granted this inequality, we quickly complete the proof. Using an SVD $\mathsf{M} = \mathsf{U}\Sigma\mathsf{V}^*$, we compute

$$\begin{aligned}\|\mathsf{PM}\|^{2(2q+1)} = \|\mathsf{PMM}^*\mathsf{P}\|^{2q+1} &= \left\|(\mathsf{U}^*\mathsf{PU})\cdot\Sigma^2\cdot(\mathsf{U}^*\mathsf{PU})\right\|^{2q+1} \\ &\leq \left\|(\mathsf{U}^*\mathsf{PU})\cdot\Sigma^{2(2q+1)}\cdot(\mathsf{U}^*\mathsf{PU})\right\| = \left\|\mathsf{P}(\mathsf{MM}^*)^{2(2q+1)}\mathsf{P}\right\| \\ &= \left\|\mathsf{P}(\mathsf{MM}^*)^q\mathsf{M}\cdot\mathsf{M}^*(\mathsf{MM}^*)^q\mathsf{P}\right\| = \|\mathsf{P}(\mathsf{MM}^*)^q\mathsf{M}\|^2.\end{aligned}$$

We have used the unitary invariance of the spectral norm in the second and fourth relations. The inequality (2.52) applies because $\mathsf{U}^*\mathsf{PU}$ is an orthogonal projector. Take a square root to finish the argument.

Now, we turn to the claim (2.52). This relation follows immediately from [11, Thm. IX.2.10], but we offer a direct argument based on more elementary considerations. Let $\mathbf{x}$ be a unit vector at which

$$\mathbf{x}^*(\mathsf{RDR})\mathbf{x} = \|\mathsf{RDR}\|.$$

We must have $\mathsf{R}\mathbf{x} = \mathbf{x}$. Otherwise, $\|\mathsf{R}\mathbf{x}\| < 1$ because R is an orthogonal projector, which implies that the unit vector $\mathbf{y} = \mathsf{R}\mathbf{x}/\|\mathsf{R}\mathbf{x}\|$ verifies

$$\mathbf{y}^*(\mathsf{RDR})\mathbf{y} = \frac{(\mathsf{R}\mathbf{x})^*(\mathsf{RDR})(\mathsf{R}\mathbf{x})}{\|\mathsf{R}\mathbf{x}\|^2} = \frac{\mathbf{x}^*(\mathsf{RDR})\mathbf{x}}{\|\mathsf{R}\mathbf{x}\|^2} > \mathbf{x}^*(\mathsf{RDR})\mathbf{x}.$$

Writing x_j for the entries of $\mathbf{x}$ and d_j for the diagonal entries of D, we find that

$$\begin{aligned}\|\mathsf{RDR}\|^t = [\mathbf{x}^*(\mathsf{RDR})\mathbf{x}]^t = [\mathbf{x}^*\mathsf{D}\mathbf{x}]^t &= \left[\sum\nolimits_j d_j x_j^2\right]^t \\ &\leq \left[\sum\nolimits_j d_j^t x_j^2\right] = \mathbf{x}^*\mathsf{D}^t\mathbf{x} = (\mathsf{R}\mathbf{x})^*\mathsf{D}^t(\mathsf{R}\mathbf{x}) \leq \left\|\mathsf{RD}^t\mathsf{R}\right\|.\end{aligned}$$

The inequality is Jensen's, which applies because $\sum x_j^2 = 1$ and the function $z \mapsto |z|^t$ is convex for $t \geq 1$. □

2.9 Error bounds via linear algebra

We are now prepared to develop a deterministic error analysis for the proto-algorithm described in §2.1.3. To begin, we must introduce some notation. Afterward, we establish the key error bound, which strengthens a result from the literature [17, Lem. 4.2]. Finally, we explain why the power method can be used to improve the performance of the proto-algorithm.

2.9.1 Setup

Let A be an $m \times n$ matrix that has a singular value decomposition $\mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^*$, as described in Section 2.3.2.2. Roughly speaking, the proto-algorithm tries to approximate the subspace spanned by the first k left singular vectors, where k is now a fixed number. To perform the analysis, it is appropriate to partition the singular value decomposition as follows.

$$\mathsf{A} = \mathsf{U} \begin{array}{c} \begin{matrix} k & n-k \end{matrix} \\ \begin{bmatrix} \Sigma_1 & \\ & \Sigma_2 \end{bmatrix} \end{array} \begin{array}{c} \begin{matrix} n \end{matrix} \\ \begin{bmatrix} \mathsf{V}_1^* \\ \mathsf{V}_2^* \end{bmatrix} \end{array} \begin{array}{c} k \\ n-k \end{array} \tag{2.53}$$

The matrices Σ_1 and Σ_2 are square. We will see that the left unitary factor U does not play a significant role in the analysis.

Let Ω be an $n \times \ell$ test matrix, where ℓ denotes the number of samples. We assume only that $\ell \geq k$. Decompose the test matrix in the coordinate system determined by the right unitary factor of A:

$$\Omega_1 = \mathsf{V}_1^* \, \Omega \quad \text{and} \quad \Omega_2 = \mathsf{V}_2^* \, \Omega. \tag{2.54}$$

The error bound for the proto-algorithm depends critically on the properties of the matrices Ω_1 and Ω_2. With this notation, the sample matrix Y can be expressed as

$$\mathsf{Y} = \mathsf{A}\Omega = \mathsf{U} \begin{array}{c} \begin{matrix} \ell \end{matrix} \\ \begin{bmatrix} \Sigma_1\Omega_1 \\ \Sigma_2\Omega_2 \end{bmatrix} \end{array} \begin{array}{c} k \\ n-k \end{array}$$

It is a useful intuition that the block $\Sigma_1\Omega_1$ in (2.9.1) reflects the gross behavior of A, while the block $\Sigma_2\Omega_2$ represents a perturbation.

2.9.2 A deterministic error bound for the proto-algorithm

The proto-algorithm constructs an orthonormal basis Q for the range of the sample matrix Y, and our goal is to quantify how well this basis captures the action of the input A. Since $\mathsf{Q}\mathsf{Q}^* = \mathsf{P}_\mathsf{Y}$, the challenge is to obtain bounds on the approximation error

$$\left\| \mathsf{A} - \mathsf{Q}\mathsf{Q}^*\mathsf{A} \right\| = \left\| (\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A} \right\|.$$

The following theorem shows that the behavior of the proto-algorithm depends on the interaction between the test matrix and the right singular vectors of the input matrix, as well as the singular spectrum of the input matrix.

Theorem 2.9.1 (Deterministic error bound)**.** *Let A be an $m \times n$ matrix with singular value decomposition $\mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^*$, and fix $k \geq 0$. Choose a test matrix Ω, and construct the sample matrix $\mathsf{Y} = \mathsf{A}\Omega$. Partition Σ as specified in* (2.53)*, and define Ω_1 and Ω_2 via* (2.54)*. Assuming that Ω_1 has full row rank, the approximation error satisfies*

$$\left\| (\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A} \right\|^2 \leq \left\| \Sigma_2 \right\|^2 + \left\| \Sigma_2\Omega_2\Omega_1^\dagger \right\|^2, \tag{2.55}$$

where $\|\cdot\|$ denotes either the spectral norm or the Frobenius norm.

Theorem 2.9.1 sharpens the result [17, Lem. 2], which lacks the squares present in (2.55). This refinement yields slightly better error estimates than the earlier bound, and it has consequences for the probabilistic behavior of the error when the test matrix Ω is random. The proof here is different in spirit from the earlier analysis; our argument is inspired by the perturbation theory of orthogonal projectors [125].

Proof. We establish the bound for the spectral-norm error. The bound for the Frobenius-norm error follows from an analogous argument that is slightly easier.

Let us begin with some preliminary simplifications. First, we argue that the left unitary factor U plays no essential role in the argument. In effect, we execute the proof for an auxiliary input matrix $\widetilde{\mathsf{A}}$ and an associated sample matrix $\widetilde{\mathsf{Y}}$ defined by

$$\widetilde{\mathsf{A}} = \mathsf{U}^*\mathsf{A} = \begin{bmatrix} \Sigma_1\mathsf{V}_1^* \\ \Sigma_2\mathsf{V}_2^* \end{bmatrix} \quad \text{and} \quad \widetilde{\mathsf{Y}} = \widetilde{\mathsf{A}}\Omega = \begin{bmatrix} \Sigma_1\Omega_1 \\ \Sigma_2\Omega_2 \end{bmatrix}. \tag{2.56}$$

Owing to the unitary invariance of the spectral norm and to Proposition 2.8.4, we have the identity

$$\|(\mathsf{I} - \mathsf{P}_{\mathsf{Y}})\mathsf{A}\| = \big\|\mathsf{U}^*(\mathsf{I} - \mathsf{P}_{\mathsf{Y}})\mathsf{U}\widetilde{\mathsf{A}}\big\| = \big\|(\mathsf{I} - \mathsf{P}_{\mathsf{U}^*\mathsf{Y}})\widetilde{\mathsf{A}}\big\| = \big\|(\mathsf{I} - \mathsf{P}_{\widetilde{\mathsf{Y}}})\widetilde{\mathsf{A}}\big\|. \tag{2.57}$$

In view of (2.57), it suffices to prove that

$$\big\|(\mathsf{I} - \mathsf{P}_{\widetilde{\mathsf{Y}}})\widetilde{\mathsf{A}}\big\| \le \big\|\Sigma_2\big\|^2 + \big\|\Sigma_2\Omega_2\Omega_1^\dagger\big\|^2. \tag{2.58}$$

Second, we assume that the number k is chosen so the diagonal entries of Σ_1 are strictly positive. Suppose not. Then Σ_2 is zero because of the ordering of the singular values. As a consequence,

$$\operatorname{range}(\widetilde{\mathsf{A}}) = \operatorname{range}\begin{bmatrix} \Sigma_1\mathsf{V}_1^* \\ 0 \end{bmatrix} = \operatorname{range}\begin{bmatrix} \Sigma_1\Omega_1 \\ 0 \end{bmatrix} = \operatorname{range}(\widetilde{\mathsf{Y}}).$$

This calculation uses the decompositions presented in (2.56), as well as the fact that both V_1^* and Ω_1 have full row rank. We conclude that

$$\big\|(\mathsf{I} - \mathsf{P}_{\widetilde{\mathsf{Y}}})\widetilde{\mathsf{A}}\big\| = 0,$$

so the error bound (2.58) holds trivially. (In fact, both sides are zero.)

The main argument is based on ideas from perturbation theory. To illustrate the concept, we start with a matrix related to $\widetilde{\mathsf{Y}}$:

$$\mathsf{W} = \begin{matrix} \\ \left[\begin{array}{c} \Sigma_1\Omega_1 \\ 0 \end{array}\right] \end{matrix} \begin{matrix} \ell \\ \begin{matrix} k \\ n-k \end{matrix} \end{matrix}$$

The matrix W has the same range as a related matrix formed by "flattening out" the spectrum of the top block. Indeed, since $\Sigma_1\Omega_1$ has full row rank,

$$\operatorname{range}(\mathsf{W}) = \operatorname{range} \begin{matrix} \\ \left[\begin{array}{c} \mathsf{I} \\ 0 \end{array}\right] \end{matrix} \begin{matrix} k \\ \begin{matrix} k \\ n-k \end{matrix} \end{matrix}$$

The matrix on the right-hand side has full column rank, so it is legal to apply the formula (2.50) for an orthogonal projector, which immediately yields

$$\mathsf{P}_{\mathsf{W}} = \begin{bmatrix} \mathsf{I} & 0 \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathsf{I} - \mathsf{P}_{\mathsf{W}} = \begin{bmatrix} 0 & 0 \\ 0 & \mathsf{I} \end{bmatrix}. \tag{2.59}$$

In words, the range of W aligns with the first k coordinates, which span the same subspace as the first k left singular vectors of the auxiliary input matrix $\widetilde{\mathsf{A}}$. Therefore, range(W) captures the action of $\widetilde{\mathsf{A}}$, which is what we wanted from range($\widetilde{\mathsf{Y}}$).

We treat the auxiliary sample matrix $\widetilde{\mathsf{Y}}$ as a perturbation of W, and we hope that their ranges are close to each other. To make the comparison rigorous, let us emulate the arguments outlined in the last paragraph. Referring to the display (2.56), we flatten out the top block of $\widetilde{\mathsf{Y}}$ to obtain the matrix

$$\mathsf{Z} = \widetilde{\mathsf{Y}} \cdot \Omega_1^\dagger \Sigma_1^{-1} = \begin{bmatrix} \mathsf{I} \\ \mathsf{F} \end{bmatrix} \quad \text{where} \quad \mathsf{F} = \Sigma_2 \Omega_2 \Omega_1^\dagger \Sigma_1^{-1}. \tag{2.60}$$

Let us return to the error bound (2.58). The construction (2.60) ensures that range(Z) $\subset$ range($\widetilde{\mathsf{Y}}$), so Proposition 2.8.5 implies that the error satisfies

$$\left\| (\mathsf{I} - \mathsf{P}_{\widetilde{\mathsf{Y}}}) \widetilde{\mathsf{A}} \right\| \le \left\| (\mathsf{I} - \mathsf{P}_{\mathsf{Z}}) \widetilde{\mathsf{A}} \right\|.$$

Squaring this relation, we obtain

$$\left\| (\mathsf{I} - \mathsf{P}_{\widetilde{\mathsf{Y}}}) \widetilde{\mathsf{A}} \right\|^2 \le \left\| (\mathsf{I} - \mathsf{P}_{\mathsf{Z}}) \widetilde{\mathsf{A}} \right\|^2 = \left\| \widetilde{\mathsf{A}}^* (\mathsf{I} - \mathsf{P}_{\mathsf{Z}}) \widetilde{\mathsf{A}} \right\| = \left\| \Sigma^* (\mathsf{I} - \mathsf{P}_{\mathsf{Z}}) \Sigma \right\|. \tag{2.61}$$

The last identity follows from the definition $\widetilde{\mathsf{A}} = \Sigma \mathsf{V}^*$ and the unitary invariance of the spectral norm. Therefore, we can complete the proof of (2.58) by producing a suitable bound for the right-hand side of (2.61).

To continue, we need a detailed representation of the projector $\mathsf{I} - \mathsf{P}_{\mathsf{Z}}$. The construction (2.60) ensures that Z has full column rank, so we can apply the formula (2.50) for an orthogonal projector to see that

$$\mathsf{P}_{\mathsf{Z}} = \mathsf{Z}(\mathsf{Z}^*\mathsf{Z})^{-1}\mathsf{Z}^* = \begin{bmatrix} \mathsf{I} \\ \mathsf{F} \end{bmatrix} (\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1} \begin{bmatrix} \mathsf{I} \\ \mathsf{F} \end{bmatrix}^*.$$

Expanding this expression, we determine that the complementary projector satisfies

$$\mathsf{I} - \mathsf{P}_{\mathsf{Z}} = \begin{bmatrix} \mathsf{I} - (\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1} & -(\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1}\mathsf{F}^* \\ -\mathsf{F}(\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1} & \mathsf{I} - \mathsf{F}(\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1}\mathsf{F}^* \end{bmatrix}. \tag{2.62}$$

The partitioning here conforms with the partitioning of Σ. When we conjugate the matrix by Σ, copies of Σ_1^{-1}, presently hidden in the top-left block, will cancel to happy effect.

The latter point may not seem obvious, owing to the complicated form of (2.62). In reality, the block matrix is less fearsome than it looks. Proposition 2.8.2, on the perturbation of inverses, shows that the top-left block verifies

$$\mathsf{I} - (\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1} \preccurlyeq \mathsf{F}^*\mathsf{F}.$$

The bottom-right block satisfies

$$\mathsf{I} - \mathsf{F}(\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1}\mathsf{F}^* \preccurlyeq \mathsf{I}$$

because the conjugation rule guarantees that $\mathsf{F}(\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1}\mathsf{F}^* \succcurlyeq \mathsf{0}$. We abbreviate the off-diagonal blocks with the symbol $\mathsf{B} = -(\mathsf{I} + \mathsf{F}^*\mathsf{F})^{-1}\mathsf{F}^*$. In summary,

$$\mathsf{I} - \mathsf{P}_\mathsf{Z} \preccurlyeq \begin{bmatrix} \mathsf{F}^*\mathsf{F} & \mathsf{B} \\ \mathsf{B}^* & \mathsf{I} \end{bmatrix}.$$

This relation exposes the key structural properties of the projector. Compare this relation with the expression (2.59) for the "ideal" projector $\mathsf{I} - \mathsf{P}_\mathsf{W}$.

Moving toward the estimate required by (2.61), we conjugate the last relation by $\mathsf{\Sigma}$ to obtain

$$\mathsf{\Sigma}^*(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{\Sigma} \preccurlyeq \begin{bmatrix} \mathsf{\Sigma}_1^*\mathsf{F}^*\mathsf{F}\mathsf{\Sigma}_1 & \mathsf{\Sigma}_1^*\mathsf{B}\mathsf{\Sigma}_2 \\ \mathsf{\Sigma}_2^*\mathsf{B}^*\mathsf{\Sigma}_1 & \mathsf{\Sigma}_2^*\mathsf{\Sigma}_2 \end{bmatrix}.$$

The conjugation rule demonstrates that the matrix on the left-hand side is psd, so the matrix on the right-hand side is too. Proposition 2.8.3 results in the norm bound

$$\|\mathsf{\Sigma}^*(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{\Sigma}\| \le \|\mathsf{\Sigma}_1^*\mathsf{F}^*\mathsf{F}\mathsf{\Sigma}_1\| + \|\mathsf{\Sigma}_2^*\mathsf{\Sigma}_2\| = \|\mathsf{F}\mathsf{\Sigma}_1\|^2 + \|\mathsf{\Sigma}_2\|^2.$$

Recall that $\mathsf{F} = \mathsf{\Sigma}_2\mathsf{\Omega}_2\mathsf{\Omega}_1^\dagger\mathsf{\Sigma}_1^{-1}$, so the factor $\mathsf{\Sigma}_1$ cancels neatly. Therefore,

$$\|\mathsf{\Sigma}^*(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{\Sigma}\| \le \left\|\mathsf{\Sigma}_2\mathsf{\Omega}_2\mathsf{\Omega}_1^\dagger\right\|^2 + \|\mathsf{\Sigma}_2\|^2.$$

Finally, introduce the latter inequality into (2.61) to complete the proof. □

2.9.3 Analysis of the power scheme

Theorem 2.9.1 suggests that the performance of the proto-algorithm depends strongly on the relationship between the large singular values of A listed in $\mathsf{\Sigma}_1$ and the small singular values listed in $\mathsf{\Sigma}_2$. When a substantial proportion of the mass of A appears in the small singular values, the constructed basis Q may have low accuracy. Conversely, when the large singular values dominate, it is much easier to identify a good low-rank basis.

To improve the performance of the proto-algorithm, we can run it with a closely related input matrix whose singular values decay more rapidly [67, 112]. Fix a positive integer q, and set

$$\mathsf{B} = (\mathsf{A}\mathsf{A}^*)^q\mathsf{A} = \mathsf{U}\mathsf{\Sigma}^{2q+1}\mathsf{V}^*.$$

We apply the proto-algorithm to B, which generates a sample matrix $\mathsf{Z} = \mathsf{B}\mathsf{\Omega}$ and constructs a basis Q for the range of Z. Section 2.4.5 elaborates on the implementation details, and describes a reformulation that sometimes improves the accuracy when the scheme is executed in finite-precision arithmetic. The following result describes how well we can approximate the **original** matrix A within the range of Z.

Theorem 2.9.2 (Power scheme)**.** *Let* A *be an* $m \times n$ *matrix, and let* Ω *be an* $n \times \ell$ *matrix. Fix a nonnegative integer* q*, form* $\mathsf{B} = (\mathsf{A}^*\mathsf{A})^q\mathsf{A}$*, and compute the sample matrix* $\mathsf{Z} = \mathsf{B}\Omega$*. Then*

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{A}\| \le \|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{B}\|^{1/(2q+1)}.$$

Proof. We determine that

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{A}\| \le \|(\mathsf{I} - \mathsf{P}_\mathsf{Z})(\mathsf{A}\mathsf{A}^*)^q\mathsf{A}\|^{1/(2q+1)} = \|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{B}\|^{1/(2q+1)}$$

as a direct consequence of Proposition 2.8.6. □

Let us illustrate how the power scheme interacts with the main error bound (2.55). Let σ_{k+1} denote the $(k+1)$th singular value of A. First, suppose we approximate A in the range of the sample matrix $\mathsf{Y} = \mathsf{A}\Omega$. Since $\|\Sigma_2\| = \sigma_{k+1}$, Theorem 2.9.1 implies that

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\| \le \left(1 + \left\|\Omega_2\Omega_1^\dagger\right\|^2\right)^{1/2}\sigma_{k+1}. \tag{2.63}$$

Now, define $\mathsf{B} = (\mathsf{A}\mathsf{A}^*)^q\mathsf{A}$, and suppose we approximate A within the range of the sample matrix $\mathsf{Z} = \mathsf{B}\Omega$. Together, Theorem 2.9.2 and Theorem 2.9.1 imply that

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{A}\| \le \|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{B}\|^{1/(2q+1)} \le \left(1 + \left\|\Omega_2\Omega_1^\dagger\right\|^2\right)^{1/(4q+2)}\sigma_{k+1}$$

because σ_{k+1}^{2q+1} is the $(k+1)$th singular value of B. In effect, the power scheme drives down the suboptimality of the bound (2.63) exponentially fast as the power q increases. In principle, we can make the extra factor as close to one as we like, although this increases the cost of the algorithm.

2.9.4 Analysis of truncated SVD

Finally, let us study the truncated SVD described in Remark 13. Suppose that we approximate the input matrix A inside the range of the sample matrix Z. In essence, the truncation step computes a best rank-k approximation $\widehat{\mathsf{A}}_{(k)}$ of the compressed matrix $\mathsf{P}_\mathsf{Z}\mathsf{A}$. The next result provides a simple error bound for this method; this argument was proposed by Ming Gu.

Theorem 2.9.3 (Analysis of Truncated SVD)**.** *Let* A *be an* $m \times n$ *matrix with singular values* $\sigma_1 \ge \sigma_2 \ge \sigma_3 \ge \dots$*, and let* Z *be an* $m \times \ell$ *matrix, where* $\ell \ge k$*. Suppose that* $\widehat{\mathsf{A}}_{(k)}$ *is a best rank-*k *approximation of* $\mathsf{P}_\mathsf{Z}\mathsf{A}$ *with respect to the spectral norm. Then*

$$\left\|\mathsf{A} - \widehat{\mathsf{A}}_{(k)}\right\| \le \sigma_{k+1} + \|(\mathsf{I} - \mathsf{P}_\mathsf{Z})\mathsf{A}\|.$$

Proof. Apply the triangle inequality to split the error into two components.

$$\left\|\mathsf{A} - \widehat{\mathsf{A}}_k\right\| \le \left\|\mathsf{A} - \mathsf{P}_\mathsf{Z}\mathsf{A}\right\| + \left\|\mathsf{P}_\mathsf{Z}\mathsf{A} - \widehat{\mathsf{A}}_{(k)}\right\|. \tag{2.64}$$

We have already developed a detailed theory for estimating the first term. To analyze the second term, we introduce a best rank-k approximation $\mathsf{A}_{(k)}$ of the matrix A. Note that

$$\left\|\mathsf{P}_{\mathsf{Z}}\mathsf{A} - \widehat{\mathsf{A}}_{(k)}\right\| \le \left\|\mathsf{P}_{\mathsf{Z}}\mathsf{A} - \mathsf{P}_{\mathsf{Z}}\mathsf{A}_{(k)}\right\|$$

because $\widehat{\mathsf{A}}_{(k)}$ is a best rank-k approximation to the matrix $\mathsf{P}_{\mathsf{Z}}\mathsf{A}$, whereas $\mathsf{P}_{\mathsf{Z}}\mathsf{A}_{(k)}$ is an undistinguished rank-k matrix. It follows that

$$\left\|\mathsf{P}_{\mathsf{Z}}\mathsf{A} - \widehat{\mathsf{A}}_{(k)}\right\| \le \left\|\mathsf{P}_{\mathsf{Z}}(\mathsf{A} - \mathsf{A}_{(k)})\right\| \le \left\|\mathsf{A} - \mathsf{A}_{(k)}\right\| = \sigma_{k+1}. \tag{2.65}$$

The second inequality holds because the orthogonal projector is a contraction; the last identity follows from Mirsky's theorem [97]. Combine (2.64) and (2.65) to reach the main result. □

Remark 20. In the randomized setting, the truncation step appears to be less damaging than the error bound of Theorem 2.9.3 suggests, but we currently lack a complete theoretical understanding of its behavior.

2.10 Gaussian test matrices

The error bound in Theorem 2.9.1 shows that the performance of the proto-algorithm depends on the interaction between the test matrix Ω and the right singular vectors of the input matrix A. Algorithm 4.1 is a particularly simple version of the proto-algorithm that draws the test matrix according to the standard Gaussian distribution. The literature contains a wealth of information about these matrices, which allows us to perform a very precise error analysis.

We focus on the real case in this section. Analogous results hold in the complex case, where the algorithm even exhibits superior performance.

2.10.1 Technical background

A *standard Gaussian matrix* is a random matrix whose entries are independent standard normal variables. The distribution of a standard Gaussian matrix is rotationally invariant: If U and V are orthonormal matrices, then $\mathsf{U}^*\mathsf{G}\mathsf{V}$ also has the standard Gaussian distribution.

Our analysis requires detailed information about the properties of Gaussian matrices. In particular, we must understand how the norm of a Gaussian matrix and its pseudoinverse vary. We summarize the relevant results and citations here, reserving the details for Appendix 2.12.

Proposition 2.10.1 (Expected norm of a scaled Gaussian matrix)**.** *Fix matrices* S, T*, and draw a standard Gaussian matrix* G*. Then*

$$\left(\mathbb{E}\left\|\mathsf{S}\mathsf{G}\mathsf{T}\right\|_{\mathrm{F}}^2\right)^{1/2} = \left\|\mathsf{S}\right\|_{\mathrm{F}}\left\|\mathsf{T}\right\|_{\mathrm{F}} \quad \textit{and} \tag{2.66}$$

$$\mathbb{E}\left\|\mathsf{S}\mathsf{G}\mathsf{T}\right\| \le \left\|\mathsf{S}\right\|\left\|\mathsf{T}\right\|_{\mathrm{F}} + \left\|\mathsf{S}\right\|_{\mathrm{F}}\left\|\mathsf{T}\right\|. \tag{2.67}$$

The identity (2.66) follows from a direct calculation. The second bound (2.67) relies on methods developed by Gordon [62, 63]. See Propositions 2.12.1 and 2.12.2.

Proposition 2.10.2 (Expected norm of a pseudo-inverted Gaussian matrix)**.** *Draw a $k \times (k+p)$ standard Gaussian matrix G with $k \geq 2$ and $p \geq 2$. Then*

$$\left(\mathbb{E}\left\|\mathsf{G}^{\dagger}\right\|_{\mathrm{F}}^{2}\right)^{1/2} = \sqrt{\frac{k}{p-1}} \quad \text{and} \tag{2.68}$$

$$\mathbb{E}\left\|\mathsf{G}^{\dagger}\right\| \leq \frac{\mathrm{e}\sqrt{k+p}}{p}. \tag{2.69}$$

The first identity is a standard result from multivariate statistics [99, p. 96]. The second follows from work of Chen and Dongarra [25]. See Proposition 2.12.4 and 2.12.5.

To study the probability that Algorithm 4.1 produces a large error, we rely on tail bounds for functions of Gaussian matrices. The next proposition rephrases a well-known result on concentration of measure [14, Thm. 4.5.7]. See also [83, §1.1] and [82, §5.1].

Proposition 2.10.3 (Concentration for functions of a Gaussian matrix)**.** *Suppose that h is a Lipschitz function on matrices:*

$$|h(\mathsf{X}) - h(\mathsf{Y})| \leq L\,\|\mathsf{X} - \mathsf{Y}\|_{\mathrm{F}} \quad \text{for all } \mathsf{X}, \mathsf{Y}.$$

Draw a standard Gaussian matrix G. Then

$$\mathbb{P}\left\{h(\mathsf{G}) \geq \mathbb{E}\,h(\mathsf{G}) + Lt\right\} \leq \mathrm{e}^{-t^2/2}.$$

Finally, we state some large deviation bounds for the norm of a pseudo-inverted Gaussian matrix.

Proposition 2.10.4 (Norm bounds for a pseudo-inverted Gaussian matrix)**.** *Let G be a $k \times (k+p)$ Gaussian matrix where $p \geq 4$. For all $t \geq 1$,*

$$\mathbb{P}\left\{\left\|\mathsf{G}^{\dagger}\right\|_{\mathrm{F}} \geq \sqrt{\frac{12k}{p}} \cdot t\right\} \leq 4t^{-p} \quad \text{and} \tag{2.70}$$

$$\mathbb{P}\left\{\left\|\mathsf{G}^{\dagger}\right\| \geq \frac{\mathrm{e}\sqrt{k+p}}{p+1} \cdot t\right\} \leq t^{-(p+1)}. \tag{2.71}$$

Compare these estimates with Proposition 2.10.2. It seems that (2.70) is new; we were unable to find a comparable analysis in the random matrix literature. Although the form of (2.70) is not optimal, it allows us to produce more transparent results than a fully detailed estimate. The bound (2.71) essentially appears in the work of Chen and Dongarra [25]. See Propositions 2.12.3 and Theorem 2.12.6 for more information.

2.10.2 Average-case analysis of Algorithm 4.1

We separate our analysis into two pieces. First, we present information about expected values. In the next subsection, we describe bounds on the probability of a large deviation.

We begin with the simplest result, which provides an estimate for the expected approximation error in the Frobenius norm. All proofs are postponed to the end of the section.

Theorem 2.10.5 (Average Frobenius error)**.** *Suppose that* A *is a* ***real*** $m \times n$ *matrix with singular values* $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \dots$. *Choose a target rank* $k \geq 2$ *and an oversampling parameter* $p \geq 2$, *where* $k + p \leq \min\{m, n\}$. *Draw an* $n \times (k + p)$ *standard Gaussian matrix* Ω, *and construct the sample matrix* $\mathsf{Y} = \mathsf{A}\Omega$. *Then the expected approximation error*

$$\mathbb{E}\,\|(\mathsf{I} - \mathsf{P}_{\mathsf{Y}})\mathsf{A}\|_{\mathrm{F}} \leq \left(1 + \frac{k}{p-1}\right)^{1/2} \left(\sum\nolimits_{j>k} \sigma_j^2\right)^{1/2}.$$

This theorem predicts several intriguing behaviors of Algorithm 4.1. The Eckart–Young theorem [54] shows that $(\sum_{j>k} \sigma_j^2)^{1/2}$ is the minimal Frobenius-norm error when approximating A with a rank-k matrix. This quantity is the appropriate benchmark for the performance of the algorithm. If the small singular values of A are very flat, the series may be as large as $\sigma_{k+1}\sqrt{\min\{m, n\} - k}$. On the other hand, when the singular values exhibit some decay, the error may be on the same order as σ_{k+1}.

The error bound always exceeds this baseline error, but it may be polynomially larger, depending on the ratio between the target rank k and the oversampling parameter p. For p small (say, less than five), the error is somewhat variable because the small singular values of a nearly square Gaussian matrix are very unstable. As the oversampling increases, the performance improves quickly. When $p \sim k$, the error is already within a constant factor of the baseline.

The error bound for the spectral norm is somewhat more complicated, but it reveals some interesting new features.

Theorem 2.10.6 (Average spectral error)**.** *Under the hypotheses of Theorem 2.10.5,*

$$\mathbb{E}\,\|(\mathsf{I} - \mathsf{P}_{\mathsf{Y}})\mathsf{A}\| \leq \left(1 + \sqrt{\frac{k}{p-1}}\right) \sigma_{k+1} + \frac{\mathrm{e}\sqrt{k+p}}{p} \left(\sum\nolimits_{j>k} \sigma_j^2\right)^{1/2}.$$

Mirsky [97] has shown that the quantity σ_{k+1} is the minimum spectral-norm error when approximating A with a rank-k matrix, so the first term in Theorem 2.10.6 is analogous with the error bound in Theorem 2.10.5. The second term represents a new phenomenon: we also pay for the Frobenius-norm error in approximating A. Note that, as the amount p of oversampling increases, the polynomial factor in the second term declines much more quickly than the factor in the first term. When $p \sim k$, the factor on the σ_{k+1} term is constant, while the factor on the series has order $k^{-1/2}$

We also note that the bound in Theorem 2.10.6 implies

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_{\mathsf{Y}})\mathsf{A}\| \le \left[1+\sqrt{\frac{k}{p-1}}+\frac{\mathrm{e}\sqrt{k+p}}{p}\cdot\sqrt{\min\{m,n\}-k}\right]\sigma_{k+1},$$

so the average spectral-norm error always lies within a small polynomial factor of the baseline σ_{k+1}.

