THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A BLOCK INCREMENTAL ALGORITHM FOR COMPUTING DOMINANT SINGULAR SUBSPACES

By

CHRISTOPHER G. BAKER

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2004

The members of the Committee approve the Thesis of Christopher G. Baker defended on April 19, 2004.

Kyle Gallivan
Professor Directing Thesis

Anuj Srivastava
Committee Member

Robert van Engelen
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

This thesis is dedicated, even when I am not:

to my parents, who forbade me to attend law school,

and to Kelly, for supporting me throughout the entire process.
I promise I will pay off one day.

ACKNOWLEDGMENTS

I would first like to acknowledge my gratitude to my committee members, Professor Anuj Srivastava and Professor Robert van Engelen. They served willingly for a student who was not always willing to work.

I am obliged to Dr. Younes Chahlaoui, for many discussions and much advice regarding this thesis. I am grateful as well to Dr. Pierre-Antoine Absil, for his advisements on writing. Were it not for these two men, the French translation of my thesis would not have been possible.

To the many people at the FSU School of Computational Science and Information Technology, where this research was conducted: the systems group kept things working; Mimi Burbank provided neverending help in typesetting this thesis; and I received much assistance and encouragement from members of the CSIT faculty, in particular Professor Gordon Erlebacher, Professor Yousuff Hussaini, and Professor David Banks.

I would do a great disservice to overlook the contributions of Professor Paul Van Dooren. I owe much to his counsel. He imparted unto me much insight into the nature of the methods discussed in this thesis. His suggestions and criticisms are the fiber of this document, and I proudly count him as a *de facto* advisor on this thesis.

Finally, I wish to thank my advisor, Professor Kyle Gallivan. Without his support—academic, technical, and of course, financial—this work would not have been possible. His ability to explain complicated topics and to illustrate subtle points is unmatched. Perhaps more remarkable is his willingness to do so. I am indebted to him for inpsiring my appreciation and interest for the field of numerical linear algebra, just as he attempts to do with everyone he meets. He is a professor in the greatest sense of the word.

This research was partially supported by the National Science Foundation under grant CCR-9912415.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis presents and evaluates a generic algorithm for incrementally computing the dominant singular subspaces of a matrix. The relationship between the generality of the results and the necessary computation is explored, and it is shown that more efficient computation can be obtained by relaxing the algebraic constraints on the factoriation. The performance of this method, both numerical and computational, is discussed in terms of the algorithmic parameters, such as block size and acceptance threshhold. Bounds on the error are presented along with a posteriori approximations of these bounds. Finally, a group of methods are proposed which iteratively improve the accuracy of computed results and the quality of the bounds.

CHAPTER 1

INTRODUCTION

The *Singular Value Decomposition* (SVD) is one of the most useful matrix decompositions, both analytically and numerically. The SVD is related to the eigenvalue decomposition of a matrix. The eigenvalue decomposition for a symmetric matrix $B \in \mathbb{R}^{n\times n}$ is $B = Q\Lambda Q^T$, where Q is an $n \times n$ orthogonal matrix and Λ is an $n \times n$ diagonal real matrix.

For a non-symmetric or non-square matrix $A \in \mathbb{R}^{m\times n}$, such a decomposition cannot exist. The SVD provides an analog. Given a matrix $A \in \mathbb{R}^{m\times n}, m \geq n$, the SVD of A is:

$$A = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T,$$

where U and V are $m \times m$ and $n \times n$ orthogonal matrices, respectively, and Σ is a diagonal matrix whose elements $\sigma_1, \ldots, \sigma_n$ are real, non-negative and non-increasing. The σ_i are the singular values of A, and the columns of U and V are the left and right singular vectors of A, respectively. Often times, the SVD is abbreviated to ignore the right-most columns of U corresponding to the zero matrix below Σ. This is referred to in the literature as a *thin SVD* [1] or a *singular value factorization* [2]. The thin SVD is written as $A = U\Sigma V^T$, where U now denotes an $m \times n$ matrix with orthonormal columns, and Σ and V are the same as above.

The existence of the SVD of a matrix can be derived from the eigenvalue decomposition. Consider the matrix $B = A^T A \in R^{n\times n}$. B is symmetric and therefore has real eigenvalues and an eigenvalue decomposition $B = V\Lambda V^T$. Assuming that A is full rank and constructing a matrix U as follows

$$U = AV\Lambda^{-1/2},$$

is easily shown to give the SVD of A:

$$A = U\Lambda^{1/2}V^T = U\Sigma V^T.$$

It should also be noted that the columns of U are the eigenvectors of the matrix AA^T.

One illustration of the meaning of the SVD is to note that the orthogonal transformation V applied to the columns of A yields a new matrix, $AV = U\Sigma$, with orthogonal columns of non-increasing norm. It is clear then that each column v_i of V is, under A, sent in the direction u_i with magnitude σ_i. Therefore, for any vector $b = \sum_{i=1}^{n} \beta_i v_i$,

$$\begin{aligned} Ab &= A\sum_{i=1}^{n} \beta_i v_i \\ &= \sum_{i=1}^{n} (\beta_i \sigma_i) u_i. \end{aligned}$$

The subspace spanned by the left singular vectors of A is called the left singular subspace of A. Similarly, the subspace spanned by the right singular vectors of A is called the right singular subspace of A. Given $k \leq n$, the singular vectors associated with the largest k singular values of A are the rank-k dominant singular vectors. The subspaces associated with the rank-k dominant singular vectors are the rank-k dominant singular subspaces.

Consider the first k columns of U and V (U_k and V_k), along with the diagonal matrix containing the first k singular values (Σ_k). There are four optimality statements that can be made regarding the matrices U_k, Σ_k, and V_k:

1. $\|A - U_k \Sigma_k V_k^T\|_2$ is minimal over all $m \times n$ matrices of rank $\leq k$.
2. $\|A^T A - V_k \Sigma_k^2 V_k^T\|_2$ is minimal over all $n \times n$ symmetric matrices of rank $\leq k$.
3. $\|AA^T - U_k \Sigma_k^2 U_k^T\|_2$ is minimal over all $m \times m$ symmetric matrices of rank $\leq k$.
4. $\text{trace}(U_k^T AA^T U_k)$ is maximal over isometries of rank $\leq k$.

One commonly used technique for dimensionality reduction of large data sets is *Principal Component Analysis* (PCA). Given a set of random variables, the goal of PCA is to determine a coordinate system such that the variances of any projection of the data set lie on the axes. These axes are the principal components. Stated more formally, assume that the column vectors of the matrix A each contain samples of the m random variables. The goal is to find an isometry P so that $B = P^T A$ with the constraint that the covariance $\text{cov}(B)$ of B is

diagonal with maximum trace. It follows that

$$\begin{aligned} \text{cov}(B) &= \text{E}[BB^T] \\ &= \text{E}[P^T AA^T P] \\ &= P^T \text{E}[AA^T] P. \end{aligned}$$

Assuming that the mean of the columns of A is zero, the isometry P which maximizes the trace of $\text{cov}(B)$ is U_k, a result of optimality Statement 4.

These vectors (principal components) can be used to project the data onto a lower dimensional space under which the variance is maximized and uncorrelated, allowing for more efficient analysis of the data. This technique has been used successfully for many problems in computer vision, such as face and handwriting recognition.

Another technique, related to PCA, is that of the *Karhunen-Loeve Transform* (KLT). The KLT involves computing a low-rank subspace (the K-L basis) under which A is best approximated. This subspace is the dominant left singular subspace of A, and can be derived from optimality Statement 2 or 3. KLT has been successfully employed in image/signal coding and compression applications.

Another application of the SVD is that of the *Proper Orthogonal Decomposition* (POD). POD seeks to produce an orthonormal basis which captures the dominant behavior of a large-scale dynamical system based on observations of the system's state over time. Known also as the *Empirical Eigenfunction Decomposition* [3], this technique is motivated by interpreting the matrix A as a time series of discrete approximations to a function on a spatial domain. The dominant SVD can then be interpreted as follows, with

- U_k are a discrete approximation to the spatial eigenfunctions of the function represented by the columns of A, known as the *characteristic eddies* [4],
- ΣV_k^T are the k coefficients that are used in the linear combination of the eigenfunctions to approximate each column of A.

A consequence of optimality Statement 1, using U_k, Σ_k, and V_k minimizes the discrete energy norm difference between the function represented by the columns of A and the rank-k factorization $U_k \Sigma_k V_k^T$.

Sirovich [5] introduced the methods of snapshots to efficiently produce this basis. Given a dynamical system, the method of snapshots saves instantaneous solutions of the system (the

snapshots) produced via a direct numerical simulation. The snapshots may be spaced across time and/or system parameters. The SVD of these snapshots then provides an orthonormal basis that approximates the eigenfunctions of the system.

This orthonormal basis can be used for multiple purposes. One use is compression, by producing a low-rank factorization of the snapshots to reduce storage. Another technique is to use the coordinates of the snapshots in this lower-dimensional space to interpolate between the snapshots, giving solutions of the system at other time steps or for other system parameters. A third use combines POD with the Galerkin projection technique to produce a reduced-order model of the system. This reduced-order model evolves in a lower dimensional space than the original system, allowing it to be used in real-time and/or memory-constrained scenarios. Each solution of the reduced-order system is then represented in original state-space using the orthonormal basis produced by the POD.

A common trait among these applications–PCA and POD–is the size of the data. For the computer vision cases, the matrix A contains a column for each image, with the images usually being very large. In the case of the POD, each column of A may represent a snapshot of a flow field. These applications usually lead to a matrix that has many more rows than columns. It is matrices of this type that are of interest in this thesis, and it is assumed throughout this thesis that $m \gg n$, unless stated otherwise.

For such a matrix, there are methods which can greatly increase the efficiency of the SVD computation. The R-SVD [1], instead of computing the SVD of the matrix A directly, first computes a QR factorization of A:

$$A = QR.$$

From this, an SVD of the $n \times n$ matrix R can be computed using a variety of methods, yielding the SVD of A as follows:

$$\begin{aligned} A &= QR \\ &= Q(\tilde{U}\tilde{\Sigma}\tilde{V}^T) \\ &= (Q\tilde{U})\tilde{\Sigma}\tilde{V}^T \\ &= U\Sigma V^T. \end{aligned}$$

To compute the R-SVD of A requires approximately $6mn^2 + O(n^3)$, and $4mn^2 + 2mnk + O(n^3)$ to produce only k left singular vectors. This is compared to $14mn^2 + O(n^3)$ for the

Golub-Reinsch SVD of A. This is clearly more efficient for matrices with many more rows than columns.

Another method is to compute the SVD via the eigendecomposition of A^TA, as suggested by the derivation of the SVD given earlier. The eigenvalue decomposition of A^TA gives $A^TA = V\Lambda V^T$, so that

$$(AV)^T(AV) = \Lambda = \Sigma^T\Sigma.$$

This method requires mn^2 operations to form A^TA and $2mnk + O(n^3)$ to compute the first k columns of $U = AV^T\Sigma_1^{-1}$. However, obtaining the SVD via A^TA is more sensitive to rounding errors than working directly with A.

The methods discussed require knowing A in its entirety. They are typically referred to as *batch methods* because they require that all of A is available to perform the SVD. In some scenarios, such as when producing snapshots for a POD-based method, the columns of A will be produced incrementally. It is advantageous to perform the computation as the columns of A become available, instead of waiting until all columns of A are available before doing any computation, thereby hiding some of the latency in producing the columns. Another case is when the SVD of a matrix must be updated by appending some number of columns. This is typical when performing PCA on a growing database. Applications with this property are common, and include document retrieval, active recognition, and signal processing.

These characteristics on the availability of A have given rise to a class of *incremental methods*. Given the SVD of a matrix $A = U\Sigma V^T$, the goal is to compute the SVD of the related matrix $A_+ = \begin{bmatrix} A & P \end{bmatrix}$. Incremental (or recursive) methods are thus named because they update the current SVD using the new columns, instead of computing the updated SVD from scratch. These methods do this in a manner which is more efficient than the $O(mn^2)$ algorithmic complexity incurred at each step using a batch method.

Just as with the batch methods, the classical incremental methods produce a full SVD of A. However, for many of the applications discussed thus far, only the dominant singular vectors and values of A are required. Furthermore, for large matrices A, with $m \gg n$, even the thin SVD of A (requiring $O(mn)$ memory) may be too large and the cost ($O(mn^2)$) may be too high. There may not be enough memory available for the SVD of A. Furthermore, an extreme memory hierarchy may favor only an incremental access to the columns of A, while penalizing (or prohibiting) writes to the distant memory.

These constraints, coupled with the need to only compute the dominant singular vectors

and values of A, prompted the formulation of a class of low-rank, incremental algorithms. These methods track a low-rank representation of A based on the SVD. As a new group of columns of A become available, this low-rank representation is updated. Then the part of this updated factorization corresponding to the weaker singular vectors and values is truncated. In this manner, the dominant singular subspaces of the matrix A can be tracked, in an incremental fashion, requiring a fraction of the memory and computation needed to compute the full SVD of A.

In Chapter 2, this thesis reviews the current methods for incrementally computing the dominant singular subspaces. Chapter 3 describes a generic algorithm which unifies the current methods. This new presentation gives insight into the nature of the problem of incrementally computing the dominant singular subspaces of a matrix. Based on this insight, a novel implementation is proposed that is more efficient than previous methods, and the increased efficiency is illustrated empirically in Chapter 4.

These low-rank incremental algorithms produce only approximations to the dominant singular subspaces of A. Chapter 5 discusses the sources of error in these approximations and revisits the attempts by previous authors to bound this error. Those works are considered in light of the current presentation of the algorithm, and the effect of algorithmic parameters upon the computed results is explored. Chapter 6 explores methods for correcting the subspaces computed by the incremental algorithm, when a second pass through the data matrix A from column 1 to column n is allowed. Three methods are described, and each is evaluated empirically. Finally, a summary of this thesis and a discussion of future work is given in Chapter 7.

CHAPTER 2

CURRENT METHODS

This section discusses three methods from the literature to incrementally update the dominant singular subspaces of a matrix A. The methods are divided into categories, characterized by the updating technique and the form of the results at the end of each step.

The UDV methods are characterized by the production of a factorization in SVD-like form, consisting of two orthonormal bases and a non-negative diagonal matrix. The QRW methods produce orthonormal bases for the singular subspaces along with a small, square matrix. This matrix contains the current singular values, along with rotations that transform the two bases to the singular bases.

2.1 UDV methods

In some applications, it is not enough to have bases for the dominant singular subspaces. A basis for a subspace defines an essentially unique coordinate system, so that when comparing two different objects, the same basis must be used to compute their coordinates. Therefore, many applications require the left and right singular bases, those bases composed of ordered left and right singular vectors. The methods described below compute approximations to the dominant singular vectors and values of A at each step.

2.1.1 GES-based methods

In [6], Gu and Eisenstat propose a stable and fast algorithm for updating the SVD when appending a single column or row to a matrix with a known SVD. The topic of their paper is the production of an updated complete SVD, and does not concern the tracking of the

dominant space. However, the work is relevant as the foundation for other algorithms that track only the dominant subspaces.