Let us continue with the proofs of these results.

Theorem 2.10.5. Let V be the right unitary factor of A. Partition $\mathsf{V} = [\mathsf{V}_1 \mid \mathsf{V}_2]$ into blocks containing, respectively, k and $n-k$ columns. Recall that

$$\Omega_1 = \mathsf{V}_1^*\Omega \quad \text{and} \quad \Omega_2 = \mathsf{V}_2^*\Omega.$$

The Gaussian distribution is rotationally invariant, so $\mathsf{V}^*\Omega$ is also a standard Gaussian matrix. Observe that Ω_1 and Ω_2 are **nonoverlapping** submatrices of $\mathsf{V}^*\Omega$, so these two matrices are not only standard Gaussian but also stochastically independent. Furthermore, the rows of a (fat) Gaussian matrix are almost surely in general position, so the $k \times (k+p)$ matrix Ω_1 has full row rank with probability one.

Hölder's inequality and Theorem 2.9.1 together imply that

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_{\mathsf{Y}})\mathsf{A}\|_{\mathrm{F}} \le \left(\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_{\mathsf{Y}})\mathsf{A}\|_{\mathrm{F}}^2\right)^{1/2} \le \left(\left\|\Sigma_2\right\|_{\mathrm{F}}^2 + \mathbb{E}\left\|\Sigma_2\Omega_2\Omega_1^{\dagger}\right\|_{\mathrm{F}}^2\right)^{1/2}.$$

We compute this expectation by conditioning on the value of Ω_1 and applying Proposition 2.10.1 to the scaled Gaussian matrix Ω_2. Thus,

$$\begin{aligned}\mathbb{E}\left\|\Sigma_2\Omega_2\Omega_1^{\dagger}\right\|_{\mathrm{F}}^2 = \mathbb{E}\left(\mathbb{E}\left[\left\|\Sigma_2\Omega_2\Omega_1^{\dagger}\right\|_{\mathrm{F}}^2 \mid \Omega_1\right]\right) &= \mathbb{E}\left(\|\Sigma_2\|_{\mathrm{F}}^2\left\|\Omega_1^{\dagger}\right\|_{\mathrm{F}}^2\right) \\ &= \|\Sigma_2\|_{\mathrm{F}}^2\cdot\mathbb{E}\left\|\Omega_1^{\dagger}\right\|_{\mathrm{F}}^2 = \frac{k}{p-1}\cdot\|\Sigma_2\|_{\mathrm{F}}^2,\end{aligned}$$

where the last expectation follows from relation (2.68) of Proposition 2.10.2. In summary,

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_{\mathsf{Y}})\mathsf{A}\|_{\mathrm{F}} \le \left(1+\frac{k}{p-1}\right)^{1/2}\|\Sigma_2\|_{\mathrm{F}}.$$

Observe that $\|\Sigma_2\|_{\mathrm{F}}^2 = \sum_{j>k}\sigma_j^2$ to complete the proof. □

Theorem 2.10.6. The argument is similar to the proof of Theorem 2.10.5. First, Theorem 2.9.1 implies that

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_{\mathsf{Y}})\mathsf{A}\| \le \mathbb{E}\left(\|\Sigma_2\|^2 + \left\|\Sigma_2\Omega_2\Omega_1^{\dagger}\right\|^2\right)^{1/2} \le \|\Sigma_2\| + \mathbb{E}\left\|\Sigma_2\Omega_2\Omega_1^{\dagger}\right\|.$$

We condition on Ω_1 and apply Proposition 2.10.1 to bound the expectation with respect to Ω_2. Thus,

$$\begin{aligned}\mathbb{E}\left\|\Sigma_2\Omega_2\Omega_1^\dagger\right\| &\le \mathbb{E}\left(\|\Sigma_2\|\left\|\Omega_1^\dagger\right\|_{\mathrm{F}} + \|\Sigma_2\|_{\mathrm{F}}\left\|\Omega_1^\dagger\right\|\right)\\ &\le \|\Sigma_2\|\left(\mathbb{E}\left\|\Omega_1^\dagger\right\|_{\mathrm{F}}^2\right)^{1/2} + \|\Sigma_2\|_{\mathrm{F}}\cdot\mathbb{E}\left\|\Omega_1^\dagger\right\|.\end{aligned}$$

where the second relation requires Hölder's inequality. Applying both parts of Proposition 2.10.2, we obtain

$$\mathbb{E}\left\|\Sigma_2\Omega_2\Omega_1^\dagger\right\| \le \sqrt{\frac{k}{p-1}}\,\|\Sigma_2\| + \frac{\mathrm{e}\sqrt{k+p}}{p}\,\|\Sigma_2\|_{\mathrm{F}}\,.$$

Note that $\|\Sigma_2\| = \sigma_{k+1}$ to wrap up. □

2.10.3 Probabilistic error bounds for Algorithm 4.1

We can develop tail bounds for the approximation error, which demonstrate that the average performance of the algorithm is representative of the actual performance. We begin with the Frobenius norm because the result is somewhat simpler.

Theorem 2.10.7 (Deviation bounds for the Frobenius error)**.** *Frame the hypotheses of Theorem 2.10.5. Assume further that $p \ge 4$. For all $u, t \ge 1$,*

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\|_{\mathrm{F}} \le \left(1 + t\cdot\sqrt{12k/p}\right)\left(\sum\nolimits_{j>k}\sigma_j^2\right)^{1/2} + ut\cdot\frac{\mathrm{e}\sqrt{k+p}}{p+1}\cdot\sigma_{k+1},$$

with failure probability at most $5t^{-p} + 2\mathrm{e}^{-u^2/2}$.

To parse this theorem, observe that the first term in the error bound corresponds with the expected approximation error in Theorem 2.10.5. The second term represents a deviation above the mean.

An analogous result holds for the spectral norm.

Theorem 2.10.8 (Deviation bounds for the spectral error)**.** *Frame the hypotheses of Theorem 2.10.5. Assume further that $p \ge 4$. For all $u, t \ge 1$,*

$$\begin{aligned}&\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\|\\ &\qquad\le \left[\left(1 + t\cdot\sqrt{12k/p}\right)\sigma_{k+1} + t\cdot\frac{\mathrm{e}\sqrt{k+p}}{p+1}\left(\sum\nolimits_{j>k}\sigma_j^2\right)^{1/2}\right] + ut\cdot\frac{\mathrm{e}\sqrt{k+p}}{p+1}\sigma_{k+1},\end{aligned}$$

with failure probability at most $5t^{-p} + \mathrm{e}^{-u^2/2}$.

The bracket corresponds with the expected spectral-norm error while the remaining term represents a deviation above the mean. Neither the numerical constants nor the precise form of the bound are optimal because of the slackness in Proposition 2.10.4. Nevertheless, the theorem gives a fairly good picture of what is actually happening.

We acknowledge that the current form of Theorem 2.10.8 is complicated. To produce more transparent results, we make appropriate selections for the parameters u, t and bound the numerical constants.

Corollary 2.10.9 (Simplified deviation bounds for the spectral error)**.** *Frame the hypotheses of Theorem 2.10.5, and assume further that $p \geq 4$. Then*

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\| \leq \left(1 + 17\sqrt{1 + k/p}\right) \sigma_{k+1} + \frac{8\sqrt{k+p}}{p+1} \left(\sum\nolimits_{j>k} \sigma_j^2\right)^{1/2},$$

with failure probability at most $6\mathrm{e}^{-p}$. Moreover,

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\| \leq \left(1 + 8\sqrt{(k+p) \cdot p \log p}\right) \sigma_{k+1} + 3\sqrt{k+p} \left(\sum\nolimits_{j>k} \sigma_j^2\right)^{1/2},$$

with failure probability at most $6p^{-p}$.

Proof. The first part of the result follows from the choices $t = \mathrm{e}$ and $u = \sqrt{2p}$, and the second emerges when $t = p$ and $u = \sqrt{2p \log p}$. Another interesting parameter selection is $t = p^{c/p}$ and $u = \sqrt{2c \log p}$, which yields a failure probability $6p^{-c}$. □

Corollary 2.10.9 should be compared with [91, Obs. 4.4–4.5]. Although our result contains sharper error estimates, the failure probabilities are usually worse. The error bound (2.9) presented in §2.1.5 follows after further simplification of the second bound from Corollary 2.10.9.

We continue with a proof of Theorem 2.10.8. The same argument can be used to obtain a bound for the Frobenius-norm error, but we omit a detailed account.

Theorem 2.10.8. Since Ω_1 and Ω_2 are independent from each other, we can study how the error depends on the matrix Ω_2 by conditioning on the event that Ω_1 is not too irregular. To that end, we define a (parameterized) event on which the spectral and Frobenius norms of the matrix $\Omega_1^\dagger$ are both controlled. For $t \geq 1$, let

$$E_t = \left\{ \Omega_1 : \left\|\Omega_1^\dagger\right\| \leq \frac{\mathrm{e}\sqrt{k+p}}{p+1} \cdot t \quad \text{and} \quad \left\|\Omega_1^\dagger\right\|_\mathrm{F} \leq \sqrt{\frac{12k}{p}} \cdot t \right\}.$$

Invoking both parts of Proposition 2.10.4, we find that

$$\mathbb{P}\left(E_t^c\right) \leq t^{-(p+1)} + 4t^{-p} \leq 5t^{-p}.$$

Consider the function $h(\mathsf{X}) = \left\|\mathsf{\Sigma}_2 \mathsf{X} \mathsf{\Omega}_1^\dagger\right\|$. We quickly compute its Lipschitz constant L with the lower triangle inequality and some standard norm estimates:

$$\begin{aligned} |h(\mathsf{X}) - h(\mathsf{Y})| &\le \left\|\mathsf{\Sigma}_2(\mathsf{X} - \mathsf{Y})\mathsf{\Omega}_1^\dagger\right\| \\ &\le \|\mathsf{\Sigma}_2\| \, \|\mathsf{X} - \mathsf{Y}\| \, \left\|\mathsf{\Omega}_1^\dagger\right\| \le \|\mathsf{\Sigma}_2\| \, \left\|\mathsf{\Omega}_1^\dagger\right\| \, \|\mathsf{X} - \mathsf{Y}\|_{\mathrm{F}} \,. \end{aligned}$$

Therefore, $L \le \|\mathsf{\Sigma}_2\| \left\|\mathsf{\Omega}_1^\dagger\right\|$. Relation (2.67) of Proposition 2.10.1 implies that

$$\mathbb{E}[h(\mathsf{\Omega}_2) \mid \mathsf{\Omega}_1] \le \|\mathsf{\Sigma}_2\| \left\|\mathsf{\Omega}_1^\dagger\right\|_{\mathrm{F}} + \|\mathsf{\Sigma}_2\|_{\mathrm{F}} \left\|\mathsf{\Omega}_1^\dagger\right\|.$$

Applying the concentration of measure inequality, Proposition 2.10.3, conditionally to the random variable $h(\mathsf{\Omega}_2) = \left\|\mathsf{\Sigma}_2 \mathsf{\Omega}_2 \mathsf{\Omega}_1^\dagger\right\|$ results in

$$\mathbb{P}\left\{ \left\|\mathsf{\Sigma}_2 \mathsf{\Omega}_2 \mathsf{\Omega}_1^\dagger\right\| > \|\mathsf{\Sigma}_2\| \left\|\mathsf{\Omega}_1^\dagger\right\|_{\mathrm{F}} + \|\mathsf{\Sigma}_2\|_{\mathrm{F}} \left\|\mathsf{\Omega}_1^\dagger\right\| + \|\mathsf{\Sigma}_2\| \left\|\mathsf{\Omega}_1^\dagger\right\| \cdot u \;\middle|\; E_t \right\} \le \mathrm{e}^{-u^2/2}.$$

Under the event E_t, we have explicit bounds on the norms of $\mathsf{\Omega}_1^\dagger$, so

$$\mathbb{P}\left\{ \left\|\mathsf{\Sigma}_2 \mathsf{\Omega}_2 \mathsf{\Omega}_1^\dagger\right\| > \|\mathsf{\Sigma}_2\| \sqrt{\frac{12k}{p}} \cdot t + \|\mathsf{\Sigma}_2\|_{\mathrm{F}} \frac{\mathrm{e}\sqrt{k+p}}{p+1} \cdot t + \|\mathsf{\Sigma}_2\| \frac{\mathrm{e}\sqrt{k+p}}{p+1} \cdot ut \;\middle|\; E_t \right\} \le \mathrm{e}^{-u^2/2}.$$

Use the fact $\mathbb{P}\,(E_t^c) \le 5t^{-p}$ to remove the conditioning. Therefore,

$$\mathbb{P}\left\{ \left\|\mathsf{\Sigma}_2 \mathsf{\Omega}_2 \mathsf{\Omega}_1^\dagger\right\| > \|\mathsf{\Sigma}_2\| \sqrt{\frac{12k}{p}} \cdot t + \|\mathsf{\Sigma}_2\|_{\mathrm{F}} \frac{\mathrm{e}\sqrt{k+p}}{p+1} \cdot t + \|\mathsf{\Sigma}_2\| \frac{\mathrm{e}\sqrt{k+p}}{p+1} \cdot ut \right\} \le 5t^{-p} + \mathrm{e}^{-u^2/2}.$$

Insert the expressions for the norms of $\mathsf{\Sigma}_2$ into this result to complete the probability bound. Finally, introduce this estimate into the error bound from Theorem 2.9.1. □

2.10.4 Analysis of the power scheme

Theorem 2.10.6 makes it clear that the performance of the randomized approximation scheme, Algorithm 4.1, depends heavily on the singular spectrum of the input matrix. The power scheme outlined in Algorithm 4.3 addresses this problem by enhancing the decay of spectrum. We can combine our analysis of Algorithm 4.1 with Theorem 2.9.2 to obtain a detailed report on the behavior of the performance of the power scheme using a Gaussian matrix.

Corollary 2.10.10 (Average spectral error for the power scheme)**.** *Frame the hypotheses of Theorem 2.10.5. Define* $\mathsf{B} = (\mathsf{A}\mathsf{A}^*)^q \mathsf{A}$ *for a nonnegative integer* q*, and construct the sample matrix* $\mathsf{Z} = \mathsf{B}\mathsf{\Omega}$*. Then*

$$\mathbb{E}\,\|(\mathsf{I} - \mathsf{P}_{\mathsf{Z}})\mathsf{A}\| \le \left[\left(1 + \sqrt{\frac{k}{p-1}} \right) \sigma_{k+1}^{2q+1} + \frac{\mathrm{e}\sqrt{k+p}}{p} \left(\sum\nolimits_{j>k} \sigma_j^{2(2q+1)} \right)^{1/2} \right]^{1/(2q+1)}.$$

Proof. By Hölder's inequality and Theorem 2.9.2,

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_\mathsf{Z})\mathsf{A}\| \le \left(\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_\mathsf{Z})\mathsf{A}\|^{2q+1}\right)^{1/(2q+1)} \le \left(\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_\mathsf{Z})\mathsf{B}\|\right)^{1/(2q+1)}.$$

Invoke Theorem 2.10.6 to bound the right-hand side, noting that $\sigma_j(\mathsf{B}) = \sigma_j^{2q+1}$. □

The true message of Corollary 2.10.10 emerges if we bound the series using its largest term σ_{k+1}^{4q+2} and draw the factor σ_{k+1} out of the bracket:

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_\mathsf{Z})\mathsf{A}\| \le \left[1 + \sqrt{\frac{k}{p-1}} + \frac{\mathrm{e}\sqrt{k+p}}{p}\cdot\sqrt{\min\{m,n\}-k}\right]^{1/(2q+1)}\sigma_{k+1}.$$

In words, as we increase the exponent q, the power scheme drives the extra factor in the error to one exponentially fast. By the time $q \sim \log(\min\{m,n\})$,

$$\mathbb{E}\,\|(\mathsf{I}-\mathsf{P}_\mathsf{Z})\mathsf{A}\| \sim \sigma_{k+1},$$

which is the baseline for the spectral norm.

In most situations, the error bound given by Corollary 2.10.10 is substantially better than the estimates discussed in the last paragraph. For example, suppose that the tail singular values exhibit the decay profile

$$\sigma_j \lesssim j^{(1+\varepsilon)/(4q+2)} \quad \text{for } j > k \text{ and } \varepsilon > 0.$$

Then the series in Corollary 2.10.10 is comparable with its largest term, which allows us to remove the dimensional factor $\min\{m,n\}$ from the error bound.

To obtain large deviation bounds for the performance of the power scheme, simply combine Theorem 2.9.2 with Theorem 2.10.8. We omit a detailed statement.

Remark 21. We lack an analogous theory for the Frobenius norm because Theorem 2.9.2 depends on Proposition 2.8.6, which is not true for the Frobenius norm. It is possible to obtain some results by estimating the Frobenius norm in terms of the spectral norm.

2.11 SRFT test matrices

Another way to implement the proto-algorithm from §2.1.3 is to use a structured random matrix so that the matrix product in Step 2 can be performed quickly. One type of structured random matrix that has been proposed in the literature is the *subsampled random Fourier transform*, or SRFT, which we discussed in §2.4.6. In this section, we present bounds on the performance of the proto-algorithm when it is implemented with an SRFT test matrix. In contrast with the results for Gaussian test matrices, the results in this section hold for both real and complex input matrices.

2.11.1 Construction and Properties

Recall from §2.4.6 that an SRFT is a tall $n \times \ell$ matrix of the form $\Omega = \sqrt{n/\ell} \cdot \mathsf{DFR}^*$ where

- D is a random $n \times n$ diagonal matrix whose entries are independent and uniformly distributed on the complex unit circle;
- F is the $n \times n$ unitary discrete Fourier transform; and
- R is a random $\ell \times n$ matrix that restricts an n-dimensional vector to ℓ coordinates, chosen uniformly at random.

Up to scaling, an SRFT is just a section of a unitary matrix, so it satisfies the norm identity $\|\Omega\| = \sqrt{n/\ell}$. The critical fact is that an appropriately designed SRFT approximately preserves the geometry of an **entire subspace of vectors**.

Theorem 2.11.1 (The SRFT preserves geometry)**.** *Fix an $n \times k$ orthonormal matrix V, and draw an $n \times \ell$ SRFT matrix Ω where the parameter ℓ satisfies*

$$4\left[\sqrt{k} + \sqrt{8\log(kn)}\right]^2 \log(k) \le \ell \le n.$$

Then

$$0.40 \le \sigma_k(\mathsf{V}^*\Omega) \quad \textit{and} \quad \sigma_1(\mathsf{V}^*\Omega) \le 1.48$$

with failure probability at most $\mathrm{O}(k^{-1})$.

In words, the kernel of an SRFT of dimension $\ell \sim k\log(k)$ is unlikely to intersect a fixed k-dimensional subspace. In contrast with the Gaussian case, the logarithmic factor $\log(k)$ in the lower bound on ℓ cannot generally be removed (Remark 23).

Theorem 2.11.1 follows from a straightforward variation of the argument in [134], which establishes equivalent bounds for a real analog of the SRFT, called the *subsampled randomized Hadamard transform* (SRHT). We omit further details.

Remark 22. For large problems, we can obtain better numerical constants [134, Thm. 3.2]. Fix a small, positive number ι. If $k \gg \log(n)$, then sampling

$$\ell \ge (1 + \iota) \cdot k\log(k)$$

coordinates is sufficient to ensure that $\sigma_k(\mathsf{V}^*\Omega) \ge \iota$ with failure probability at most $\mathrm{O}(k^{-\mathrm{c}\iota})$. This sampling bound is essentially optimal because $(1 - \iota) \cdot k\log(k)$ samples are not adequate in the worst case; see Remark 23.

Remark 23. The logarithmic factor in Theorem 2.11.1 is **necessary** when the orthonormal matrix V is particularly evil. Let us describe an infinite family of worst-case examples. Fix an integer k, and let $n = k^2$. Form an $n \times k$ orthonormal matrix V by regular decimation of the $n \times n$ identity matrix. More precisely, V is the matrix whose jth row has a unit entry in column $(j-1)/k$ when $j \equiv 1 \pmod{k}$ and is zero otherwise. To see why this type of matrix is nasty, it is helpful to consider the auxiliary matrix $\mathsf{W} = \mathsf{V}^*\mathsf{DF}$. Observe that, up to scaling and modulation of columns, W consists of k copies of a $k \times k$ DFT concatenated horizontally.

Suppose that we apply the SRFT $\Omega = \mathsf{DFR}^*$ to the matrix V^*. We obtain a matrix of the form $\mathsf{X} = \mathsf{V}^*\Omega = \mathsf{WR}^*$, which consists of ℓ random columns sampled from W. Theorem 2.11.1 certainly cannot hold unless $\sigma_k(\mathsf{X}) > 0$. To ensure the latter event occurs, we must pick at least one copy each of the k distinct columns of W. This is the coupon collector's problem [98, Sec. 3.6] in disguise. To obtain a complete set of k coupons (i.e., columns) with nonnegligible probability, we must draw at least $k\log(k)$ columns. The fact that we are sampling without replacement does not improve the analysis appreciably because the matrix has too many columns.

2.11.2 Performance guarantees

We are now prepared to present detailed information on the performance of the proto-algorithm when the test matrix Ω is an SRFT.

Theorem 2.11.2 (Error bounds for SRFT)**.** *Fix an $m \times n$ matrix A with singular values $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \dots$. Draw an $n \times \ell$ SRFT matrix Ω, where*

$$4\left[\sqrt{k} + \sqrt{8\log(kn)}\right]^2 \log(k) \leq \ell \leq n.$$

Construct the sample matrix $\mathsf{Y} = \mathsf{A}\Omega$. Then

$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\| \leq \sqrt{1 + 7n/\ell} \cdot \sigma_{k+1} \quad \text{and}$$
$$\|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}\|_\mathrm{F} \leq \sqrt{1 + 7n/\ell} \cdot \left(\sum\nolimits_{j>k} \sigma_j^2\right)^{1/2}$$

with failure probability at most $\mathrm{O}(k^{-1})$.

As we saw in §2.10.2, the quantity σ_{k+1} is the minimal spectral-norm error possible when approximating A with a rank-k matrix. Similarly, the series in the second bound is the minimal Frobenius-norm error when approximating A with a rank-k matrix. We see that both error bounds lie within a polynomial factor of the baseline, and this factor decreases with the number ℓ of samples we retain.

The likelihood of error with an SRFT test matrix is substantially worse than in the Gaussian case. The failure probability here is roughly k^{-1}, while in the Gaussian case, the failure probability is roughly $\mathrm{e}^{-(\ell-k)}$. This qualitative difference is not an artifact of the analysis; discrete sampling techniques inherently fail with higher probability.

Matrix approximation schemes based on SRFTs often perform much better in practice than the error analysis here would indicate. While it is not generally possible to guarantee accuracy with a sampling parameter less than $\ell \sim k \log(k)$, we have found empirically that the choice $\ell = k + 20$ is adequate in almost all applications. Indeed, SRFTs sometimes perform even *better* than Gaussian matrices (see, e.g., Figure 2.8).

We complete the section with the proof of Theorem 2.11.2.

Theorem 2.11.2. Let V be the right unitary factor of matrix A, and partition $\mathsf{V} = [\mathsf{V}_1 \mid \mathsf{V}_2]$ into blocks containing, respectively, k and $n - k$ columns. Recall that

$$\Omega_1 = \mathsf{V}_1^* \Omega \quad \text{and} \quad \Omega_2 = \mathsf{V}_2^* \Omega.$$

where Ω is the conjugate transpose of an SRFT. Theorem 2.11.1 ensures that the submatrix Ω_1 has full row rank, with failure probability at most $\mathrm{O}(k^{-1})$. Therefore, Theorem 2.9.1 implies that

$$\|\!|(\mathsf{I} - \mathsf{P}_\mathsf{Y})\mathsf{A}|\!\| \leq \|\!|\Sigma_2|\!\| \left[1 + \left\|\Omega_1^\dagger\right\|^2 \cdot \|\Omega_2\|^2\right]^{1/2},$$

where $\|\!|\cdot|\!\|$ denotes either the spectral norm or the Frobenius norm. Our application of Theorem 2.11.1 also ensures that the spectral norm of $\Omega_1^\dagger$ is under control.

$$\left\|\Omega_1^\dagger\right\|^2 \leq \frac{1}{0.40^2} < 7.$$

We may bound the spectral norm of Ω_2 deterministically.

$$\|\Omega_2\| = \|\mathsf{V}_2^* \Omega\| \leq \|\mathsf{V}_2^*\| \, \|\Omega\| = \sqrt{n/\ell}$$

since V_2 and $\sqrt{\ell/n} \cdot \Omega$ are both orthonormal matrices. Combine these estimates to complete the proof. □

Acknowledgments

The authors have benefited from valuable discussions with many researchers, among them Inderjit Dhillon, Petros Drineas, Ming Gu, Edo Liberty, Michael Mahoney, Vladimir Rokhlin, Yoel Shkolnisky, and Arthur Szlam. In particular, we would like to thank Mark Tygert for his insightful remarks on early drafts of this paper. The example in Section 2.7.2 was provided by François Meyer of the University of Colorado at Boulder. The example in Section 2.7.3 comes from the FERET database of facial images collected under the FERET program, sponsored by the DoD Counterdrug Technology Development Program Office. The work reported was initiated during the program *Mathematics of Knowledge and Search Engines* held at IPAM in the fall of 2007. Finally, we would like to thank the anonymous referees, whose thoughtful remarks have helped us to improve the manuscript dramatically.

Appendix

2.12 On Gaussian matrices

This appendix collects some of the properties of Gaussian matrices that we use in our analysis. Most of the results follow quickly from material that is already available in the literature. One fact, however, requires a surprisingly difficult new argument. We focus on the real case here; the complex case is similar but actually yields better results.

2.12.1 Expectation of norms

We begin with the expected Frobenius norm of a scaled Gaussian matrix, which follows from an easy calculation.

Proposition 2.12.1. *Fix real matrices* S, T*, and draw a standard Gaussian matrix* G*. Then*

$$\left(\mathbb{E}\,\|\mathsf{SGT}\|_{\mathrm{F}}^2\right)^{1/2} = \|\mathsf{S}\|_{\mathrm{F}}\,\|\mathsf{T}\|_{\mathrm{F}}\,.$$

Proof. The distribution of a Gaussian matrix is invariant under orthogonal transformations, and the Frobenius norm is also unitarily invariant. As a result, it represents no loss of generality to assume that S and T are diagonal. Therefore,

$$\mathbb{E}\,\|\mathsf{SGT}\|_{\mathrm{F}}^2 = \mathbb{E}\left[\sum\nolimits_{jk} |s_{jj} g_{jk} t_{kk}|^2\right] = \sum\nolimits_{jk} |s_{jj}|^2 |t_{kk}|^2 = \|\mathsf{S}\|_{\mathrm{F}}^2\,\|\mathsf{T}\|_{\mathrm{F}}^2\,.$$

Since the right-hand side is unitarily invariant, we have also identified the value of the expectation for general matrices S and T. □

The literature contains an excellent bound for the expected spectral norm of a scaled Gaussian matrix. The result is due to Gordon [62, 63], who established the bound using a sharp version of Slepian's lemma. See [83, §3.3] and [34, §2.3] for additional discussion.

Proposition 2.12.2. *Fix real matrices* S, T*, and draw a standard Gaussian matrix* G*. Then*

$$\mathbb{E}\,\|\mathsf{SGT}\| \le \|\mathsf{S}\|\,\|\mathsf{T}\|_{\mathrm{F}} + \|\mathsf{S}\|_{\mathrm{F}}\,\|\mathsf{T}\|\,.$$

2.12.2 Spectral norm of pseudoinverse

Now, we turn to the pseudoinverse of a Gaussian matrix. Recently, Chen and Dongarra developed a good bound on the probability that its spectral norm is large. The statement here follows from [25, Lem. 4.1] after an application of Stirling's approximation. See also [91, Lem. 2.14]

Proposition 2.12.3. *Let* G *be an* $m \times n$ *standard Gaussian matrix with* $n \geq m \geq 2$*. For each* $t > 0$*,*

$$\mathbb{P}\left\{\left\|\mathsf{G}^{\dagger}\right\| > t\right\} \leq \frac{1}{\sqrt{2\pi(n-m+1)}}\left[\frac{\mathrm{e}\sqrt{n}}{n-m+1}\right]^{n-m+1} t^{-(n-m+1)}.$$

We can use Proposition 2.12.3 to bound the expected spectral norm of a pseudo-inverted Gaussian matrix.

Proposition 2.12.4. *Let* G *be a* $m \times n$ *standard Gaussian matrix with* $n - m \geq 1$ *and* $m \geq 2$*. Then*

$$\mathbb{E}\left\|\mathsf{G}^{\dagger}\right\| < \frac{\mathrm{e}\sqrt{n}}{n-m}$$

Proof. Let us make the abbreviations $p = n - m$ and

$$C = \frac{1}{\sqrt{2\pi(p+1)}}\left[\frac{\mathrm{e}\sqrt{n}}{p+1}\right]^{p+1}.$$

We compute the expectation by way of a standard argument. The integral formula for the mean of a nonnegative random variable implies that, for all $E > 0$,

$$\begin{aligned}\mathbb{E}\left\|\mathsf{G}^{\dagger}\right\| = \int_0^{\infty} \mathbb{P}\left\{\left\|\mathsf{G}^{\dagger}\right\| > t\right\} \mathrm{d}t &\leq E + \int_E^{\infty} \mathbb{P}\left\{\left\|\mathsf{G}^{\dagger}\right\| > t\right\} \mathrm{d}t \\ &\leq E + C\int_E^{\infty} t^{-(p+1)}\,\mathrm{d}t = E + \frac{1}{p}CE^{-p},\end{aligned}$$

where the second inequality follows from Proposition 2.12.3. The right-hand side is minimized when $E = C^{1/(p+1)}$. Substitute and simplify. □

2.12.3 Frobenius norm of pseudoinverse

The squared Frobenius norm of a pseudo-inverted Gaussian matrix is closely connected with the trace of an inverted Wishart matrix. This observation leads to an exact expression for the expectation.

Proposition 2.12.5. *Let* G *be an* $m \times n$ *standard Gaussian matrix with* $n - m \geq 2$*. Then*

$$\mathbb{E}\left\|\mathsf{G}^{\dagger}\right\|_{\mathrm{F}}^2 = \frac{m}{n-m-1}.$$

Proof. Observe that

$$\left\|\mathsf{G}^{\dagger}\right\|_{\mathrm{F}}^{2} = \operatorname{trace}\left[(\mathsf{G}^{\dagger})^{*}\mathsf{G}^{\dagger}\right] = \operatorname{trace}\left[(\mathsf{G}\mathsf{G}^{*})^{-1}\right].$$

The second identity holds almost surely because the Wishart matrix $\mathsf{G}\mathsf{G}^{*}$ is invertible with probability one. The random matrix $(\mathsf{G}\mathsf{G}^{*})^{-1}$ follows the inverted Wishart distribution, so we can compute its expected trace explicitly using a formula from [99, p. 97]. □

On the other hand, very little seems to be known about the tail behavior of the Frobenius norm of a pseudo-inverted Gaussian matrix. The following theorem, which is new, provides an adequate bound on the probability of a large deviation.

Theorem 2.12.6. *Let G be an $m \times n$ standard Gaussian matrix with $n - m \geq 4$. For each $t \geq 1$,*

$$\mathbb{P}\left\{\left\|\mathsf{G}^{\dagger}\right\|_{\mathrm{F}}^{2} > \frac{12m}{n-m} \cdot t\right\} \leq 4t^{-(n-m)/2}.$$

Neither the precise form of Theorem 2.12.6 nor the constants are ideal; we have focused instead on establishing a useful bound with minimal fuss. The rest of the section is devoted to the rather lengthy proof. Unfortunately, most of the standard methods for producing tail bounds fail for random variables that do not exhibit normal or exponential concentration. Our argument relies on special properties of Gaussian matrices and a dose of brute force.

2.12.3.1 Technical background

We begin with a piece of notation. For any number $q \geq 1$, we define the L_q norm of a random variable Z by

$$\mathbb{E}^{q}(Z) = \left(\mathbb{E}\,|Z|^{q}\right)^{1/q}.$$

In particular, the L_q norm satisfies the triangle inequality.

We continue with a collection of technical results. First, we present a striking fact about the structure of Gaussian matrices [55, §3.5].

Proposition 2.12.7. *For $n \geq m$, an $m \times n$ standard Gaussian matrix is orthogonally equivalent with a random bidiagonal matrix*

$$\mathsf{L} = \begin{bmatrix} X_n & & & & & \\ Y_{m-1} & X_{n-1} & & & & \\ & Y_{m-2} & X_{n-2} & & & \\ & & \ddots & \ddots & & \\ & & & Y_1 & X_{n-(m-1)} & \end{bmatrix}_{m\times n}, \tag{2.72}$$

where, for each j, the random variables X_j^2 and Y_j^2 follow the χ^2 distribution with j degrees of freedom. Furthermore, these variates are mutually independent.

We also require the moments of a chi-square variate, which are expressed in terms of special functions.

Proposition 2.12.8. *Let Ξ be a χ^2 variate with k degrees of freedom. When $0 \le q < k/2$,*

$$\mathbb{E}\left(\Xi^q\right) = \frac{2^q \Gamma(k/2+q)}{\Gamma(k/2)} \quad \text{and} \quad \mathbb{E}\left(\Xi^{-q}\right) = \frac{\Gamma(k/2-q)}{2^q \Gamma(k/2)}.$$

Proof. Recall that a χ^2 variate with k degrees of freedom has the probability density function

$$f(t) = \frac{1}{2^{k/2}\Gamma(k/2)} t^{k/2-1} \mathrm{e}^{-t/2}, \quad \text{for } t \ge 0.$$

By the integral formula for expectation,

$$\mathbb{E}(\Xi^q) = \int_0^\infty t^q f(t)\,\mathrm{d}t = \frac{2^q\Gamma(k/2+q)}{\Gamma(k/2)},$$

where the second equality follows from Euler's integral expression for the gamma function. The other calculation is similar. □

To streamline the proof, we eliminate the gamma functions from Proposition 2.12.8. The next result bounds the positive moments of a chi-square variate.

Lemma 2.12.9. *Let Ξ be a χ^2 variate with k degrees of freedom. For $q \ge 1$,*

$$\mathbb{E}^q(\Xi) \le k + q.$$

Proof. Write $q = r + \theta$, where $r = \lfloor q \rfloor$. Repeated application of the functional equation $z\Gamma(z) = \Gamma(z+1)$ yields

$$\mathbb{E}^q(\Xi) = \left[\frac{2^\theta \Gamma(k/2+\theta)}{\Gamma(k/2)} \cdot \prod_{j=1}^{r} (k + 2(q-j)) \right]^{1/q}.$$

The gamma function is logarithmically convex, so

$$\frac{2^\theta \Gamma(k/2+\theta)}{\Gamma(k/2)} \le \frac{2^\theta \cdot \Gamma(k/2)^{1-\theta} \cdot \Gamma(k/2+1)^\theta}{\Gamma(k/2)} = k^\theta \le \left[\prod_{j=1}^{r} (k + 2(q-j)) \right]^{\theta/r}.$$

The second inequality holds because k is smaller than each term in the product, hence is smaller than their geometric mean. As a consequence,

$$\mathbb{E}^q(\Xi) \le \left[\prod_{j=1}^{r} (k + 2(q-j)) \right]^{1/r} \le \frac{1}{r} \sum_{j=1}^{r} (k + 2(q-j)) \le k + q.$$

The second relation is the inequality between the geometric and arithmetic mean. □

Finally, we develop a bound for the negative moments of a chi-square variate.

Lemma 2.12.10. *Let Ξ be a χ^2 variate with k degrees of freedom, where $k \geq 5$. When $2 \leq q \leq (k-1)/2$,*

$$\mathbb{E}^q\left(\Xi^{-1}\right) < \frac{3}{k}.$$

Proof. We establish the bound for $q = (k-1)/2$. For smaller values of q, the result follows from Hölder's inequality. Proposition 2.12.8 shows that

$$\mathbb{E}^q\left(\Xi^{-1}\right) = \left[\frac{\Gamma(k/2-q)}{2^q\Gamma(k/2)}\right]^{1/q} = \left[\frac{\Gamma(1/2)}{2^q\Gamma(k/2)}\right]^{1/q}.$$

Stirling's approximation ensures that $\Gamma(k/2) \geq \sqrt{2\pi}\cdot(k/2)^{(k-1)/2}\cdot \mathrm{e}^{-k/2}$. Since the value $\Gamma(1/2) = \sqrt{\pi}$,

$$\mathbb{E}^q\left(\Xi^{-1}\right) \leq \left[\frac{\sqrt{\pi}}{2^q\sqrt{2\pi}\cdot(k/2)^q\cdot \mathrm{e}^{-q-1/2}}\right]^{1/q} = \frac{\mathrm{e}}{k}\left[\frac{\mathrm{e}}{2}\right]^{1/2q} < \frac{3}{k},$$

where we used the assumption $q \geq 2$ to complete the numerical estimate. □

2.12.3.2 Proof of Theorem 2.12.6

Let G be an $m \times n$ Gaussian matrix, where we assume that $n - m \geq 4$. Define the random variable

$$Z = \left\|\mathsf{G}^\dagger\right\|_{\mathrm{F}}^2.$$

Our goal is to develop a tail bound for Z. The argument is inspired by work of Szarek [130, §6] for square Gaussian matrices.

The first step is to find an explicit, tractable representation for the random variable. According to Proposition 2.12.7, a Gaussian matrix G is orthogonally equivalent with a bidiagonal matrix L of the form (2.72). Making an analogy with the inversion formula for a triangular matrix, we realize that the pseudoinverse of L is given by

$$\mathsf{L}^\dagger = \begin{bmatrix} X_n^{-1} & & & & \\ \frac{-Y_{m-1}}{X_nX_{n-1}} & X_{n-1}^{-1} & & & \\ & \frac{-Y_{m-2}}{X_{n-1}X_{n-2}} & X_{n-2}^{-1} & & \\ & & \ddots & \ddots & \\ & & & \frac{-Y_1}{X_{n-(m-2)}X_{n-(m-1)}} & X_{n-(m-1)}^{-1} \\ & & & & \end{bmatrix}_{n\times m}.$$

Because $\mathsf{L}^\dagger$ is orthogonally equivalent with $\mathsf{G}^\dagger$ and the Frobenius norm is unitarily invariant, we have the relations

$$Z = \left\|\mathsf{G}^\dagger\right\|_{\mathrm{F}}^2 = \left\|\mathsf{L}^\dagger\right\|_{\mathrm{F}}^2 \leq \sum_{j=0}^{m-1}\frac{1}{X_{n-j}^2}\left(1 + \frac{Y_{m-j}^2}{X_{n-j+1}^2}\right),$$

where we have added an extra subdiagonal term (corresponding with $j = 0$) so that we can avoid exceptional cases later. We abbreviate the summands as

$$W_j = \frac{1}{X_{n-j}^2}\left(1 + \frac{Y_{m-j}^2}{X_{n-j+1}^2}\right), \quad j = 0, 1, 2, \dots, m-1.$$

Next, we develop a large deviation bound for each summand by computing a moment and invoking Markov's inequality. For the exponent $q = (n-m)/2$, Lemmas 2.12.9 and 2.12.10 yield

$$\begin{aligned}
\mathbb{E}^q(W_j) &= \mathbb{E}^q(X_{n-j}^{-2}) \cdot \mathbb{E}^q\left[1 + \frac{Y_{m-j}^2}{X_{n-j+1}^2}\right] \\
&\le \mathbb{E}^q(X_{n-j}^{-2})\left[1 + \mathbb{E}^q(Y_{m-j}^2) \cdot \mathbb{E}^q(X_{n-j+1}^{-2})\right] \\
&\le \frac{3}{n-j}\left[1 + \frac{3(m-j+q)}{n-j+1}\right] \\
&= \frac{3}{n-j}\left[1 + 3 - \frac{3(n-m+1-q)}{n-j+1}\right]
\end{aligned}$$

Note that the first two relations require the independence of the variates and the triangle inequality for the L_q norm. The maximum value of the bracket evidently occurs when $j = 0$, so

$$\mathbb{E}^q(W_j) < \frac{12}{n-j}, \quad j = 0, 1, 2, \dots, m-1.$$

Markov's inequality results in

$$\mathbb{P}\left\{W_j \ge \frac{12}{n-j} \cdot u\right\} \le u^{-q}.$$

Select $u = t \cdot (n-j)/(n-m)$ to reach

$$\mathbb{P}\left\{W_j \ge \frac{12}{n-m} \cdot t\right\} \le \left[\frac{n-m}{n-j}\right]^q t^{-q}.$$

To complete the argument, we combine these estimates by means of the union bound and clean up the resulting mess. Since $Z \le \sum_{j=0}^{m-1} W_j$,

$$\mathbb{P}\left\{Z \ge \frac{12m}{n-m} \cdot t\right\} \le t^{-q} \sum_{j=0}^{m-1} \left[\frac{n-m}{n-j}\right]^q.$$

To control the sum on the right-hand side, observe that

$$\begin{aligned}
\sum_{j=0}^{m-1} \left[\frac{n-m}{n-j}\right]^q &< (n-m)^q \int_0^m (n-x)^{-q}\,\mathrm{d}x \\
&< \frac{(n-m)^q}{q-1}(n-m)^{-q+1} = \frac{2(n-m)}{n-m-2} \le 4,
\end{aligned}$$

where the last inequality follows from the hypothesis $n - m \ge 4$. Together, the estimates in this paragraph produce the advertised bound.

Remark 24. It would be very interesting to find more conceptual and extensible argument that yields accurate concentration results for inverse spectral functions of a Gaussian matrix.

Bibliography

[1] D. Achlioptas, Database-friendly random projections: Johnson–Lindenstrauss with binary coins, J. Comput. System Sci., 66 (2003), pp. 671–687.

[2] D. Achlioptas and F. McSherry, Fast computation of low-rank matrix approximations, J. Assoc. Comput. Mach., 54 (2007), pp. Art. 9, 19 pp. (electronic).

[3] N. Ailon and B. Chazelle, Approximate nearest neighbors and the fast Johnson–Lindenstrauss transform, in STOC '06: Proc. 38th Ann. ACM Symp. Theory of Computing, 2006, pp. 557–563.

[4] N. Ailon and E. Liberty, Fast dimension reduction using Rademacher series on dual BCH codes, in STOC '08: Proc. 40th Ann. ACM Symp. Theory of Computing, 2008.

[5] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy, Tracking join and self-join sizes in limited storage, in Proc. 18th ACM Symp. Principles of Database Systems (PODS), 1999, pp. 10–20.

[6] N. Alon, Y. Matias, and M. Szegedy, The space complexity of approximating frequency moments, in STOC '96: Proc. 28th Ann. ACM Symp. Theory of Algorithms, 1996, pp. 20–29.

[7] S. Arora, E. Hazan, and S. Kale, A fast random sampling algorithm for sparsifying matrices, in Approximation, randomization and combinatorial optimization: Algorithms and Techniques, Springer, Berlin, 2006, pp. 272–279.

[8] A. R. Barron, Universal approximation bounds for superpositions of a sigmoidal function, IEEE Trans. Inform. Theory, 39 (1993), pp. 930–945.

[9] I. Beichl, The Metropolis algorithm, Comput. Sci. Eng., (2000), pp. 65–69.

[10] M.-A. Bellabas and P. J. Wolfe, On sparse representations of linear operators and the approximation of matrix products, in Proc. 42nd Ann. Conf. Information Sciences and Systems (CISS), 2008, pp. 258–263.

[11] R. Bhatia, Matrix Analysis, no. 169 in GTM, Springer, Berlin, 1997.

[12] Å. Björck, Numerics of Gram–Schmidt orthogonalization, Linear Algebra Appl., 197–198 (1994), pp. 297–316.

[13] ——, Numerical Methods for Least Squares Problems, SIAM, Philadelphia, PA, 1996.

[14] V. Bogdanov, Gaussian Measures, American Mathematical Society, Providence, RI, 1998.

[15] J. Bourgain, On Lipschitz embedding of finite metric spaces in Hilbert space, Israel J. Math., 52 (1985), pp. 46–52.

[16] C. Boutsidis and P. Drineas, Random projections for nonnegative least squares, Linear Algebra Appl., 431 (2009), pp. 760–771.

[17] C. Boutsidis, P. Drineas, and M. W. Mahoney, An improved approximation algorithm for the column subset selection problem, in Proc. 20th Ann. ACM–SIAM Symp. Discrete Algorithms (SODA), 2009.

[18] C. Boutsidis, M. W. Mahoney, and P. Drineas, Unsupervised feature selection for principal components analysis, in Proc. ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining (KDD), Aug. 2008.

[19] E. Candès and J. K. Romberg, Sparsity and incoherence in compressive sampling, Inverse Problems, 23 (2007), pp. 969–985.

[20] E. Candès, J. K. Romberg, and T. Tao, Robust uncertainty principles: Exact signal reconstruction from highly incomplete Fourier information, IEEE Trans. Inform. Theory, 52 (2006), pp. 489–509.

[21] E. J. Candès, Compressive sampling, in Proc. 2006 Intl. Cong. Mathematicians, Madrid, 2006.

[22] E. J. Candès and B. Recht, Exact matrix completion via convex optimization, Found. Comput. Math., 9 (2009), pp. 717–772.

[23] E. J. Candès and T. Tao, The power of convex relaxation: Near-optimal matrix completion, IEEE Trans. Inform. Theory, 56 (2010), pp. 2053–2080.

[24] B. Carl, Inequalities of Bernstein–Jackson-type and the degree of compactness in Banach spaces, Ann. Inst. Fourier (Grenoble), 35 (1985), pp. 79–118.

[25] Z. Chen and J. J. Dongarra, Condition numbers of Gaussian random matrices, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 603–620.

[26] H. Cheng, Z. Gimbutas, P.-G. Martinsson, and V. Rokhlin, On the compression of low rank matrices, SIAM J. Sci. Comput., 26 (2005), pp. 1389–1404 (electronic).

[27] A. Çivril and M. Magdon-Ismail, On selecting a maximum volume sub-matrix of a matrix and related problems, Theoret. Comput. Sci., 410 (2009), pp. 4801–4811.

[28] K. L. Clarkson, Subgradient and sampling algorithms for ℓ_1 regression, in Proc. 16th Ann. ACM–SIAM Symp. Discrete Algorithms (SODA), 2005, pp. 257–266.

[29] K. L. Clarkson and D. P. Woodruff, Numerical linear algebra in the streaming model, in STOC '09: Proc. 41st Ann. ACM Symp. Theory of Computing, 2009.

[30] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker, Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps, Proc. Natl. Acad. Sci. USA, 102 (2005), pp. 7426–7431.

[31] A. Dasgupta, P. Drineas, B. Harb, R. Kumar, and M. W. Mahoney, Sampling algorithms and coresets for ℓ_p regression, SIAM J. Comput., 38 (2009), pp. 2060–2078.

[32] S. Dasgupta and A. Gupta, An elementary proof of the Johnson–Lindenstrauss lemma, Computer Science Dept. Tech. Report 99-006, Univ. California at Berkeley, Mar. 1999.

[33] A. d'Aspremont, Subsampling algorithms for semidefinite programming. Available at `arXiv:0803.1990`, Apr. 2009.

[34] K. R. Davidson and S. J. Szarek, Local operator theory, random matrices, and Banach spaces, in Handbook of Banach Space Geometry, W. B. Johnson and J. Lindenstrauss, eds., Elsevier, 2002, pp. 317–366.

[35] J. Demmel, I. Dumitriu, and O. Holtz, Fast linear algebra is stable, Numer. Math., 108 (2007), pp. 59–91.

[36] A. Deshpande and L. Rademacher, Efficient volume sampling for row/column subset selection. Available at `arXiv:1004.4057`, Apr. 2010.

[37] A. Deshpande, L. Rademacher, S. Vempala, and G. Wang, Matrix approximation and projective clustering via volume sampling, in Proc. 17th Ann. ACM–SIAM Symp. Discrete Algorithms (SODA), 2006, pp. 1117–1126.

[38] A. Deshpande and S. Vempala, Adaptive sampling and fast low-rank matrix approximation, in Approximation, randomization and combinatorial optimization, vol. 4110 of LNCS, Springer, Berlin, 2006, pp. 292–303.

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

[40] J. Dongarra and F. Sullivan, The top 10 algorithms, Comput. Sci. Eng., (2000), pp. 22–23.

[41] D. L. Donoho, Compressed sensing, IEEE Trans. Inform. Theory, 52 (2006), pp. 1289–1306.

[42] D. L. Donoho, M. Vetterli, R. A. DeVore, and I. Daubechies, Data compression and harmonic analysis, IEEE Trans. Inform. Theory, 44 (1998), pp. 2433–2452.

[43] P. Drineas, A. Frieza, R. Kannan, S. Vempala, and V. Vinay, Clustering of large graphs via the singular value decomposition, Machine Learning, 56 (2004), pp. 9–33.

[44] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, Clustering in large graphs and matrices, in Proc. 10th Ann. ACM Symp. on Discrete Algorithms (SODA), 1999, pp. 291–299.

[45] P. Drineas, R. Kannan, and M. W. Mahoney, Fast Monte Carlo algorithms for matrices. I. Approximating matrix multiplication, SIAM J. Comput., 36 (2006), pp. 132–157.

[46] ——, Fast Monte Carlo algorithms for matrices. II. Computing a low-rank approximation to a matrix, SIAM J. Comput., 36 (2006), pp. 158–183 (electronic).

[47] ——, Fast Monte Carlo algorithms for matrices. III. Computing a compressed approximate matrix decomposition, SIAM J. Comput., 36 (2006), pp. 184–206.

[48] P. Drineas and M. W. Mahoney, On the Nyström method for approximating a Gram matrix for improved kernel-based learning, J. Mach. Learn. Res., 6 (2005), pp. 2153–2175.

[49] ——, A randomized algorithm for a tensor-based generalization of the singular value decomposition, Linear Algebra Appl., 420 (2007), pp. 553–571.

[50] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, Subspace sampling and relative-error matrix approximation: Column-based methods, in APPROX and RANDOM 2006, J. D. et al., ed., no. 4110 in LNCS, Springer, Berlin, 2006, pp. 321–326.

[51] ——, Relative-error CUR matrix decompositions, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 844–881.

[52] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós, Faster least squares approximation, Num. Math., (2009). To appear. Available at `arXiv:0710.1435`.

[53] A. Dvoretsky, Some results on convex bodies and Banach spaces, in Proc. Intl. Symp. Linear Spaces, Jerusalem, 1961, pp. 123–160.

[54] C. Eckart and G. Young, The approximation of one matrix by another of lower rank, Psychometrika, 1 (1936), pp. 211–218.

[55] A. Edelman, Eigenvalues and condition numbers of random matrices, Ph.D. thesis, Mathematics Dept., Massachusetts Inst. Tech., Boston, MA, May 1989.

[56] B. Engquist and O. Runborg, Wavelet-based numerical homogenization with applications, in Multiscale and Multiresolution Methods: Theory and Applications, T. J. B. et al., ed., vol. 20 of LNCSE, Springer, Berlin, 2001, pp. 97–148.

[57] A. Frieze, R. Kannan, and S. Vempala, Fast Monte Carlo algorithms for finding low-rank approximations, in Proc. 39th Ann. IEEE Symp. Foundations of Computer Science (FOCS), 1998, pp. 370–378.

[58] ——, Fast Monte Carlo algorithms for finding low-rank approximations, J. Assoc. Comput. Mach., 51 (2004), pp. 1025–1041. (electronic).

[59] A. Y. Garnaev and E. D. Gluskin, The widths of a Euclidean ball, Dokl. Akad. Nauk. SSSR, 277 (1984), pp. 1048–1052. In Russian.

[60] A. Gittens and J. A. Tropp, Error bounds for random matrix approximation schemes. Available at `arXiv:0911.4108`.

[61] G. H. Golub and C. F. van Loan, Matrix Computations, Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins Univ. Press, Baltimore, MD, 3rd ed., 1996.

[62] Y. Gordon, Some inequalities for Gaussian processes and applications, Israel J. Math., 50 (1985), pp. 265–289.

[63] ——, Gaussian processes and almost spherical sections of convex bodies, Ann. Probab., 16 (1988), pp. 180–188.

[64] S. A. Goreinov, E. E. Tyrtyshnikov, and N. L. Zamarashkin, Theory of pseudo-skeleton matrix approximations, Linear Algebra Appl., 261 (1997), pp. 1–21.

[65] L. Grasedyck and W. Hackbusch, Construction and arithmetics of $\mathcal{H}$-matrices, Computing, 70 (2003), pp. 295–334.

[66] L. Greengard and V. Rokhlin, A new version of the fast multipole method for the Laplace equation in three dimensions, Acta Numer., 17 (1997), pp. 229–269.

[67] M. Gu, 2007. Personal communication.

[68] M. Gu and S. C. Eisenstat, Efficient algorithms for computing a strong rank-revealing QR factorization, SIAM J. Sci. Comput., 17 (1996), pp. 848–869.

[69] N. Halko, P.-G. Martinsson, Y. Shkolnisky, and M. Tygert, An algorithm for the principal component analysis of large data sets, 2010.

[70] S. Har-Peled, Matrix approximation in linear time. Manuscript. Available at `http://valis.cs.uiuc.edu/~sariel/research/papers/05/lrank/`, 2006.

[71] T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer, Berlin, 2nd ed., 2008.

[72] R. A. Horn and C. R. Johnson, Matrix Analysis, Cambridge Univ. Press, Cambridge, 1985.

[73] P. Indyk and R. Motwani, Approximate nearest neighbors: Toward removing the curse of dimensionality, in STOC '98: Proc. 30th Ann. ACM Symp. Theory of Computing, 1998, pp. 604–613.

[74] W. B. Johnson and J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space, Contemp. Math., 26 (1984), pp. 189–206.

[75] D. R. Karger, Random sampling in cut, flow, and network design problems, Math. Oper. Res., 24 (1999), pp. 383–413.

[76] ——, Minimum cuts in near-linear time, J. ACM, 47 (2000), pp. 46–76.

[77] B. S. Kašin, On the widths of certain finite-dimensional sets and classes of smooth functions, Izv. Akad. Nauk. SSSR Ser. Mat., 41 (1977), pp. 334–351, 478. In Russian.

[78] J. Kleinberg, Two algorithms for nearest neighbor search in high dimensions, in STOC '97: Proc. 29th ACM Symp. Theory of Computing, 1997, pp. 599–608.

[79] 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.

[80] E. Kushilevitz, R. Ostrovski, and Y. Rabani, Efficient search for approximate nearest neighbor in high-dimensional spaces, SIAM J. Comput., 30 (2000), pp. 457–474.

[81] D. Le and D. S. Parker, Using randomization to make recursive matrix algorithms practical, J. Funct. Programming, 9 (1999), pp. 605–624.

[82] M. Ledoux, The Concentration of Measure Phenomenon, no. 89 in MSM, American Mathematical Society, Providence, RI, 2001.

[83] M. Ledoux and M. Talagrand, Probability in Banach Spaces: Isoperimetry and Processes, Springer, Berlin, 1991.

[84] W. S. Lee, P. L. Bartlett, and R. C. Williamson, Efficient agnostic learning of neural networks with bounded fan-in, IEEE Trans. Inform. Theory, 42 (1996), pp. 2118–2132.

[85] Z. Leyk and H. Woźniakowski, Estimating the largest eigenvector by Lanczos and polynomial algorithms with a random start, Num. Linear Algebra Appl., 5 (1998), pp. 147–164.

[86] E. Liberty, Accelerated dense random projections, Ph.D. thesis, Computer Science Dept., Yale University, New Haven, CT, 2009.

[87] E. Liberty, N. Ailon, and A. Singer, Dense fast random projections and lean Walsh transforms, in APPROX and RANDOM 2008, A. G. et al., ed., no. 5171 in LNCS, Springer, Berlin, 2008, pp. 512–522.

[88] E. Liberty, F. 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.

[89] M. W. Mahoney and P. Drineas, CUR matrix decompositions for improved data analysis, Proc. Natl. Acad. Sci. USA, 106 (2009), pp. 697–702.

[90] P.-G. Martinsson, V. Rokhlin, Y. Shkolnisky, and M. Tygert, ID: A software package for low-rank approximation of matrices via interpolative decompositions, version 0.2, 2008.

[91] P.-G. Martinsson, V. Rokhlin, and M. Tygert, A randomized algorithm for the approximation of matrices, Computer Science Dept. Tech. Report 1361, Yale Univ., New Haven, CT, 2006.

[92] P.-G. Martinsson, A. Szlam, and M. Tygert, Normalized power iterations for the computation of SVD. Manuscript., Nov. 2010.

[93] J. Matoušek, Lectures on Discrete Geometry, Springer, Berlin, 2002.

[94] F. McSherry, Spectral Methods in Data Analysis, Ph.D. thesis, Computer Science Dept., Univ. Washington, Seattle, WA, 2004.

[95] N. Metropolis and S. Ulam, The Monte Carlo method, J. Amer. Statist. Assoc., 44 (1949), pp. 335–341.

[96] V. D. Milman, A new proof of A. Dvoretsky's theorem on cross-sections of convex bodies, Funkcional. Anal. i Priložen, 5 (1971), pp. 28–37.

[97] L. Mirsky, Symmetric gauge functions and unitarily invariant norms, Quart. J. Math. Oxford Ser. (2), 11 (1960), pp. 50–59.

[98] R. Motwani and P. Raghavan, Randomized Algorithms, Cambridge Univ. Press, Cambridge, 1995.

[99] R. J. Muirhead, Aspects of Multivariate Statistical Theory, Wiley, New York, NY, 1982.

[100] S. Muthukrishnan, Data Streams: Algorithms and Applications, Now Publ., Boston, MA, 2005.

[101] D. Needell, Randomized Kaczmarz solver for noisy linear systems, BIT, 50 (2010), pp. 395–403.

[102] N. H. Nguyen, T. T. Do, and T. D. Tran, A fast and efficient algorithm for low-rank approximation of a matrix, in STOC '09: Proc. 41st Ann. ACM Symp. Theory of Computing, 2009.

[103] C.-T. Pan, On the existence and computation of rank-revealing LU factorizations, Linear Algebra Appl., 316 (2000), pp. 199–222.

[104] C. H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala, Latent semantic indexing: A probabilistic analysis, in Proc. 17th ACM Symp. Principles of Database Systems (PODS), 1998.

[105] ——, Latent semantic indexing: A probabilistic analysis, J. Comput. System Sci., 61 (2000), pp. 217–235.

[106] D. S. Parker and B. Pierce, The randomizing FFT: An alternative to pivoting in Gaussian elimination, Computer Science Dept. Tech. Report CSD 950037, Univ. California at Los Angeles, 1995.

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

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

[109] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes: The Art of Scientific Computing, Cambridge University Press, Cambridge, 3rd ed., 2007.

[110] A. Rahimi and B. Recht, Random features for large-scale kernel machines, in Proc. 21st Ann. Conf. Advances in Neural Information Processing Systems (NIPS), 2007.

[111] B. Recht, M. Fazel, and P. Parillo, Guaranteed minimum-rank solutions ot matrix equations via nuclear-norm minimization, SIAM Rev., 52 (2010), pp. 471–501.

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

[113] V. Rokhlin and M. Tygert, A fast randomized algorithm for overdetermined linear least-squares regression, Proc. Natl. Acad. Sci. USA, 105 (2008), pp. 13212–13217.

[114] S. Roweis, EM algorithms for PCA and SPCA, in Proc. 10th Ann. Conf. Advances in Neural Information Processing Systems (NIPS), MIT Press, 1997, pp. 626–632.

[115] M. Rudelson, Random vectors in the isotropic position, J. Funct. Anal., 164 (1999), pp. 60–72.

[116] M. Rudelson and R. Vershynin, Sampling from large matrices: An approach through geometric functional analysis, J. Assoc. Comput. Mach., 54 (2007), pp. Art. 21, 19 pp. (electronic).

[117] A. F. Ruston, Auerbach's theorem, Math. Proc. Cambridge Philos. Soc., 56 (1964), pp. 476–480.

[118] T. Sarlós, Improved approximation algorithms for large matrices via random projections, in Proc. 47th Ann. IEEE Symp. Foundations of Computer Science (FOCS), 2006, pp. 143–152.

[119] S. Shalev-Shwartz and N. Srebro, Low ℓ_1-norm and guarantees on sparsifiability, in ICML/COLT/UAI Sparse Optimization and Variable Selection Workshop, July 2008.

[120] X. Shen and F. G. Meyer, Low-dimensional embedding of fMRI datasets, Neuroimage, 41 (2008), pp. 886–902.

[121] N. D. Shyamalkumar and K. Varadarajaran, Efficient subspace approximation algorithms, in Proc. 18th Ann. ACM–SIAM Symp. Discrete Algorithms (SODA), 2007, pp. 532–540.

[122] L. Sirovich and M. Kirby, Low-dimensional procedure for the characterization of human faces., J. Optical Soc. Amer. A, 4 (1987), pp. 519–524.

[123] D. Spielman and N. Srivastasa, Graph sparsification by effective resistances, in STOC '08: Proc. 40th Ann. ACM Symp. Theory of Computing, 2008.

[124] G. Stewart, Accelerating the orthogonal iteration for the eigenvectors of a Hermitian matrix, Num. Math., 13 (1969), pp. 362–376. 10.1007/BF02165413.

[125] G. W. Stewart, Perturbation of pseudo-inverses, projections, and linear least squares problems, SIAM Rev., 19 (1977), pp. 634–662.

[126] ——, Four algorithms for the efficient computation of truncated pivoted QR approximations to a sparse matrix, Numer. Math., 83 (1999), pp. 313–323.

[127] ——, The decompositional approach to matrix computation, Comput. Sci. Eng., (2000), pp. 50–59.

[128] T. Strohmer and R. Vershynin, A randomized Kaczmarz algorithm with exponential convergence, J. Fourier Anal. Appl., 15 (2009), pp. 262–278.

[129] J. Sun, Y. Xie, H. Zhang, and C. Faloutsos, Less is more: Compact matrix decomposition for large sparse graphs, Stat. Anal. Data Min., 1 (2008), pp. 6–22.

[130] S. J. Szarek, Spaces with large distance from ℓ_∞^n and random matrices, Amer. J. Math., 112 (1990), pp. 899–942.

[131] A. Szlam, M. Maggioni, and R. R. Coifman, Regularization on graphs with function-adapted diffsion processes, J. Mach. Learn. Res., 9 (2008), pp. 1711–1739.

[132] L. N. Trefethen and D. B. III, Numerical Linear Algebra, SIAM, Philadelphia, PA, 1997.

[133] J. A. Tropp, On the conditioning of random subdictionaries, Appl. Comput. Harmon. Anal., 25 (2008), pp. 1–24.

[134] ——, Improved analysis of the subsampled randomized Hadamard transform, Adv. Adaptive Data Anal., 3 (2011). To appear. Available at `arXiv:1011.1595`.

[135] J. von Neumann and H. H. Goldstine, Numerical inverting of matrices of high order, Bull. Amer. Math. Soc., 53 (1947), pp. 1021–1099.

[136] ——, Numerical inverting of matrices of high order. II, Proc. Amer. Math. Soc., 2 (1952), pp. 188–202.

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

Chapter 3

An algorithm for the principal component analysis of large data sets

Nathan Halko, Per-Gunnar Martinsson, Yoel Shkolnisky, Mark Tygert

Note: *The work described in this chapter was carried out in collaboration with Professor Yoel Shkolnisky of Tel Aviv University, Professor Mark Tygert of New York University, and Professor Per-Gunnar Martinsson of the University of Colorado. It appeared in the SIAM Journal of Scientific Computation in 2011 (volume 33, number 5, pages 2580–2594) under the title: "An Algorithm for the Principal Component Analysis of Large Data Sets"*

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.*

Keywords: algorithm, principal component analysis, PCA, SVD, singular value decomposition, low rank

AMS classification: 65F15, 65C60, 68W20

3.1 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 3.3 below. The proofs in [5] and [13] 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 [5] and [13]; the present article adapts the algorithm for use with data sets that are too large to be stored in the random-access memory (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, say 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

$$A \approx U \Sigma V^{\top}, \tag{3.1}$$

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. The algorithm summarized in Section 3.3 below is most efficient when k is substantially less than $\min(m, n)$; in typical real-world applications, $k \ll \min(m, n)$. Most often, the relevant measure of the quality of the approximation in (3.1) is the spectral norm of the discrepancy $A - U \Sigma V^{\top}$; see, for example, Section 3.3 below. The present article focuses on the spectral norm, though our methods produce similar accuracy in the Frobenius/Hilbert-Schmidt norm (see, for example, [5]).