The kernel step in their algorithm is the efficient tridiagonalization of a "broken arrowhead" matrix, having the form:

$$B = \begin{bmatrix} \Sigma_i & z \\ & \rho \end{bmatrix} = \begin{bmatrix} \sigma_1 & & & \zeta_1 \\ & \ddots & & \vdots \\ & & \sigma_k & \zeta_n \\ & & & \rho \end{bmatrix} = W \Sigma_{i+1} Q^T,$$

where $W, \Sigma_{i+1}, Q \in \mathbb{R}^{(i+1)\times(i+1)}$. Their algorithm is capable of computing the SVD of B in $O(i^2)$ computations instead of the $O(i^3)$ computations required for a dense SVD. This is done stably and efficiently by relating the SVD of the structured matrix to a function of a special form, that admits efficient evaluation using the *fast multipole method*.

Chandrasekaran et al. [7, 8] propose an algorithm for tracking the dominant singular subspace and singular values, called the Eigenspace Update Algorithm (EUA). Given an approximation to the dominant SVD of the first i columns of A, $A_{(1:i)} \approx U_i \Sigma_i V_i^T$, and the next column, a, the EUA updates the factors as follows:

$$\begin{aligned} a_\perp &= (I - U_i U_i^T) a \\ u &= a_\perp / \|a_\perp\|_2 \\ T &= \begin{bmatrix} \Sigma_i & U_i^T a \\ 0 & u^T a \end{bmatrix} = W \hat{\Sigma}_{i+1} Q^T \\ \hat{U}_{i+1} &= \begin{bmatrix} U_i & u \end{bmatrix} W \\ \hat{V}_{i+1} &= \begin{bmatrix} V_i & 0 \\ 0 & 1 \end{bmatrix} Q. \end{aligned}$$

U_{i+1}, Σ_{i+1}, and V_{i+1}^T are obtained by truncating the least significant singular values and vectors from $\hat{U}_{i+1}$, $\hat{\Sigma}_{i+1}$, and $\hat{V}_{i+1}^T$, respectively. All vectors corresponding to the singular values lower than some user-specified threshold, ϵ, are truncated. The EUA was the first algorithm to adaptively track the dominant subspace.

The SVD of T can be obtained either via a standard dense SVD algorithm or by utilizing the GES method mentioned above. The GES produces U_{i+1}, V_{i+1}, and Σ_{i+1} in $O(mk)$. However, the overhead of this method makes it worthwhile only for large values k. Otherwise, a dense SVD of T produces U_{i+1}, V_{i+1}, and Σ_{i+1} in $O(mk^2)$. Note also that the arrowhead-based method is only possible if a single row or column is used to update the SVD at

each step. Later methods allow more efficient inclusion of multiple rows or columns. The formation of the intermediate matrices in the algorithms discussed is rich in computation involving block matrix operations, taking advantage of the memory hierarchy of modern machines. Furthermore, it will be shown later that bringing in multiple columns at a time can yield better numerical performance. The UDV and QRW methods in the following sections utilize block algorithms to reap these benefits.

2.1.2 Sequential Karhunen-Loeve

In [9], Levy and Lindenbaum propose an approach for incrementally computing the a basis for the dominant left singular subspace. Their algorithm, the *Sequential Karhunen-Loeve* (SKL), allows a block of columns to be brought in on each step, and the authors recommend a block size which minimizes the overall complexity of the algorithm, assuming the number of columns per block is under user control. While the work of Levy and Lindenbaum concerns finding the KL basis (the dominant left singular basis), their technique can be modified to compute a low-rank factorization of A without dramatically affecting the performance.

The update is analogous to that of the EUA. The incoming vectors P (of size $m \times l$) are separated into components, $U^T P$ and $\tilde{P}^T P$, contained in and orthogonal to the current dominant space:

$$B' = \left[\begin{array}{c|c} B & P \end{array}\right] = \left[\begin{array}{c|c} U & \tilde{P} \end{array}\right] \left[\begin{array}{c|c} D & U^T P \\ \hline 0 & \tilde{P}^T P \end{array}\right] \left[\begin{array}{c|c} V^T & 0 \\ \hline 0 & I \end{array}\right] = U' D' V'^T.$$

Next, the SVD of D' is computed,

$$D' = \tilde{U} \tilde{D} \tilde{V}^T.$$

The SVD of B' clearly is

$$B' = U' D' V'^T = (U' \tilde{U}) \tilde{D} (V' \tilde{V})^T.$$

Finally, the rank of the dominant space is determined, based on a user specified threshold and the noise space is truncated.

The SVD of D' is computed in a negligible $O(k^3)$ per step, but the formation of the dominant part of $U'\tilde{U}$ requires $2m(k+l)k$. Combined with the formation of U' from U and P in $4m(k+l)l$, this yields a total complexity of $2mn\frac{k^2+3lk+2l^2}{l}$ to process the entire matrix A. It is shown that, assuming a fixed size for k, a block size l can be determined that minimizes

the total operations. They show that the value of $l = \frac{k}{\sqrt{2}}$ yields a minimal operation count of $(4\sqrt{2}+6)mnk \approx 12mnk$. The authors make qualitative claims about the convergence of the approximation under certain assumptions, but they give neither quantitative explanation nor rigorous analysis.

In [10], Brand proposes an algorithm similar to that of Levy and Lindenbaum. By using an update identical to that of the SKL, followed by a dense SVD of the $k \times k$ middle matrix and a matrix multiplication against the large current approximate basis, the algorithm has a larger leading coefficient than is necessary. Brand's contribution is the ability of his algorithm to handle missing or uncertain values in the input data, a feature not pursued in this thesis.

2.2 QRW methods

The defining characteristic of the UDV-based algorithms is that they produce at each step an approximation to the singular bases, instead of some other bases for the dominant singular subspaces. However, if the goal of the algorithm is to track the dominant subspace, then all that is required is to separate the dominant subspace from the noise subspace at each step, so that the basis for the noise space can be truncated. This is the technique that the QRW methods use to their computational advantage.

In [11], Chahlaoui, Gallivan and Van Dooren propose an algorithm for tracking the dominant singular subspace. Their Incremental QRW (IQRW) algorithm produces approximate bases for the dominant left singular subspace in $8mnk + O(nk^3)$ operations. It can also produce both bases for the left and right dominant singular subspaces in $10mnk + O(nk^3)$ operations.

This efficiency over the earlier algorithms is a result of a more efficient kernel step. On each step, the next column of A (denoted by a) is used to update the dominant basis. Given a current low-rank factorization, QRW^T, the step begins by updating the existing transformation in a Gram-Schmidt-type procedure identical to those used in the previously discussed algorithms:

$$
\begin{aligned}
z &= Q^T a \\
\hat{a} &= a - Qz \\
\rho &= \|\hat{a}\|_2 \\
q &= \hat{a}/\rho.
\end{aligned}
$$

This produces a new factorization

$$
\left[\begin{array}{cc} QR & a \end{array}\right] = \left[\begin{array}{cc} Q & q \end{array}\right] \left[\begin{array}{cc} R & z \\ 0 & \rho \end{array}\right] \left[\begin{array}{cc} W^T & 0 \\ 0 & 1 \end{array}\right] = \hat{Q}\hat{R}\hat{W}^T.
$$

The crucial difference between this algorithm and the UDV algorithms occurs next, in the downdating step. Where the previous algorithms compute the SVD of $\hat{R} = U_R \Sigma_R V_R^T$, the IQRW computes transformations G_u and G_v that decouple dominant and noise spaces in $R_{up} = G_u^T \hat{R} G_v$, allowing the noise space to be discarded without the (possible) extra work involved in forming the singular vectors. G_u and G_v are produced using the SVD of the $(k+1) \times (k+1)$ matrix $\hat{R}$. More specifically, only "smallest" left singular vector u_{k+1} is needed, though the $O(k^3)$ cost for an SVD of $\hat{R}$ should be negligible if $m \gg n \gg k$. These transformations are constructed so that their application to $\hat{Q}$ and $\hat{W}^T$ can be more efficient than a dense matrix-matrix multiplication. The cost of the reduced complexity is that the singular vectors are not available at each step. However, as is shown later, this method allows greater flexibility in the trade-off between performance and accuracy of the incremental calculation.

An error analysis is presented in [11] that considers the effect of truncation at each step. Error bounds are derived that are essentially independent of the problem size, allowing the use of the algorithm for large problem sizes. Also, to quell concerns about numerical problems from the Gram-Schmidt procedure used in the update step, an error analysis that bounds the loss of orthogonality in the computed basis vectors is also presented.

This thesis builds upon the work of the previous authors, mainly that of [11]. Chapter 3 defines a generic incremental algorithm that unifies the current methods. An exploration of this algorithm reveals a block implementation that is more efficient than any previous method.

CHAPTER 3

A BLOCK INCREMENTAL ALGORITHM

This chapter outlines a generic block, incremental technique for estimating the dominant left and right singular subspaces of a matrix. The technique is flexible, in that it can be adapted to the specific requirements of the application, e.g. left space bases only, left and right space bases, singular vectors, etc. Such variants are discussed in this chapter, and their efficiency is characterized using their operation count.

Section 3.1 introduces a generic technique for isolating the dominant subspaces of a matrix from the dominated subspaces. Section 3.2 describes an algorithm for incrementally tracking the dominant left and/or right singular subspaces of a matrix, using the technique introduced in Section 3.1. Section 3.3 discusses implementations of this generic algorithm, and analyzes the relationship between the operation count of the algorithm and the computational results. Furthermore, a new algorithm is proposed which has a lower operation count than existing methods. Finally, Section 3.4 discusses techniques for obtaining the singular vectors using methods which do not explicitly produce them.

3.1 A Generic Separation Factorization

Given an $m \times (k+l)$ matrix M and its QR factorization,

$$M = \begin{bmatrix} \overbrace{Q_1}^{k+l} & \overbrace{Q_2}^{m-k-l} \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R,$$

consider the SVD of R and partition it conformally as

$$R = U \Sigma V^T = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1 & V_2 \end{bmatrix}^T,$$

where U_2, Σ_2, and V_2 contain the smallest l left singular vectors, values and right singular vectors of R, respectively. Let the orthogonal transformations G_u and G_v be such that they block diagonalize the singular vectors of R,

$$G_u^T U = \begin{bmatrix} T_u & 0 \\ 0 & S_u \end{bmatrix} \quad \text{and} \quad G_v^T V = \begin{bmatrix} T_v & 0 \\ 0 & S_v \end{bmatrix}. \tag{3.1}$$

Applying these transformations to R yields $R_{new} = G_u^T R G_v$. G_u and G_v rotate R to a coordinate system where its left and right singular bases are block diagonal. It follows that R_{new} has the form

$$\begin{aligned} R_{new} &= G_u^T R G_v = G_u^T U \Sigma V^T G_v \\ &= \begin{bmatrix} T_u & 0 \\ 0 & S_u \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} T_v^T & 0 \\ 0 & S_v^T \end{bmatrix} \\ &= \begin{bmatrix} T_u \Sigma_1 T_v^T & 0 \\ 0 & S_u \Sigma_2 S_v^T \end{bmatrix}. \end{aligned} \tag{3.2}$$

The SVD of the block diagonal matrix R_{new} has a block diagonal structure. This gives a new factorization of M:

$$M = Q_1 R = (Q_1 G_u)(G_u^T R G_v) G_v^T = \hat{Q} R_{new} G_v^T = \hat{Q} \begin{bmatrix} T_u \Sigma_1 T_v^T & 0 \\ 0 & S_u \Sigma_2 S_v^T \end{bmatrix} G_v^T,$$

whose partitioning identifies bases for the dominant left and right singular subspaces of M in the first k columns of $\hat{Q}$ and G_v. It should be noted that G_u is not uniquely defined by Equation (3.1). This definition admits any G_u whose first k columns are some orthonormal basis for the dominant left singular subspace of R, and whose last l columns therefore are some orthonormal basis for the dominated (weaker) left singular subspace of R. This is also the case, *mutatis mutandis*, for G_v.

3.2 An Incremental Method

The factorization of the previous section can be used to define a generic method that requires only one pass through the columns of an $m \times n$ matrix A to compute approximate bases for the left and right dominant singular subspaces. The procedure begins with a QR factorization of the first k columns of A, denoted $A_{(1:k)} = Q_0 R_0$, and with the right space basis initialized to $W_0 = I_k$.

The first expansion step follows with $i = 1$ and $s_0 = k$, where i refers to the step/iteration of the algorithm and $s_i = s_{i-1} + l_i$ refers to the number of columns of A that have been

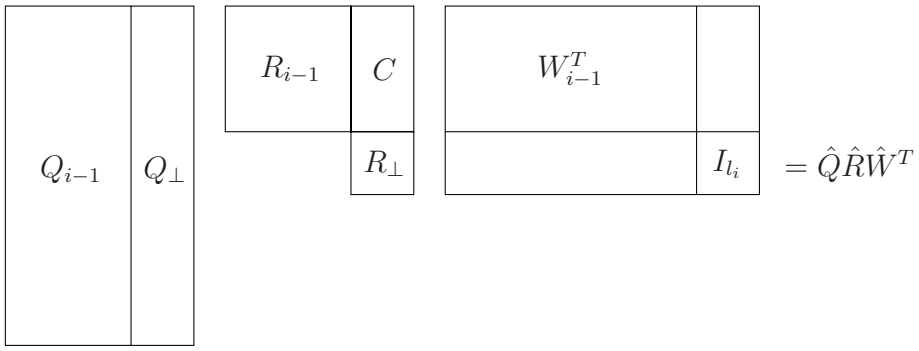

Figure 3.1: The structure of the expand step.

"processed" after completing step i. The l_i incoming columns of A, denoted $A_+ = A_{(s_{i-1}+1:s_i)}$, are used to expand Q_{i-1} and R_{i-1} via a Gram-Schmidt procedure:

$$\begin{aligned} C &= Q_{i-1}^T A_+ \\ A_\perp &= A_+ - Q_{i-1}Q_{i-1}^T A_+ = A_+ - Q_{i-1}C \\ A_\perp &= Q_\perp R_\perp. \end{aligned}$$

The $l_i \times l_i$ identity is appended to W_{i-1}^T to expand the $k \times s_{i-1}$ matrix to $k + l_i \times s_i$, producing a new factorization

$$\begin{bmatrix} Q_{i-1}R_{i-1}W_{i-1}^T & A_+ \end{bmatrix} = \hat{Q}\hat{R}\hat{W}^T, \tag{3.3}$$

the structure of which is shown in Figure 3.1.

Transformations G_u and G_v are constructed to satisfy Equation (3.1). These transformations are applied to the block triangular matrix $\hat{R}$ to put it in a block diagonal form that isolates the dominant singular subspaces from the dominated subspaces, as follows:

$$\begin{aligned} \hat{Q}\hat{R}\hat{W}^T &= \hat{Q}(G_u G_u^T)\hat{R}(G_v G_v^T)\hat{W}^T \\ &= (\hat{Q}G_u)(G_u^T \hat{R} G_v)(G_v^T \hat{W}^T) \\ &= \bar{Q}\bar{R}\bar{W}^T. \end{aligned}$$

The structure of $\bar{Q}\bar{R}\bar{W}^T$ is shown in Figure 3.2.