The procedure of the present article works to minimize the total number of times that the algorithm has to access each entry of the matrix A being approximated. A related strategy is to minimize the total number of disk seeks and to maximize the dimensions of the approximation that can be constructed with a given amount of RAM; the algorithm in [8] takes this latter approach.

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 3.2 explains the motivation behind the algorithm. Section 3.3 outlines the algorithm. Section 3.4 details the implementation for very large matrices. Section 3.5 quantifies the main factors influencing the running-time of the algorithm. Section 3.6 illustrates the performance of the algorithm via several numerical examples. Section 3.7 applies the algorithm to a data set of interest in biochemical imaging. Section 3.8 draws some conclusions and proposes directions for further research.

3.2 Informal description of the algorithm

In this section, we provide a brief, heuristic description. Section 3.3 below provides more details on the algorithm described intuitively in the present section.

Suppose that k, m, and n are positive integers with $k < m$ and $k < n$, and A is a real $m \times n$ matrix. We will construct an approximation to A such that

$$\|A - U \Sigma V^{\top}\|_2 \approx \sigma_{k+1}, \tag{3.2}$$

where 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 diagonal real $k \times k$ matrix whose entries are all nonnegative, $\|A - U \Sigma V^{\top}\|_2$ is the spectral (l^2-operator) norm of $A - U \Sigma V^{\top}$, and σ_{k+1} is the $(k+1)$st greatest

singular value of A. To do so, we select nonnegative integers i and l such that $l \geq k$ and $(i+2)k \leq n$ (for most applications, $l = k + 2$ and $i \leq 2$ is sufficient; $\|A - U\,\Sigma\,V^\top\|_2$ will decrease as i and l increase), and then identify an orthonormal basis for "most" of the range of A via the following two steps:

1. Using a random number generator, form a real $n \times l$ matrix G whose entries are independent and identically distributed Gaussian random variables of zero mean and unit variance, and compute the $m \times ((i+1)l)$ matrix
$$H = \left(\ A\,G \ \middle|\ A\,A^\top\,A\,G \ \middle|\ \ldots \ \middle|\ (A\,A^\top)^{i-1}\,A\,G \ \middle|\ (A\,A^\top)^{i}\,A\,G\ \right). \tag{3.3}$$
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
$$H = Q\,R. \tag{3.4}$$
(See, for example, Chapter 5 in [4] for details concerning the construction of such a matrix Q.)

Intuitively, the columns of Q in (3.4) constitute an orthonormal basis for most of the range of A. Moreover, the somewhat simplified algorithm with $i = 0$ is sufficient except when the singular values of A decay slowly; see, for example, [5].

Notice that Q may have many fewer columns than A, that is, k may be substantially less than n (this is the case for most applications of principal component analysis). This is the key to the efficiency of the algorithm.

Having identified a good approximation to the range of A, we perform some simple linear algebraic manipulations in order to obtain a good approximation to A, via the following four steps:

3. Compute the $n \times ((i+1)l)$ product matrix
$$T = A^\top\,Q. \tag{3.5}$$
4. Form an SVD of T,
$$T = \tilde{V}\,\tilde{\Sigma}\,W^\top, \tag{3.6}$$
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 [4] for details concerning the construction of such an SVD.)
5. Compute the $m \times ((i+1)l)$ product matrix
$$\tilde{U} = Q\,W. \tag{3.7}$$
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 matrices U, Σ, and V obtained via Steps 1–6 above satisfy (3.2); in fact, they satisfy the more detailed bound (3.8) described below.

3.3 Summary of the algorithm

In this section, we will construct a low-rank (say, rank k) approximation $U\,\Sigma\,V^\top$ to any given real matrix A, such that

$$\|A - U\,\Sigma\,V^\top\|_2 \le \sqrt{(Ckn)^{1/(2i+1)} + \min(1, C/n)}\;\sigma_{k+1} \tag{3.8}$$

with high probability (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 $(k+1)$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 (3.8), 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 $\|A - U\,\Sigma\,V^\top\|_2$ is the spectral (l^2-operator) norm of $A - U\,\Sigma\,V^\top$, that is,

$$\|A - U\,\Sigma\,V^\top\|_2 = \max_{x \in \mathbb{R}^n : \|x\|_2 \neq 0} \frac{\|(A - U\,\Sigma\,V^\top)x\|_2}{\|x\|_2}, \tag{3.9}$$

$$\|x\|_2 = \sqrt{\sum_{j=1}^{n} (x_j)^2}. \tag{3.10}$$

To simplify the presentation, we will assume that $n \le m$ (if $n > m$, then the user can apply the algorithm to $A^\top$). In this section, we summarize the algorithm; see [5] and [13] for an in-depth discussion, including proofs of more detailed variants of (3.8).

The minimal value of the spectral norm $\|A - B\|_2$, minimized over all rank-k matrices B, is σ_{k+1} (see, for example, Theorem 2.5.3 in [4]). Hence, (3.8) 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 help suppress the smaller singular values, we apply $A\,(A^\top A)^i$, too. Once we have identified "most" of the range of A, we perform some linear-algebraic manipulations in order to recover an approximation satisfying (3.8).

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 and 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

$$H^{(0)} = A\,G, \tag{3.11}$$

$$H^{(1)} = A\,(A^\top H^{(0)}), \tag{3.12}$$

$$H^{(2)} = A\,(A^\top H^{(1)}), \tag{3.13}$$

$$\vdots$$

$$H^{(i)} = A\,(A^\top H^{(i-1)}). \tag{3.14}$$

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

$$H = \left(\; H^{(0)} \;\middle|\; H^{(1)} \;\middle|\; \ldots \;\middle|\; H^{(i-1)} \;\middle|\; H^{(i)} \;\right). \tag{3.15}$$

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

$$H = Q\,R. \tag{3.16}$$

(See, for example, Chapter 5 in [4] for details concerning the construction of such a matrix Q.)

3. Compute the $n \times ((i+1)l)$ product matrix

$$T = A^\top Q. \tag{3.17}$$

4. Form an SVD of T,

$$T = \tilde{V}\,\tilde{\Sigma}\,W^\top, \tag{3.18}$$

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 [4] for details concerning the construction of such an SVD.)

5. Compute the $m \times ((i+1)l)$ product matrix

$$\tilde{U} = Q\,W. \tag{3.19}$$

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 (3.8) (we omit the proof; see [5] for proofs of similar, more general bounds).

Remark 25. *In the present paper, we assume that the user specifies the rank k of the approximation $U\,\Sigma\,V^\top$ being constructed. See [5] for techniques for determining the rank k adaptively, such that the accuracy $\|A - U\,\Sigma\,V^\top\|_2$ satisfying (3.8) also meets a user-specified threshold.*

Remark 26. *Variants of the fast Fourier transform (FFT) permit additional accelerations; see [5], [7], and [14]. 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.*

Remark 27. *The algorithm described in the present section can underflow or overflow when the range of the floating-point exponent is inadequate for representing simultaneously both the spectral norm $\|A\|_2$ and its $(2i+1)$st power $(\|A\|_2)^{2i+1}$. A convenient alternative is the algorithm described in [8]; another solution is to process $A/\|A\|_2$ rather than A.*

3.4 Out-of-core computations

With suitably large matrices, some steps in Section 3.3 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 random-access memory (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 3.4.1 and 3.4.2.

3.4.1 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 $A^\top$, we immediately perform all computations associated with this particular entry (or row or column) that contribute to the matrix product.

3.4.2 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 $A^\top$ instead). To construct the matrix product in (3.11), 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 (3.17), 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 (3.12) similarly, forming $F = A^\top H^{(0)}$ first, and $H^{(1)} = A\,F$ second. Constructing the matrix products in (3.13)–(3.14) is analogous.

3.5 Computational costs

In this section, we tabulate the computational costs of the algorithm described in Section 3.3, for the particular out-of-core implementations described in Subsections 3.4.1 and 3.4.2. We will be using the notation from Section 3.3, including the integers i, k, l, m, and n, and the $m \times n$ matrix A.

Remark 28. *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 [4]).*

3.5.1 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 3.3 have the following costs:

(1) Forming $H^{(0)}$ in (3.11) costs $C_A + \mathcal{O}(l\,N_A)$ flops. Forming any of the matrix products in (3.12)–(3.14) costs $2C_A + \mathcal{O}(l\,N_A)$ flops. Forming H in (3.15) 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 (3.16) costs $\mathcal{O}(i^2l^2m)$ flops.

(3) Forming T in (3.17) costs $C_A + \mathcal{O}(il\,N_A)$ flops.

(4) Forming the SVD of T in (3.18) costs $\mathcal{O}(i^2l^2n)$ flops.

(5) Forming $\tilde{U}$ in (3.19) costs $\mathcal{O}(i^2l^2m)$ 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

$$C_{\text{on-the-fly}} = 2(i+1)\,C_A + \mathcal{O}(il\,N_A + i^2l^2m) \tag{3.20}$$

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. In practice, we choose $l \approx k$ (usually a good choice is $l = k+2$).

3.5.2 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 3.3 have the following costs (assuming for convenience that $j > 2\,(i+1)\,l\,(m+n)$):

(1) Forming $H^{(0)}$ in (3.11) 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 (3.12)–(3.14) 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 (3.15) 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 (3.16) costs $\mathcal{O}(i^2l^2m)$ flops.

(3) Forming T in (3.17) 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 (3.18) costs $\mathcal{O}(i^2l^2n)$ flops.

(5) Forming $\tilde{U}$ in (3.19) costs $\mathcal{O}(i^2l^2m)$ flops.

(6) Forming U, Σ, and V in Step 6 costs $\mathcal{O}(k(m+n))$ flops.

In practice, we choose $l \approx k$ (usually a good choice is $l = k + 2$). 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

$$C_{\text{flops}} = \mathcal{O}(ilmn + i^2l^2m) \tag{3.21}$$

flops,

$$C_{\text{accesses}} = \mathcal{O}(imn/j) \tag{3.22}$$

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

$$C_{\text{words}} = \mathcal{O}(imn) \tag{3.23}$$

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

3.6 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 3.3 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.

3.6.1 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

$$A = E\,S\,F, \tag{3.24}$$

where E and F 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

$$S_{j,j} = \begin{cases} 10^{-4(j-1)/19}, & j = 1, 2, \ldots, 19, \text{ or } 20 \\ 10^{-4}/(j-20)^{1/10}, & j = 21, 22, \ldots, n-1, \text{ or } n. \end{cases} \tag{3.25}$$

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

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

$$A = E\,S\,F, \tag{3.26}$$

where E and F 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

$$S_{j,j} = \begin{cases} 1.00, & j = 1, 2, \text{ or } 3 \\ 0.67, & j = 4, 5, \text{ or } 6 \\ 0.34, & j = 7, 8, \text{ or } 9 \\ 0.01, & j = 10, 11, \text{ or } 12 \\ 0.01 \cdot \frac{n-j}{n-13}, & j = 13, 14, \ldots, n-1, \text{ or } n. \end{cases} \tag{3.27}$$

Clearly, $S_{1,1}$, $S_{2,2}$, ..., $S_{n-1,n-1}$, $S_{n,n}$ 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 following list describes the headings of the tables:

- 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 3.3 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.

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

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

- 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 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 [6]). 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

$$\alpha\, w_1 + \beta\, w_2 + \gamma\, w_3 + \delta, \tag{3.28}$$

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,

$$\alpha = a\, r\, (\cos\varphi)\, \sin\theta, \tag{3.29}$$

$$\beta = b\, r\, (\sin\varphi)\, \sin\theta, \tag{3.30}$$

$$\gamma = c\, r \cos\theta, \tag{3.31}$$

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 (3.28) and the (normalized) right 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 (3.28) and the (normalized) right singular vector associated with the second greatest singular value, as well as the inner product of w_3 and the (normalized) right singular vector associated with the third greatest singular value. Needless to say, the inner products (*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 (3.20).

3.6.2 Measured data

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

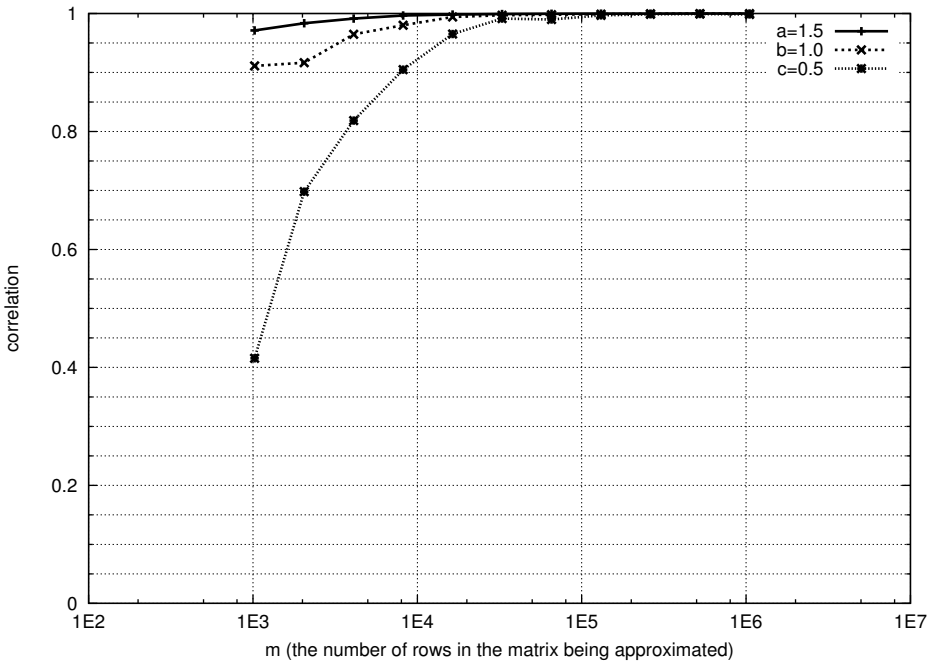


FIG. 1A. *Convergence for the third example (the computational simulation).*

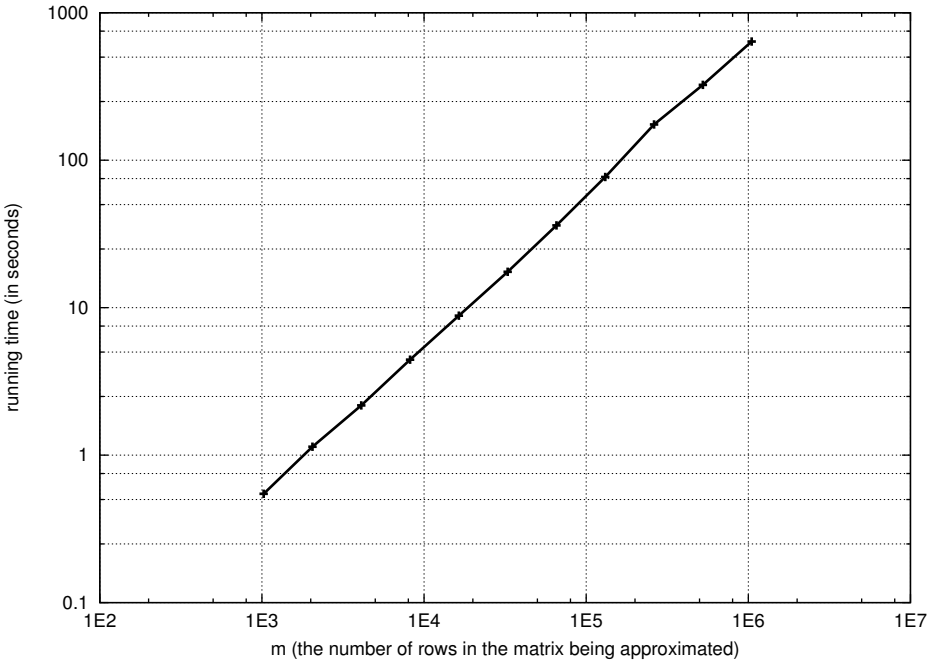


FIG. 1B. *Timing for the third example (the computational simulation).*

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 [10] and [11], 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 3.6).

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.

While this example does not directly provide a reasonable means for performing face recognition or any other task of image processing, it does indicate that the sheer brute force of linear algebra (that is, computing a low-rank approximation) can be used directly for processing (or preprocessing) a very large data set. When used alone, this kind of brute force is inadequate for face recognition and other tasks of image processing; most tasks of image processing can benefit from more specialized methods (see, for example, [9], [10], and [11]). Nonetheless, the ability to compute principal component analyses of very large data sets could prove helpful, or at least convenient.

3.7 An application

In this section, we apply the algorithm of the present paper to a data set of interest in a currently developing imaging modality known as single-particle cryo-electron microscopy. For an overview of the field, see [3], [12], and their compilations of references; in particular, [12] provides an alternative to the algorithm of the present paper that is generally preferable for the specific application discussed in the present section.

The data set consists of 10,000 two-dimensional images of the (three-dimensional) charge density map of the E. coli 50S ribosomal subunit, projected from uniformly random orientations, then added to white Gaussian noise whose magnitude is 32 times larger than the original images', and finally rotated by 0, 1, 2, ..., 358, 359 degrees. The entire data set thus consists of 3,600,000 images, each 129 pixels wide and 129 pixels high; the matrix being processed is 3,600,000 $\times$ 129^2. We set $i = 1$, $k = 250$, and $l = k + 2$, where i, k, and l are the parameters from Section 3.3 above. Processing the data set required 5.5 hours on two 2.8 GHz quad-core Intel Xeon x5560 microprocessors with 48 GB of random-access memory.

Figure 3a displays the 250 computed singular values. Figure 3b displays the computed right singular vectors corresponding to the 25 greatest computed singular values. Figure 3c displays several noisy projections, their versions before adding the white Gaussian noise, and their denoised versions. Each denoised image is the projection of the corresponding noisy image on the computed right singular vectors associated with the 150 greatest computed singular values. The denoising is clearly satisfactory.

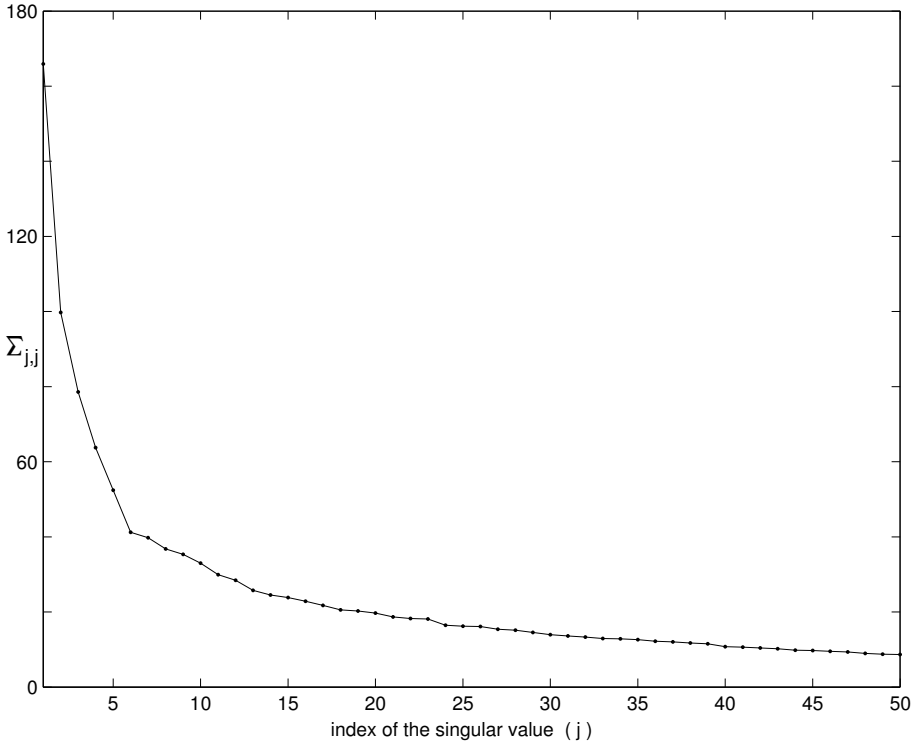


FIG. 2A. *Singular values computed for the fourth example (the database of images).*

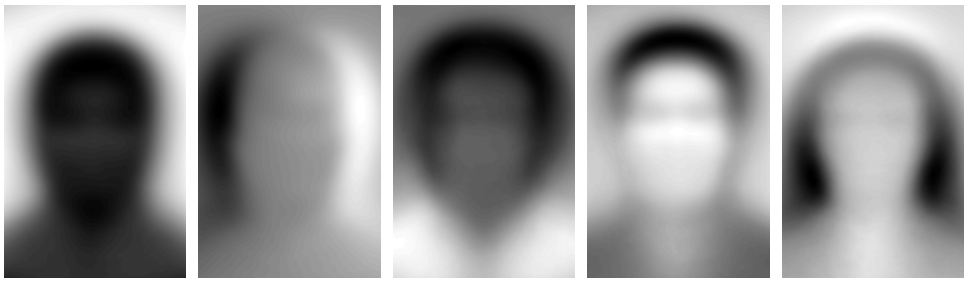


FIG. 2B. *Dominant singular vectors computed for the fourth example (the database of images).*

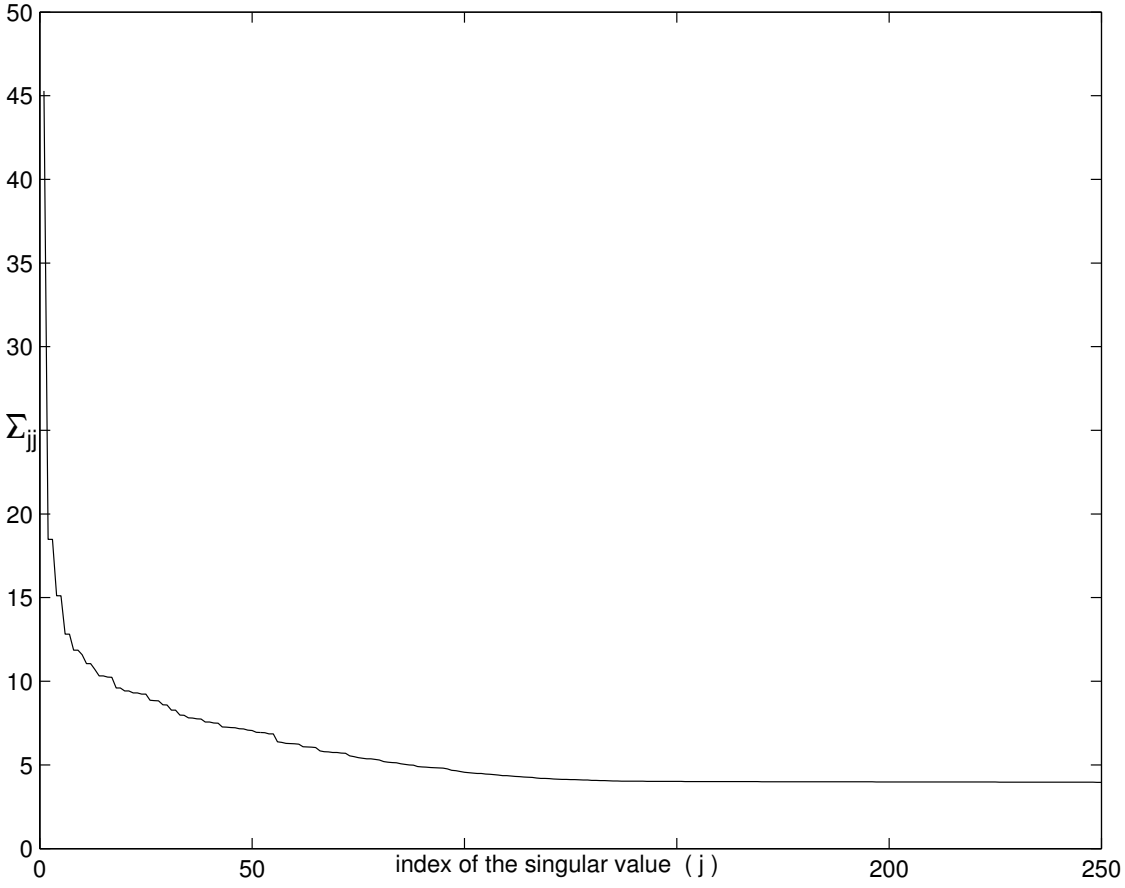


FIG. 3A. *Singular values computed for the E. coli data set.*

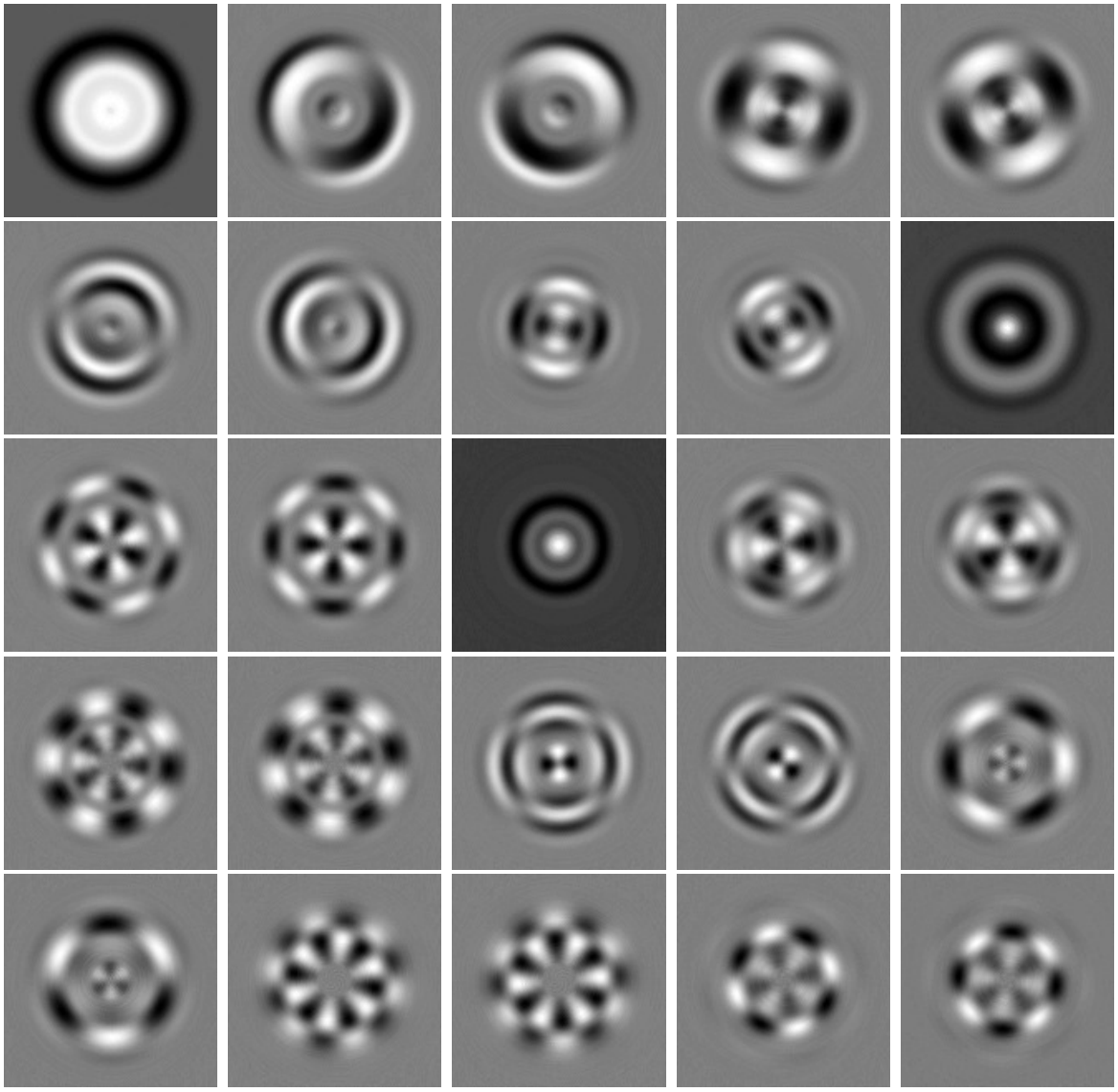


FIG. 3B. *Dominant singular vectors computed for the E. coli data set.*

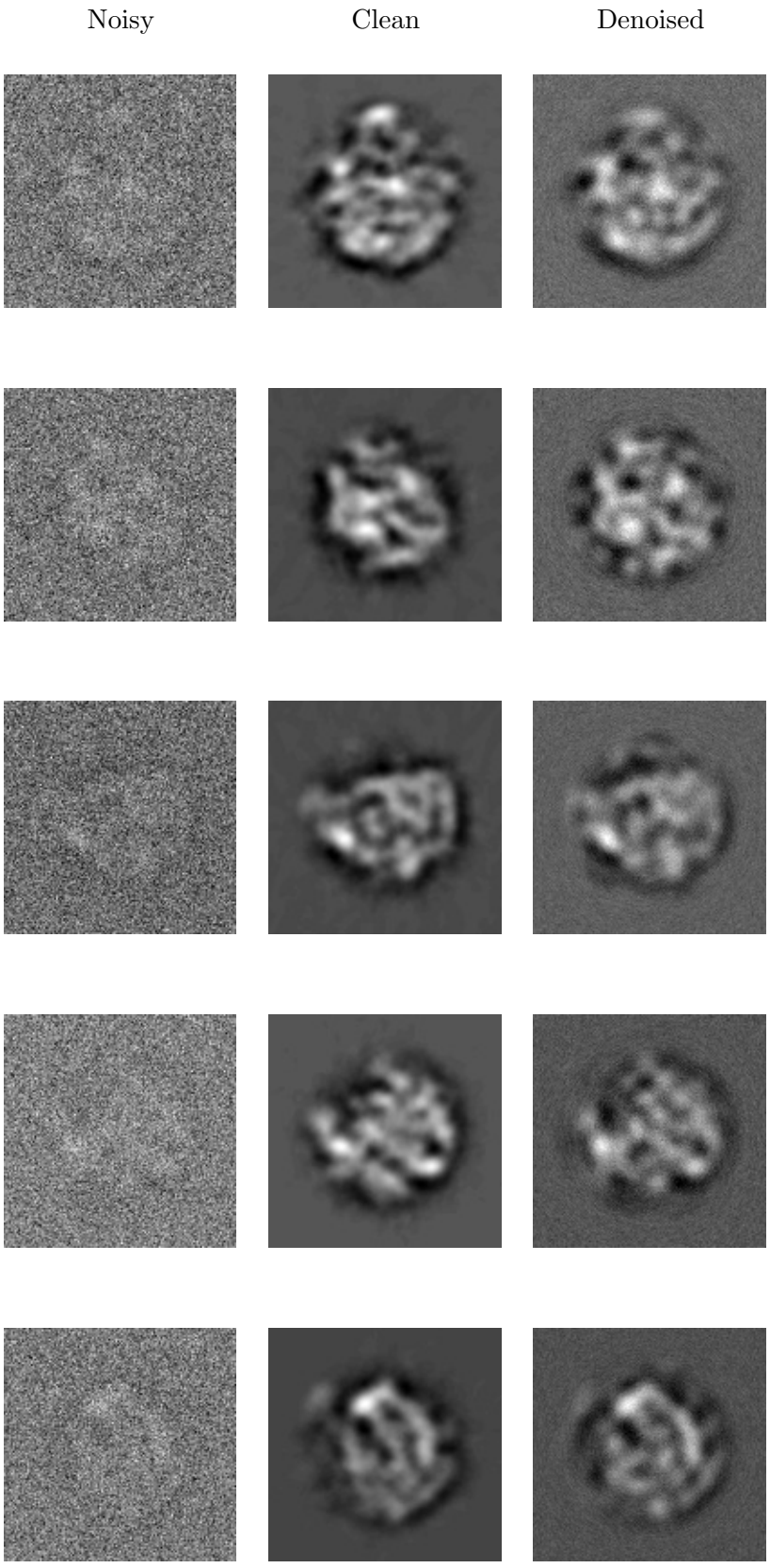


FIG. 3C. *Noisy, clean, and denoised images for the E. coli data set.*

3.8 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 approximation). 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 [6] (see also Section 3.4 of [14]).

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

$$p_{j,k}(D) = \max_{1 \le q \le k} \sqrt{\frac{\|(D^\top D)^j \, \omega^{(q)}\|_2}{\|(D^\top D)^{j-1} \, \omega^{(q)}\|_2}}, \tag{3.32}$$

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, [6]). 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

$$p_{j,k}(D) \ge \|D\|_2/2 \tag{3.33}$$

is greater than

$$1 - \left(\frac{2n}{(2j-1) \cdot 16^j} \right)^{k/2}. \tag{3.34}$$

The probability in (3.34) 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 (3.8), choosing $D = A - U\,\Sigma\,V^\top$, where A, U, Σ, and V are the matrices from (3.8). 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], [6], or Section 3.4 of [14].

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. Yoel Shkolnisky was supported in part by Israel Science Foundation grant 485/10. 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.

Bibliography

[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] J. Frank, *Three-dimensional electron microscopy of macromolecular assemblies: Visualization of biological molecules in their native state*, Oxford University Press, Oxford, UK, 2006.

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

[5] N. Halko, P.-G. Martinsson, and J. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), issue 2.

[6] 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.

[7] 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.

[8] P.-G. Martinsson, A. Szlam, and M. Tygert, *Normalized power iterations for the computation of SVD*, Proceedings of the Neural and Information Processing Systems (NIPS) Workshop on Low-Rank Methods for Large-Scale Machine Learning, Vancouver, Canada (2011), available at http://www.math.ucla.edu/∼aszlam/npisvdnipsshort.pdf.

[9] H. Moon and P. J. Phillips, *Computational and performance aspects of PCA-based face-recognition algorithms*, Perception, 30 (2001), pp. 303–321.

[10] 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.

[11] 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.

[12] C. Ponce and A. Singer, *Computing steerable principal components of a large set of images and their rotations*, IEEE Trans. Image Process., to appear. Available at http://math.princeton.edu/∼amits/publications/LargeSetPCA.pdf.

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

[14] 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.

Chapter 4

A randomized MapReduce algorithm for computing the singular value decomposition of large matrices

4.1 Introduction

Randomized algorithms for computing matrix decompositions have shown promise for processing massive data sets in distributed computing environments. By reorganizing the computations, they provide fast and accurate methods that require very few passes through the data and are therefore well suited for communication constrained environments. Single core, serial versions of the methods have been tested on large data sets Chapter 2 and *out-of-core* variants have proved successful on even larger data sets Chapter 3. We now take one step further by considering a fully distributed computing environment with multiple processors and distributed storage to continue the progression of processing ever larger data sets.

This chapter describes modifications of the algorithms in Chapter 2 necessary to adapt them to a distributed computing environment involving a large number of processors, distributed memory, and slow communication between nodes. The implementation leverages the Hadoop distributed computing framework and the MapReduce programming model. We provide a thorough discussion of this environment in order to understand settings and capabilities of the algorithm. A distributed orthogonalization scheme is described that uncouples memory requirements from both dimensions of the input matrix. We compare the randomized algorithm to a Lanczos solver also implemented in the same environment and show that randomized algorithms out perform classical techniques in terms of speed, accuracy and scalability. Finally we present a number of experiments with extremely large sparse matrices using Hadoop clusters on Amazon's Elastic Compute Cloud.

4.2 Problem Formulation

The problem we wish to solve is to compute approximate rank k singular value decompositions of extremely large matrices. In particular, matrices that are so large the approximate factors are even too large to fit into the RAM of a single computer. For matrix A, the problem is solved by constructing a suitable low dimensional space that approximates the range of A. We find an orthonormal matrix Q such that

$$\|\mathsf{A} - \mathsf{Q}\mathsf{Q}^T\mathsf{A}\| \approx \min_{\mathrm{rank}(\mathsf{X}) \leq k} \|\mathsf{A} - \mathsf{X}\| \tag{4.1}$$

In other words, the columns of Q form an orthonormal basis for the approximate range of A. Matrix Q must be constructed with distributed techniques since we assume it is too large to fit in memory. By partitioning the input matrix A into blocks of s rows, memory requirements are kept proportional to sk. To form the approximate rank k singular value decomposition, we project A onto the low dimensional subspace spanned by the columns of Q. From the projected matrix we form a small $O(k)$ dimensional matrix that fits comfortably in memory and classical methods are used to complete the factorization.

Randomized sampling techniques are used to construct the matrix Q. That is, given a target rank k and an oversampling parameter p, draw an $n \times k+p$ random matrix Ω and form the product $\mathsf{Y} = \mathsf{A}\Omega$. The range of Y closely approximates the range of A. Orthogonalizing the columns of Y produces the $m \times k+p$ matrix Q of equation (4.1). The key observation of randomized methods is that p does not need to be large to produce nearly optimal results.

4.3 Algorithm overview

The Stochastic Singular Value Decomposition is composed of Algorithms 4.4 and 5.1 of Chapter 2. Additionally, modifications are made to adapt the method to a distributed computing environment which we describe in §4.6 and §4.7.

Algorithm 4.3: Stochastic Singular Value Decomposition

Given an $m \times n$ matrix A, a target rank k, an oversampling parameter p, and a number of power iterations q, the following algorithm computes an approximate rank k singular value decomposition $\mathsf{A} \approx \mathsf{U}\Sigma\mathsf{V}^T$.

Draw an $n \times k + p$ random matrix Ω.
Form the product $\mathsf{Y} = \mathsf{A}\Omega$.
Orthogonalize the columns of $\mathsf{Y} \to \mathsf{Q}$.
for $i = 1..q$
    Form the product $\mathsf{Y} = \mathsf{A}\mathsf{A}^T\mathsf{Q}$.
    Orthogonalize the columns of $\mathsf{Y} \to \mathsf{Q}$.
end
Form the projection $\mathsf{B} = \mathsf{Q}^T\mathsf{A}$.
Compute the factorization $\tilde{\mathsf{U}}\Sigma^2\tilde{\mathsf{U}}^T = \mathsf{B}\mathsf{B}^T$.
Solve $\tilde{\mathsf{V}}^T = \Sigma^{-1}\tilde{\mathsf{U}}^T\mathsf{B}$.
Set $\mathsf{U} = \mathsf{Q}\tilde{\mathsf{U}}(:, 1{:}\, k)$.
Set $\mathsf{V} = \tilde{\mathsf{V}}(:, 1{:}\, k)$.

4.4 Large scale distributed computing environment

MapReduce $\mathcal{MR}$ is a functional programming abstraction. Google popularized its use in large scale distributed computing [8] and it is now widely available in open source as Hadoop. Its power comes from distilling the complexities of parallel programming down to two operations: a map $\mathcal{M}$ and a reduce $\mathcal{R}$. We provide some details of Hadoop and MapReduce to understand what is provided by the framework and the abstraction level we can use to design algorithms. First, we will cover the MapReduce programming model and how algorithms can be cast into these two simple operations. Then by confining operations and communication to the MapReduce model, we discuss how Hadoop is able to automatically take care of the details of the distributed computation.

4.4.1 MapReduce

MapReduce $\mathcal{MR}$ is a programming model. Let us separate this from Hadoop which is the implementation of the MapReduce programming model or Mahout for example which is a software library. It is a way to organize algorithms which gives control to the user to define the map and reduce functions while passing many of the details and complications of distributed computation to the framework to handle. A MapReduce job actually consists of three phases:

(1) map

(2) shuffle and sort

(3) reduce

with the complexities of (2) handled by the framework. This phase is also the only chance for global communication. Another important point is that the map function is split into potentially thousands of tasks performing the same function on small separate chunks of the data. The flow is then to operate locally on the data, communicate, then operate with a different function to complete the job. At this point the *map-reduce* terminology is not clear and we can instead think of the job as:

input data	$\mapsto$	(1) process phase	$\mapsto$	(2) communicate	$\mapsto$	(3) process phase	$\mapsto$	output data

The abstraction is of course not limited to just one job. Output of one MapReduce job becomes the input of another and in this way, $\mathcal{MR}$ jobs can be chained together to accommodate complex algorithms. The main restriction being that in order for the framework to handle communication, the user needs to define the algorithm in terms of:

local computation	$\mapsto$	communi- cation	$\mapsto$	local computation	$\mapsto$	communi- cation	$\mapsto$	etc ...

To give meaning to the terms map and reduce we consider the data format required. Data is always in a $< key, value >$ pair known as a record. For example, a corpus of documents could have each document keyed by a unique integer id and the value is the body of the document, or a matrix could be keyed by row number with each row as the value. The job of the map phase is then to assign intermediate keys to the data so that all the values associated with the same key are mapped together to the same reducer. The reducer is commonly used to aggregate data thus reducing many values associated with the same key down to one. Just as it is possible for algorithms to chain together many MapReduce jobs, we are also free to do simply a map only job to achieve a parallel batch processing task. For example, with the corpus of documents we could translate each one to Russian with each map task working a small subset of documents. No reducer or shuffling is needed here and the mappers would simply write the translated documents back to the filesystem.

A single MapReduce cycle is quite limiting. Not very many problems can be solved if we only allow one communication phase. Starting from the bottom up we define the organization of work that is capable of executing more complex algorithms.

Tasks The basic work unit. Computation inside a task is similar to computing on a single computer. In fact, a separate Java Virtual Machine (JVM) is used for each task so that the outside world (other tasks) are invisible to us inside the task. Here we do local computation. There are two types of tasks: map and reduce, and they differ in how the input data is presented to them. Generally, map tasks perform computation that is possible via a sequential read of the input data, and reduce tasks do computation on intermediate data or a reordering of the input data.

Jobs Basic operations. A job is composed of all the identical tasks operating on different chunks of data in parallel. As a programmer, one specifies the functionality of the job and the framework handles the details of executing and coordinating the tasks. Generally a job consists of both map and reduce stages, a MapReduce job, but there can be map only jobs as well.

Program *(Algorithm)* By putting together the basic operations and combining jobs we can build complex programs that have the desired functionality at a massive scale. Jobs can be executed in parallel or sequentially. The output of one job can be the input of another, or separate data can be processed by each job.

We now, as users, understand the contract to adhere to: we are provided with chunks of input data in *key-value* pairs, we provide a map function that operates locally and determines intermediate *key-value* pairs, we finally provide a reduce function that operates on all values associated with the same intermediate key. Following this model allows Hadoop to handle much of the burden of distributed computations and allows the user to focus on the functionality of their algorithms. Lets look at how Hadoop accomplishes this.

4.4.2 Hadoop

Hadoop is a framework for large scale distributed computing that handles the complexities of parallelization, fault tolerance, locality optimization and load balancing [2]. By following the

MapReduce contract outlined above the user is free to focus on functionality of the program while the framework handles the complexities of managing many machines working together. The confinement of communication to the shuffle and sort phase allows Hadoop to gracefully manage machine failures which is a common occurrence while orchestrating potentially thousands of computers.

A typical Hadoop cluster, as of 2010 [20], is composed of commodity computers similar to these specs:

Processor 2 quad-core 2-2.25 GHz CPUs

Memory 16-24 GB ECC RAM

Storage 4 $\times$ 1TB SATA disks

Network Gigabit Ethernet

Commodity does not mean cheap, rather just that the machines are not special purpose and readily available. These give the best performance for the price. They are arranged in a master slave relationship with the master coordinating tasks, data flow and scheduling of the slaves which perform the actual work. Two groups of master slaves are at play here.

The first group provides the locality optimization and load balancing characteristics of Hadoop. This group manages the location of the data which is stored locally on each machine. There is no central data store instead the data is spread out across the cluster. Each *datanode* (slave) communicates to the *namenode* (master) what data it is storing on its disk. When executing a program with input data spread across the cluster, the *namenode* can be queried to find out where the data resides and the map task can be executed on the *datanode* holding its chunk of the input data. This idea of bringing the computation to the data, and not the other way around, saves bandwidth which is the scarcest resource in a distributed environment.

The second group provides the parallelization and fault tolerant characteristics of Hadoop by managing the execution of a program. The *jobtracker* (master), equipped with the locations of the data from the *namenode*, assigns tasks to the *tasktrackers* (slaves). The *jobtracker* periodically pings each *tasktracker* for information about progress of the tasks it was assigned. If the *tasktracker* fails to respond or is slow due to other competing processes on the machine or faulty hardware, the task can be killed and rescheduled to another *tasktracker*. The actual computation of a map or reduce function is done by a child process spawned and managed by the *tasktracker*. It is important to note this step since each child process runs in its own separate JVM. Sharing information or variables between child processes is impossible which supports our earlier conclusion that the only communication available is during shuffle and sort and not possible in a map or reduce phase.

Both masters, the *namenode* and the *jobtracker* are located on the same machine while each slave node runs a *datanode*, *tasktracker* and both map and reduce child processes. The treatment of Hadoop and MapReduce given here is by no means comprehensive. A full understanding of the system requires hands on experience to gain intuition about how this complex system behaves, good starting references are the Hadoop wiki [2] and [20].

4.4.3 Mahout

Apache Mahout is an open source machine learning library [3]. Mahout utilizes the MapReduce programming model and Hadoop to implement scalable, distributed algorithms. Algorithms 4.4 and 5.1 of Chapter 2 (pages 26,29) were incorporated into Mahout 0.5 as the Stochastic Singular Value Decomposition, or ssvd. Also available in Mahout is a distributed Lanczos SVD algorithm described in §4.8. With both methods implemented in Mahout we have a prime environment to study characteristics of the algorithms on a distributed system.

4.5 A MapReduce matrix multiplication algorithm

As an introduction to MapReduce algorithms and for later use in ssvd and Lanczos algorithms, we present two matrix multiplication schemes. We make an effort in the formal descriptions of MapReduce algorithms to avoid task indexing as much as possible to keep an uncluttered presentation. The verbal descriptions accompanying the algorithms should clarify in more detail how they operate.

An important operation when dealing with Krylov methods and the ssvd is clearly the matrix vector product.

$$\mathbf{y} = \mathsf{A}\mathbf{x} \tag{4.2}$$

Equally important when decomposing non-symmetric matrices of irregular dimension is

$$\mathbf{y} = \mathsf{A}^T\mathsf{A}\mathbf{x} \tag{4.3}$$

Luckily both of these operations can be performed in MapReduce in a distributed manner with just a single pass over the input matrix A. A variety of ways to do these products is explained in [17], but we outline the one convenient to the way the data is stored. Matrices are stored in row major order where each row is a record keyed by row index and valued with the row vector

$$\begin{gathered} < 1, [a_{11}\ a_{12} \dots a_{1n}] > \\ < 2, [a_{21}\ a_{22} \dots a_{2n}] > \\ \vdots \\ < m, [a_{m1}\ a_{m2} \dots a_{mn}] > \end{gathered} \tag{4.4}$$

Records are stored in a binary sequence file. Mahout has further specialized matrix formats to optimize for sequential access to the elements and have a low memory footprint [18], much smaller than a native Java array structure. Hadoop then partitions the records (rows) into groups of a size determined by the input split, say 128MB, and computation for a particular group is brought to the data rather than the data being sent around the cluster on the network. Therefore, if we can restrict our matrix vector product to only use rows from a particular block at a time, then the computation will use local I/O and not burden the network, the scarcest resource. For equation (4.2) this is simple and requires only a map phase to complete. Assuming we have M input splits so that blocks of rows $\mathsf{A}_1, \mathsf{A}_2, \dots, \mathsf{A}_M$ are distributed around the cluster

$$\mathsf{A}\mathbf{x} = \begin{bmatrix} \mathsf{A}_1\mathbf{x} \\ \vdots \\ \mathsf{A}_i\mathbf{x} \\ \vdots \\ \mathsf{A}_M\mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_i \\ \vdots \\ \mathbf{y}_M \end{bmatrix} \tag{4.5}$$

Each map task operates only on one block of rows A_i and outputs the corresponding section of $\mathbf{y}$, $\mathbf{y}_i$. The full vector $\mathbf{x}$ needs to be provided to each of the M tasks. This will incur $O(Mn)$ network.

ALGORITHM 4.5: MATRIX MULTIPLICATION $\mathsf{A}\mathbf{x}$

This algorithm forms the product $\mathbf{y} = \mathsf{A}\mathbf{x}$ *assuming* A *is stored in row major format.*

Map
  Iterate A_{row}
    $\mathbf{y}_{row} = \langle \mathsf{A}_{row}, \mathbf{x} \rangle$
  output y

Equation (4.3) can also be done with one pass through A with the help of a reduce phase. Conveniently and by no accident, the row major format of A begs an outer product

$$\mathsf{C} = \sum_{i=1}^{M} \mathsf{A}_i^T \mathsf{A}_i \tag{4.6}$$

which requires access to only one row (or block) of A at a time. The product $\mathsf{C}\mathbf{x}$ is then a sum of column vectors

$$\begin{aligned} \mathsf{C}\mathbf{x} &= \textstyle\sum_{i=1}^{M} \mathsf{A}_i^T (\mathsf{A}_i \mathbf{x}) \\ &= \textstyle\sum_{i=1}^{M} \mathbf{v}^{(i)} \\ &= \mathbf{y} \end{aligned} \tag{4.7}$$

Each map task then computes a $\mathsf{v}^{(i)}$ using only its particular block of A, and outputs a key value pair

$$< j, \mathbf{v}_j^{(i)} > \tag{4.8}$$

which is the j^{th} entry of the partial sum vector $\mathbf{v}^{(i)}$. If R reducers are used, each reducer will receive all the partial sum entries for about $\frac{1}{R}$ rows so it can complete the sums and output the entries of $\mathbf{y}$. In Algorithm 4.5 we refer to $\mathbf{v}^{(i)}$ as $\mathbf{y}_{partial}$. In addition to the input vector $\mathbf{x}$ needing to be replicated to each node, we have another $O(Mn)$ transfer of data to shuffle and sort for the reducers. Of course we are not limited to $\mathbf{x}$ being a one dimensional vector. If $\mathbf{x} \in \mathbb{R}^{n \times k}$ is a tall skinny matrix, the procedure is the same except we pay extra data movement costs: $O(Mnk)$ for equation (4.2) and $2 \cdot O(Mnk)$ for equation (4.3). We still key by row number with the rows of the partial sums as values.

ALGORITHM 4.5: MATRIX MULTIPLICATION $\mathsf{A}^T\mathsf{A}\mathbf{x}$

This algorithm forms the product $\mathbf{y} = \mathsf{A}^T\mathsf{A}\mathbf{x}$ *assuming* A *is stored in row major format.*

Map
  Iterate A_{row}
    $\mathbf{y}_{partial} = \langle \mathsf{A}_{row}, \mathbf{x} \rangle \cdot \mathsf{A}_{row}^T$
  output $\mathbf{y}_{partial}$
Reduce
    $\mathbf{y} = \sum \mathbf{y}_{partial}$
  output $\mathbf{y}$

The notion of block matrix operations is only theoretical in Hadoop, at least for the operations at hand. Records (rows) are read into the mapper one by one so the block outer product of equation (4.7) is computed row by row using a combiner to compute the partial sums. If an entire block is needed at one time, the user code needs to allocate a buffer to hold the incoming rows.

Remark 29. *An optimization to eliminate data transfer in equation (4.2) is to generate* $\mathbf{x}$ *on the fly, or in our case, generate a random matrix* Ω *on the fly. To form a row of* Y *we sum outer products of row* A_i *and* Ω*.*

$$\begin{aligned} \mathsf{Y}(i,:) &= \mathsf{A}(i,:)\Omega \\ &= \textstyle\sum_j \mathsf{A}(i,j)\Omega(j,:) \end{aligned} \tag{4.9}$$

If A *is sparse, then the sum can be substantially smaller than* n *terms.* ***The elements of*** Ω ***are computed individually given a seed term and the*** (i,j) ***index of*** Ω***.***

4.6 Distributed Orthogonalization

The two major operations of the ssvd are matrix multiplication as described in §4.5 and the orthogonalization of $\mathsf{Y} = \mathsf{A}\Omega$. The following section will describe the distributed orthogonalization scheme found in [14, 16].

4.6.1 Givens Rotations

A Givens rotation $G(i,k,\theta)$ is an orthogonal transformation of matrix Y effecting just rows (or columns) i and k of a matrix

$$\begin{array}{cc} G^T\mathsf{Y} & \mathsf{Y}G \\ rows & columns \end{array} \tag{4.10}$$

Given rotations are attractive for distributed orthogonalization because of this property since it allows one to operate on distinct chunks of a matrix in parallel. For any element y_{kp} in row Y_k we can zero it out by choosing θ such that

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}^T \begin{bmatrix} y_{ip} \\ y_{kp} \end{bmatrix} = \begin{bmatrix} \tilde{y}_{ip} \\ 0 \end{bmatrix} \tag{4.11}$$

In practice, θ is not computed in favor of the values $c = \cos\theta$ and $s = \sin\theta$. The transformation is applied to both rows, independent of the other rows of Y, giving a linear combination

$$\begin{array}{rrcl} & \tilde{\mathsf{Y}}_i &=& c\mathsf{Y}_i - s\mathsf{Y}_k \\ & \tilde{\mathsf{Y}}_k &=& s\mathsf{Y}_i + c\mathsf{Y}_k \\ \text{where ...} & & & \\ & \tilde{\mathsf{Y}}_k &=& [\tilde{y}_{k1}, \ldots, 0_{kp}, \ldots, \tilde{y}_{kn}] \end{array} \tag{4.12}$$

It is easy to see that application of a Givens rotation takes only time linear to the number of elements in the row. A key observation of equation (4.12) is if both rows share leading zeros, then they are preserved by the linear combinations in the new vectors. This property admits a Cantor-esque pattern of obtaining a QR factorization. The rotations are of the form $G^T(i-1, i, \theta)$ so always two adjacent rows are operated on at once.

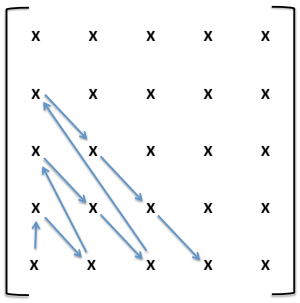


Figure 4.1: Cantor pattern

The process of Givens QR orthogonalization produces a sequence of rotations G_j^T that we can accumulate on the identity matrix to get Q^T. Of course for a tall skinny $r \times \ell$ matrix Y we will want the thin QR factorization and avoid ever having the large $r \times r$ matrix Q in memory. We can do this by accumulating on a slice of the identity one of two ways

$$\mathsf{Q}_{thin}^T = [\mathsf{I}_{\ell\times\ell} \quad 0_{r-\ell\times\ell}] \cdot G_N^T \cdots G_2^T G_1^T \tag{4.13}$$

or

$$\begin{array}{rcl} \mathsf{Q}^T &=& G_N^T \cdots G_2^T G_1^T \cdot \mathsf{I}_{r\times r} \\ \mathsf{Q}_{thin}^T &=& \left(\mathsf{Q}\left(:, 1:\ell\right)\right)^T \end{array} \tag{4.14}$$

Equation (4.13) can form the thin Q directly but requires that the rotations be applied in reverse order so that they need to be saved and applied at the end. Equation (4.14) can be done on the fly to avoid storing all the rotations. It is a bit misleading that it looks like we need to intermediately form the full dense Q. Using the pattern of Figure 4.1, equation (4.14) can be done in $O(r\ell)$ storage.

4.6.2 Overview

Before we launch into the details of the orthogonalization scheme, it is instructive to first take a high level and concise view and point out some key features. The method will be made precise in the sections that follow. There are two schemes at play: the *streaming* QR algorithm forms the initial factorizations of blocks of Y, secondly, a *merge* phase glues together the individual factorizations to provide a thin QR factorization of the entire matrix Y. The granularity is determined by 3 factors

r the size of the initial factorizations.

s the number of rows in the input split handed to each mapper.

M number of map tasks.

The number of rows s handed to each map task must be at least r so that $s > r$. Assume that $s = 3r$. Within each map task, the *streaming* algorithm computes thin QR factorizations of sub blocks of Y_i

$$\begin{bmatrix} \mathsf{Y}_i^{(1)} \\ \mathsf{Y}_i^{(2)} \\ \mathsf{Y}_i^{(3)} \end{bmatrix} = \begin{bmatrix} \mathsf{Q}_1\mathsf{R}_1 \\ \mathsf{Q}_2\mathsf{R}_2 \\ \mathsf{Q}_3\mathsf{R}_3 \end{bmatrix} = \begin{bmatrix} \mathsf{Q}_1 & & \\ & \mathsf{Q}_2 & \\ & & \mathsf{Q}_3 \end{bmatrix}^{s\times 3\ell} \begin{bmatrix} \mathsf{R}_1 \\ \mathsf{R}_2 \\ \mathsf{R}_3 \end{bmatrix}^{3\ell\times\ell} \tag{4.15}$$

The *merge* algorithm then glues together the 3 pieces in the map task to form the thin factorization for Y_i.

$$\begin{bmatrix} \mathsf{Q}_1 & \leftarrow & \leftarrow \\ \leftarrow & \mathsf{Q}_2 & \leftarrow \\ \leftarrow & \leftarrow & \mathsf{Q}_3 \end{bmatrix} \begin{bmatrix} \mathsf{R}_1 & \\ \mathsf{R}_2 & \uparrow \\ \mathsf{R}_3 & \uparrow \end{bmatrix} \Longrightarrow \begin{bmatrix} \\ \mathsf{Q} \\ \\ \end{bmatrix}^{s\times\ell} \begin{bmatrix} \mathsf{R} \end{bmatrix}^{\ell\times\ell} \tag{4.16}$$

The output of each map task is a thin QR factorization of a block Y_i. The merge is performed again with these factorizations to glue together the final factorization of Y. To recap

(1) In each mapper, the *streaming* algorithm produces $z = \lfloor \frac{s}{r} \rfloor$ factorizations of order r.

(2) The order r factorizations in each mapper are glued together by *merge* to form a factorization of order s.

(3) The M order s factorizations from each mapper are glued together by another *merge* to form the final factorization of order m.

The trick to all of this is that each of the Q's in equation (4.16) can be migrated to the left independently of each other so that we only need to have one Q in memory at a time. We now describe in detail the *streaming* and *merge* algorithms.

4.6.3 Streaming QR

The streaming QR algorithm takes place inside each map task. For simplicity we drop the subscript that assigns blocks of Y to each mapper and just refer to block Y_i as Y. In §4.5 we pointed out that the product $\mathsf{Y} = \mathsf{A}\Omega$ is actually done row by row so that rows of Y become available one at a time. The streaming QR capitalizes on this and starts factorizing as soon as rows become available. One caveat is the order they become available. Following the Cantor pattern of Figure 4.1 we would start with the last row first. The remedy is to do the factorization 'upside down'. The order of rows only effects Q,

$$\mathsf{Q}(j,:)\mathsf{R} = \mathsf{Y}(j,:) \tag{4.17}$$

R is independent of row ordering. Taking this into consideration, we simply reverse the row ordering of Q when we are finished. For simplicity, assume that rows of Y are given in 'correct' order from last to first.

The key to the streaming QR is a sliding buffer of height $\ell + 1$. Assume we have collected $\ell + 1$ rows of Y and have zeroed out all entries below the diagonal leaving a row of zeros in the bottom of R.

$$\begin{bmatrix} \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & \mathrm{x} & \mathrm{x} \\ 0 & 0 & \mathrm{x} \\ 0 & 0 & 0 \end{bmatrix} = \left[\begin{array}{c} \mathsf{R}^{\ell\times\ell} \\ \hline \mathbf{0}^{1\times\ell} \end{array}\right] \tag{4.18}$$

As the next row of Y is computed, append it to the top and discard the bottom row of zeros. Continue the Cantor pattern by zeroing out the sub-diagonal.

$$\begin{bmatrix} \mathrm{y} & \mathrm{y} & \mathrm{y} \\ \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & \mathrm{x} & \mathrm{x} \\ 0 & 0 & \mathrm{x} \end{bmatrix} \rightarrow \begin{bmatrix} \mathrm{x} & \mathrm{x} & \mathrm{x} \\ \mathbf{0} & \mathrm{x} & \mathrm{x} \\ 0 & \mathbf{0} & \mathrm{x} \\ 0 & 0 & \mathbf{0} \end{bmatrix} \tag{4.19}$$

Following equation (4.14), as each Givens rotation is computed and applied to Y, we also accumulate Q^T by applying each rotation to the identity matrix of order r. To avoid collecting a dense $r \times r$ matrix, just as we discard the bottom zero row of the buffer, we can also discard the bottom row of Q^T. This is justified since the thin factorization will contain only the first ℓ rows of Q^T (leading ℓ columns of Q), and the discarded row would not have been involved in any more rotations. Additionally, since no rotation has touched rows above the buffer, the upper part of Q^T need not be carried around since it is still just the identity. In this way we keep memory requirements for Q^T to $O(r\ell)$.

The computation continues until r rows have been processed. Q and R are flushed to disk and a new factorization is started. There are $z = \lfloor \frac{s}{r} \rfloor$ such factorizations output from streaming QR

$$\underset{r \times \ell}{\mathsf{Q}} \quad \underset{\ell \times \ell}{\mathsf{R}} \quad = \quad \underset{r \times \ell}{\mathsf{Y}} \tag{4.20}$$

4.6.4 Merging factorizations

The merging algorithm described here will be applied twice to obtain a final QR factorization of the full matrix Y. First, it is applied to the results of the streaming QR. Here the z order r factorizations are merged into one order s factorization per mapper. Then a second pass merges each of the M order s factorizations into a final order m QR factorization of Y.

4.6.4.1 Merge R

The basic building block is the merging of two R's by pushing the mass of the lower block up into the top block, creating a block of zeros that can be discarded.

$$\begin{bmatrix} \mathsf{R}_1 \\ \mathsf{R}_2 \uparrow \end{bmatrix} \mapsto \begin{bmatrix} \mathsf{R} \\ 0 \end{bmatrix} \tag{4.21}$$

The merge follows the Cantor pattern on the bottom R_2, and traces along the diagonal in the top R_1.