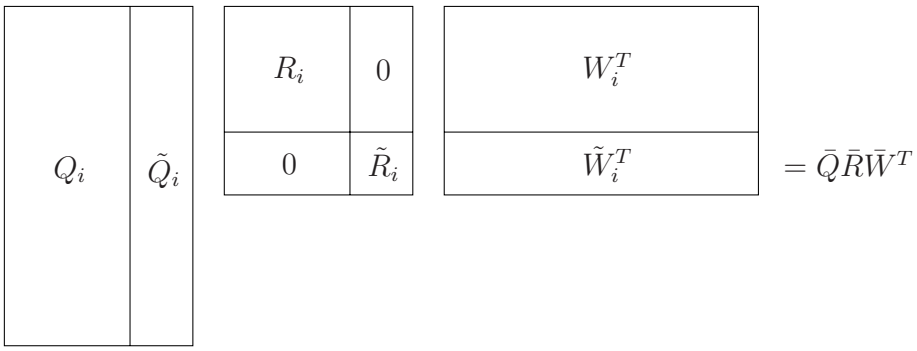

Figure 3.2: The result of the deflate step.

The dominant singular subspaces for $\left[\begin{array}{cc} Q_{i-1}R_{i-1}W_{i-1}^T & A_+ \end{array}\right]$ are contained in the first k columns of $\bar{Q}$ and rows of $\bar{W}^T$. The columns $\tilde{Q}_i$, rows $\tilde{W}_i^T$, and the last columns and rows in $\bar{R}$ are truncated to yield Q_i, R_i, and W_i^T, which are $m \times k$, $k \times k$, and $k \times s_i$, respectively. This rank-k factorization is an approximation to the first s_i columns of A.

The output at step i includes

- Q_i - an approximate basis for the dominant left singular space of $A_{(1:s_i)}$,
- W_i - an approximate basis for the dominant right singular space of $A_{(1:s_i)}$, and
- R_i - a $k \times k$ matrix whose SVD contains the transformations that rotate Q_i and W_i to the dominant singular vectors. The singular values of R_i are estimates for the singular values of $A_{(1:s_i)}$.

Note that after the i-th step, there exists an orthogonal matrix V_i embedding W_i and relating the first s_i columns to the current approximation of A and the discarded data up to this point:

$$A_{(1:s_i)}V_i \;=\; A_{(1:s_i)}\left[\begin{array}{cc} \overbrace{W_i}^{k_i} & \overbrace{W_i^\perp}^{s_i-k_i} \end{array}\right] = \left[\begin{array}{cccc} \overbrace{Q_iR_i}^{k_i} & \overbrace{\tilde{Q}_1\tilde{R}_1}^{d_1} & \cdots & \overbrace{\tilde{Q}_i\tilde{R}_i}^{d_i} \end{array}\right].$$

More specifically, after the final step f of the algorithm, there exists V_f such that

$$A_{(1:s_f)}V_f = A\left[\begin{array}{cc} W_f & W_f^\perp \end{array}\right] = \left[\begin{array}{ccccc} Q_fR_f & \tilde{Q}_1\tilde{R}_1 & \cdots & \tilde{Q}_f\tilde{R}_f \end{array}\right],$$

yielding the following additive decomposition:

$$A = Q_f R_f W_f^T + \left[\begin{array}{ccc} \tilde{Q}_1 \tilde{R}_1 & \ldots & \tilde{Q}_f \tilde{R}_f \end{array} \right] W_f^{\perp T}.$$

This property is proven in Appendix A and is used to construct bounds [12] on the error of the computed factorization.

3.3 Implementing the Incremental SVD

In this section, the current methods for incrementally computing dominant singular subspaces are described in terms of the generic algorithmic framework of Section 3.2. The computation in this framework can be divided into three steps:

1. The Gram-Schmidt expansion of Q, R, and W. This step is identical across all methods and requires $4mkl$ to compute the coordinates of A_+ onto Q and the residual $A_\perp$, and $4ml^2$ to compute an orthonormal basis for $A_\perp$, the cost per step is $4ml(k+l)$. The total cost over $\frac{n}{l}$ steps is $4mn(k+l)$, one of two cost-dominant steps in each method.

2. Each method constructs G_u and G_v based on the SVD of $\hat{R}$. Whether the SVD is computed using classical methods (as in the SKL and IQRW) or accelerated methods (using the GES as in the EUA) determines the complexity. All methods discussed in this thesis depend on the ability to perform this step in at most $O(k^3)$ computations, which is negligible when $k \ll n \ll m$.

3. Finally, the computation of G_u and G_v varies across methods, as does the application of these transformations to $\hat{Q}$ and $\hat{W}$. The methods are distinguished by the form of computation in Steps 2 and 3.

The following subsections describe the approach taken by each method to implement Steps 2 and 3. The methods discussed are the EUA, SKL, and IQRW algorithms, and the Generic Incremental algorithm (GenInc). Complexity is discussed in terms of operation count, with some discussion of memory requirements and the exploitation of a memory hierarchy.

3.3.1 Eigenspace Update Algorithm

Recall that in the generic framework, the end products of each step are bases for the dominant singular subspaces and a low-rank matrix encoding the singular values and the rotations necessary to transforms the current bases to the singular bases. The EUA, however, does more than that. Instead of producing the factorization in a QBW form, EUA outputs in a UDV form. The bases output by this method are the singular bases, i.e., those bases composed of the ordered singular vectors.

This is accomplished by the choice of the transformations G_u and G_v. For G_u and G_v, the EUA uses the singular vectors of $\hat{R}$, $G_u = \hat{U}$ and $G_v = \hat{V}$. These choices clearly satisfy the conditions in Equation (3.1), as

$$\begin{aligned} G_u^T \hat{U} &= \begin{bmatrix} U_1^T \\ U_2^T \end{bmatrix} \begin{bmatrix} U_1 & U_2 \end{bmatrix} = \begin{bmatrix} I_{k_i} & 0 \\ 0 & I_{d_i} \end{bmatrix} \quad \text{and} \\ G_v^T \hat{V} &= \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} \begin{bmatrix} V_1 & V_2 \end{bmatrix} = \begin{bmatrix} I_{k_i} & 0 \\ 0 & I_{d_i} \end{bmatrix}. \end{aligned}$$

By using the singular vectors for G_u and G_v, this method puts R_{new} into a diagonal form, such that the Q_i and W_i produced at each step are approximations to the dominant singular bases. That is, the method produces the singular value decomposition of a matrix that is near to $A_{(1:s_i)}$.

The EUA uses the Gram-Schmidt expansion described in Section 3.2. The matrix $\hat{R}$ is a broken arrowhead matrix, having the form

$$\hat{R} = \begin{bmatrix} \sigma_1 & & & \zeta_1 \\ & \ddots & & \vdots \\ & & \sigma_k & \zeta_n \\ & & & \rho \end{bmatrix} = \hat{U}\hat{\Sigma}\hat{V}^T.$$

For the SVD of the $\hat{R}$, the authors propose two different methods. The first utilizes the Gu and Eisenstat SVD (GES) [6]. By employing this method, the singular values of $\hat{R}$ can be computed and G_u and G_v can be applied in $O(mk \log_2^2 \epsilon)$ (where ϵ is the machine precision). With a complexity of over $(52)^2 mk$ per update (for a machine with 64-bit IEEE floats), this complexity is prohibitive except when k is large. Furthermore, GES requires a broken arrowhead matrix, restricting this approach to a scalar ($l_i = 1$) algorithm.

If the GES method is not used, however, then the computation of $\hat{Q}G_u$ and $\hat{W}G_v$ require $2mk^2$ and $2s_ik^2$, respectively. Passing through all n columns of A, along with the Gram-

Schmidt update at each step, requires $8mnk + 2mnk^2 + O(n^2k^2) + O(nk^3)$. For this number of computations, the R-SVD could have been used to produce the exact SVD of A.

In terms of operation count, the EUA is only interesting in the cases where the efficiency of GES can be realized. Even in these cases, where the overall computation is $O(mnk)+O(n^2k)+O(nk^2)$, the leading order constant (at least $\log_2^2 \epsilon$) exceeds those of other methods. As for the non-GES, matrix multiply method, the $O(mnk^2) + O(n^2k^2) + O(nk^3)$ is much slower than that of other methods, which only grow linearly with k in the dominant terms.

3.3.2 Sequential Karhunen-Loeve

The SKL method proceeds in a manner similar to the EUA. Using the Gram-Schmidt expansion described in the generic framework, the SKL method expands the current bases for the dominant singular subspaces to reflect the incoming columns A_+.

The SKL method also uses the singular vectors U and V for the transformations G_u and G_v. Again, these specific transformations produce not just bases for the dominant singular subspaces, but the dominant singular bases.

The authors are interested in approximating the Karhunen-Loeve basis (the basis composed of the dominant left singular vectors). They do not consider the update of the right singular basis. However, as this thesis is interested in the computation of an approximate singular value decomposition, the update of the right singular subspace and calculations of complexity are included (although the added complexity is not significant when $m \gg n$). In the remainder of this thesis when discussing the SKL it is assumed that the right singular vectors are produced at each step, in addition to the left singular vectors. To make this explicit, this two-sided version of the of the SKL algorithm is referred to as the SKL-LR.

Producing the singular vectors and values of the $k_i + l_i \times k_i + l_i$ matrix $\hat{R}$ requires a dense SVD, at $O(k_i^3 + l_i^3)$ operations. The matrix multiply $\hat{Q}G_u$, producing only the first k_i columns, requires $2mk_i(k_i + l_i)$ operations. Likewise, producing the first k_i columns of $\hat{W}G_v$ requires $2s_ik_i(k_i + l_i)$. Combining this with the effort for the SVD of $\hat{R}$ and the Gram-Schmidt expansion, the cost per step of the SKL-LR algorithm is $2m(k_i^2 + 3k_il_i + 2l_i^2) + O(k_i^3) + 2s_ik_i(k_i + l_i)$. Fixing k_i and l_i and totaling this over n/l

steps of the algorithm, the total cost is

$$\begin{aligned} C_{skl} &= \sum_i 2m(k_i^2 + 3k_i l_i + 2l_i^2) + O(k_i^3) + 2s_i(k_i + l_i) \\ &\approx \sum_{i=1}^{n/l} \left[2m(k^2 + 3kl + 2l^2) + O(k^3) + 2ilk(k + l)\right] \\ &= 2mn\frac{k^2 + 3kl + 2l^2}{l} + O(\frac{nk^3}{l}) + 2lk(k + l)\sum_{i=1}^{n/l} i \\ &= 2mn\frac{k^2 + 3kl + 2l^2}{l} + O(\frac{nk^3}{l}) + \frac{n^2k(k + l)}{l}. \end{aligned}$$

Neglecting the terms not containing m (appropriate when $k \ll n \ll m$ or when only updating the left basis), the block size $l = k/\sqrt{2}$ minimizes the number of floating point operations performed by the algorithm. Substituting this value of l into the total operation count yields a complexity of $(6 + 4\sqrt{2})mnk + (1 + \sqrt{2})n^2k + O(nk^2) \leq 12mnk + 3n^2k + O(nk^2)$.

Note that this linear complexity depends on the ability to choose $l_i = \frac{k_i}{\sqrt{2}}$ at each step. Many obstacles may prevent this. The SKL-LR algorithm requires $m(k+l)$ memory to hold $\hat{Q}$ and mk workspace for the multiplication $\hat{Q}G_u$. If memory is limited, then the size of k cuts into the available space for l. Also, as in the active recognition scenario, an update to the dominant singular vectors might be required more often than every $\frac{k_i}{\sqrt{2}}$ snapshots, so that $l < \frac{k}{\sqrt{2}}$. As l approaches 1, the cost for each step approaches $2mk^2 + 6mk$, with the cost over the entire matrix approaching $2mnk^2 + 6mnk + n^2k^2 + O(nk^3)$, which is a higher complexity than computing the leading k singular vectors and values of A using the R-SVD.

3.3.3 Incremental QRW

The Incremental QRW method of Chahlaoui et al. [12] is only described for the scalar case ($l = 1$), though the authors allude to a "block" version. A presentation of this block version is described here.

Using the same Gram-Schmidt update employed by the generic algorithm and the previously described methods, the authors describe the construction of G_u and G_v. The transformation G_u is constructed such that

$$G_u^T U_2 = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix},$$

and G_v such that $G_u^T \hat{R} G_v = R_{new}$ is upper triangular. It is easily shown that

$$\begin{aligned} R_{new}^T \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix} &= \begin{bmatrix} 0 \\ R_3^T \end{bmatrix} \\ &= \tilde{V}_2 \Sigma_2 \end{aligned}$$

where $\tilde{V}_2 = G_v^T V_2$ are the d_i weaker right eigenvectors of R_{new}.

If $\hat{R}$ is non-singular (as assumed by the authors), and therefore Σ_2 is as well, then $\tilde{V}_2 = \begin{bmatrix} 0 \\ R_3^T \end{bmatrix} \Sigma_2^{-1}$ and, by its triangularity and orthogonality, $\tilde{V}_2 = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix}$. This "transform-and-triangularize" technique of the IQRW satisfies Equation (3.1). Furthermore, the resulting factorization of each step contains an upper triangular matrix instead of a dense block, detailing the source of the name IQRW.

To compute the transformations G_u and G_v at each step, the authors propose two different methods. In the case where the left and right subspaces are tracked, both $\hat{Q} G_u$ and $\hat{W} G_v$ are computed. Therefore, the applications of G_u and G_v must be as efficient as possible. The authors present a technique that uses interleaved Givens rotations to build G_u from U_2, while at the same time applying the rotations to $\hat{R}$, and building G_v to keep $G_u \hat{R}$ upper triangular. These rotations are applied to $\hat{Q}$ and $\hat{W}$ as they are constructed and applied to $\hat{R}$. To transform U_2 to $\begin{bmatrix} 0 \\ I_d \end{bmatrix}$ requires $kl + l^2/2$ rotations. Applying these rotations to $\hat{Q}$ then requires $6mkl + 3ml^2$. Furthermore, each rotation introduces an element of fill-in into the triangular matrix $\hat{R}$, which must be eliminated by G_v. Absorbing these rotations into $\hat{W}$ requires $6s_ikl + 3s_il^2$ flops.

Including Steps 1 and 2 of the generic algorithm, the overall complexity of the two-sided, Givens-based technique (over n/l steps) is $10mnk + 7mnl + 3n^2k + \frac{3}{2}n^2l + O(\frac{n}{l}k^3)$. This Givens-based method was proposed to lower the cost of updating the right basis. The motivation for this construction of G_u and G_v was to minimize the amount of work required to update the right basis at each step. This is significant when $l = 1$, the scenario in which this algorithm was generated, because the update step is performed n times, instead of $\frac{n}{l}$. For this scenario, the Givens-based IQRW requires only $10mnk + 3n^2k + O(nk^3)$.

However, if the right basis is not tracked (and $\hat{W} G_v$ therefore is not needed), the authors propose a different technique that reduces the cost of the left basis update. By using Householder reflectors instead of Givens rotations, G_u can be composed of l reflectors of order $k+1$ to $k+l$. The cost of applying these to $\hat{Q}$ is $4mkl + 2ml^2$. Restoring $G_u^T \hat{R}$ to upper

triangular is done via an RQ factorization, of negligible cost. Including Steps 1 and 2, the complexity of this Householder-based IQRW (over $\frac{n}{l}$ update steps) is $8mnk+6mnl+O(\frac{n}{l}k^3)$. When used in the $l = 1$ scenario, the complexity becomes $8mnk + O(nk^3)$.

It should be noted that this method of defining G_u and G_v does not succeed when $\hat{R}$ is singular. A simple counterexample illustrates this. If $U_2 = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix}$, then $G_u = I$, and $G_u^T \hat{R}$ is still upper triangular. Then G_v also is the identity, and

$$R_{new} = \hat{R} = \begin{bmatrix} R_1 & R_2 \\ 0 & 0 \end{bmatrix}.$$

The dominant and dominated singular subspaces are clearly not separated in this R_{new}.

A modification of this transform-and-triangularize method is always successful. Switch the roles of G_u and G_v, so that

$$G_v^T V_2 = \tilde{V}_2 = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix}$$

and G_u restores $\hat{R}G_v$ to upper triangular. This is the a well-known technique for subspace tracking described in Chapter 5 of [2]. Using this method, it can be shown that the dominant and dominated subspaces are decoupled in the product R_{new}. However, there is a drawback to constructing G_u in this fashion. In the cases where $m \gg n$ (the assumption made in [12] and this thesis), the goal is to minimize the number of operations on the large, $m \times k + l$ matrix $\hat{Q}$. When G_u is constructed, as in the IQRW, based on U_2, it can be applied more efficiently than when G_u is constructed to retriangularize $\hat{R}G_v$.

3.3.4 Generic Incremental Algorithm

The separation technique described in Section 3.2 is the heart of the generic incremental SVD. It requires only that the first k columns of G_u are a basis for the dominant singular subspace of $\hat{R}$ and that the last d columns of G_u are a basis for the dominated singular subspace. The EUA and the SKL-LR both compose G_u specifically from the basis composed of the left singular vectors. This is done in order to produce a UDV factorization at every step.

Alternatively, the IQRW uses bases for the dominant subspaces which leave $G_u^T \hat{R} G_v$ in an upper triangular form. The upper triangular middle matrix offers advantages over an unstructured square matrix in that the storage requirement is cut in half, the system formed

by the matrix is easily solved, and operations such as tridiagonalization and multiplication can be performed more efficiently. While the middle matrix R is only $k \times k$, the above operations can become significant as k becomes large, especially as R exceeds cache size.

This section describes an implementation, the Generic Incremental Algorithm (GenInc). The GenInc is similar to the IQRW, in that it focuses on lowering the operation count via a special construction of G_u and G_v. The GenInc has two variants, each which is preferable under different parametric circumstances. These variants are described in the following subsection.

3.3.4.1 GenInc - Dominant Space Construction

Recall that the generic block algorithm requires that orthogonal transformations G_u and G_v be constructed which block diagonalize the left and right singular vectors of $\hat{R}$:

$$G_u^T U = \begin{bmatrix} T_u & 0 \\ 0 & S_u \end{bmatrix} \quad \text{and} \quad G_v^T V = \begin{bmatrix} T_v & 0 \\ 0 & S_v \end{bmatrix}.$$

This is equivalent to constructing transformations G_u and G_v such that

$$G_u^T U_1 T_u^T = \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} \quad \text{and} \quad G_v^T V_1 T_v^T = \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} \tag{3.4}$$

or

$$G_u^T U_2 S_u^T = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix} \quad \text{and} \quad G_v^T V_2 S_v^T = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix}. \tag{3.5}$$

This means that the T_u and S_u transformations can be specified to rotate the singular bases to other bases that are more computationally friendly. Since only the first k_i columns of the products $\hat{Q}G_u$ and $\hat{W}G_v$ are to be kept, it is intuitive that working with Equation (3.4) may be more computationally promising. As the construction of T_u and G_u, based on U_1, is analogous to that of T_v and G_v, based instead on V_1, the process is described below for the left transformations only, with the right side transformations proceeding in the same manner, *mutatis mutandis*.

The remaining issue is how much structure can be exploited in $G_1 = U_1 T_u^T$. Since only the first k_i columns of $\hat{Q}G_u$, $\hat{Q}G_u \begin{bmatrix} I_{k_i} & 0 \end{bmatrix}^T = \hat{Q}G_1$ are needed and G_1 must be a basis for the dominant left singular subspace, there is a limit on the number of zeroes that can be introduced with T_u^T. Taking the $k_i + d_i \times k_i$ matrix U_1, construct an orthogonal matrix T_u that transforms U_1 to an upper-trapezoidal matrix, "notching" the lower left-hand corner

$$G_1 = U_1 T_u^T = \begin{bmatrix} \upsilon_{1,1} & \upsilon_{1,2} & \upsilon_{1,3} \\ \upsilon_{2,1} & \upsilon_{2,2} & \upsilon_{2,3} \\ \upsilon_{3,1} & \upsilon_{3,2} & \upsilon_{3,3} \\ \upsilon_{4,1} & \upsilon_{4,2} & \upsilon_{4,3} \\ \upsilon_{5,1} & \upsilon_{5,2} & \upsilon_{5,3} \\ \upsilon_{6,1} & \upsilon_{6,2} & \upsilon_{6,3} \end{bmatrix} T_u^T = \begin{bmatrix} \eta_{1,1} & \eta_{1,2} & \eta_{1,3} \\ \eta_{2,1} & \eta_{2,2} & \eta_{2,3} \\ \eta_{3,1} & \eta_{3,2} & \eta_{3,3} \\ \eta_{4,1} & \eta_{4,2} & \eta_{4,3} \\ 0 & \eta_{5,2} & \eta_{5,3} \\ 0 & 0 & \eta_{6,3} \end{bmatrix}$$

Figure 3.3: The notching effect of T_u on U_1, with $k_i = 3, d_i = 3$.

as illustrated in Figure 3.3. This is done by computing the RQ factorization $U_1 = G_1 T_u$. $G_1 = U_1 T_u^T$ is the upper trapezoidal matrix shown in Figure 3.3

It follows that G_1 is of the form

$$G_1 = \begin{bmatrix} B \\ U \end{bmatrix},$$

where B is an $d_i \times k_i$ dense block and U is a $k_i \times k_i$ upper-triangular matrix. Any G_u embedding G_1 as

$$G_u = \begin{bmatrix} G_1 & G_1^\perp \end{bmatrix}$$

clearly satisfies Equation (3.4), for any $G_1^\perp$ that completes the space.

Computing the first k_i columns of $\hat{Q} G_u$ then consists of the following:

$$\begin{aligned} \hat{Q} G_u \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} &= \hat{Q} \begin{bmatrix} G_1 & G_1^\perp \end{bmatrix} \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} \\ &= \hat{Q} G_1 \\ &= \hat{Q} \begin{bmatrix} B \\ U \end{bmatrix} \\ &= \hat{Q}_{(1:d_i)} B + \hat{Q}_{(d_i+1:d_i+k_i)} U. \end{aligned}$$

This computation requires $2md_ik_i$ for the product $\hat{Q}_{(1:d_i)}B$ and mk_i^2 for the product $\hat{Q}_{(d_i+1:d_i+k_i)}U$ (because of the triangularity of U), for a total of $mk_i(2d_i + k_i)$ to produce the first k_i columns of $\hat{Q}G_u$. Similarly, the cost to produce the first k_i columns of $\hat{W}G_v$ is $s_ik_i(2d_i + k_i)$.

The total cost per step, including the Gram-Schmidt update and the SVD of $\hat{R}$, is $4ml_i(k_{i-1} + l_i) + mk_i(2d_i + k_i) + s_ik_i(2d_i + k_i) + O(k_i^3 + d_i^3)$. Assuming fixed values of k_i and

l_i on each step, the total cost of the algorithm over the entire matrix becomes

$$\begin{aligned} C_{ds} &= \sum_i \left[4ml_i(k_{i-1}+l_i) + mk_i(2d_i+k_i) + s_ik_i(2d_i+k_i) + O(k_i^3+d_i^3)\right] \\ &\approx \sum_{i=1}^{n/l} \left[4ml(k+l) + mk(2l+k) + ilk(2l+k) + O(k^3+l^3)\right] \\ &= 6mnk + 4mnl + \frac{mnk^2}{l} + O(\frac{nk^3}{l} + nl^2) + lk(2l+k)\sum_{i=1}^{n/l} i \\ &= 6mnk + 4mnl + \frac{mnk^2}{l} + O(\frac{nk^3}{l} + nl^2) + \frac{n^2(2kl+k^2)}{2l}. \end{aligned}$$

Fixing m, n, and k and neglecting terms not containing m, the block size $l = \frac{k}{2}$ minimizes the overall complexity of the algorithm, yielding a complexity of $10mnk + 2n^2k + O(nk^2)$. The memory requirement is just $m(k+l)$ for the matrix $\hat{Q}$. Note that the dominant space construction requires no work space. This is because the triangular matrix multiplication to form $\hat{Q}G_1$ can be performed in-situ, and the rectangular matrix multiplication can be accumulated in this same space.

3.3.4.2 GenInc - Dominated Space Construction

It is worthwhile to investigate the result of forming G_u based on Equation (3.5). Recall that in this scenario, G_u is defined such that

$$G_u^T U_2 S_u^T = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix}.$$

This requires that the last d_i columns of G_u are equal to $U_2S_u^T$. First a transformation S_u^T that notches the upper-right hand corner of U_2 as shown in Figure 3.4 is constructed.
Any G_u of the form

$$G_u = \begin{bmatrix} G_2^\perp & G_2 \end{bmatrix}$$

satisfies Equation (3.5), for some completion of the basis, $G_2^\perp$. However, unlike before, the completed basis $G_2^\perp$ is needed, because the first k_i columns of $\hat{Q}G_u$ are given by the quantity $\hat{Q}G_2^\perp$.

In this case, G_u can be obtained directly from Equation (3.5) using Householder reflectors, such that

$$G_u^T(U_2S_u^T) = H_{d_i}^T \dots H_1^T(U_2S_u^T) = \begin{bmatrix} 0 \\ I_{d_i} \end{bmatrix}. \tag{3.6}$$

$$G_2 = U_2 S_u^T = \begin{bmatrix} \upsilon_{1,1} & \upsilon_{1,2} & \upsilon_{1,3} \\ \upsilon_{2,1} & \upsilon_{2,2} & \upsilon_{2,3} \\ \upsilon_{3,1} & \upsilon_{3,2} & \upsilon_{3,3} \\ \upsilon_{4,1} & \upsilon_{4,2} & \upsilon_{4,3} \\ \upsilon_{5,1} & \upsilon_{5,2} & \upsilon_{5,3} \\ \upsilon_{6,1} & \upsilon_{6,2} & \upsilon_{6,3} \end{bmatrix} S_u^T = \begin{bmatrix} \eta_{1,1} & 0 & 0 \\ \eta_{2,1} & \eta_{2,2} & 0 \\ \eta_{3,1} & \eta_{3,2} & \eta_{3,3} \\ \eta_{4,1} & \eta_{4,2} & \eta_{4,3} \\ \eta_{5,1} & \eta_{5,2} & \eta_{5,3} \\ \eta_{6,1} & \eta_{6,2} & \eta_{6,3} \end{bmatrix}$$

Figure 3.4: The notching effect of S_u on U_2, with $k_i = 3, d_i = 3$.

This is done by taking the QL factorization of the matrix $U_2 S_u^T$. Bischof and Van Loan describe in [13] how the product of d Householder reflectors can be represented in a block form, $H_1 \dots H_d = I + WY^T$. This allows the product of reflectors to be applied as a single, rank-d update instead of d rank-1 updates, exposing BLAS-3 primitives that can exploit a memory hierarchy.

By virtue of the notched form of $U_2 S_u^T$ and its having orthonormal columns, each of the H_i Householder reflectors above is of order $k_i + 1$. This structure is also seen in the block representation of G_u. It is shown in Appendix B that such a matrix has the following structure:

$$G = I + WY^T = I + \begin{bmatrix} L_1 \\ B_1 \end{bmatrix} \begin{bmatrix} U & B_2 & L_2 \end{bmatrix}$$

such that L_1 and L_2 are $d_i \times d_i$ lower-triangular matrices, U is a $d_i \times d_i$ upper-triangular matrix, B_1 is an $k_i \times d_i$ dense matrix, and B_2 is a $d_i \times k_i - d_i$ dense matrix. Producing the first k_i columns of $\hat{Q} G_u$ by applying this factored transformation then consists of the following:

$$\begin{aligned} \hat{Q} G_u \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} &= \hat{Q}(I + WY^T) \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} \\ &= \hat{Q} \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} + \hat{Q} \begin{bmatrix} L_1 \\ B_1 \end{bmatrix} \begin{bmatrix} U & B_2 & L_2 \end{bmatrix} \begin{bmatrix} I_{k_i} \\ 0 \end{bmatrix} \\ &= \hat{Q}_{(1:k_i)} + (\hat{Q}_{(1:d_i)} L_1 + \hat{Q}_{(d_i+1:d_i+k_i)} B_1) \begin{bmatrix} U & B_2 \end{bmatrix} \\ &= \hat{Q}_{(1:k_i)} + M \begin{bmatrix} U & B_2 \end{bmatrix} \\ &= \hat{Q}_{(1:k_i)} + \begin{bmatrix} MU & MB_2 \end{bmatrix}. \end{aligned}$$

The computational cost associated with this is

- md_i^2 for the triangular matrix multiplication of $\hat{Q}_{(1:d_i)}L_1$,
- $2mk_id_i$ for the matrix multiplication of $\hat{Q}_{(d_i+1:d_i+k_i)}B_1$,
- md_i^2 for the triangular matrix multiplication of $M = \hat{Q}_{(1:d_i)}L_1 + \hat{Q}_{(d_i+1:d_i+k_i)}B_1$ by U, and
- $2m(k_i - d_i)d_i$ for the matrix multiplication of MB_2.

The total cost for the factored update of the first k_i columns of $\hat{Q}$ then is $4mk_id_i$.

Note that this update is linear in m, k_i and d_i, as opposed to the update using the G_u, which had a complexity $mk_i(2d_i + k_i)$. The "dominant space" G_u is simpler to compute, requiring only the notching of U_1 and two matrix multiplications. However, the more complicated "dominated space" G_u has a lower complexity when $l \leq \frac{k}{2}$.

Including the Gram-Schmidt update and the SVD of $\hat{R}$, the cost per step when using the dominated space construction of G_u and G_v is $4ml_i(k_i + l_i) + 4mk_id_i + 4s_ik_id_i + O(k_i^3 + d_i^3)$. Assuming maximum values for k_i and l_i on each step, the total cost of the algorithm is

$$\begin{aligned}
C_{ws} &= \sum_i \left[4ml_i(k_i + l_i) + 4mk_id_i + 4s_ik_id_i + O(k_i^3 + d_i^3)\right] \\
&\approx \sum_{i=1}^{n/l} \left[4ml(k + l) + 4mkl + 4ilkl + O(k^3 + l^3)\right] \\
&= 8mnk + 4mnl + O(\frac{nk^3}{l} + nl^2) + 4kl^2 \sum_{i=1}^{n/l} i \\
&= 8mnk + 4mnl + O(\frac{nk^3}{l} + nl^2) + 2n^2k.
\end{aligned}$$

If $1 \leq l \leq \frac{k}{2}$, the motivating case, then this can be bounded:

$$\begin{aligned}
C_{ws} &= 8mnk + 4mnl + O(\frac{nk^3}{l} + nl^2) + 2n^2k \\
&\leq 8mnk + 2mnk + O(nk^3 + nk^2) + 2n^2k \\
&= 10mnk + 2n^2k + O(nk^3).
\end{aligned}$$

In this case, the cost related to computing the SVD of $\hat{R}$ is more significant than in the dominant space construction. This is a consequence of the smaller block sizes l_i, requiring that more steps of the algorithm be performed in order to pass through A.