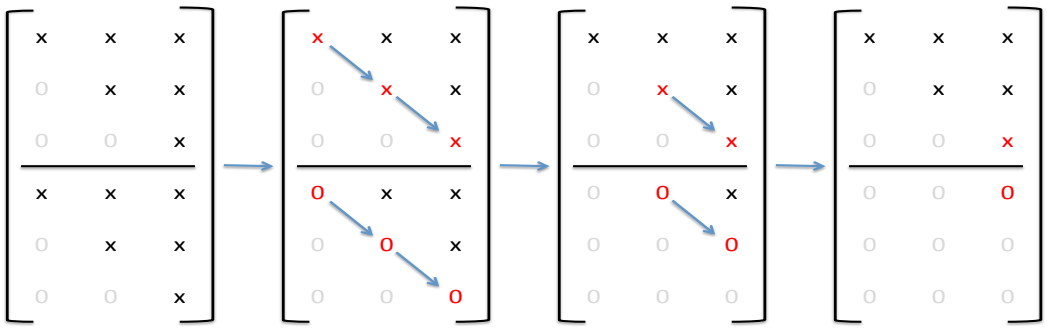


Figure 4.2: R merge sequence

4.6.4.2 Merge Q^T

Conceptually, it is best to consider accumulating Q^T so we can do it in a row-wise fashion similar to R. (In practice, rows of Q are accumulated to facilitate the outer product $\mathsf{Q}^T\mathsf{A}$.) We would like to apply the same sequence of rotations obtained from equation (4.21) to the corresponding Q matrix

$$\begin{bmatrix} \mathsf{Q}_1^T & \\ & \mathsf{Q}_2^T \end{bmatrix} \rightarrow \begin{bmatrix} \hat{\mathsf{Q}}_1^T & \hat{\mathsf{Q}}_2^T \\ \mathsf{X} & \mathsf{X} \end{bmatrix} \tag{4.22}$$

Unfortunately, this quadruples the $O(r\ell)$ memory requirement. Luckily we can side step the bloated memory by applying the sequence twice to the two Q's independently, once to get $\hat{\mathsf{Q}}_1^T$ and again to get $\hat{\mathsf{Q}}_2^T$. Assume the sequence obtained in equation (4.21) is G_{seq}^T, then

$$G_{seq}^T \begin{bmatrix} \mathsf{Q}_1^T \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{\mathsf{Q}}_1^T \\ \mathsf{X} \end{bmatrix} \quad \text{and} \quad G_{seq}^T \begin{bmatrix} 0 \\ \mathsf{Q}_2^T \end{bmatrix} = \begin{bmatrix} \hat{\mathsf{Q}}_2^T \\ \mathsf{X} \end{bmatrix} \tag{4.23}$$

which produce the same Q hat's as in equation (4.22) using only $2 \cdot O(r\ell)$ memory. Consider the case of merging 3 factorizations which would naively require 9 times the memory. The algorithm takes pairs so that first we merge R_2 into R_1 to form $\tilde{\mathsf{R}}$, then merge R_3 into $\tilde{\mathsf{R}}$.

$$\underset{G_{seq1}^T}{\begin{bmatrix} \mathsf{R}_1 \\ \mathsf{R}_2 \uparrow \end{bmatrix}} \to \begin{bmatrix} \tilde{\mathsf{R}} \\ 0 \end{bmatrix} \quad \text{then} \quad \underset{G_{seq2}^T}{\begin{bmatrix} \tilde{\mathsf{R}} \\ \mathsf{R}_3 \uparrow \end{bmatrix}} \to \begin{bmatrix} \mathsf{R} \\ 0 \end{bmatrix} \tag{4.24}$$

To compute $\hat{\mathsf{Q}}$'s we apply the sequence $G_{seq2}^T G_{seq1}^T$ to the corresponding rows. For example, $\hat{\mathsf{Q}}_2^T$ is computed by

$$G_{seq1}^T \begin{bmatrix} 0 \\ \mathsf{Q}_2^T \end{bmatrix} \to \begin{bmatrix} \tilde{\mathsf{Q}} \\ \mathsf{X} \end{bmatrix} \quad \text{then} \quad G_{seq2}^T \begin{bmatrix} \tilde{\mathsf{Q}} \\ 0 \end{bmatrix} \to \begin{bmatrix} \hat{\mathsf{Q}}_2^T \\ \mathsf{X} \end{bmatrix} \tag{4.25}$$

To compute $\hat{\mathsf{Q}}_3^T$, notice that G_{seq1}^T will be applied to a block of zeros

$$G_{seq1}^T \begin{bmatrix} 0 \\ 0 \end{bmatrix} \to \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{then} \quad G_{seq2}^T \begin{bmatrix} 0 \\ \mathsf{Q}_3^T \end{bmatrix} \to \begin{bmatrix} \hat{\mathsf{Q}}_3^T \\ \mathsf{X} \end{bmatrix} \tag{4.26}$$

Here we can skip applying G_{seq1}^T and merge Q_3^T directly from the third block to the top. To compute $\hat{\mathsf{Q}}_1^T$ notice again we have a double block of zeros. If our sequence of Givens rotations we formed from merging the R's in a different order, we could avoid applying both sequences and apply only the second one. For clarity, redefine the G_{seq}^T's as

$$\underset{G_{seq1}^T}{\begin{bmatrix} \mathsf{R}_2 \\ \mathsf{R}_3 \uparrow \end{bmatrix}} \to \begin{bmatrix} \tilde{\mathsf{R}} \\ 0 \end{bmatrix} \quad \text{then} \quad \underset{G_{seq2}^T}{\begin{bmatrix} \mathsf{R}_1 \\ \tilde{\mathsf{R}} \uparrow \end{bmatrix}} \to \begin{bmatrix} \mathsf{R} \\ 0 \end{bmatrix} \tag{4.27}$$

Now we can compute $\hat{\mathsf{Q}}_1^T$ as

$$G_{seq2}^T \begin{bmatrix} \mathsf{Q}_1^T \\ 0 \end{bmatrix} \to \begin{bmatrix} \hat{\mathsf{Q}}_1^T \\ \mathsf{X} \end{bmatrix} \tag{4.28}$$

Finally, consider merging many factorizations together. We take 7 for a small example but this process could easily extend to 100's of blocks. When computing $\hat{\mathsf{Q}}_4^T$ we will have blocks of zeros on the top and bottom. If we merge $\mathsf{R}_2, \mathsf{R}_3$ into R_1 and likewise merge $\mathsf{R}_6, \mathsf{R}_7$ into R_5, then we only need apply two sequences of rotations to Q_4^T to obtain $\hat{\mathsf{Q}}_4^T$.

There is a further optimization that saves us re-merging the top R's. Consider computing $\hat{\mathsf{Q}}_5^T$. Following the pattern in equation (4.3) we would merge $\mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4$ into R_1 to create $\mathsf{R}^\sharp$. But we already computed the merge of $\mathsf{R}_2, \mathsf{R}_3$ into R_1. Therefore, we can take the previous $\mathsf{R}^\sharp$, call it $\mathsf{R}_{(4)}^\sharp$ and create $\mathsf{R}_{(5)}^\sharp$ by simply merging in R_4.

$$\mathsf{R}_{(i)}^\sharp = \mathsf{R}_{(i-1)}^\sharp \leftarrow \mathsf{R}_{i-1} \tag{4.29}$$

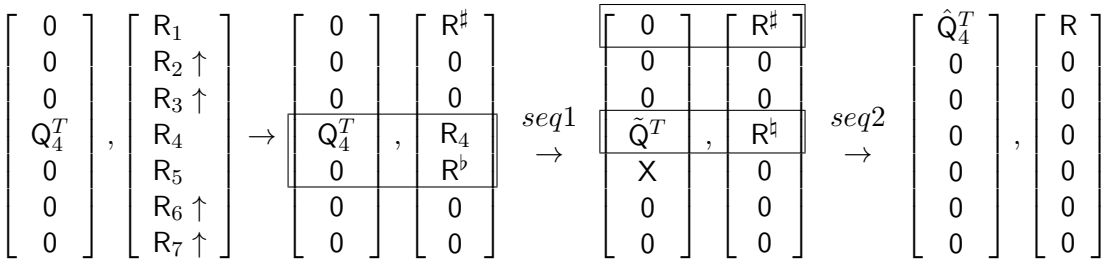


Figure 4.3: Q merge

A similar optimization can be done with $\mathsf{R}^\flat$ though it is not done in practice. The R_i's reside in an iterator that feeds them one by one in order $\mathsf{R}_1, \mathsf{R}_2, \ldots$. If we had access to them in reverse order, we would compute $\mathsf{R}^\flat$ by merging the R's into R_7 in reverse order.

$$\mathsf{R}^\flat_{(1)} = \mathsf{R}_7 \leftarrow \mathsf{R}_6, \mathsf{R}_5, \ldots, \mathsf{R}_2 \tag{4.30}$$

To compute $\mathsf{R}^\flat_{(2)}$ we can just *undo* the merge of R_2. In general,

$$\mathsf{R}^\flat_{(i)} = \mathsf{R}^\flat_{(i-1)} \quad \textit{undo} \quad \mathsf{R}_i \tag{4.31}$$

To undo the merge, if

$$\ldots G_3^T G_2^T G_1^T \tag{4.32}$$

is the sequence that merged R_i, then

$$G_1 G_2 G_3 \ldots \tag{4.33}$$

will *undo* the merge. This requires that we save the rotations in factored form, which up until this point has not been necessary since we can also recompute the rotations each time we need them.

4.6.5 Analysis

As pointed out in §4.6.2, the streaming QR and the first phase merge are done in serial. Since there is extra work involved in the merge we will want to examine the cost and benefits of doing this two phase approach in serial as opposed to just using the streaming QR to construct the full order s factorization. We first discuss a direct factorization of the full $s \times \ell$ matrix, then we discuss the non-optimized merge and finally the optimizations available to speed this up.

Application of a Givens rotation involves just two rows and can be done in time linear to the number of elements in each row. Computing each c and s is done in constant time. For a tall skinny $s \times \ell$ matrix, $O(\ell^2 s + \ell s)$ flops are required to compute R. Accumulating Q we apply $\sim \ell s$ rotations to an $s \times s$ identity requiring $O(\ell s^2)$ flops. The same conditions apply to the streaming QR except on smaller order r matrices. The catch of course is that we do the process z times.

To merge two R's as in equation (4.21) requires $O(\ell^3)$ flops and to merge Q^T as in equation (4.28), or rather *migrate* a Q^T upwards, requires $O(\ell^2 r)$ flops. The total for one merge step

method	flops	memory
direct	$O(\ell s^2 + \ell^2 s)$	$O(\ell s)$
streaming	$z \cdot O(\ell r^2 + \ell^2 r)$	$O(\ell r)$
naive	$z \cdot O(z\ell^2 r + z\ell^3)$	$O(\ell r)$
$\mathsf{R}^\sharp$	$z \cdot O(\frac{z}{2}\ell^2 r + \frac{z}{2}\ell^3)$	$O(\ell r)$
$\mathsf{R}^\sharp$, $\mathsf{R}^\flat$	$z \cdot O(2\ell^2 r + 4\ell^3)$	$O(\ell s)$
distributed	$O(\frac{z}{2}\ell^2 r + z\ell^3)$	$O(\ell r)$

Table 4.1: Flops and memory requirements for QR methods. s is the number of rows per input spit, r is blocksize (parameter `-r`), $\ell = k + p$ and $z = \lfloor \frac{s}{r} \rfloor$ is the number of blocks. For the second phase merge scheme set $s \leftarrow r$ and $M \leftarrow z$.

then involves $O(\ell^2 r + \ell^3)$ flops. A naive merge scheme would *not* involve pre-merging any R's before migrating Q^T. For each of the z blocks we pay $O(z\ell^3)$ flops for merging R. To migrate Q^T then involves on average $z/2$ migrations for a total of $O(z\ell^2 r)$ flops. Again we need to repeat this for each of the z blocks giving $z \cdot O(z\ell^2 r + z\ell^3)$ flops for the total merge.

The optimizations in equations (4.29) and (4.31) provide some speed-up of the naive merge scheme. Recursively forming $\mathsf{R}^\sharp$ as in (4.29) cuts out a factor of 2 but unfortunately does not change the order of the method. We still need $z \cdot z/2$ total merges. If we also form $\mathsf{R}^\flat$ via equation (4.31) we get the scheme in Figure 4.3 (with the exception that $\mathsf{R}^\sharp, \mathsf{R}^\flat$ are computed recursively at step (1) instead of from scratch as it appears in Figure 4.3). This scheme requires 4 merges to form $\mathsf{R}^\flat, \mathsf{R}^\sharp, \mathsf{R}^\natural$ and R, and 2 merges to form $\tilde{\mathsf{Q}}^T$ and $\hat{\mathsf{Q}}^T$ for each of the z blocks. Table 4.1 summarizes the cost of the methods discussed.

The direct method and the streaming QR plus naive merge have the same asymptotic scaling if done in serial, $O(\ell s^2)$. (To see this take $z = r/s$ and streaming plus naive from Table 4.1. Let $r \nearrow s$ or $r \searrow \ell$.) Intuitively, each method needs to zero $s\ell - \ell^2/2$ elements in matrix Y and so long as we do not 'un-zero' a number that has already been zeroed, the only difference in methods is the pattern of zeroing entries. Equality of methods assumes that rotations are computed once to zero entries in Y and stored costing $O(s\ell)$ memory, then applied to form Q. To avoid using extra memory we can recompute the sequence for each merge costing an additional $O(\frac{s^2\ell^3}{r^2})$. Assuming $r \sim \ell^{\frac{3}{2}}$, recomputing the rotations nearly doubles the work. The $\mathsf{R}^\sharp$ optimization cuts the entire cost in half, both the expense of recomputing rotations and the cost of migrating Q's upward bringing the methods back on equal footing flops-wise. Note the merge scheme still has the advantage of much reduced memory. The $\mathsf{R}^\sharp, \mathsf{R}^\flat$ optimization cuts the quadratic dependence on s, the largest dimension, to give $O(s\ell^2)$. It is not a total panacea though. Assuming a serial computation we can recursively accumulate $\mathsf{R}^\sharp$ without extra memory. But for $\mathsf{R}^\flat$ we need access to all the rotations so they can be *undone* one by one, thus increasing memory cost. Finally, in a distributed environment we cannot even recursively compute $\mathsf{R}^\sharp$. However we can form $\mathsf{R}^\sharp$ and $\mathsf{R}^\flat$ from scratch as in Figure 4.3 and thus migrate Q only a constant number of times. Either way, a carefully implemented merge scheme is linear in the largest dimension s which is much better than a direct QR computation.

4.6.6 Q-less optimizations

A downside of the above scheme is recomputing and applying the rotations necessary to form Q explicitly. Although, numerical experiments indicate that the orthogonalization process takes a very small percentage of total execution time. Ultimately, we need to form the matrix B^T which we can do by simulating the action of Q via

$$\begin{aligned} \mathsf{B}^T &= \mathsf{A}^T\mathsf{Q} \\ &= \mathsf{A}^T\left(\mathsf{Y}\mathsf{R}^{-1}\right) \end{aligned} \tag{4.34}$$

The Tall and Skinny QR algorithm of [6] presents a very similar technique to what we have described in §4.6, with the exception that Q is not formed explicitly.

Another option is to compute the Cholesky factors of the symmetric matrix $\mathsf{Y}^T\mathsf{Y} = \mathsf{R}^T\mathsf{R}$. Partial sums $\left(\mathsf{Y}_i^T\mathsf{Y}_i\right)$ can be formed in a distributed fashion and sent to a sinlge reducer for summing. The R factor is formed in the reducer and is the same factor of the QR decomposition of Y. Accuracy issues can arise in truely low-rank situations since $\sigma\left(\mathsf{Y}^T\mathsf{Y}\right) = \sigma^2\left(\mathsf{Y}\right)$. If Y captures all the action of a low-rank matrix, then the trailing singular values will be small and become even smaller when squared. In practical situations this is of little concern. Most large matrices are very noisy with slowly decaying singular values. Methods that use Q-less optimizations are currently under development in Mahout.

4.7 SSVD MapReduce Implementation

We now describe the Stochastic Singular Value Decomposition algorithm, *ssvd*, in term of its constituent MapReduce jobs. This section closely follows the work of [12, 15, 13].

Algorithm 4.7: Stochastic Singular Value Decomposition

The ssvd algorithm produces rank k matrices $\mathsf{U}, \mathsf{V}, \Sigma$ that form an approximate singular value decomposition of matrix A.

Q-job
Bt-job
for $i = 1..q$
  ABt-job
  Bt-job
end for
Serial step:
  compute $\tilde{\mathsf{U}}\Sigma^2\tilde{\mathsf{U}}^T = \mathsf{B}\mathsf{B}^T$
U-job
V-job

4.7.1 Q-job

Q-job is a map only job that performs the multiplication, $\mathsf{A}\Omega$, and does the first two phases of orthogonalization. Input is a block of s rows A_i and output is a thin QR factorization of the corresponding block Y_i. The number of rows s provided each mapper can vary if the density pattern is non uniform across the rows. Recall, the input splits are determined by size in megabytes. The rows are fed in one by one and multiplied by Ω to give a row of Y_i as in Remark 29. As rows of Y_i become available they are sent to the streaming QR algorithm of §4.6.3 where an order r factorization is computed from the sub-block $\mathsf{Y}_i^{(j)}$ for $j = 1, \ldots, z$. Once the z factorizations have been computed, the merge algorithm of §4.6.4 glues them together to output a thin QR decomposition for Y_i.

ALGORITHM 4.7.1: Q-JOB

This algorithm forms the product $\mathsf{Y} = \mathsf{A}\Omega$, *performs the streaming* QR *factorization and the first level merge.*

Map
  Iterate A_{row}
    $\mathsf{Y}_{row} = \mathsf{A}_{row} \cdot \Omega$
    $\mathsf{Y}_{row} \rightarrow streaming\mathsf{QR}$
  output $\mathsf{Q}_i^{r\times\ell}$ and $\mathsf{R}_i^{\ell\times\ell}$.
  $merge\{\mathsf{Q}_i, \mathsf{R}_i\}_{i=1}^{z}$
  output $\mathsf{Q}^{s\times\ell}\mathsf{R}^{\ell\times\ell}$

Though we described in §4.6.4.2 how to accumulated rows Q^T, we actually want to accumulate rows of Q here. The purpose being to set up the projection step in the next job with conveniently oriented vectors. Matrices are always written row-wise so to obtain columns of a matrix without incurring a transpose step, write the matrix as rows of the transpose.

4.7.1.1 Cost and memory requirements

Each of the M map tasks performs the following work:

(m1)	matrix multiply $\mathsf{A}_{rows} \cdot \Omega$	$O(sn\ell)$
(m2)	z streaming QR	$z \cdot O(\ell r^2 + \ell^2 r)$
(m3)	z merges with $\mathsf{R}^\sharp$ optimization	$z \cdot O(\frac{z}{2}\ell^2 r + \frac{z}{2}\ell^3)$

If A is sparse then the dominant cost will be merging the dense blocks of intermediate Q matrices in (m3). Also as ℓ grows, cost of recomputing and applying Givens rotations to the R matrices becomes a large factor. Memory requirements are merely $O(r\ell)$ independent of either dimension of the matrix or the input split size s. Rows of the input matrix are used one by one so we avoid having the full block of input in memory.

4.7.2 Bt-job

Bt-job has a map and reduce phase. The front end of the map job finalizes the orthogonalization started in Q-job. Let $\tilde{\mathsf{Q}}_i$, $\tilde{\mathsf{R}}_i$ be the output of map task i from Q-job. Then for map task i in Bt-job, we read $\tilde{\mathsf{Q}}_i$ and $\tilde{\mathsf{R}}_1, \ldots, \tilde{\mathsf{R}}_M$ from disk in order to form the final block of Q. Now we begin to read in the rows of matrix A to form $\mathsf{A}^T\mathsf{Q} = \mathsf{B}^T$ via outer product. Partial sums are sent in blocks to the reducers to be summed up. If this Bt-job is the final phase, meaning that there are no power iterations or that this is the last iteration of them, we can output the upper triangular partial sums of $\left(\mathsf{B}\mathsf{B}^T\right)_{partial}$ using only the block B_i^T in memory. These partial sums are collected and summed during the serial step.

ALGORITHM 4.7.2: BT-JOB

This algorithm completes the second level merge and computes the product $\mathsf{B}^T = \mathsf{A}^T\mathsf{Q}$. *Optionally, partial sums of* $\mathsf{B}\mathsf{B}^T$ *are formed and output to disk.*

Map
- $\mathsf{Q}_i \leftarrow$ merge $\tilde{\mathsf{Q}}_i, \tilde{\mathsf{R}}_1, \ldots, \tilde{\mathsf{R}}_M$
- **output** Q_i as block of final Q.
- **Iterate** A_{row}
  - $\mathsf{B}^T_{partial} = (\mathsf{A}_{row})^T \cdot \mathsf{Q}_{row}$
- **output** $\mathsf{B}^T_{partial}$

Reduce
- $\mathsf{B}^T = \sum \mathsf{B}^T_{partial}$
- ***Option***
  - $\left(\mathsf{B}\mathsf{B}^T\right)_{partial} = \mathsf{B}_i\mathsf{B}_i^T$

4.7.2.1 Cost and memory requirements

Each of the M map tasks and R reduce tasks perform the following work:

(m1)	M merges with partial $\mathsf{R}^\sharp$ optimization	$M \cdot O(\frac{1}{2}\ell^2 s + \ell^3)$
(m2)	projection $(\mathsf{A}_{rows})^T \cdot \mathsf{Q}$	$O(sn\ell)$
(r1)	sum outer products	$M \cdot O(\frac{n}{R}\ell)$

The $\mathsf{R}^\sharp$ optimization is a bit different than the one done in Q-job. We can save merging Q blocks only $\frac{M}{2}$ times by premerging $\mathsf{R}^\sharp$, but we cannot share $\mathsf{R}^\sharp$ between tasks so that the full R merge needs to be done in each task. Memory requirements are $O(\max\{s, \texttt{-oh}\} \cdot \ell)$. Chunks of B^T are written to disk by each reducer. Use equation (4.50) to determine the size of each chunk for use in ABt-job.

4.7.3 ABt-job

ABt-job has much the same functionality as Q-job. We need to do a matrix multiplication $\mathsf{AB}^T = \mathsf{AA}^T\mathsf{Q}$ and orthogonalize the result. Unfortunately it is more complicated than Q-job since we cannot produce B^T on the fly as we could with Ω, instead, B^T is stored on disk. To limit access to B^T the rows of A are preloaded into memory so that we can use the partial columns, that is, columns of the row blocks A_i, for outer products and read the sequence of B^T blocks only once per mapper. In Q-job, our blocks Y_i were available in the mapper. With ABt-job, we need to accumulate partial sums of Y_i in the reducer so we bring the streaming QR and first level merge computation there as well.

$$\mathsf{A}_{block}\mathsf{B}^T \quad = \quad \sum_j \mathsf{A}_{block}(:,j)\mathsf{B}^T(j,:) \tag{4.35}$$

ALGORITHM 4.7.3: ABT-JOB

This algorithm computes the product AB^T*, performs the streaming* QR *factorization and the first level merge.*

Map
- **Iterate** A_{row}
  - $\mathsf{A}_{block}(i,:) = \mathsf{A}_{row}$
- **foreach** B^T_{row}
  - $\mathsf{Y}_{partial} = \mathsf{A}_{block}(:,j) \cdot \mathsf{B}^T_{row}$
- **output** $\mathsf{Y}_{partial}$

Reduce
- $\mathsf{Y} = \sum \mathsf{Y}_{partial}$
- $\mathsf{Y} \rightarrow streaming\mathsf{QR}$
- **output** $\mathsf{Q}_i^{r\times\ell}$ and $\mathsf{R}_i^{\ell\times\ell}$
- $merge\{\mathsf{Q}_i, \mathsf{R}_i\}_{i=1}^{z}$
- **output** $\mathsf{Q}^{s\times\ell}\mathsf{R}^{\ell\times\ell}$

4.7.3.1 Cost and memory requirements

Since A_{block} is preloaded into memory, the memory requirement is the size of the input split plus a block of B^T. Use equation (4.50) to determine this size in Bt-job. §4.7.1.1 describes the costs of the job with the exception that (m2) and (m3) are performed in the reducer.

4.7.4 U-job

U-job is a simple map only job that computes the final rank k factor U of the singular value decomposition. The rank ℓ factorization

$$\tilde{\mathsf{U}}\Sigma\tilde{\mathsf{U}}^T = \mathsf{B}\mathsf{B}^T \tag{4.36}$$

was computed in core so that the small $\ell \times \ell$ factor $\tilde{\mathsf{U}}$ fits comfortably in memory. Redefine $\tilde{\mathsf{U}}$ by extracting the first k columns

$$\tilde{\mathsf{U}} \leftarrow \tilde{\mathsf{U}}(:, 1\colon k) \tag{4.37}$$

ALGORITHM 4.7.4: U-JOB

This algorithm computes the rank k factor U of the singular value decomposition

Map
Iterate Q_{row}
$\mathsf{U}_{row} = \mathsf{Q}_{row}\tilde{\mathsf{U}}$
output U_{row}

4.7.5 V-job

V-job is again a map only job forming the rows of the rank k factor V

$$\mathsf{V}(i, :) = (\mathsf{B}(:, i))^T \tilde{\mathsf{U}}\Sigma^{-1} \tag{4.38}$$

where $\tilde{\mathsf{U}}$ is as in (4.37) and Σ is the uppermost leftmost block of k singular values. Recall that B is stored columnwise as rows of B^T.

ALGORITHM 4.7.5: V-JOB

This algorithm computes the rank k factor V of the singular value decomposition

Map
Iterate B^T_{row}
$\mathsf{V}_{row} = \mathsf{B}^T_{row}\tilde{\mathsf{U}}\Sigma^{-1}$
output V_{row}

4.8 Lanczos

The Lanczos method [11] is very popular for solving large, sparse, symmetric eigenproblems. In addition it is also used on non-symmetric matrices to find the singular value decomposition since the SVD is closely related to the solution of the eigenproblem on the symmetric matrix $\mathsf{A}^T\mathsf{A}$. The defining characteristics of the method, and those that make it so popular, are (1) the input matrix A is only accessed via a matrix vector multiply $\mathsf{A}^T\mathsf{A}x$. This eliminates random access requirements to the elements of A and does not produce *fill-in* during computation. And (2), the method converges quickly so that a subset of the eigenpairs can be accurately generated without computing the full decomposition.

Unfortunately, numerical stability issues are also a prevalent feature of the algorithm and care must be taken to avoid loss of orthogonality among the Krylov basis vectors. Mahout's solution is to compute a basis of size k as requested by the user. No restart or re-orthogonalization is used. Instead, there is a seperate routine to verify the accuracy of the computed eigen pairs and only keep the ones deemed to have converged. We only study the construction of the k sized basis and the resulting eigenpairs, foregoing the verification process. Just like the ssvd, we understand that the tail end of the computed spectrum will loose accuracy and need to be discarded. The method and implementation we present here describes the svd implementation in Mahout.

4.8.1 Description of method

The Krylov subspace for a symmetric $n \times n$ matrix A is

$$\mathcal{K}(\mathsf{A}, \mathbf{q}, n) = \left[\mathbf{q},\ \mathsf{A}\mathbf{q},\ \mathsf{A}^2\mathbf{q},\ \ldots\ ,\ \mathsf{A}^{n-1}\mathbf{q}\right] \tag{4.39}$$

The vector $\mathbf{q}$ is usually chosen to be in the range of A so that $\mathbf{q} = \mathsf{A}\boldsymbol{\omega}$ with $\boldsymbol{\omega}$ a random vector or a constant vector of unit norm for example. Remarkably, the QR factorization (non-pivoted) of the Krylov space gives orthonormal matrix Q such that $\mathsf{Q}(:,1) = \mathbf{q}/\|\mathbf{q}\|$ and

$$\mathsf{AQ} = \mathsf{QT} \tag{4.40}$$

where T is an order n symmetric, tridiagonal matrix. Directly equating the elements in equation (4.40) we find

$$\mathsf{A}\mathbf{q}_j = \beta_{j-1}\mathbf{q}_{j-1} + \alpha_j\mathbf{q}_j + \beta_j\mathbf{q}_{j+1} \tag{4.41}$$

where α's are diagonal elements of T and β's are both sub and super diagonals. The Lanczos procedure follows from repeatedly solving this three term recursion to obtain the elements α, β in T and the columns of the orthonormal matrix Q. For practical purposes we needn't solve all the way to n to obtain information of interest. Stopping computation at a $k < n$ gives a k dimensional symmetric, tridiagonal matrix T_k that is similar to A projected into the truncated Krylov space. The eigenvalues of T_k closely approximate the extremal eigenvalues of A and can be very efficiently extracted. The eigen-decomposition $\mathsf{T}_k = \mathsf{X}\Lambda\mathsf{X}^T$ also yields approximate eigenvectors of A via

$$\underset{n \times k}{\mathsf{V}} \quad = \quad \underset{n \times k}{\mathsf{Q}} \quad \underset{k \times k}{\mathsf{X}} \tag{4.42}$$

4.8.2 Implementation

The three term recursion in equation (4.41) is quite elegant, requiring only 3 dense vectors to be held in memory at one time. Unfortunately, we loose orthogonality among the $\mathbf{q}$'s in finite precision arithmetic. A simple solution is forego the recursion and orthogonalize $\mathbf{q}_{j+1}$ against all previous basis vectors $\mathbf{q}_i$ for $i = 1, \ldots, j$. Letting $\mathsf{Q}_j = \begin{bmatrix}\mathbf{q}_1 & \ldots & \mathbf{q}_j\end{bmatrix}$ be the basis computed by the j^{th} step , we induct $\mathbf{q}_{j+1}$ into the basis as follows

$$\begin{array}{rcl} \mathbf{q} & = & \left(I - \mathsf{Q}_j\mathsf{Q}_j^T\right)\mathsf{A}\mathbf{q}_j \\ \eta & = & \|\mathbf{q}\| \\ \mathbf{q}_{j+1} & = & \mathbf{q}/\eta \end{array} \tag{4.43}$$

It can easily be the case, especially dealing with sparse matrices, that the $O(kn)$ storage of the dense basis vectors is large. To avoid holding the entire basis Q in memory consider,

$$\begin{array}{rcl} \mathbf{y} & = & \mathsf{A}\mathbf{q}_j \\ \left(I - \mathsf{Q}_j\mathsf{Q}_j^T\right)y & = & y - \sum_{i=1}^{j}\langle\mathbf{q}_i, y\rangle\mathbf{q}_i \end{array} \tag{4.44}$$

Each basis vector $\mathbf{q}_i$ can be read in one by one from disk to limit memory requirements. Again we are back to only 3 dense vectors in memory: y, $\mathbf{q}_i$ and a vector $\mathbf{q}$ to accumulate the sum

$$\begin{array}{rcl} \mathbf{q} & = & y \\ \mathbf{q} & \leftarrow & \mathbf{q} - \langle\mathbf{q}_i, y\rangle\mathbf{q}_i \end{array} \tag{4.45}$$

Full orthogonalization costs $O(kn)$ flops and perhaps more importantly requires access to the disk. However, Figure 4.15 on page 178 suggests the time required to do this is surprisingly small.

A problem still exists in this method. The vector $\mathbf{q}$ in equation (4.44) is orthogonal to within machine precision ε of the rest of the basis. That is

$$\|\mathsf{Q}_j^T\mathbf{q}\| < \varepsilon \tag{4.46}$$

Substituting $\mathbf{q} = \eta \cdot \mathbf{q}_{j+1}$ we find that $\eta \ll 1$ can amplify the error

$$\|\mathsf{Q}_j^T\mathbf{q}_{j+1}\| < \varepsilon/\eta \tag{4.47}$$

A strategy described in [7] and employed in ARPACK and Matlab [19] is to monitor η. If $\eta < 1/\sqrt{2}$ or some given threshold, then set $\mathbf{y} = \mathbf{q}_{j+1}$ and repeat the projection in equation (4.44). These implementations assume that Q is in memory and thus pay only the flop penalty for the extra reorthogonalization step. Mahout, however, is geared toward larger problems and thus assumes Q to be stored on disk (actually both disk storage, hdfsBackedLanczosState, and an in-memory, LanczosState are available). Re-projection is not done, instead the algorithm detects if parameters get beyond a safe limit and stops computation with a warning message.

The orthogonalization described takes place as a serial process on one node. The part of the algorithm that is parallelized is the interaction with the matrix A. Since we are computing the singular value decomposition, the matrix product is

$$\mathbf{y} = \mathsf{A}^T\mathsf{A}\mathbf{q}_j \tag{4.48}$$

Notice this will compute the right singular vectors of A. The matrix product $\mathsf{A}\mathsf{A}^T$ is not as easily computed as $\mathsf{A}^T\mathsf{A}$ in §4.5.

When the iterative computation has come to a halt we compute the eigen-decomposition $\mathsf{T}_k = \mathsf{X}\Lambda\mathsf{X}^T$. To obtain the right singular vectors of A from X use equation (4.42). The dense $O(kn)$ matrix is again avoided in memory by sweeping through the vectors of Q k times

$$\mathsf{V}(:,j) = \sum_{i=1}^{k} \mathsf{X}(i,j)\mathsf{Q}(:,i), \quad \text{for } j = 1,\ldots,k \tag{4.49}$$

ALGORITHM 4.8.2: LANCZOS SVD

Given an $m \times n$ matrix A and a desired rank k, Lanczos SVD computes the k dimensional eigen-decomposition, $\mathsf{V}\Sigma^2\mathsf{V}^T$, of $\mathsf{A}^T\mathsf{A}$ which yields the right singular vectors V and singular values Σ of A.

$\mathbf{q} = \mathsf{A}^T\mathsf{A}\boldsymbol{\omega}$
$\mathbf{q}_1 = \mathbf{q}/\|\mathbf{q}\|$
while $i \leq k$ **do**
  MapReduce step: Algorithm 4.5
    $\mathbf{q} = \mathsf{A}^T\mathsf{A}\mathbf{q}_i$
  Serial step:
    for $j = 1..i$
      $\mathbf{q} \leftarrow \mathbf{q} - \langle \mathbf{q}_j, \mathbf{q} \rangle \mathbf{q}_j$
      $\mathbf{q}_{i+1} = \mathbf{q}/\|\mathbf{q}\|$
      collect: α, β
    end for
end while
compute $\mathsf{X}\Lambda\mathsf{X}^T = T$
$\mathsf{V} = \mathsf{Q}\mathsf{X}$
$\sigma_i = \sqrt{\lambda_i}$

4.8.3 Scalability

Lanczos method and its related variations, Arnoldi and unsymmetric Lanczos, are extremely efficient and well suited for some computing environments. Specifically, the ideal situation would be

(1) A subset of $k \ll n$ eigenpairs are desired as opposed to the full decomposition.