Furthermore, unlike in the dominant space construction, this dominated space construction requires some working space, in addition to the $m(k+l)$ space required to store the matrix $\hat{Q}$. This work space requires ml memory, and is necessary to store the matrix $M = \hat{Q}_{(1:d_i)}L_1 + \hat{Q}_{(d_i+1:d_i+k_i)}B_1$.

3.3.5 Operation Count Comparison

The operation counts for the two GenInc implementations are lower than that of previous methods. Noting Table 3.1, the SKL-LR algorithm, has a total complexity of approximately $12mnk$. The Gram-Schmidt update and SVD of $\hat{R}$ are identical for both the GenInc and SKL-LR algorithms. The only difference is the update of the dominant singular bases. This imposes a cost of $2mk(k+l)$ for the SKL-LR algorithm, but only $2mk(k+l) - mk^2$ for the dominant space GenInc. Furthermore, when $l_i \leq k/2$, the lower-complexity, dominated space GenInc may be used, requiring only $4mkl$.

Table 3.1: Computational and memory costs of the block algorithms.

Algorithm	SKL-LR	GenInc (d.s.)	GenInc (w.s.)
Complexity	$12mnk$	$10mnk$	$\leq 10mnk$
Work Space	mk	none	ml

For equivalent values of k and l, the SKL-LR algorithm has a higher operation count than the GenInc. The only benefit provided by this extra work is that the SKL-LR computes an approximation to the singular bases, whereas the GenInc provides arbitrary bases for the dominant singular subspaces. This shortcoming of the GenInc can be addressed and is the subject of the next section.

3.4 Computing the Singular Vectors

The efficiency of the GenInc method arises from the ability to use less expensive computational primitives (triangular matrix multiplies instead of general matrix multiplies) because of special structure imposed on the transformations used to update the bases at each step. The SKL-LR method, on the other hand, always outputs a basis composed of the dominant singular vectors. Whether or not this is necessary varies across applications. If the basis is

to be used for a low-rank factorization of the data,

$$A = U\Sigma V^T \approx U_k \Sigma_k V_k^T = Q_k R_k W_k^T$$

then the choice of basis is irrelevant Also, if the basis is to be used to compute or apply a projector matrix for the dominant subspace, then the choice of basis does not matter, as the projector is unique for a subspace. However, if the application calls for computing the coordinates of some vector(s) with respect to a specific basis, then that specific basis must be used, explicitly or implicitly. It is the case in many image processing applications that the user wishes to have the coordinates of some vector(s) with respect to the dominant singular basis. In this case, the basis output by the GenInc is not satisfactory.

However, as the basis Q_i output by the GenInc at each step represents the same space as does the basis U_i output by the SKL-LR algorithm, then it follows that there is a $k_i \times k_i$ rotation matrix relating the two:

$$U_i = Q_i X.$$

This transformation X is given by the SVD of $R_i = U_r \Sigma_r V_r^T$, as

$$U_i \Sigma_i V_i^T = Q_i R_i W_i^T = (Q_i U_r)\Sigma_r (V_r W_i)^T.$$

In the case of the Dominant Space GenInc, the matrix X is readily available. Recall from Equation (3.2) that the SVD of R_i (the $k_i \times k_i$ principal submatrix of $\hat{R}$) is given by

$$R_i = T_u \Sigma T_v^T$$

where T_u and T_v^T are the matrices used to notch the dominant singular subspaces of $\hat{R}$ at step i. In addition to producing the matrices Q_i, R_i, and V_i at each step, GenInc can also output the transformations T_u and T_v, which rotate Q_i and W_i to the dominant singular bases U_i and V_i.

Computing the coordinates of some vector b with respect to U_i consists of computing the coordinates with respect to Q_i and using T_u to rotate to the coordinate space defined by U_i, as follows:

$$U_i^T b = (Q_i T_u)^T b = T_u^T (Q_i^T b).$$

The production of $Q_i^T b$ requires the same number of operations as that of $U_i^T b$, and the additional rotation by T_u^T occurs in k_i-space, and is negligible when $k_i \ll m$. As this

rotation matrix is small ($k_i \times k_i$), the storage space is also negligible compared to the space required for the $m \times k_i$ matrix Q_i (and even compared to the $s_i \times k_i$ matrix W_i).

Unlike the Dominant Space construction, when the GenInc is performed using the Dominated Space construction of G_u and G_v, the matrices T_u and T_v are not computed. The matrix S_u is used to notch the dominated subspace basis U_2, and G_u is constructed to drive $U_2 S_u^T$ to $\begin{bmatrix} 0 & I_{d_i} \end{bmatrix}^T$. This is achieved using Householder reflectors, as illustrated in Equation (3.6). It is nevertheless possible to compute T_u during this step. Recall that G_u is defined to block-diagonalize U, such that

$$G_u^T U = G_u^T \begin{bmatrix} U_1 & U_2 \end{bmatrix} = \begin{bmatrix} T_u & 0 \\ 0 & S_u \end{bmatrix}.$$

By applying G_u^T to U_1, during or after constructing it from $U_2 S_u^T$, T_u can be constructed. If T_v is also needed, it can be constructed in a similar manner, by applying G_v^T to V_1.

Therefore, the dominant singular bases can be made implicitly available using the GenInc, at a lower cost than explicitly producing them with the SKL-LR. It should be noted that when the dominant singular subspaces are explicitly required on every step (e.g., to visualize the basis vectors), they can still be formed, by rotating the bases Q_i and W_i by T_u and T_v, respectively. The cost for this is high, and in such a case it is recommended to use a method (such as the SKL-LR) that explicitly produces the singular basis.

In cases where the singular bases are required only occasionally, a hybrid method is the most efficient. Using such a method, the SKL-LR separation step is used whenever the explicit singular bases are required, and the less expensive GenInc is otherwise employed. This results in a savings of computation on the off-steps, at the expense of storing a $k_i \times k_i$ block matrix instead of just the k_i singular values composing Σ_i at step i.

3.5 Summary

This chapter describes a generic incremental algorithm which unifies previous methods for incrementally computing the dominant singular subspaces of a matrix. The current methods–the EUA, SKL-LR, and IQRW algorithms–are shown to fit into this framework. Each of these methods implements the generic algorithm in order to achieve specific results. A new method (GenInc) is proposed which achieves a lower operation count by relaxing the criteria on the computed factorization.

Table 3.2 shows the computational complexity of the different implementations of the generic algorithm, as they relate to the block size. The EUA (using the Gu-Eisenstat SVD) only exists for a scalar update and has a higher operation count than other algorithms. The IQRW has a low operation count for a scalar update, but the operation count is not as competitive for non-trivial block sizes.

Table 3.2: Complexity of implementations for different block size scenarios, for two-sided algorithms.

Scenario	EUA	IQRW	SKL-LR	GenInc
l fixed	-	$10mnk + 7mnl$	$\frac{2mnk^2}{l} + 6mnk + 4mnl$	$8mnk + 4mnl$ (w) $\frac{mnk^2}{l} + 6mnk + 4mnl$ (d)
$l = 1$	$(\log_2^2 \epsilon)mnk$	$10mnk$	$2mnk^2 + 6mnk$	$8mnk$ (w)
l optimal	-	-	$12mnk$ $(l = \frac{k}{\sqrt{2}})$	$10mnk$ $(l = \frac{k}{2})$ (d)

The operation count of the SKL-LR method relies on the ability to choose as optimal the block size. Many things prevent this from being realized. The greatest obstacle to this is that there may not be storage available to accumulate the optimal number of columns for an update. This is further complicated by the higher working space requirements of the SKL-LR (Table 3.1).

These issues are resolved by the introduction of a new algorithm, the GenInc. By considering a generic formulation of the incremental subspace tracking algorithm and relaxing some of the constraints imposed by the previous method–triangular storage and the explicit production of singular vectors–a new method is proposed that has a lower a complexity than other methods. Two different constructions of the GenInc (the dominant subspace and dominated subspace constructions) allow the algorithm to maintain a relatively low operation count, regardless of block size. Proposals are made to extend the GenInc, to implicitly produce the current dominant singular vectors, in the case that the application demands them, and a UDV-QBW hybrid algorithm is proposed for cases where an explicit representation of the singular vectors is required.

The discussion in this chapter centers around the operation count of the algorithms. The next chapter evaluates the relationship between operation count and run-time performance of these algorithms, focusing on the performance of individual primitives under different algorithmic parameters.

CHAPTER 4

ALGORITHM PERFORMANCE COMPARISON

Chapter 3 discussed the current low-rank incremental methods in the context of the generic framework, and presented a novel algorithm based on this framework. The predicted performance of these methods was discussed in terms of their operation counts. This chapter explores the circumstances under which performance can be predicted by operation count.

Section 4.1 proposes the questions targeted by the experiments in this chapter. Section 4.2 describes the testing platform. More specifically, the test data is be discussed, along with the computational libraries that are employed. Lastly, Section 4.3 presents the results of the performance experiments and analyze them.

4.1 Primitive Analysis

The lower complexity of the GenInc as compared to the SKL-LR is a result of exploiting the structure of the problem, allowing the use of primitives with lower operation complexity. More specifically, triangular matrix multiplies (`TRMM`) as opposed to general matrix multiplies (`GEMM`). The complexity, in terms of the operation count for the primitives employed by all of the algorithms, is given in Table 4.1.

Note that the only differences between the SKL-LR and the GenInc are

- smaller `GEMM`s for the GenInc,
- two triangular matrix multiplies in the GenInc, not present in the SKL-LR, and
- two very small `GEQRF`s in the GenInc, not present in the SKL-LR.

The small `GEQRF` primitives being negligible, the performance difference between the GenInc and the SKL-LR rests on the performance difference between the `TRMM` primitive

Table 4.1: Cost of algorithms in terms of primitive. $(\cdot)^*$ represents a decrease in complexity of the GenInc over the SKL-LR, while $(\cdot)^\dagger$ represents an increase in complexity of the GenInc over the SKL-LR.

Primitive	SKL-LR	GenInc (d.s.)	GenInc (w.s.)
General Matrix Multiply (`GEMM`)	$k \times m \times l$ $m \times k \times l$ $m \times (k+l) \times k^\dagger$	$k \times m \times l$ $m \times k \times l$ $m \times l \times k^*$	$k \times m \times l$ $m \times k \times l$ $m \times l \times k^*$ $m \times l \times (k-l)^*$
Triangular Matrix Multiply (`TRMM`)		$m \times k \times k^\dagger$	$m \times l \times l^\dagger$ $m \times l \times l^\dagger$
SVD (`GESVD`)	$(k+l) \times (k+l)$	$(k+l) \times (k+l)$	$(k+l) \times (k+l)$
QR decomposition (`GEQRF`)	$m \times l$	$m \times l$	$m \times l$

and the `GEMM` primitive. A triangular matrix multiply of order $m \times k \times k$ requires only half of the floating point operations as does a general matrix multiply of the same size. Furthermore, it has the added benefit of being computable in-situ, which cuts in half the memory footprint and may lead to improved cache performance. For the general matrix multiply, the multiplication of matrices A and B requires storage space for the output matrix in addition to the input matrices.

However, if the `TRMM` is not faster than the `GEMM` primitive, then the GenInc algorithm should be slower than the SKL-LR. For a naive implementation of both primitives, the `TRMM` should be faster. However, it is sometimes the case when using "optimized" implementations of these primitives, that some primitives are more optimized than others. More specifically, commercial implementations of numerical libraries may put more effort into optimizing the commonly used primitives (such as `GEMM` and `GESVD`) and less effort on the less-used primitives (like `TRMM`). The result is that the `TRMM` primitive may actually be slower than the `GEMM` primitive in some optimized libraries.

This effect must be always be considered when implementing high performance numerical algorithms. If the primitives required by the algorithms do not perform well on a given architecture with a given set of libraries, then it naturally follows that the algorithm does not perform well (using the given implementation of the prescribed primitives). This is a well-known and oft-explored fact in the field of high-performance scientific computing. In

such a case, the developer has the choice of either using different primitives to implement the algorithm or of rewriting the primitives to perform better on the target architecture.

The GenInc algorithm requires that a call to the `TRMM` primitive of order $m \times k \times k$ and a `GEMM` primitive of order $m \times l \times k$ perform better than a single `GEMM` primitive of order $m \times (k + l) \times k$. In such cases where the extra time is large enough to cover the overhead required by the GenInc algorithm (namely, the construction of G_u and G_v), then the GenInc algorithm outperforms the SKL-LR algorithm. However, if the `TRMM` primitive is not significantly faster than the `GEMM` primitive (for the same inputs), then the GenInc performs worse, despite its lower operation count.

It is possible to compare the running-time performance of the SKL-LR algorithm against that of the GenInc algorithm. The performance of the stand-alone primitives (`GEMM` and `TRMM`) constitute the major computational difference between the two and is considered in the discussion of performance between the two algorithms.

4.2 Methodology

Implementations were created, and timings were collected for three of the algorithms: the SKL-LR, IQRW, and GenInc. Each was compared against the same baseline. This baseline is the the low-dimensional R-SVD, computed by

- first performing a rank-n QR factorization,
- followed by an $n \times n$ SVD of the triangular factor, and
- finally a matrix multiplication to produce the k left singular vectors from the Q factor and the left singular vectors of the triangular factor.

The tests were conducted using a Sun Ultra 80, with 1024MB of system memory and dual UltraSparc-II CPUs. Each CPU has a clock speed of 450MHz and a 4MB L2 cache. The codes were written in Fortran 95, and compiled to run on a single processor. Further information regarding compiler version and options can be found in Appendix C.

4.2.1 Test Data

The data used for testing comes from two sources. The actual content of the matrices used to test is irrelevant; only the dimensions m, n, and k are important. However, these data are

representative of some of the application scenarios where sub-space tracking/model-reduction methods such as the Incremental SVD would be used.

The first dataset is from the Columbia Object Image Library (COIL-20) database. This database, commonly used for benchmarking performance in image processing applications, is a collection of images of different objects. The collection contains 72 images each of 20 different objects, for a total of 1440 images. The images for a target object are from varying angles. Each image is a 128×128 grayscale, and is represented as a single column vector of the image matrix A. The performance tests run here considered only the first 10 objects, so that A has dimensions $M = 16384$ and $N = 720$. This dataset is representative of many in image processing applications, with a fairly large value for M (64^2, 128^2, etc.) and a smaller (but significant) number of columns.

The second dataset used for performance testing is from a Computational Fluid Dynamics dataset. Each column of the matrix represents a snapshot of a 3-dimensional flow field, of dimension $64 \times 64 \times 65$. The performance tests are run with two versions of the dataset. In one, only one flow direction of the data set is included, so that the length of each column of A is $m = 64 \times 64 \times 65 = 266240$. In the other, all three directions of flow are considered, so that $m = 64 \times 64 \times 65 \times 3 = 798720$. In both cases, $n = 100$ snapshots of the flow field are considered. This dataset is representative of many CFD datasets, with fewer columns than in image databases like the COIL20, but with columns varying from large ($\approx 260,000$) to very large ($\approx 800,000$).

4.2.2 Test Libraries

The different algorithms are tested with a variety of numerical libraries:

- Netlib libraries
- SUNPERF library
- ATLAS library

The Netlib libraries are the BLAS and LAPACK sources from the Netlib repository and compiled with the Sun compiler. The SUNPERF is a hand-tuned, "optimized" version of the BLAS and LAPACK routines for use on Sun architectures. The Automatically Tuned Linear Algebra Software (ATLAS) is a profile-based, self-tuning implementation of often-used

routines from the BLAS and LAPACK numerical libraries. ATLAS is primarily intended for use on machines where no hand-tuned, architecture-specific numerical library is present.

Many architecture vendors provide implementations of the BLAS and LAPACK libraries specially tuned to the vendor's platform. Examples include

- AMD's "AMD Core Math Library" (ACML),
- Intel's "Intel Math Kernel Library" (IMKL),
- Sun's "Sun Performance Library" (SUNPERF),
- SGI's "Scientific Computing Software Library" (SCSL), and
- IBM's "Engineering and Scientific Subroutine Library" (ESSL).

For users whose platform does not provide tuned libraries, the remaining options are to download and compile the BLAS and LAPACK libraries from Netlib, or to use ATLAS or some other self-tuning software package.

4.3 Results and Analysis

The results of the performance testing are shown in Figures 4.1, 4.2, 4.3, and 4.4. The first three sets of figures show the performance of the two basic primitives (`GEMM` and `TRMM`) along with the performance of all four algorithms: the IQRW, the GenInc, the SKL-LR, and the R-SVD. This is detailed for each of the three libraries: ATLAS, SUNPERF, and Netlib. The fourth set of figures show the ratio between the performance the `TRMM` and `GEMM` primitives, along with the ratio of the performance between the GenInc and SKL-LR algorithms.

The first result to note is the improvement in performance of both of the block algorithms over that of the IQRW. With a complexity of $8mnk$ and $10mnk$ (depending on which method is used) as compared to $12mnk$ (SKL-LR) and $10mnk$ (GenInc), the IQRW methods proposed by Chahlaoui et al. have the lowest operation count of any method. However, the block methods, exploiting temporal locality and a memory hierarchies, overcome their increased operation count and yield better running times. Observing Figures 4.1 and 4.2, this is the case. For the ATLAS and SUNPERF libraries, the level-3 primitives make more efficient use of the memory hierarchy, allowing the block algorithms to outperform the lower complexity scalar algorithm.

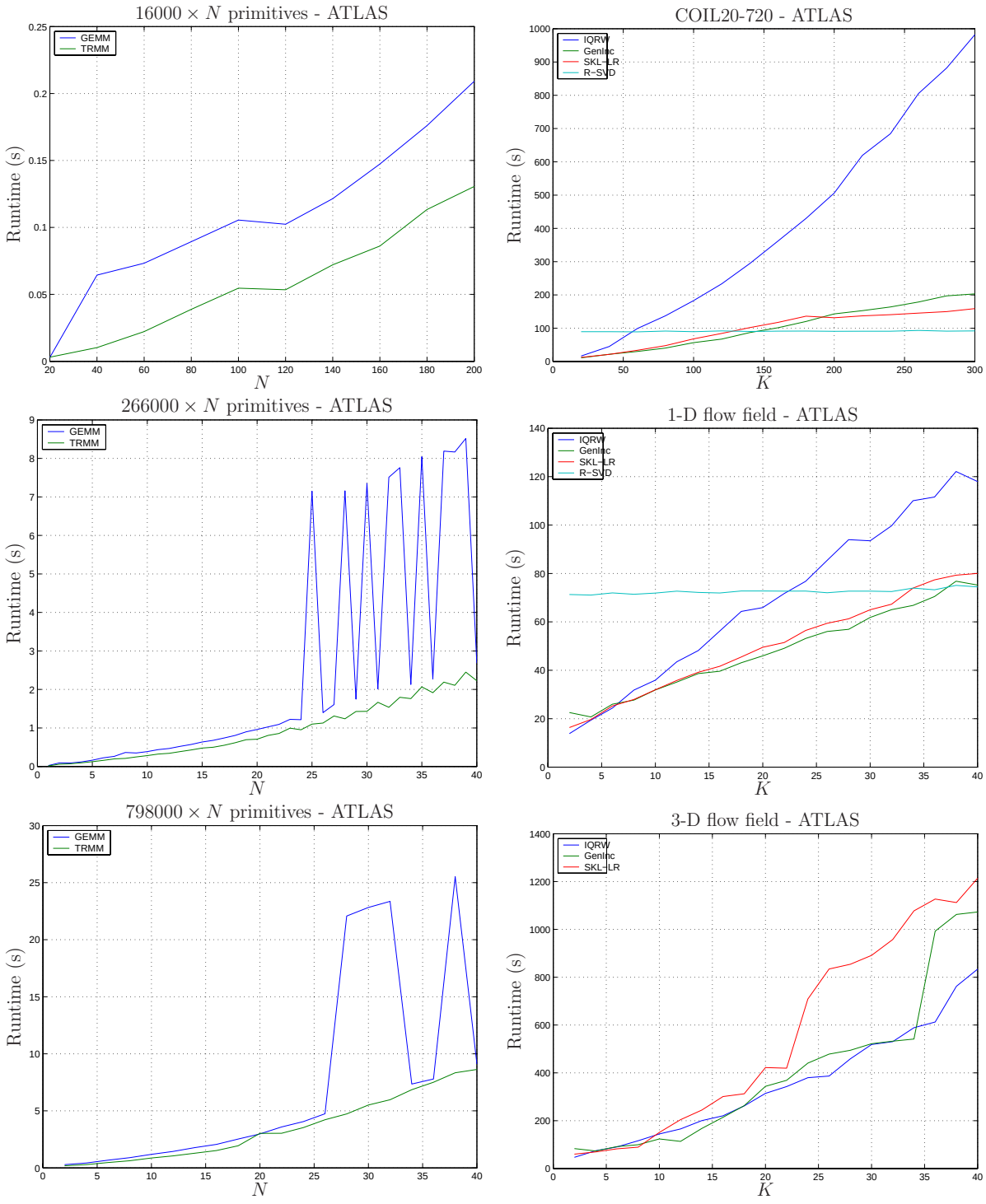

Figure 4.1: Performance of primitives and algorithms with ATLAS library.

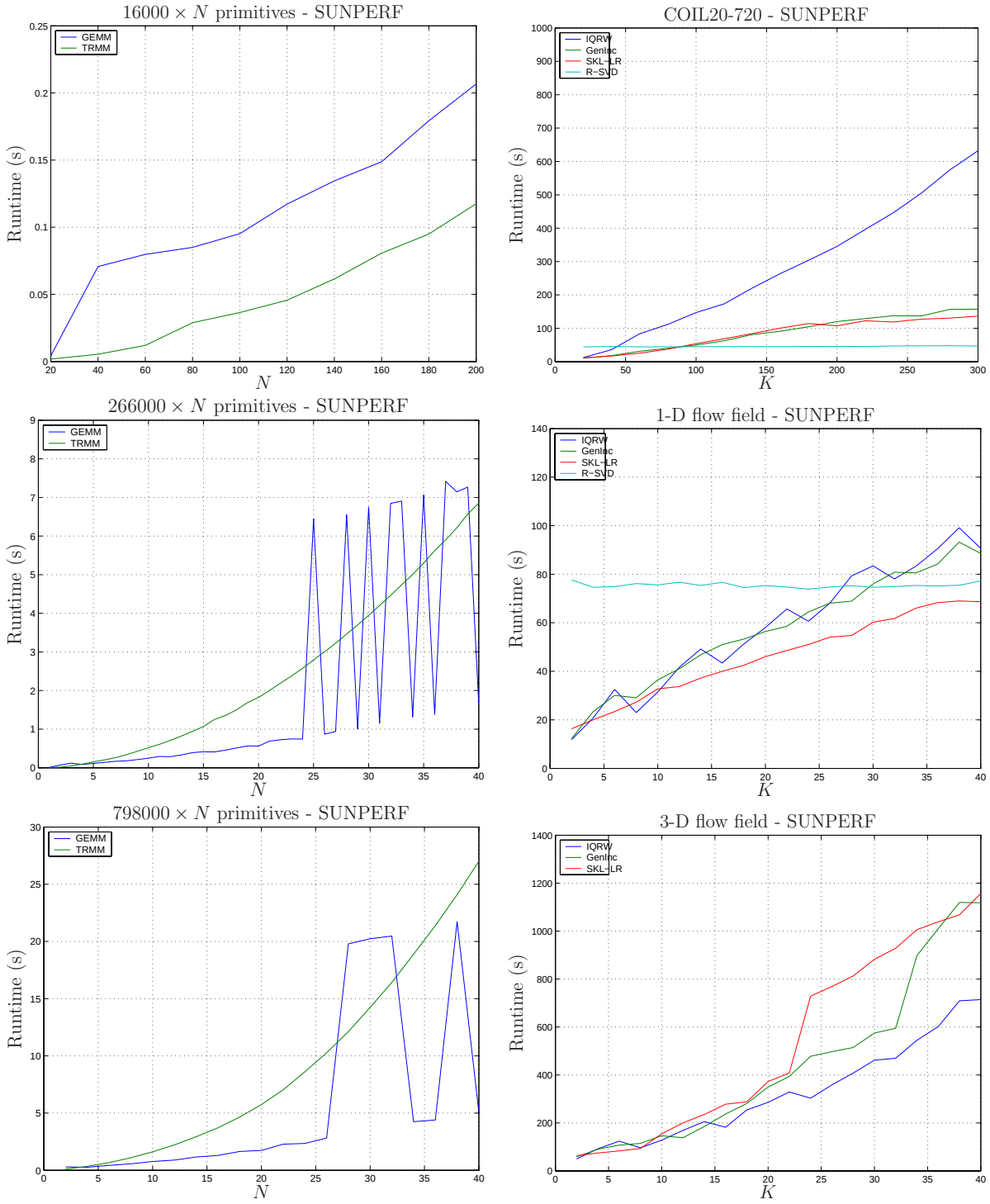

Figure 4.2: Performance of primitives and algorithms with SUNPERF library.

Note Figure 4.3, the results using the Netlib library. Not only is overall performance slower than with the two tuned libraries, but the relative performance of the three incremental algorithms is different. Because the Netlib libraries do not employ blocking to improve performance of the level-3 primitives, the IQRW algorithm is able to outperform the two block algorithms.

Next, consider the performance arguments made regarding the two block algorithms. It was argued previously that, given a `TRMM` primitive that is more efficient than the `GEMM` primitive of the same order, the GenInc should outperform the SKL-LR algorithm.

For the ATLAS library, the performance testing on the primitives show that the `TRMM` primitive runs consistently faster than the `GEMM` primitive (Figure 4.1). The `GEMM` primitives show some spiking, both for the ATLAS and SUNPERF libraries, but this is typical of blocked routines and is predictably absent from the Netlib library.

Considering the algorithm timings using the ATLAS library, the results are straightforward. In the COIL-20, 1-D CFD, and 3-D CFD scenarios, the `TRMM` primitives performs on average 50% better than the `GEMM` primitive, and it never performs worse. For both the 1-D and 3-D CFD datasets, the GenInc algorithm has a runtime lower than that of the SKL-LR algorithm, and in the case of the 3-D CFD dataset, the runtime is much better than would be predicted by operation counts. Some of this benefit may be because the `TRMM` primitive is performed in place, as opposed to the `GEMM` primitive, which requires work space. This can lead to a smaller memory footprint and less cache misses, which is more significant in the 3-D CFD scenario, where the data is so much larger than in the 1-D case.

For the COIL image database, however, the GenInc, while out-performing the SKL-LR for lower values of K, does become more expensive. This trend is not predicted by the primitive timings for the ATLAS library, where the `TRMM` stills run in half the of the `GEMM`. An explanation for this comes by way of the reduced size of m, relative to n and k. This reduces the importance of the `GEMM` and `TRMM` operations, giving more weight to other work in the GenInc: the extra QR factorizations used to construct G_u and G_v, in addition to the smaller step size.

Consider now the performance of the incremental algorithms linked against the SUNPERF library (Figure 4.2). First of all, note the lack of improvement in the `TRMM` primitive over the `GEMM` primitive. An operation that should require only half of the runtime, instead runs as much as 3 and 4 times slower. This translates, for the 1-D CFD scenario, into an

increase in runtime for the GenInc, as compared to the SKL-LR. In the case of the 3-D CFD dataset, even though the `TRMM` does not perform better than the `GEMM`, the GenInc still outperforms the SKL-LR. This is likely a result of the in-situ execution of the `TRMM` as compared to the ex-situ execution of the `GEMM`. As with the ATLAS libraries, this gives the GenInc an extra benefit over the SKL-LR for this large dataset, and in this case, is able to make up for a relatively inefficient triangular matrix multiply.

Note the performance of both algorithms using the SUNPERF library for the COIL dataset. Even though the `TRMM` primitive out-performs the `GEMM`, the GenInc under-performs compared to the SKL-LR. This can be explained, as with the ATLAS implementation, by the lessened effect of the matrix multiplication primitives, in light of a much smaller m.

4.4 Summary

This chapter illustrates the benefit of both of the block algorithms over a scalar algorithm and the computational benefit of the GenInc over the SKL-LR algorithm.

The better performance of block update algorithms, in spite of higher operation counts, over scalar update algorithms comes by utilizing BLAS-3 primitives which exploit a memory hierarchy. This is clearly illustrated in the experiments presented in Section 4.1. Codes linked with unoptimized libraries yield runtimes consistent with operation count, resulting in scalar update algorithms running faster than block update algorithms. However, codes linked with libraries optimized for a memory hierarchy yield superior performance for block update algorithms as compared to scalar update algorithms.

Furthermore, this chapter investigates the relative performance of the two block update algorithms, SKL-LR and GenInc. The increased efficiency of the GenInc is shown to be tied to the performance of specific primitives. Experiments show that, in the presence of adequate primitives, the GenInc yields better runtime performance than the SKL-LR.

The next chapter contains a discussion of the source of error in the incremental algorithm, and it discusses the attempts of previous authors to bound this error. These results are updated for a block algorithm. Improvements to the bounds are proposed and experiments are run to validate the discussion.