(2) The input matrix is sparse or admits a fast matrix vector product.

(3) The input is of large enough dimension that we cannot afford intermediate full matrices of $O(n)$.

(4) The input and resulting $O(kn)$ dense basis is small enough to fit in RAM.

Of the 4 conditions, the restriction (4) of how large we can get is the most limiting. We discussed a way to scale up by storing the basis vectors on disk, although this comes with the penalty of persistent I/O pressure as can be seen in Figure 4.4a. Furthermore we can lift (4) by distributing the matrix vector product. With this expansion of the limiting factor, namely scale, we get an incredibly powerful method capable of solving large problems in a distributed environment. To compete with Lanczos method, we must show that ssvd can produce comparable accuracy in less time and scale better and beyond in a distributed environment.

Several factors limit the scalability of the Lanczos method. First, Lanczos is an iterative method and hence a serial computation. Though parts can be distributed and parallelized such as the matrix vector product, the algorithm depends on results from previous iterations to proceed and thus is a serial computation. At each iteration, only a small amount of information is obtained and so requires at least k passes over the data.

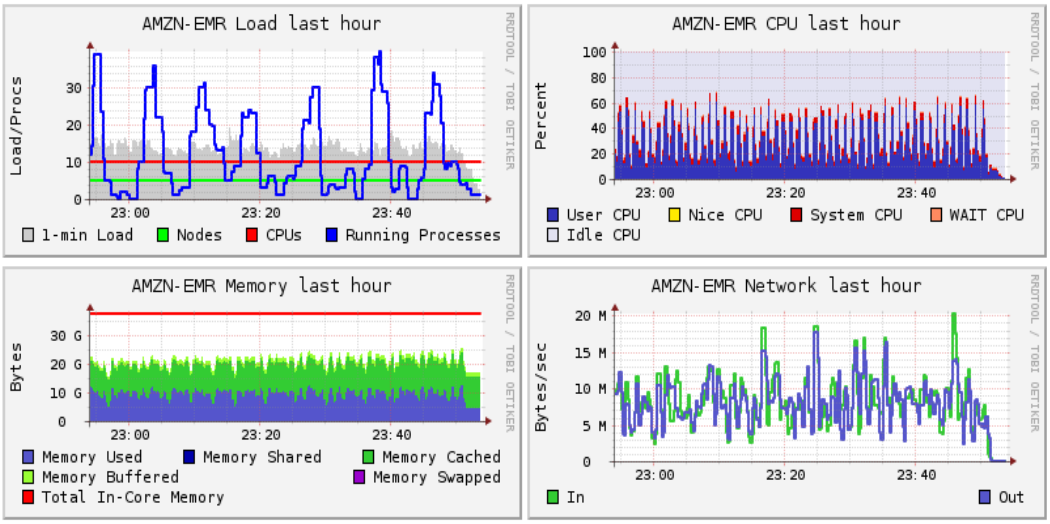
In addition, the algorithm was designed to minimize flops which is no longer the scarcest resource in a computing environment. With distributed computing, slow data transfer over the network requires communication avoiding schemes that perhaps do extra computation in order to limit data transfer.

4.9 SSVD parameters

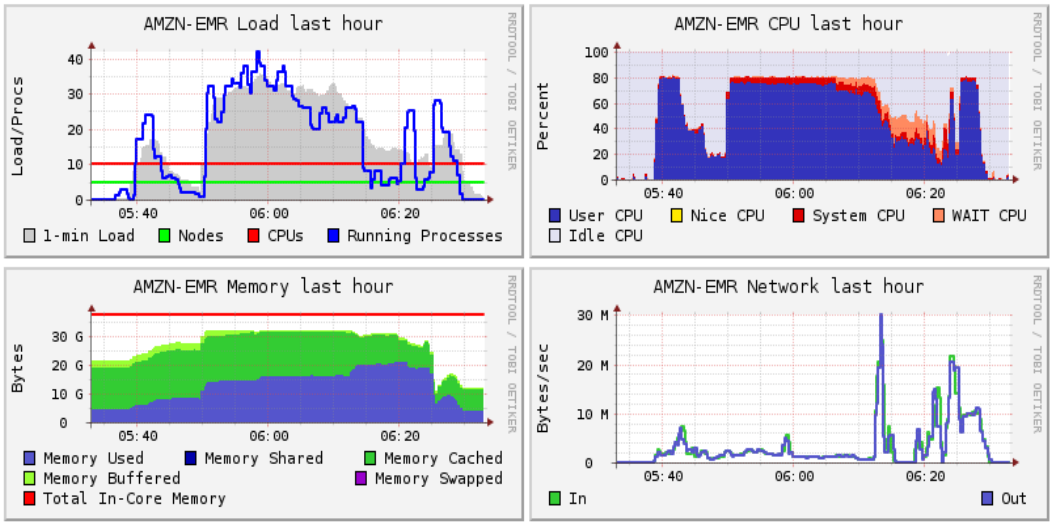
We list the usage of the important parameters in Table 4.2 in two groups. The first define the mathematical requirements while the second define the computational parameters and help us optimize the computation. Rank k, oversampling parameter p and power iteration q are given thorough treatment in Chapter 2. Figure 4.5 graphically represents the computational parameters. More information about these are found in §4.12.

4.10 The data

Our dataset is a *term-frequency document-frequency* (tf-df) matrix of Wikipedia articles, that is a matrix whose rows represent documents and whose entries are frequencies of word occurrences within the document. The raw articles, *enwiki-latest-pages-articles.xml* downloaded here [5], are a recent dump of millions of Wikipedia articles taking 32GB of space (7.2GB compressed .bz2 download). Mahout has utilities specifically to parse this data set from raw .xml files to a matrix format required by the algorithm. First we use the `seqwiki` command that converts the raw xml



(a) Ganglia screenshot during svd on Wikipedia-million data set. Each spike is an iteration involving a matrix vector multiply. Map tasks take only $20 - 30$ seconds to complete much of which is set up and IO costs. Tasks should take at least 1 minute to complete to offset these costs. This job failed at the end and thus did not complete the factorization. Note the persistent pressure on the network.



(b) Ganglia screenshot from ssvd of Wikipedia-million data set. Notice the network activity near the end of the job is computation of dense matrices U and V.

Figure 4.4: Ganglia screenshots showing network and cpu characteristics of the Lanczos and ssvd algorithms.

files to sequence files. Sequence files are key value pair binary files keyed with a document id and valued with the text of the document. Next we use `seq2sparse` that parses the sequence files into document vectors again keyed by document id and valued with a sparse row vector whose entries are the frequency of each word in the document. We create three matrices from the articles,

parameter	default	description
–rank (-k)	none	decomposition rank
–oversampling (-p)	15	oversampling
–powerIter (-q)	0	number of additional power iterations
–blockHeight (-r)	10,000	Y block height (must be $> (k+p)$)
–outerProdBlockHeight (-oh)	30,000	block height of outer products during multiplication, increase for sparse input
–abtBlockHeight (-abth)	200,000	block height of Y_i in ABtJob during AB^T multiplication, increase for extremely sparse inputs
–reduceTasks (-t)	1	number of reduce tasks (where applicable)
–minSplitSize (-s)	-1	minimum split size

Table 4.2: Important parameters of the Mahout implementation of ssvd. The first group defines the mathematical parameters for the algorithm, while the second group defines computational settings.

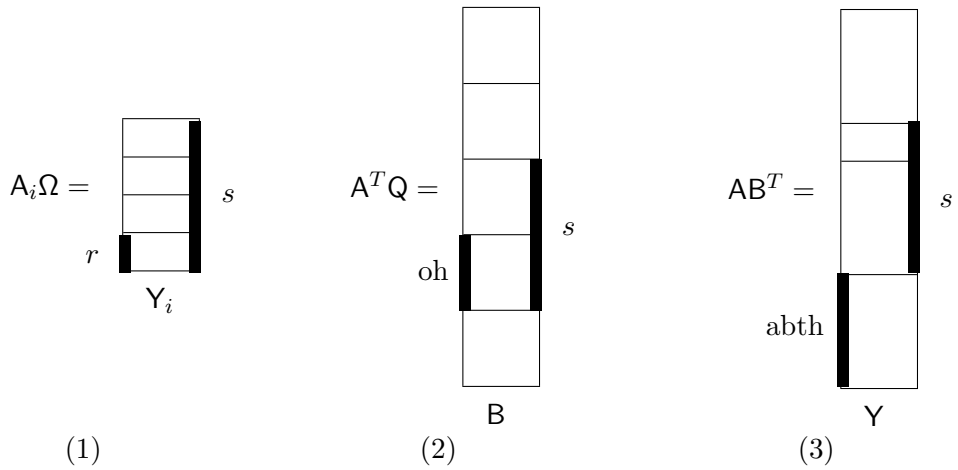


Figure 4.5: Computational parameters. s is the number of rows in an input split and this value can widely vary in sparse matrices. Manually set parameters will not evenly divide the split and there will be blocks of different sizes as in (3).

(1) a subset of 1,042,976 articles.

(2) the full corpus of 6,404,775 articles.

(3) the full corpus replicated 6 times to produce a large square matrix.

Notice the storage required for the matrix is about half original size of the corpus.

data set	documents m	terms n	size	density
wiki-million	1,042,976	976,239	1.9GB	0.01%
wiki-all	6,404,775	37,293,795	15GB	0.00056%
wiki-MAX!	38,428,650	37,293,795	90GB	0.00056%

Table 4.3: Sparse *term-frequency – document-frequency* (tf-df) matrices.

Remark 30. *For practical purposes, a dictionary of size* $> 10^6$ *is not very useful. Many of the 'terms' or tokens represent anomalies in the text such as sequences of special characters or non-normalized formatting. A treatment is given in [18] to various parameters of the parsing algorithm* `seq2sparse` *that prune away high or low frequency words for example, making the tf-df matrices much smaller and more useful.*

4.11 Machines

4.11.1 Amazon EC2 and EMR

Amazon's Elastic Compute Cloud (EC2) is a publicly available cluster computing environment [1] that provides scalable computational resources on a pay as go basis. Elastic MapReduce (EMR) is Hadoop running on EC2 instances. This type of cloud environment is a good choice for research purposes for a number of reasons. It is very cheap compared to the option of an onsite dedicated cluster. Maintainance and upkeep is done by the provider and thus does not take time away from research goals. The on-demand, pay-as-you-go pricing structure accommodates the sporadic usage needs of research. The elasticity accommodates fluctuation in problem size without having under utilized (wasted) resources. These characteristics can be found with other cloud providers, but Amazon's widespread adoption means there is a large community of users that supports itself by sharing information and experience. Finally, for benchmarking algorithms, Amazon EC2 provides a standardized environment valuable to providing unbiased metrics.

4.11.2 Instance Types

Amazon EC2 provides different instance types depending on computational needs. There are six families including a high performance computing type and a cluster GPU type, each with various configurations within them to further customize as per application. Multiple instances within a type and multiple instance types can be spun up together to create a cluster. We choose from the standard instance type which is suited for general purpose computing. Since this study is intended to benchmark the algorithm on standard computing resources, this is a good choice over the more specialized instances. Instances are normalized into a measure of compute units. One EC2 compute unit provides cpu resources equivalent to a 2007, 1.2 GHz Xeon processor. We first

	m1.large	m1.xlarge
Compute Units	4	8
Virtualized cores	2	4
RAM	7.5 GB	15 GB
storage	850 GB	1690 GB
JVM heap	1.6 GB	1.6 GB
map slots	4	8
reduce slots	2	4

Table 4.4: Specifications for two types of Amazon EC2 instance types.

examine specs of the two instance types we choose within the standard family. Table 4.4 lists these machine specifications that will later help us choose optimal parameters for the algorithm.

Essentially, the m1.xlarge instance is twice the machine of the m1.large. Both have high I/O performance and are 64-bit Linux platforms. The JVM heap is allotted per task so this parameter is the same between these two instances, the distinction being that the m1.xlarge can accommodate twice as many concurrent tasks as the m1.large. Based on these specifications we could use two m1.large's for every one m1.xlarge to build an equally powerful cluster, however, the advantage to the m1.xlarge is data locality since there is physically a denser distribution of data on half as many machines. We explore this idea further in the experiments.

4.11.3 Virtualization and shared resources.

It is important to note that cpu, memory and hard disk storage are dedicated to your instance, but network is not, it is a shared resource. Each instance on a physical device is guaranteed a minimum amount of network, but if there is surplus, other instances may use more than their share. Therefore, a positive characteristic of an algorithm will be minimal network use since this potentially can be used elsewhere by other processes. If network usage cannot be avoided, it is better to have spikes of usage rather than a constant load. Spiking network use can use excess network when needed and free up network when not needed. In contrast, the dedicated resources like cpu will lie idle if not used and cannot be shared. We would like our algorithm to then maximize cpu use and minimize or spike network use.

4.11.4 Choosing a cluster

With the input data specified and the instance types narrowed down we can start to decide how to partition the data and how many resources we will need. The basic unit of parallelism is the task (map or reduce) and is determined by the input split. The input split, measured in bytes, partitions the data into chunks (default 128MB) that become the input of each map task. The split is careful not to cut across records so to each mapper is delivered a subset of rows of the matrix of

approximately the input split size. Since our data matrix is sparse we will see a large variation in the number of rows each map task receives and consequently, a variation in the size of the dense intermediate calculations. This can cause slowdowns since the job is not complete until all the tasks have finished. If an exceptionally sparse chunk of rows is processed near the end of the job, the task will take a long time to process meanwhile the rest of the machines are waiting idle.

data set	split MB	maps	s	memory MB
wiki-all	32	491	13,000	14
	64	280	26,000	27
	128	123	52,000	53
wiki-million	32	60	18,000	18
	64	30	35,000	36
	128	15	70,000	71

Table 4.5: Calculations based on size of input data to help determine appropriate input split and cluster sizes.

The numbers in Table 4.5 are approximations to the following:

$$\begin{aligned} \text{number of maps} &= \frac{\text{size of input}}{\text{size of split}} \\ s &= \frac{\text{number of rows } m}{\text{number of maps}} \\ \text{memory} &= \text{size of } O(s\ell) \text{ dense subproblems, } \ell = 125 \end{aligned}$$

The number of rows per split s can vary greatly when dealing with sparse matrices. For example, using a 128MB split on the wiki-million data set produces splits that have as few as 20,000 rows to as many as 165,000 rows. Keeping in mind the variability of s we choose an input split that keeps the size of the dense subproblems and s at 'reasonable' levels, where reasonable is a fuzzy term. Consider if our wiki-million matrix was dense. To achieve a subproblem of about 70MB and 15 maps as we have with a 128MB split, the dense matrix would need a split of 500GB! This is beyond large for an input split in Hadoop and we would favor many more maps to divide the computation via a much smaller split.

Once we decide a good level of parallelism for our job, we consider the number of maps and the number of map slots on a chosen machine. For example, a 128MB split on the wiki-million matrix gives 15 maps and an m1.large instance can process 4 map tasks concurrently. Choosing a cluster of $4 + 1$ m1.large's gives enough map slots to process all the data at the same time. By having an equal number of tasks and slots, the faster tasks ($s < 50{,}000$) will process quickly and the machines will be waiting idle while the longest task ($s = 165{,}000$ for example) finishes. This is not the most efficient use of resources, but for a given input split size, it is the fastest. Using more machines will just increase idle resources and not provide any benefits. Using less slots than

tasks provides and interesting optimization. We still have a lower bound on runtime of the longest running task, however, faster tasks can be executed back to back in the same slot. Figure 4.6 shows an idealized scenerio. Of course it may happen that task 3 is executed in slot 1 while task 2 and task 4 are in slot 2. They will both finish at the same time leaving task 1 to execute all by itself. Now the running time is say $t_{task3} + t_{task1}$, but at least only one slot is sitting idle and less resources are being used.

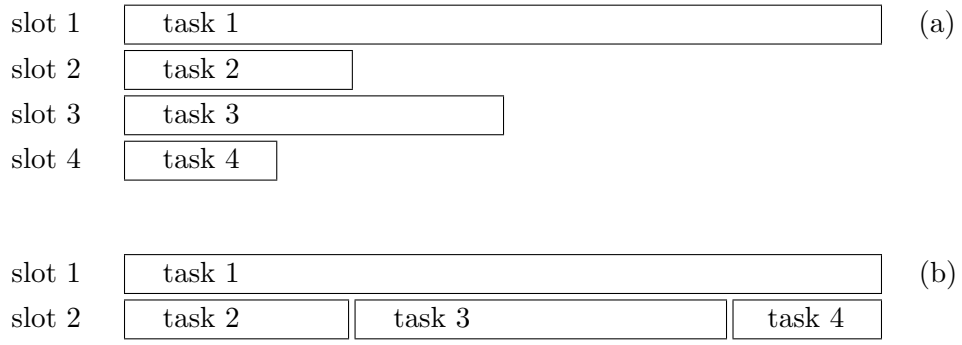


Figure 4.6: (a) represents number of tasks equal to number of slots. (b) shows having less slots than tasks can better utilize resources and not sacrifice performance. Running time is represented by the length of the boxes for each task.

4.11.5 Monitoring

It is essential to have some kind of monitoring available when working with MapReduce in Hadoop environment. We want to be able to monitor the progress of the job, see statistics and details about data flow, and assess the health and utilization of the cluster. The two presented here are web user interfaces that provide metrics on the remote cluster via ssh tunneling. The Hadoop UI's provide access to the tasktrackers, jobtracker, namenode and datanodes. Here we can view log files, job statistics and browse the hdfs file system. This UI comes standard as a part of Hadoop. Ganglia [4] is a scalable distributed monitoring tool for high performance computing systems that provides machine specific details about the cluster such as cpu and memory use.

The Hadoop jobtracker UI provides details about workflow and general job statistics. Figure 4.7 is the front page listing jobs that have been run on the cluster. Clicking on the hyperlink for a particular job brings you to Figure 4.8. Here we see much more detailed information about the job. Job progress is broken out into map and reduce jobs that are pending, running, complete and failed or killed. Clicking the links takes you to the tasktracker (not shown) to which the task is assigned where you can view more detailed information. The task logs contain error messages and verbose information for debugging. Next, the counters are useful to asses data flow and progress of the job. They are incremented periodically when a tasktracker sends a heartbeat back to the jobtracker. The heartbeat contains information about progress and an update to the counters. Some default counters come standard with every job as part of the Hadoop framework but custom counters can be implemented as well. Note that this communication is somewhat onesided. The

jobtracker aggregates statistics from all the tasktrackers so it has a global perspective on the computation, but each tasktracker has no concept of what work is being done by other tasktrackers. That is, the tasktracker cannot poll the jobtracker for global information, it can only send information about its own status through the heartbeat. Finally, completion graphs give a visual representation of progress of each task in the job. Notice the three phases color coded in the reduce graph: copy, sort, and reduce. Technically, copy and sort are part of the *shuffle* phase but are lumped together in this representation. An interesting observation is that data is copied from the local tasktracker (where the map task has been computed) to the tasktracker performing the reduce as soon as a map finishes. If we have more map tasks than available slots, the first wave of map tasks will finish and the copy will begin. The actual reduce algorithm cannot start until all the maps have finished so much time is spent in the copy phase, mostly just waiting for maps to finish. This unfortunately accrues time towards the reduce task and gives a false sense of how long the actual reduce process is taking. For example, the actual reduce portion in Bt-job takes less than a minute, however, the time listed for each reducer (including copy and sort) is around 25 minutes. One must be careful not to go crazy trying to optimize the reducer based on this metric.

While the Hadoop jobtracker UI provides information relevant to the specific jobs, Ganglia monitoring gives us information on how the machines are operating. Figure 4.4b shows four graphs that represent aggregate statistics from the entire cluster. The profile shown is a stochastic singular value decomposition being done on the wikipedia million data set without any power iterations. Looking at the top left pane Load/Procs notice initially about a 10 minute burst of computation, followed by a 35 minute job and finishing with short 5 minute computation. It is also very visible in the second pane measuring cpu utilization. The mounds of activity are Q-job, Bt-job and both U-job and V-job respectively. (U-job and V-job are actually done at the same time in parallel). We notice immediately that the algorithm is cpu bound (at least for the super sparse wikipedia data set). These same graphs are available per node along with many other metrics such as cpu wait io and memory free that are incredibly useful to monitor while running jobs.

4.12 Computational parameters

As promised this section discusses the computational parameters of Figure 4.5. The trials are carried out on the wiki-million data set with $k = 100, \ell = 25$. Using the logic of §4.11.4, we use a cluster of 4 (4 slave + 1 master) m1.large instances. With the default 128MB input split, this configuration gives 15 map tasks and 16 map slots.

As discussed in §4.6.5, blockHeight `(-r)` defines the finest level partition of the data. We compute streaming QR factorizations of $r \times \ell$ sized blocks of Y. The analysis showed that approximately the same amount of work is done for either the direct QR method or the merge scheme used. A clear advantage to the merge scheme is reduced memory. Reducing r then reduces memory use and by Figure 4.9 drastically reduces the time of Q-job.

Parameter r also comes into play in ABt-job where the streaming QR takes place in the reducer. We assume the benefits seen in Q-job are also observed there, though every ABt-job we have observed uses $r = 2,500$. ABt-job is dominated by the matrix product AB^T which requires reading B^T from disk once per input split.

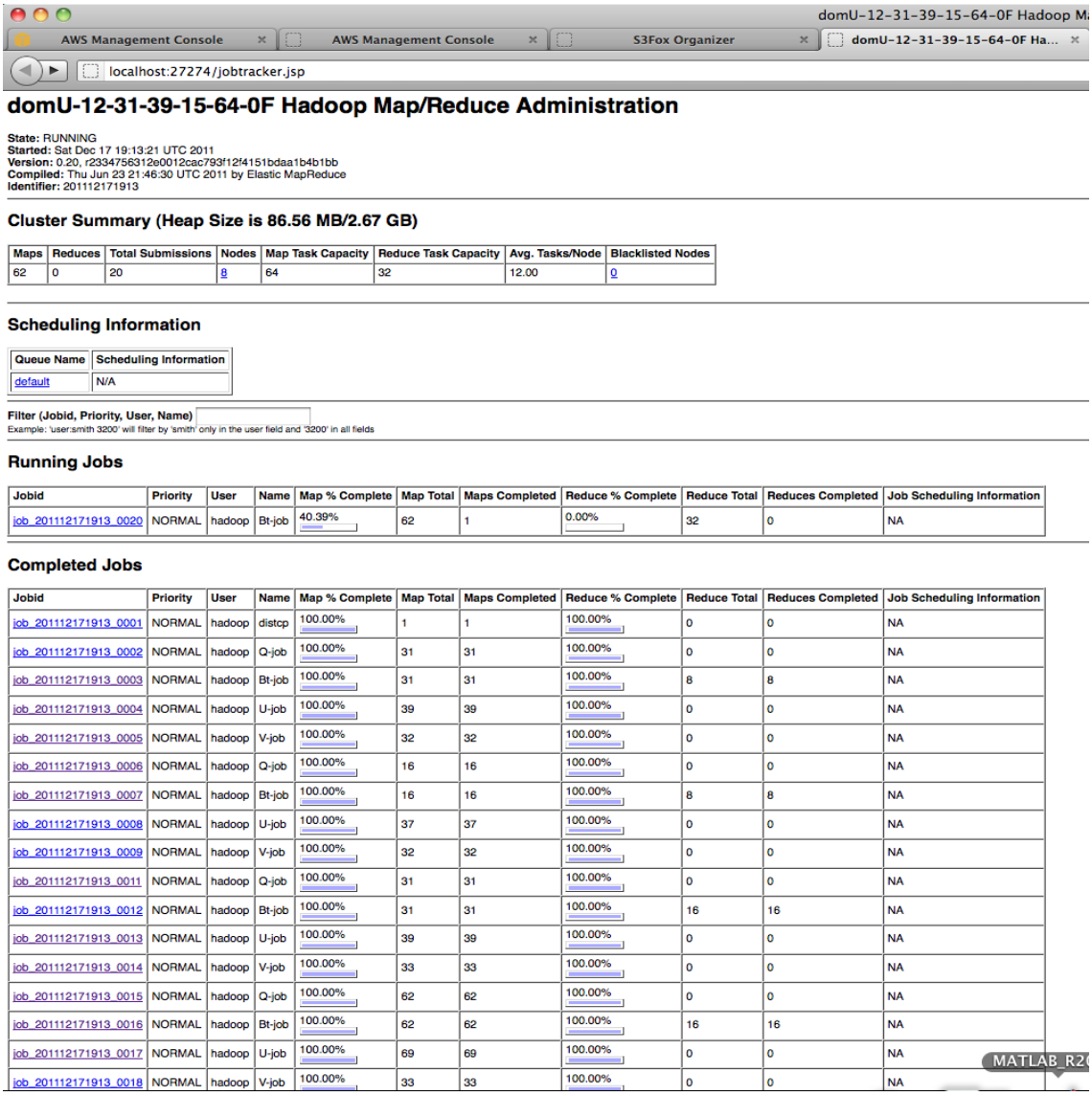


Figure 4.7: Frontpage. View of all jobs submitted to the cluster. Clicking on the job id hyperlinks gives detailed information for that job as in Figure 4.8.

The two other notions of block height, outerProdBlockHeight `(-oh)` and abtBlockHeight `(-abth)`, do not effect runtime in any noticeable way. They define the size of the blocks held in memory when accumulating $\mathsf{A}^T\mathsf{Q}$ and AB^T respectively. Their major effect is to reduce the number of spilled records, roughly doubling the block height cuts number of spilled records in half. Block height's of greater than 40,000 put too much pressure on the memory and produce "reduce copier failed" errors. Therefore, both `-oh` and `-abth` are set to 40,000.

The load on reducers for this algorithm is very light and most of the computation is done in the mappers. Reducers function to produce B^T in Bt-job and optionally the partial sums of BB^T, and to compute the streaming QR and first phase merge in ABt-job. The number of reducers determines the size B^T blocks which then effects the product AB^T. We want to use all our available

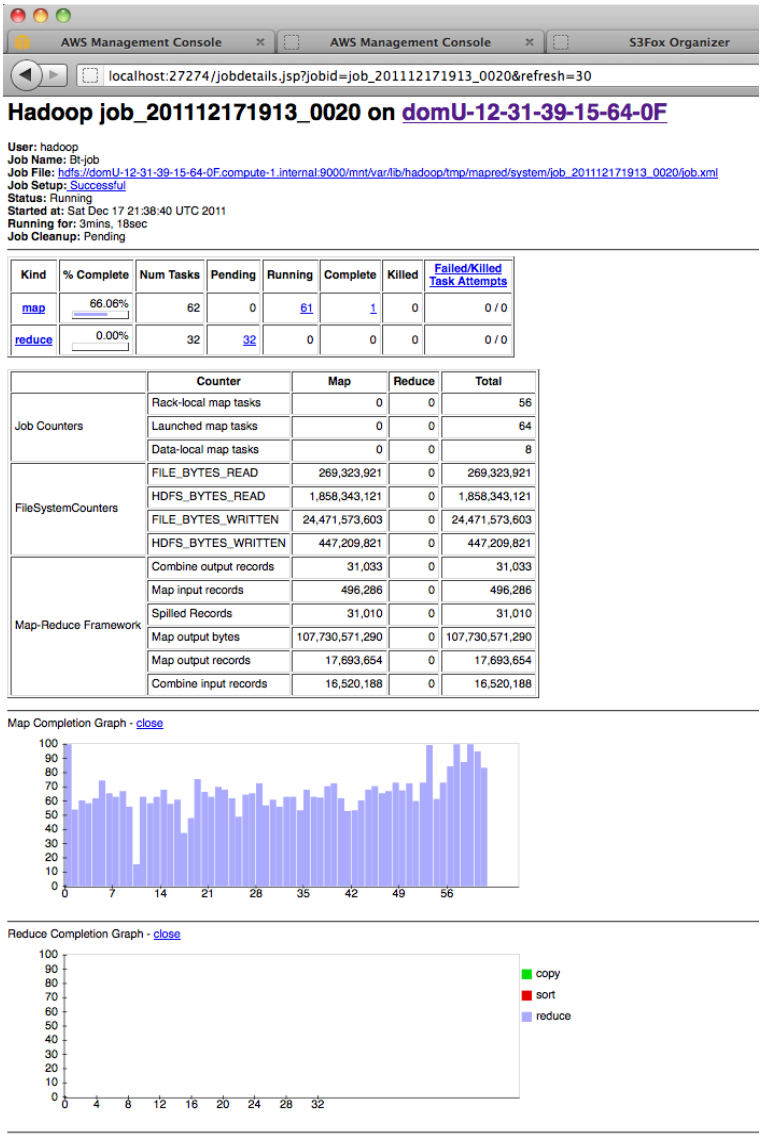
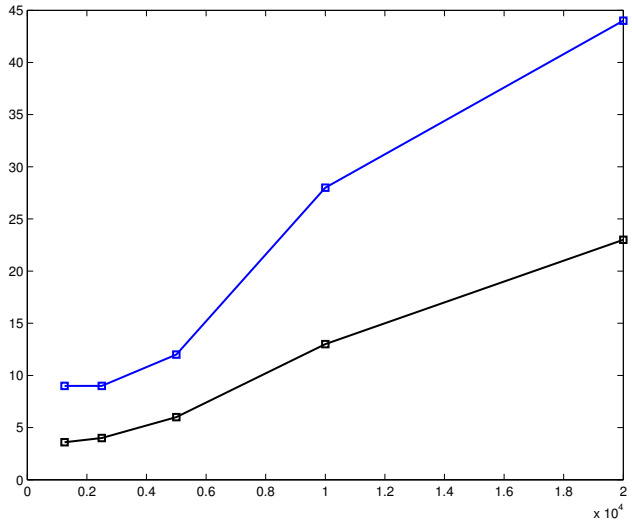


Figure 4.8: Data flow and system metrics on the job level.

reduce slots but we do not want the blocks of B^T to be too large. Unlike the number of map tasks which is determined by the input split, we manually set the exact number of reducers to fire. A good rule is

$$numReducers = \max\{\text{number of slots}, \frac{8\ell n}{1024^2 \cdot b}\} \quad \text{where } b \sim 400 \tag{4.50}$$

b is the block size in megabytes of B^T. During ABt-job, we preload the entire input split into memory and read in blocks of B^T requiring $b + inputSplit$ megabytes of memory.



blockHeight (`-r`)	1250	**2500**	5000	10000	20000
Q-job	9	**9**	12	28	44
per map	3.6	**4**	6	13	23

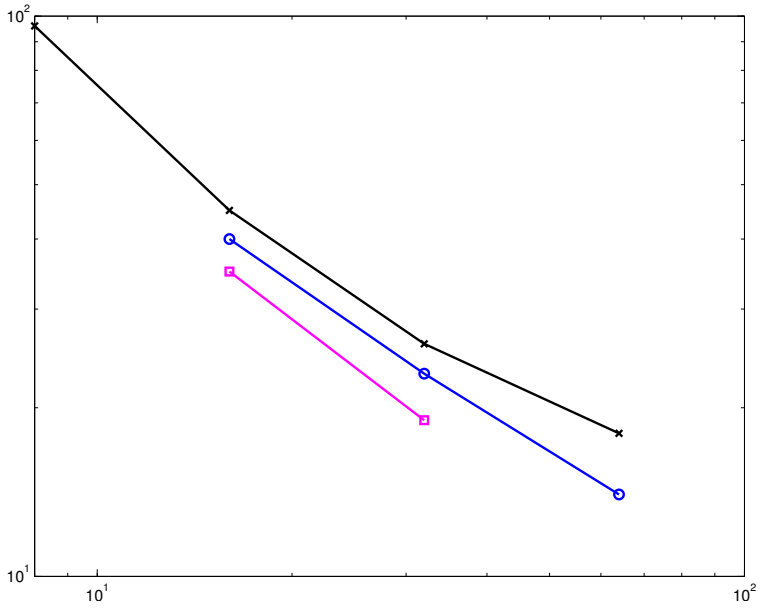
Figure 4.9: Time in minutes for various blockHeight's. We choose $r = 2{,}500$ as optimal.

4.13 Scaling the cluster

The biggest factor in scalability is the input split size. We need to be sure to keep the ratio of tasks per slot near or above one to allow processing of multiple fast tasks per slow task. However we also see that the communication step is impacted by increased parallelism. The relationship is proportional, more parallelism requires more communication. We will explore keeping the input split at default value 128MB until our cluster grows and the available number of task slots forces us to decrease the input split in order to keep number of map tasks approximately equal to available slots. Also when possible, we use a smaller split to see the benefits of better utilized resources when ratio of maps to slots is greater than one.