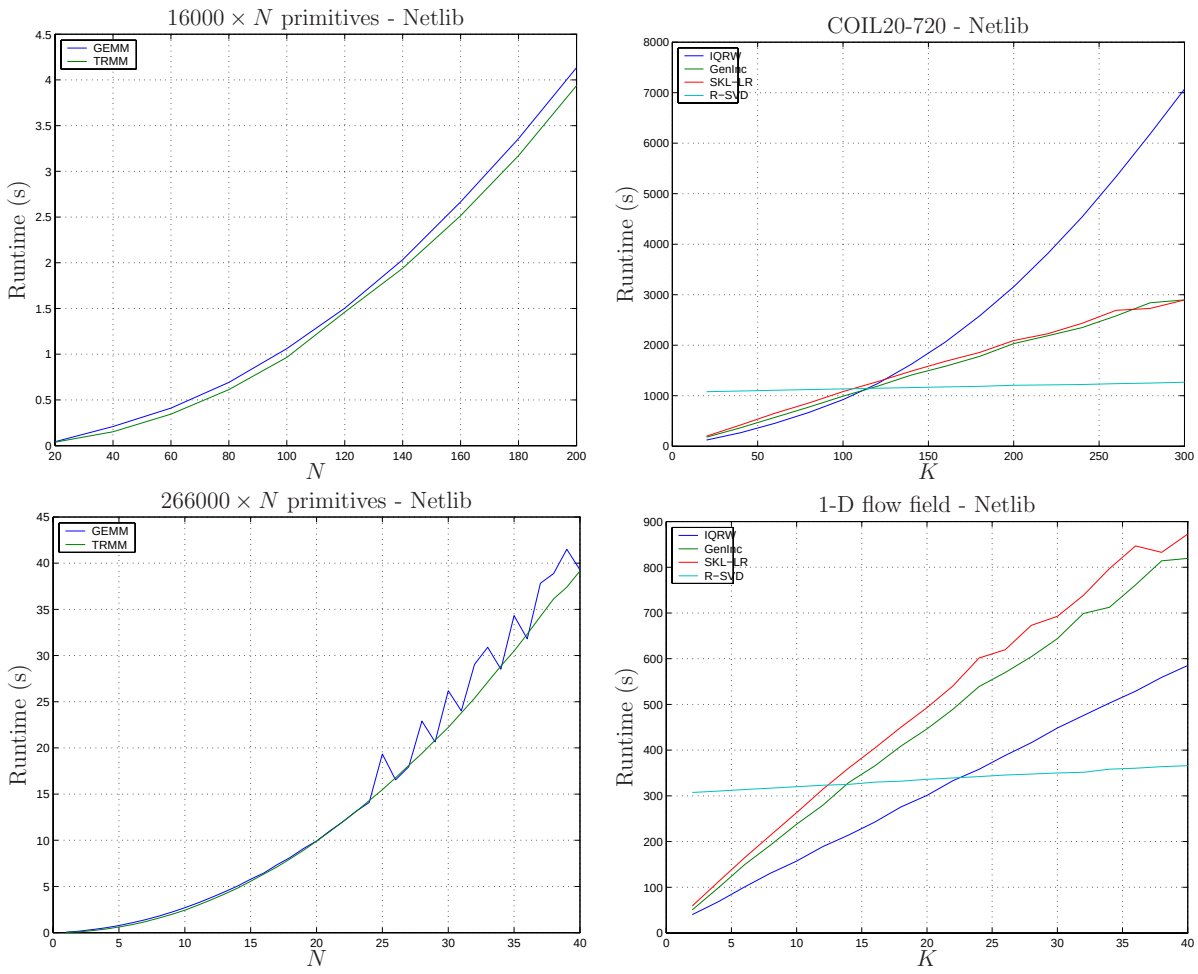

Figure 4.3: Performance of primitives and algorithms with Netlib library.

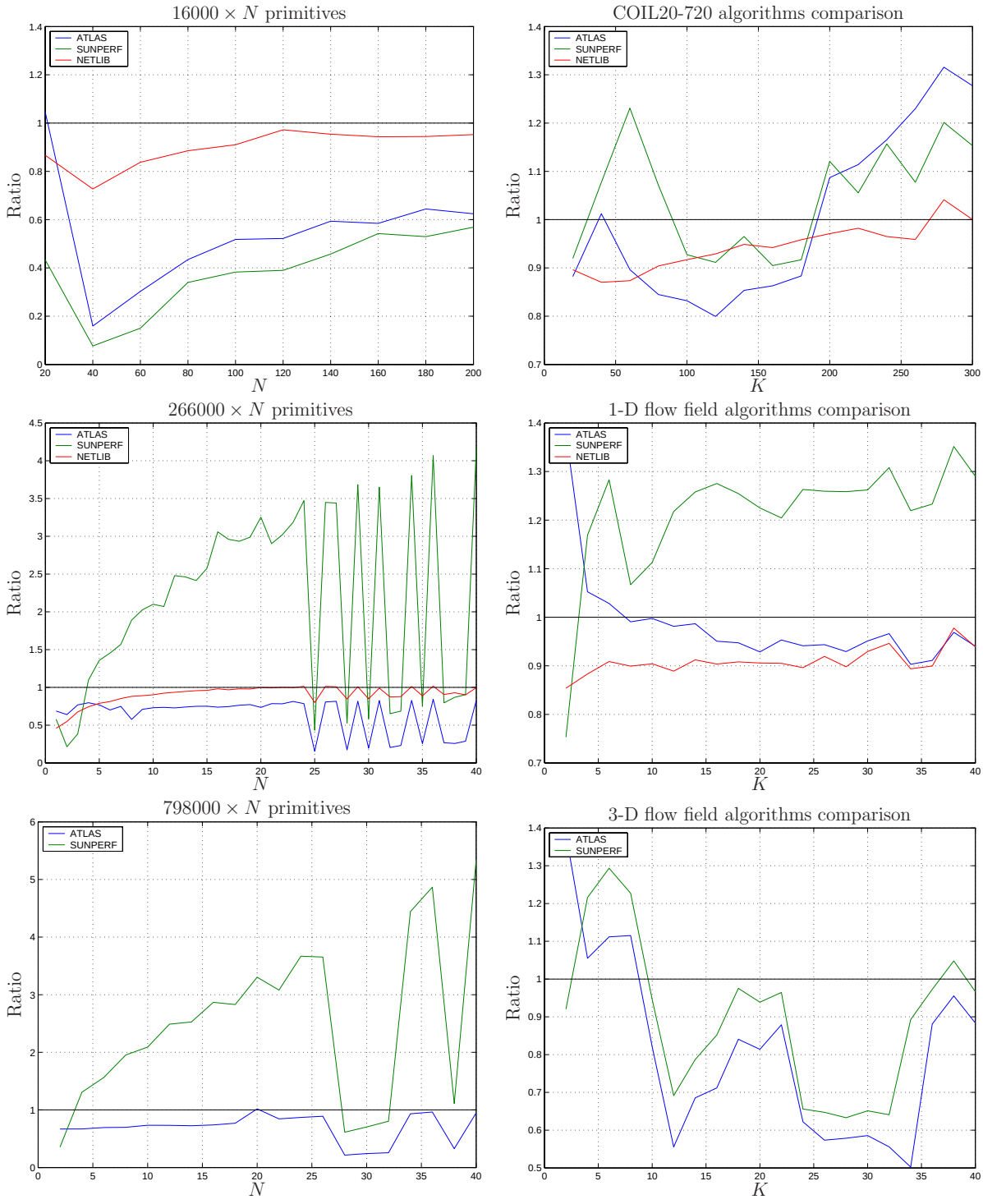

Figure 4.4: Runtime ratios of primitives (`TRMM`/`GEMM`) and algorithms (GenInc/SKL-LR) for different libraries.

CHAPTER 5

BOUNDS ON ACCURACY

The computational advantages of the low-rank, incremental algorithms over standard methods (e.g., R-SVD) do not come without a cost. At each step, the algorithms choose the optimal low-rank subspace for approximating the local matrix. However, it is a heuristic argument that these local approximations are globally optimal. Because data truncated from the factorization is lost forever, the incremental algorithms produce only approximations to the dominant SVD of the target matrix.

This chapter discusses the efforts of the previous authors to bound the errors in the incremental algorithm and evaluates the reliability of the error estimates they recommend in the context of the generic incremental algorithm framework. Two parameters whose effects on accuracy are examined are the "threshold of acceptance", δ_i, and the block size, l_i. The threshold δ_i is used to determine the value for k_{i+1} at each step i of the algorithm. That is, after updating the decomposition at step i with l_i new columns of A, δ_i is used to determine the rank of the retained factorization. The block size l_i has a profound effect both on the quality of the error estimates and the computational effort required to produce the decomposition.

The error analyses in this section depend on a theorem introduced in [12] and proven in Appendix A. This theorem states that after every step of the incremental algorithm, there exists an orthogonal matrix V_i (embedding W_i) relating the processed columns of A to the current decomposition and the truncated data, like so:

$$A_{(1:s_i)}V_i = A_{(1:s_i)}\begin{bmatrix} W_i & W_i^{\perp} \end{bmatrix} = \begin{bmatrix} Q_iR_i & \tilde{Q}_1\tilde{R}_1 & \dots & \tilde{Q}_i\tilde{R}_i \end{bmatrix}.$$

This decomposition holds for each step of the algorithm, including the last one. Therefore,

there is some orthogonal matrix V_f embedding W_f which transforms the matrix A as follows:

$$A \begin{bmatrix} W_f & W_f^\perp \end{bmatrix} = \begin{bmatrix} Q_f R_f & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix},$$

so that

$$\begin{aligned} A &= \begin{bmatrix} Q_f R_f & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix} \begin{bmatrix} W_f & W_f^\perp \end{bmatrix}^T \\ &= Q_f R_f W_f^T + \begin{bmatrix} \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix} W_f^{\perp T}. \end{aligned}$$

This splits A into two parts: the decomposition produced by the incremental algorithm and the information truncated at each step from the algorithm. This enables analyses of the accuracy of the algorithm.

Section 5.1 discusses the error in the low-rank factorization produced by a two-sided incremental method, as compared to the matrix A. The source of this error is described, as well as the efforts of previous authors to construct a priori bounds on the error. The effect of block size on the performance of the algorithm is also studied. Section 5.2 discusses the a posteriori bounds derived by previous authors on the error in the computed subspaces and singular values, as well as approximations to these bounds. Previous bounds and their estimates are reviewed in the context of the generic incremental algorithm. Finally, Section 5.3 suggests techniques for improving the quality of the bounds and evaluates their effectiveness.

5.1 A Priori Bounds

In [8], Manjunath et al. concern themselves with two measures of accuracy. With origins in image processing, each column of A represents an image. The dominant SVD then represents the ideal rank-k representation of the set of images constituting A (under the 2-norm). They are therefore concerned with the ability of this low-rank representation to approximate the individual images. The authors attempt to bound the individual errors in terms of the norm of the data truncated at each step. This is possible because the data truncated at each step is bounded in norm by the threshold of acceptance, δ_i. This is accomplished by choosing k_{i+1} (the rank for the next step) so that $\hat{\sigma}_{k_{i+1}+1}, \dots, \hat{\sigma}_{k_{i+1}+d_i}$ (the singular values of $\hat{R}_i$) are all smaller than δ_i. For the rest of this discussion, it is assumed that δ_i is held constant from step to step (although this is not necessary in practice.) Furthermore, subscripts on k_i and l_i may at times be dropped for the sake of simplicity.

Manjunath et al. propose an argument for the approximating ability of Q_i (clarifying additions by Baker in brackets):

> Note that Q_{i+1} approximates a_{i+1} [and $Q_i R_i$] to an accuracy of δ. Similarly Q_i approximates a_i to an accuracy of δ. From this it follows that Q_{i+1} approximates a_i to an accuracy of 2δ. In general, $j \leq i$, Q_i approximates a_j to an accuracy of $(i - j + 1)\delta$. Therefore if we choose δ to be ϵ/n we can guarantee that Q_n approximate all the images to an accuracy of ϵ.

This bound on the representation error of each a_i, while correct, is unnecessarily loose. Recall the decomposition for $A_{(1:i)}$ given above, for a scalar algorithm ($l = 1$):

$$A_{(1:i)} = Q_i R_i W_i^T + \left[\hat{\sigma}_{k+1}^{(1)}\tilde{q}_1 \quad \dots \quad \hat{\sigma}_{k+1}^{(i)}\tilde{q}_i\right] W_i^{\perp T},$$

where each $\hat{\sigma}_{k+1}^{(i)} \leq \delta$. Then, for $j \leq i$, the representation error for $a_j = Ae_j$ in $Q_i R_i W_i^T$ is

$$\begin{aligned}
\|a_j - Q_i R_i W_i^T e_j\|_2 &= \|A_{(1:i)}e_j - Q_i R_i W_i^T e_j\|_2 \\
&= \left\| \left[\hat{\sigma}_{k+1}^{(1)}\tilde{q}_1 \quad \dots \quad \hat{\sigma}_{k+1}^{(i)}\tilde{q}_i\right] W_i^{\perp T} e_j \right\|_2 \\
&= \left\| \left[\hat{\sigma}_{k+1}^{(j)}\tilde{q}_j \quad \dots \quad \hat{\sigma}_{k+1}^{(i)}\tilde{q}_i\right] \right\|_2 \qquad (5.1) \\
&\leq \left\| \left[\tilde{q}_j \quad \dots \quad \tilde{q}_i\right] \right\|_2 \max_{g=j}^{i} \hat{\sigma}_{k+1}^{(g)} \\
&\leq \sqrt{i - j + 1}\delta,
\end{aligned}$$

where Equation (5.1) is a result of the trapezoidal structure of $W_i^{\perp}$ and its orthonormal columns (see Appendix A). Therefore, each column of A is approximated to $\sqrt{n}\delta$ by $Q_n R_n W_n^T$, if δ is the threshold for rank-determination used at each step. To guarantee that all images are approximated to some accuracy ϵ, δ should be set to $\epsilon/\sqrt{n}$.

The second measure of accuracy that Manjunath et al. propose appears in [7]. There, the authors are concerned with the overall representation error between A and $Q_n R_n W_n^T$. It is proposed in [7] to bound the representation error as follows:

$$\|A - Q_n R_n W_n^T\|_2 \leq n\delta.$$

The authors suggest that this bound is overly conservative and that it should be approximated by $\sqrt{n}\delta$. However, using a similar analysis as above, it is easily shown that $\sqrt{n}\delta$ is a

bound on the error:

$$\begin{aligned}
\|A - Q_n R_n W_n^T\|_2 &= \left\| \left[\hat{\sigma}_{k+1}^{(1)} \tilde{q}_1 \quad \dots \quad \hat{\sigma}_{k+1}^{(n)} \tilde{q}_n \right] W_n^{\perp T} \right\|_2 \\
&= \left\| \left[\hat{\sigma}_{k+1}^{(1)} \tilde{q}_1 \quad \dots \quad \hat{\sigma}_{k+1}^{(n)} \tilde{q}_n \right] \right\|_2 \\
&\leq \sqrt{n}\delta.
\end{aligned}$$

While tighter, this bound is still conservative, and better bounds and approximations are given later.

Note that both of these bounds improve with a growing block size l. Considering a fixed block size of l requiring f iterations to pass through all columns of A, the bound on the error in the approximation is easily revised as:

$$\begin{aligned}
\|A - Q_f R_f W_f^T\|_2 &= \left\| \left[\tilde{Q}_1 \tilde{R}_1 \quad \dots \quad \tilde{Q}_f \tilde{R}_f \right] W_f^{\perp T} \right\| \\
&\leq \sqrt{f}\delta \\
&\approx \sqrt{\frac{n - k_0}{l}}\delta.
\end{aligned} \tag{5.2}$$

For a fixed threshold parameter δ, increasing the block size clearly lowers this bound. However, this only occurs if the allowable rank of the factorization is large enough so that the discarded singular values are smaller than δ. This may be not be possible due to memory limitations.

Levy et al. [9] also present an argument that the performance of the algorithm improves as the block size increases. They describe each column in A as being composed of some components each of the "true basis vectors" (the dominant k left singular vectors of A) and of the "noise basis vectors" (the dominated k left singular vectors of A. They propose that an increased block size allows the algorithm to gather in a single step, more energy from the "true basis vectors" relative to the "noise basis vectors" (which allegedly do not have support across multiple columns of A). This allows the "true basis vectors" a better chance of justifying their existence in the factorization retained by the low-rank incremental algorithm.