With relatively small clusters of 2-16 nodes there is very little load on the master and a typical scene is Figure 4.10, where the slave nodes are red hot and the master is cold blue. The master runs the namenode which keeps track of all the files in hdfs and the jobtracker which coordinates all the running tasks. A single master of this size could handle millions of files and tens of thousands of tasks which is far above our needs for these experiments. A minimum of two slave nodes are used for both the m1.large and m1.xlarge clusters. Hadoop run on only one or two machines is mostly for testing and debugging or see §4.16 for example. Also, having significantly more available slots than tasks is wasteful. These two constraints bound the size of our clusters and explain the missing values in Figure 4.11.

Figure 4.10: There is little strain on the cold blue master with only a handful of nodes and ~ 100 files to coordinate.



compute units	8	16	32	64
input split	128	128	64	32
m1.large	96	45	26	18
m1.xlarge		40	23	14
input split		64	32	
m1.xlarge		35	19	

Figure 4.11: Time listed in minutes for varying amounts of computing power. The first split category is chosen so that number of maps is approximately equal to the available slots. The second category better utilizes resources by using a smaller input split and thus more maps than slots.

4.14 Power Iterations

All the timings listed in Figure 4.11 are for the algorithm with $q = 0$. To test the power iterations we choose a cluster of 8 m1.large instances. An input split of 32MB gave the best performance

on that configuration. We ran into the same error when attempting the power iterations as we did when we were increasing `-oh` parameter in §4.12: *reduce copier failed* and *cannot allocate memory.* This time it needed to be dealt with since it was blocking normal operation of the algorithm and not just a possible performance boost.

Table 4.6 lists some parameters of an m1.large instance that are defined upon startup of the cluster. The total memory footprint of each node is calculated as

$$\text{Tasktracker heap} + \text{Datanode heap} + (\text{map slots} + \text{reduce slots}) \times \text{child heap}$$

Surprisingly, the default setting over allocates memory by almost 50%! The point of this configuration is to allow tasks to spike memory usage, but overall use far less memory then allotted [1]. If tasks actually use a lot of memory, the trick is to not over allocate memory because you assume tasks will use all the memory they are given. As suggested in Table 4.6, the memory-intensive bootstrap action both reduces heap size and reduces number of concurrent tasks. A bootstrap action is a script that runs at startup to define some parameters or perform an action. This is how, for example, Ganglia of §4.11.5 is set up. The memory-intensive bootstrap action solved the problem but seemed to under utilize the amount of available memory. We found that the 'config' column of Table 4.6 both solved the problem and restored our concurrent processing capabilities.

	default	mem-intensive	config
Tasktracker heap	1536	512	512
Datanode heap	256	512	512
child heap	1600	1024	1024
map slots	4	3	4
reduce slots	2	1	2
total mem MB	11392	5120	7168
total mem GB	11.125	5	7

Table 4.6: Default and Memory-Intensive bootstrap action settings for m1.large instance. Keep in mind total memory for this instance is 7.5GB. Heap is measured in MB. The third column 'config' are the settings we went with.

Table 4.7 gives the time for each phase of the algorithm. A power iteration requires ABt-job and Bt-job for a total of 28 minutes. This is slightly more than the original time of 26 minutes for $q = 0$.

4.15 Varying k

Finally we time the algorithm for various values of k with $p = 0$ ($\ell = k$ so we use the variables interchangeably here). Most notable about this experiment is the effect of an unequal number of

powerIters

phase	time
Q-job	4
Bt-job	**16**
ABt-job	**12**
U-job	2
V-job	2

q	time
0	26
1	52
2	80
3	107

Table 4.7: (left) Timings in minutes for the various phases of the algorithm on an 8 node m1.large cluster. The bulk of the processing time is in Bt-job and ABt-job which are recomputed for each power iteration. (right) Timings in minutes for values of q.

rows in each input split. The dense subproblem scales as $O(\frac{s^2\ell^2}{r})$, where the variability in s is amplified by larger ℓ. For $\ell \sim 125$ or less, as in the previous experiments, the amplification factor was small enough not to cause major differences in execution time across unequally sized s blocks. With increasing ℓ, the differences are exacerbated creating a major *straggler* problem. A straggler in our case is a task that receives a disproportionately large amount of rows and takes much longer to execute than the rest of the tasks. As depicted in Figure 4.6, all other tasks finish and leave the nodes idle meanwhile the straggler is still running. This is a tremendous waste of resources. Figure 4.12 shows the sorted timings of tasks when $\ell = 500$. We omit the details of this experiment but use it simply to illustrate how a straggler can dramatically slow down each job. A remedy to this problem would be to assign input splits based on number of records rather than number of bytes.

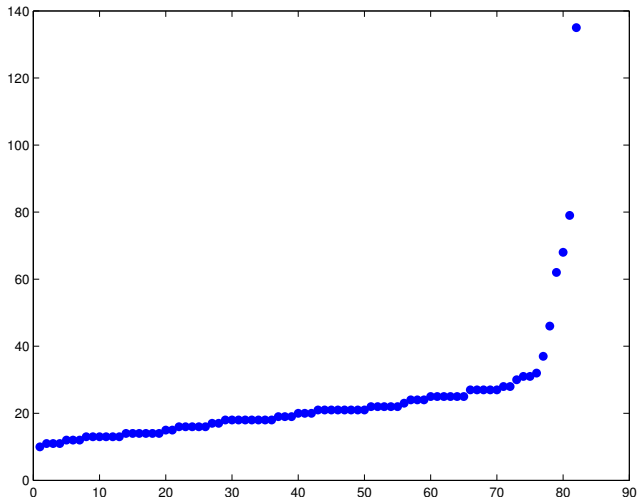


Figure 4.12: Sorted task times in minutes. Smaller values of ℓ reduce the execution time and adverse impact of a straggler task. For $\ell = 500$ as shown, the straggler task dominates execution time, leaving idle nodes for $\sim 75\%$ of the computation.

Figure 4.13 plots time in minutes for various values of k on the wiki-million data set. As noted in §4.16, Lanczos runs most efficiently on just one slave and one master setup. The times plotted for Lanczos use only this setup of $1+1$ m1.large's. For ssvd, we begin with a $4+1$ m1.large cluster for smaller k. For $k = 250$ we needed to double the heap allocated to the tasks and thus

reduce concurrent processing from 4/2 to 2/1 map/reduce slots. Therefore, we justify doubling the number of nodes so that concurrent processing ability remains constant. We could possibly have sped up ssvd for smaller values of k by increasing split size and actually decreasing the number of nodes, but we did not pursue this route.

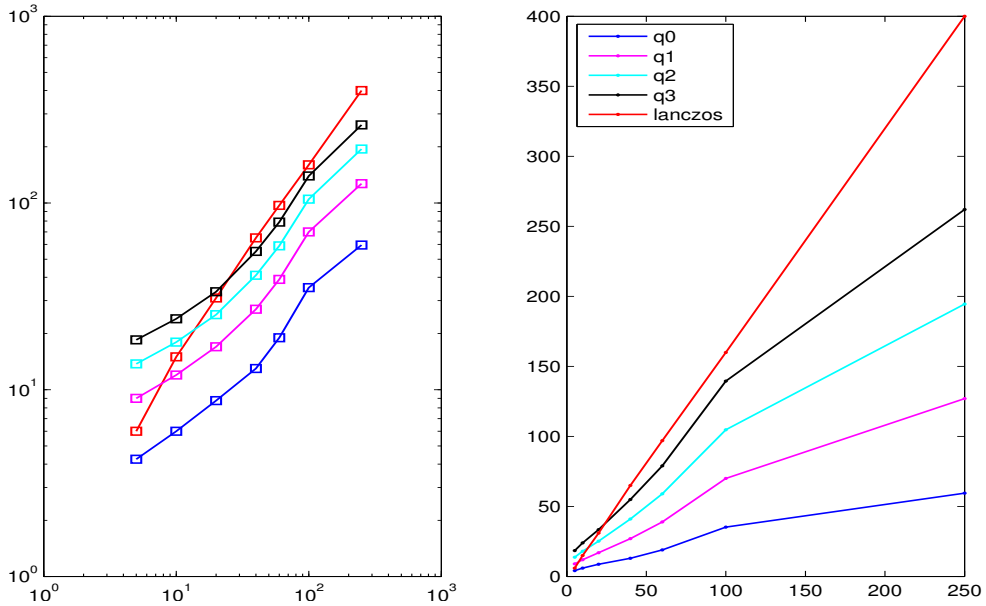


Figure 4.13: Time in minutes for various values of k on the wiki-million data set. Lanczos time is extrapolated for $k > 60$. The loglog plot on the left is to illustrate the activity for small k.

4.16 Lanczos on the cluster

To fit Lanczos algorithm on the cluster we need to consider the operations involved just as we did with the ssvd. The only MapReduce job involved is the matrix multiply of Algorithm 4.5. We perform this job k times so it is important to optimize, seconds per job can easily translate to minutes added to the final execution time. But we use caution, measuring performance in seconds is not a good idea in a virtualized shared environment. Many uncontrollable factors can produce fluctuations in running time. Recall Figure 4.4a which shows how under utilized the cpu is and how much pressure is put on the network. Contrary to the way we divided the data to share cpu load in the ssvd, here we need to do the opposite and group data together to shift the work load from the network to the cpu. To achieve this we need to increase the input split size thus decreasing the number of maps and limiting the scalability of the algorithm. Table 4.8 shows indeed that increasing split size gives faster performance. But we reach a limit. A Hadoop MapReduce job takes ~ 30 seconds to setup, also increasing split size increases time per task. Then we see a lower bound on execution time of

$$\text{setup time} + \max\{\text{task execution time}\} < \text{overall execution time}$$

Also we need to consider that increasing split size will require larger heap settings per task. By increasing task heap we reduce the number of concurrent tasks per node. This is a dead-end street to travel down when scaling. We present Table 4.8 to show that this approach only improves time slightly. Look for the lower bound in column 3. Increasing the split beyond 512M will increase task time and thus increase job time.

split	128	256	512	512*
job	1:17	1:11	1:06	1:25
task (seconds)	18	32	33	53

Table 4.8: Larger input split sizes reduce overall execution time per job, but the benefit is minimal. We used here a $2+1$ cluster of m1.large instances having 8 concurrent map slots. A split of 512MB only produces 4 map tasks so the nodes have unused resources. The last column 512* uses a $1+1$ setup.

The remainder of the algorithm is performed in serial on the master node. Figure 4.15 shows the time spent between each MapReduce job in the iteration. Because only one vector gets inducted into the basis at each iteration, it does not make sense to try and distribute the orthogonalization. Also, the timings suggest that these operations are only a small percentage of total time. To help facilitate the extensive I/O of the orthogonalization process we increase a parameter that determines how much data is buffered during read and write operations from its default of 64 kilobytes to 8 megabytes, which is just larger than the size of a dense 1,000,000 dimensional vector. Finally, we choose a cluster of only one master and one slave. This configuration dedicates one machine to the matrix vector product and the other to the serial computation, which is esentially the orthogonalization.

Though not as fast as using two slave as shown in Table 4.8, the resource use shown in Figure 4.14 is well optimized. Using a 512MB input split puts enough data into a map task to use 100% cpu, if even only for a short burst. Note that the CPU shown in Figure 4.14 is aggregate between the slave and master node. The peaks represent the slave working at 100% and the troughs are the master using $< 10\%$. The network load is greatly minimized. Since there is only one slave node, there is no network communication needed in the shuffle and sort phase. This eliminates 22MB of data transfer during the shuffle.

4.17 Wikipedia-all

Using what we have learned from the wiki-million data set, scaling up to the wiki-all data set requires only a few changes. First we increase the heap given to the map and reduce tasks. Preliminary runs gave *GC overhead limit exceeded* errors in ABt-job which indicates a *possibility* that the heap was too small. Fortunately, increasing heap to 2GB solved the problem. By doubling the heap we also need to cut in half the number of concurrent map and reduce tasks. We use a cluster of 16 m1.xlarge instances giving 64 map slots and 32 reduce slots. A 64MB split gives 280 map tasks and we use equation (4.50) to get 85 reduce tasks.

A major bottleneck in ABt-job is the network pressure from reading blocks of B^T. We need to read the *entire* B^T matrix once per input split. Multiplying the size of B^T by the number of map

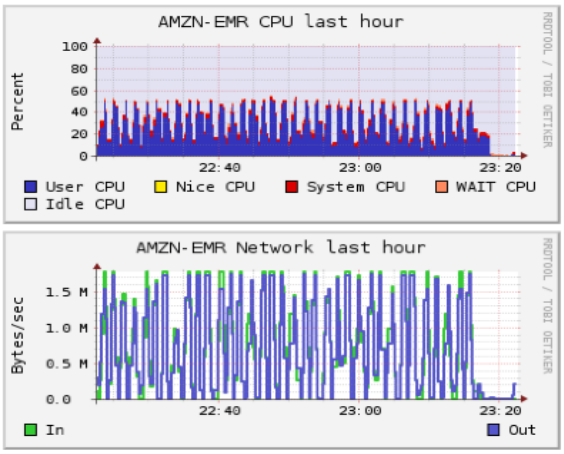


Figure 4.14: Optimized resource uses for Lanczos svd. One master plus one slave m1.large instances. Note the CPU appears to be only 50% utilized but the meter measures both cpu's. In actuality, the slave node is using 100% during matrix vector product and master is using $< 10\%$ in between to orthogonalize. Also note the scale of the network is from 0 to 1.5 megabytes per second. The network cost during shuffle has been greatly reduced by using only one node. Contrast this with Figure 4.4a that has $5 - 10$ MB/sec network use.

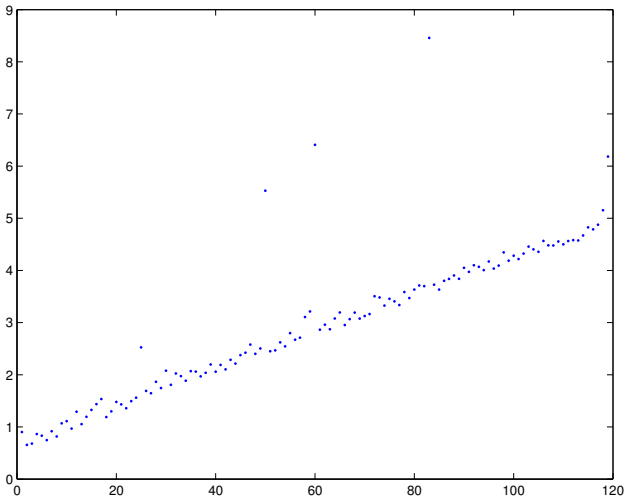


Figure 4.15: Time in seconds of the non MapReduce (serial) computation in Lanczos svd. As the basis grows we see a linear increase in computation time. Overall, the percentage of time of each iteration is very small but we pay an increasing amount every iteration.

tasks gives about 10TB over the network for a 64MB split. Figure 4.18 shows network loads during ABt-job for identical clusters. In §4.11.3 we mention how shared network resources can provide surplus bandwidth at times and at other times, bandwidth is limited or *throttled*. Figure 4.18(a) shows a throttling of the network with a maximum bandwidth of 1GB/s. Figure 4.18(b) shows the case where extra resources are available and increases bandwidth to $\sim$ 1.4GB/s.

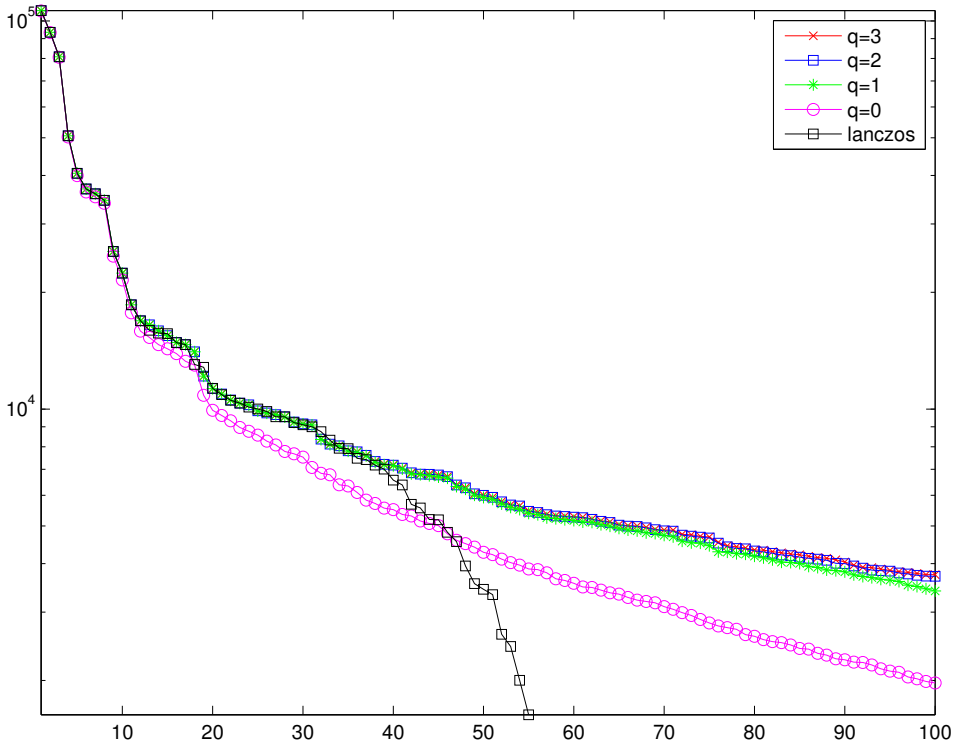


Figure 4.16: Computed spectrum of the Wikipedia million data set with $k = 100$ and $p = 25$. Only 60 values were computed for Lanczos before running out of memory (Java Heap). Just one power iteration gives excellent accuracy. Singular values for $q = 2, 3$ are indistinguishable in the plot and appear as a red x inside a blue square.

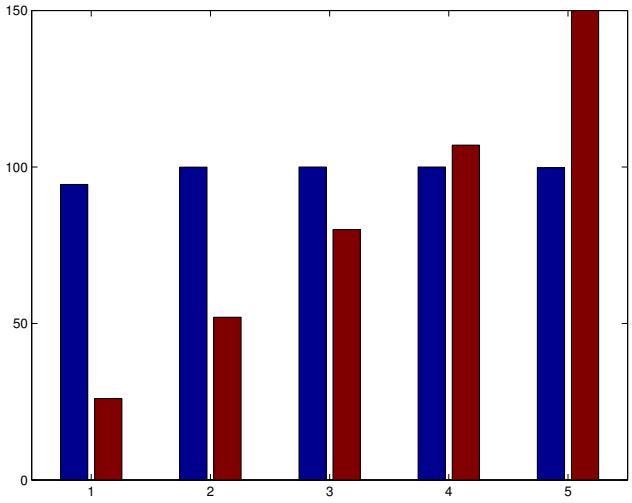
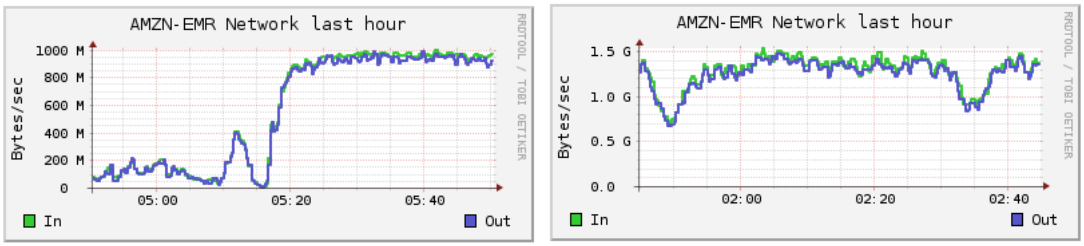


Figure 4.17: Accuracy as a percentage and time in minutes. From left to right ssvd with $q = 0, 1, 2, 3$ and lanczos. Accuracy is computed as the sum of the first 40 singular values obtained from each method. The sum is scaled by the largest sum and multiplied by 100 to produce a percentage. Order of accuracy is $q = 3$, $q = 2$, $q = 1$, lanczos, $q = 0$.

Setup parameters	
m1.xlarge	16
child heap	2048M
map slots	64
reduce slots	32
input split	64M
map tasks	280
reduce tasks	85

Execution times	minutes
Q-job	4
Bt-job	49
ABt-job	127
U-job	1.5
V-job	7
$q = 0$	61
$q = 1$	235

Table 4.9: Setup and execution times in minutes for the wiki-all data set.



(a) Throttled network

(b) Extra network resource

Figure 4.18: Network throttling has a dramatic impact on ABt-job which reads dense matrix B^T from disk once per input split.

4.18 Wikipedia-MAX

Setup for the wiki-max data set is much the same as for wiki-all. A 128MB split gives 726 maps. We only attempted the computation for $q = 1$ which took a little over 22 hours. This shows that further power iterations, $q = 2, 3, \ldots$ are possible with the additional time incurred by Bt-job and ABt-job. Figure 4.20 gives a plot of the computed singular values. Table 4.10 gives the timings of each phase of the computation and gives extrapolated times for a range of q values and larger clusters. This assumes we can half the computation time by doubling the size of the cluster as long as the number of available map slots stays below the number of map tasks, 726. A 128 node cluster provides 512 slots in our configuration.

Figure 4.19 shows the day long Ganglia profile from the wiki-max computation. The three stages of computation: Bt-job, ABt-job, and Bt-job are clearly visible. Most notable is the sustained 1.3 GB/sec network load during ABt-job which transfers a total of 50TB across the network. Reading the matrix B^T only accounts for half of this. The remaining network is a result of input splits being larger than the outer product block height `-abth`. The result is that intermediate data is spilled to disk causing extra network load. Contrast this with the network load from wiki-all with a 64MB split. There the `-abth` parameter was large enough to not allow spills and the network load was all from the reading of B^T.

Remark 31. *The computation of the wiki-all and wiki-max data sets may use safer than optimal parameter choices. We used information gathered on smaller computations and made changes only*

as necessary to handle bigger inputs. When dealing with computations that run many hours or days, tuning parameters to avoid failure is more important than tuning them to save time.

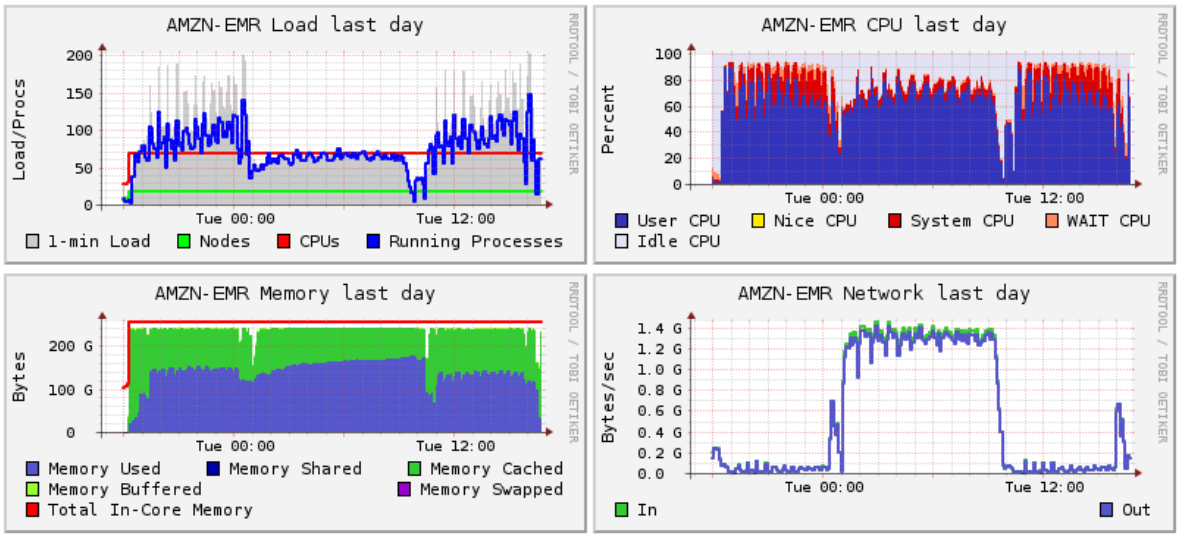


Figure 4.19: Ganglia profile for the wiki-max computation which clearly shows the 3 stages: Bt-job, ABt-job, Bt-job. Notice the extreme network load during ABt-job which shows 50TB of data was passed across the network.

Actual timings

phase	time
Q-job	20
Bt-Job	365
ABt-job	571
U-job	8
V-job	14
total	1335

Extrapolated timings

nodes	$q = 0$	$q = 1$	$q = 2$	$q = 3$
16	407	**1335**	2271	3207
32	204	668	1136	1604
64	102	334	568	802
128	51	167	284	401

Table 4.10: Timings in minutes for the wiki-max data set. The cost of the machines needed to decompose this data set makes additional runs prohibitively expensive. Extrapolated timings on the right provide an idea of scaling times assuming doubling the number of nodes halves computation time as long as number of tasks is larger than number of slots.

4.19 Lanczos comparison with SSVD

This section gathers conclusions from the results of §4.16 and §4.8.3. Overall, the ssvd outperforms the Lanczos implementation in this environment and for the data sets used.

Execution time The Hadoop environment is ill suited for iterative computations. With each pass through the data an algorithm should extract as much information as possible thus limiting

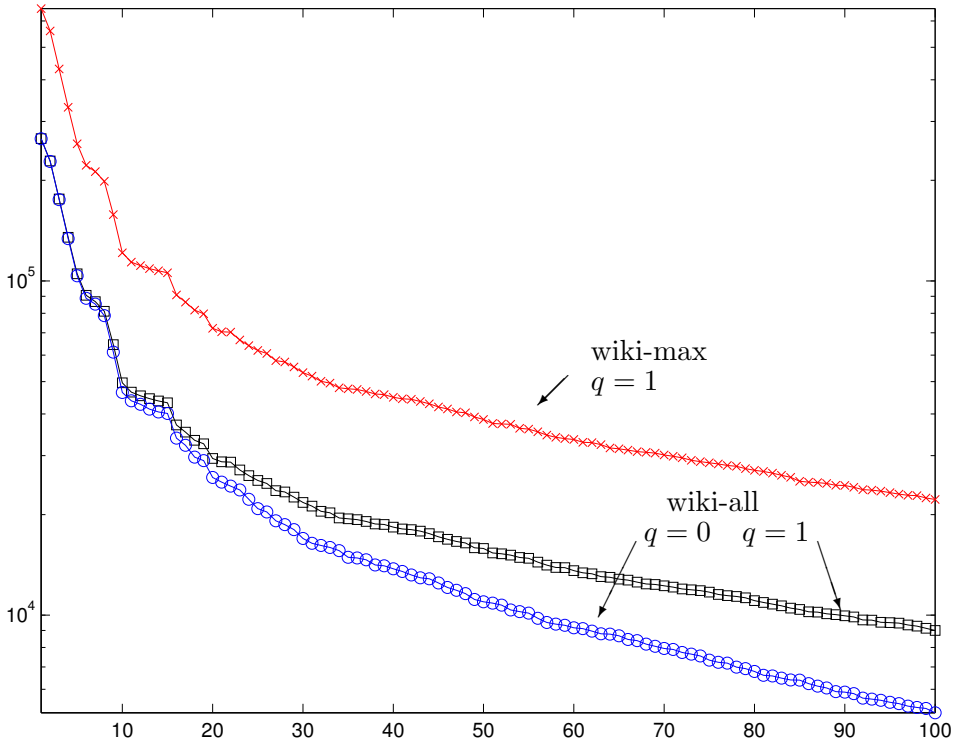


Figure 4.20: Singular values for the wiki-all and wiki-max data sets.

interaction with the data and minimizing set up and data movement costs. The Lanczos algorithm computes only a vector's worth of data each pass through the matrix. This not only requires many passes through the data incurring set up and data movement costs, but also fails to fully utilize processor capability. The ssvd visits the matrix only twice (and two additional times per power iteration). It also does computation in bulk which fully utilizes the processors and minimizes overhead costs of the framework. The ssvd is faster than the Lanczos method in this environment.

Accuracy The Lanczos method provides excellent accuracy. It was designed to do this at the expense of many passes through the data. The ssvd provides comparable accuracy. In particular, for $q = 0$, the ssvd is not as accurate as Lanczos method, however, with just one power iteration we obtain a slightly better approximation.

Scalability This is perhaps the most important feature since if a method does not scale, then it cannot compete on large data sets. The implementation of Lanczos we tested only distributed the matrix vector multiplication, orthogonalization was done in serial on a single machine. We ran into memory (java heap) problems on modest sized data sets and could not use the methods on our larger data sets. The ssvd was able to compute a rank 100 singular value decomposition of a matrix whose dimensions both exceeded 37,000,000. Theoretically, the ssvd's memory usage is not dependent on either dimension of the matrix so we believe the method will scale to much larger data sets.

4.20 Conclusion

This paper presented a fully distributed randomized algorithm for computing an approximate rank k singular value decomposition. The method scaled efficiently and ran on up to 64 cores in Amazon's Elastic Cloud Compute. We gave a treatment of the distributed computing environment: Hadoop, MapReduce, Amazon EC2 and the software library Mahout. Through careful tuning of our Hadoop cluster, we were able to decompose a matrix whose dimensions were orders of magnitude larger than in previous works. We also showed how the ssvd is well suited to distributed computation by comparing it with the classic Lanczos method. The ssvd scales far beyond Lanczos method due to its bulk data processing and non-iterative characteristics and can achieve equivalent accuracy in less time.

Acknowledgements

I would like to thank Dmitriy Lyubimov and Ted Dunning for contributing the ssvd to Mahout and for their hard work and dedication that keep open source projects like Mahout evolving. I also want to thank Dave Angulo and Lanny Ripple for their mentorship in teaching me Hadoop, MapReduce and the ways of big data and distributed computing.

Bibliography

[1] Amazon web services. http://aws.amazon.com/.

[2] Apache hadoop. http://hadoop.apache.org/.

[3] Apache mahout. http://mahout.apache.org/.

[4] Ganglia monitoring system. http://ganglia.sourceforge.net/.

[5] Wikimedia article dump. http://dumps.wikimedia.org/enwiki/latest/.

[6] P. G. Constantine and D. F. Gleich, Tall and skinny qr factorizations in mapreduce architectures, in Proceedings of the second international workshop on MapReduce and its applications, MapReduce '11, New York, NY, USA, 2011, ACM, pp. 43–50.

[7] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart, Reorthogonalization and Stable Algorithms for Updating the Gram-Schmidt QR Factorization, Mathematics of Computation, 30 (1976), pp. 772–795.

[8] Dean and Ghemwat, Mapreduce: Simplified data processing on large clusters, tech. rep., Google Inc., 2004.

[9] N. Halko, P. Martinsson, and J. Tropp, Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions., SIAM Review, 53(2) (2011), pp. 217–288.

[10] N. Halko, P.-G. Martinsson, Y. Shkolnisky, and M. Tygert, An algorithm for the principal component analysis of large data sets, SIAM Journal on Scientific Computing, 33 (2011), pp. 2580–2594.

[11] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, research of the national bureau of standards, 45 (1950).

[12] D. Lyubimov, Modified power iterations in existing ssvd code, mahout-796. https://issues.apache.org/jira/browse/MAHOUT-796.

[13] ——, Weathering thru tech days. http://weatheringthrutechdays.blogspot.com/search/label/Stochastic%20SVD.

[14] ——, Mapreduce ssvd working notes. https://issues.apache.org/jira/secure/attachment/12466791/SSVD+working+notes.pdf, December 2010.

[15] ——, Modified stochastic svd algorithm for mapreduce. https://issues.apache.org/jira/secure/attachment/12456827/Modified+stochastic+svd+algorithm+for+mapreduce.pdf, 2010.

[16] ——, Qr decomposition step using mapreduce. https://issues.apache.org/jira/secure/attachment/12459940/QR+decomposition+for+Map.pdf, November 2010.

[17] J. Norstad, A mapreduce algorithm for matrix multiplication. http://www.norstad.org/matrix-multiply/index.html.

[18] S. Owen, R. Anil, T. Dunning, and E. Friedman, Mahout In Action, Manning Publications Co., 2012.

[19] R. J. Radke, A matlab implementation of the implicitly restarted arnoldi method for solving large-scale eigenvalue problems, master of arts, Rice University, Houston, Texas, April 1996.

[20] T. White, Hadoop: The Definitive Guide, O'Reilly Media, original ed., June 2009.