An assumption in this explanation is that each column of A contains some average energy for each singular vector (dominant and dominated). Another is that each "noise basis vector" has support in only a single column of A.

If the "noise basis vectors" do have support from contiguous columns of A, then increasing the block size may have the effect of requiring that the rank of the factorization be increased in order to retain all of the vectors which meet the acceptance threshold. As the rank of the factorization effects the memory and computational requirements, the rank of the factorization may be limited by the amount of memory on a computer or by the amount of time allowed for a single incremental step.

The discussion in [9] suggests that an increased block size allows the incremental algorithms to better capture the dominant space, even if the rank of the algorithm does not grow. This statement requires the assumption that each column of A contains the average energy associated each "true basis vector". While there is some average energy associated with each "true basis vector", the amount of energy in the columns of A associated with each singular vector may not be uniform across columns. Therefore, the block size of the algorithm may have to be greatly increased in order for one update to gather more information about a particular singular vector than would have occurred with a smaller block size.

The following experiment illustrates this point. A matrix A of dimension 1000×100 is generated with elements chosen from a standard normal distribution. The incremental algorithm is applied to this matrix multiple times, with block sizes ranging from 1 to 75, always initialized with a rank $k_0 = 5$ factorization. Two types of tests are run: "constrained rank" tests, where the rank of the algorithm is never allowed to grow above 15 (the resulting rank when run for $l = 1$); and "free rank" tests, where the rank is determined entirely by the acceptance threshold δ. This experiment was run multiple times with similarly generated matrices, and the results were consistent across these runs.

Figure 5.1 shows the results of this experiment. First note that for the rank 15 experiments, the representation error lies between 92% and 94%. However, the ratio $\sigma_{16}/\sigma_1 = 90\%$ is a lower bound on the norm of the representation error for an approximation of rank 15.

Noting Figure 5.1(a), in the free rank tests, an increased block size l is met with a decreased representation error. Figure 5.1(b) shows that as the block size increases, the rank of the factorization must grow, as predicted above. However, when the rank is not allowed to grow (in the constrained rank tests), an increase in block size does not result in a significant decrease in representation error.

This illustrates the argument made above, that the bounds on the error (and the error,

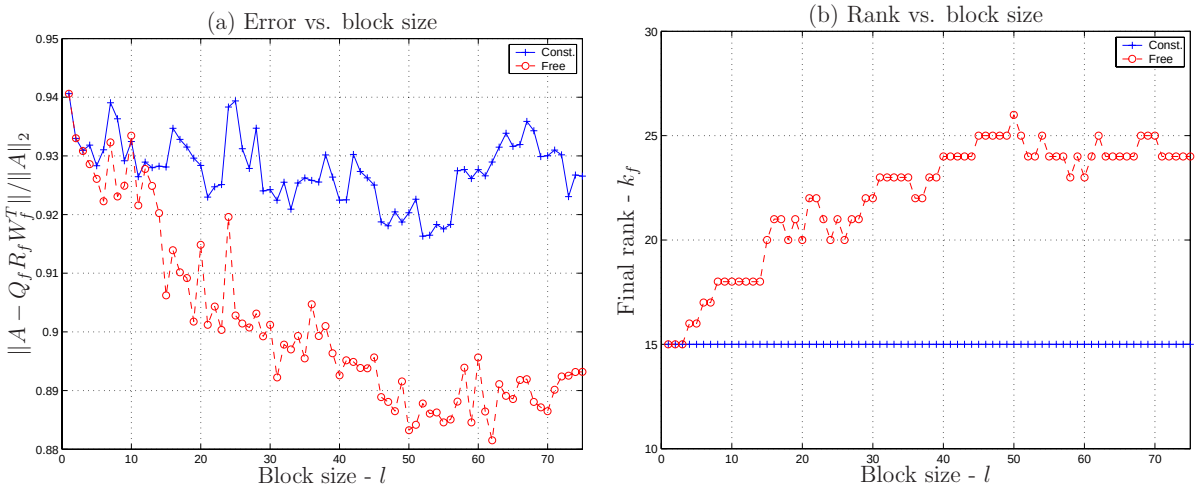

Figure 5.1: (a) The error in representation ($\|A - Q_f R_f W_f^T\|/\|A\|_2$) and (b) final rank of factorization (k_f) when running the incremental algorithm with increasing block sizes l.

in practice) decrease as the block size increases. However, this experiment confirms that for a dynamic rank algorithm based on an absolute threshold of acceptance, the rank of the algorithm tends to increase with an increasing block size. Furthermore, if the factorization rank is not allowed to increase, the performance benefits disappear.

Note as well that in the free rank tests, the error does not reduce monotonically with increasing block size. This is because the optimal block size (in terms of numerical performance) depends on the support that the dominant singular vectors have in the columns of A. Without knowing something about this energy (like the "average energy" assumption in [9]), it is difficult to make rigorous statements about the numerical performance of the incremental algorithm with respect to block size.

This section outlines the attempts at a priori bounds on the error ($\|A - Q_f R_f W_f{}^T\|_2$) by previous authors, while studying the effect of block size on this error measure. The argument of Levy et al. [9], that increased block size leads to an improved factorization, is restated and confirmed with experiments. However, the source of this improvement is shown to be due to the increased rank of the factorization, itself a result of the increased block size. Block size alone was shown not to improve performance for a class of randomly generated matrices.

This section also tightens those a priori bounds presented by Manjunath et al. and

updates them for the block algorithm. The following section will present the attempts by previous authors to construct a posteriori bounds on the performance of the incremental algorithm.

5.2 A Posteriori Bounds

The bound $\sqrt{\frac{n-k_0}{l}}\delta$ proposed in [8] is an a priori bound of the representation error, recommending a value of δ when the tolerable error is known. Because the low-rank incremental algorithms can produce bad results, it is important to have reliable a posteriori bounds on the error in the singular subspaces and singular values. Utilizing a posteriori information about the truncated data allows a tighter bound on this error. In the this section, the attempts by Chahlaoui et al. [12] to provide tight a posteriori bounds are revisited for a block algorithm. Their analysis is described and updated for the block case. Experiments are run to test the prerequisite assumptions of their bounds.

The authors of the IQRW [12] derive bounds on the error in the subspaces produced by the incremental algorithm. These bounds compare the result of the incremental algorithm with that of the true SVD. Recall that the work of Chahlaoui et al. described the incremental algorithm and resulting accuracy analysis only for the scalar case ($l = 1$).

Their analyses rest on the results of two theorems. The first is the decomposition proven in Appendix A and has been used throughout this section. Stated again, it proves the existence of a matrix V_i at each step i of the algorithm satisfying:

$$A_{(1:s_i)}V_i = A_{(1:s_i)}\begin{bmatrix} W_i & W_i^{\perp}\end{bmatrix} = \begin{bmatrix} Q_iR_i & \tilde{Q}_1\tilde{R}_1 & \dots & \tilde{Q}_i\tilde{R}_i\end{bmatrix},$$

where $\tilde{Q}_i$ and $\tilde{R}_i$ are the matrices truncated from the decomposition at step i, and $W_i^{\perp}$ is the orthogonal complement to W_i composed of the truncated $\tilde{W}_i$ and necessary permutations.

Updating their argument shows that each intermediate factorization, produced at step i, is orthogonally equivalent to a submatrix $A_{(1:s_i)}$ of A, so that

$$\hat{\sigma}_1^{(i)} \leq \sigma_1, \quad \dots, \quad \hat{\sigma}_{k_i+l_i}^{(i)} \leq \sigma_{k_i+l_i}$$

where $\hat{\sigma}_j^{(i)}$ are the singular values of $\hat{R}_i$, and σ_j are the singular values of A. Then the singular values that are discarded at each step are all smaller than $\sigma_k^{(i)}$, the k_i^{th} singular value of A.

Furthermore, consider the update at step i:

$$\begin{bmatrix} Q_i R_i & A_+ \end{bmatrix} \begin{bmatrix} W_i^T & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} Q_{i+1} & \tilde{Q}_{i+1} \end{bmatrix} \begin{bmatrix} R_{i+1} & 0 \\ 0 & \tilde{R}_{i+1} \end{bmatrix} G_v^T \begin{bmatrix} W_i^T & 0 \\ 0 & I \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} Q_i R_i & A_+ \end{bmatrix} = \begin{bmatrix} Q_{i+1} & \tilde{Q}_{i+1} \end{bmatrix} \begin{bmatrix} R_{i+1} & 0 \\ 0 & \tilde{R}_{i+1} \end{bmatrix} G_v^T \tag{5.3}$$

The matrix on the right hand side of (5.3) has singular values

$$\hat{\sigma}_1^{(i+1)}, \ldots, \hat{\sigma}_{k_i+l_i}^{(i+1)}.$$

It also has $Q_i R_i$ as a submatrix, so that

$$\hat{\sigma}_1^{(i)} \leq \hat{\sigma}_1^{(i+1)}, \quad \ldots \quad , \quad \hat{\sigma}_{k_i}^{(i)} \leq \hat{\sigma}_{k_i}^{(i+1)}.$$

The singular values of the computed factorization increase monotonically, with every step of the algorithm, toward the singular values of A.

It is shown in [12] that bounds can be obtained for the incremental algorithm by relating the final factorization $\hat{Q}_f \hat{R}_f \hat{W}_f^T$ to the matrix A. This is accomplished using the SVD-like decomposition shown at the end of Appendix A:

$$A = UMV^T = \begin{bmatrix} Q_f \hat{U} & Q_f^\perp \end{bmatrix} \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix} \begin{bmatrix} W_f \hat{V} & W_f^\perp \end{bmatrix}^T,$$

where $A_2 \doteq \begin{bmatrix} A_{1,2} \\ A_{2,2} \end{bmatrix}$ and $\mu \doteq \|A_2\|_2$.

They apply a theorem from [14], which bounds the distance between the dominant singular subspaces of

$$M = \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix}$$

and those of

$$\hat{M} = \begin{bmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{bmatrix}.$$

Using this theorem, the following bounds are given in [12]:

$$\tan\theta_k \leq \frac{\mu^2}{\hat{\sigma}_k^2 - 2\mu^2} \quad \text{if} \quad \mu \leq \frac{\hat{\sigma}_k}{\sqrt{3}} \tag{5.4}$$

$$\tan\phi_k \leq \frac{2\mu\|A\|_2}{\hat{\sigma}_k^2 - \mu^2} \quad \text{if} \quad \mu \leq \frac{7\hat{\sigma}_k^2}{16\|A\|_2}, \tag{5.5}$$

where θ_k and ϕ_k are the largest canonical angles between the left and right dominant singular subspaces of M and $\hat{M}$, respectively. These angles also represent the distance between the dominant subspaces of A and $Q_f R_f W_f^T$, so they measure the error in the subspaces computed by the incremental algorithms.

Furthermore, using these results, [12] derives relative bounds on the errors in the singular values computed by the incremental algorithm:

$$|\sigma_i - \hat{\sigma}_i| \leq \frac{\mu^2}{\sigma_i + \hat{\sigma}_i}.$$

Unfortunately, the matrix A_2 is not computed during the low-rank incremental algorithm, and neither is its norm, μ. To evaluate the bounds, it is necessary to approximate this value.

Consider the matrix A_2. Take the matrix

$$S_i = \begin{bmatrix} 0 \\ I_{d_i} \\ 0 \end{bmatrix},$$

which selects some $\tilde{Q}_i \tilde{R}_i$ as such:

$$\begin{bmatrix} \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix} S_i = \tilde{Q}_i \tilde{R}_i.$$

Then the block of columns in A_2 corresponding to $\tilde{Q}_i \tilde{R}_i$ has the same 2-norm as $\tilde{R}_i$:

$$\begin{aligned} \|\tilde{Q}_i \tilde{R}_i\|_2^2 &= \| \begin{bmatrix} \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix} S_i \|_2^2 \\ &= \|Q_f \hat{U} A_{1,2} S + Q_f^{\perp} A_{2,2} S\|_2^2 \\ &= \|Q_f \hat{U} A_{1,2} S + Q_f^{\perp} A_{2,2} S\|_2^2 \\ &= \|S^T A_{1,2}^T A_{1,2} S + S^T A_{2,2}^T A_{2,2} S\|_2 \\ &= \|S^T A_2^T A_2 S\|_2 \\ &= \|A_2 S\|_2^2. \end{aligned}$$

Denoting μ_i as the largest singular value discarded at step i, the 2-norm of A_2 can be bounded above as follows:

$$\|A_2\|_2^2 \leq \bar{\mu}^2 = \sum_{i=1}^{f} \mu_i^2.$$

The authors worry that this value may overestimate μ to the point that $\hat{\sigma}_k^2 - \bar{\mu}^2$ and $\hat{\sigma}_k^2 - 2\bar{\mu}^2$ become negative. By assuming the truncated vectors to be noise (i.e., uncorrelated

and having standard Gaussian distribution), they are able to use a result by Geman [15] to estimate μ. Defining $\hat{\mu} \doteq \max_i \mu_i$, each column of A_2 then has norm bounded by $\hat{\mu}$. Geman shows then that the expected value of $\mu = \|A_2\|_2$ has the relation:

$$\hat{\mu} \leq \mu \leq c\hat{\mu}, \quad c \approx 1 + \sqrt{\frac{n - k_0}{n}} \approx 2.$$

Chahlaoui et al. then recommend approximating μ by $\hat{\mu}$. They point out that this approximation has the advantage that, since for each discarded μ_i it is known that $\mu_i \leq \hat{\sigma}_k \leq \hat{\sigma}_1$, and therefore $\hat{\sigma}_k^2 - \hat{\mu}^2$ is always non-negative. This allows the proposed bounds to be approximated as follows:

$$\begin{aligned}
\tan\theta_k &\approx \tan\hat{\theta}_k \doteq \frac{\hat{\mu}^2}{\hat{\sigma}_k^2 - \hat{\mu}^2}, \\
\tan\phi_k &\approx \tan\hat{\phi}_k \doteq \frac{2\hat{\mu}\hat{\sigma}_1}{\hat{\sigma}_k^2 - \hat{\mu}^2}, \\
|\sigma_i - \hat{\sigma}_i| &\leq \frac{\mu^2}{\sigma_i + \hat{\sigma}_i} \approx \frac{\hat{\mu}^2}{2\hat{\sigma}_i}.
\end{aligned}$$

In [12], Chahlaoui et al. present the results of tests on matrices with randomly generated elements. These matrices are constructed to have gaps (between σ_k and σ_{k+1}) of varying size. Their tests show that, even when the necessary assumptions are not met (Equations (5.4,5.5)), the proven bounds often hold. Furthermore, they illustrate that the quantity $\hat{\mu}$ is an appropriate approximation for μ, so that the bound approximations do in practice bound the error in the computed subspaces and singular values.

However, this choice of $\hat{\mu}$ to approximate μ is not always accurate. In the best case, when the columns of A_2 are orthogonal, then $\hat{\mu} = \mu$. However, in the worst case, where the columns of A_2 are coplanar with the same norm ($\hat{\mu}$), then $\mu = \sqrt{\frac{n-k_0}{l}}\hat{\mu}$. It is always the case that $\hat{\mu}$ underestimates μ; the extent of this underestimation depends on the directional information in the discarded columns. The result is that the bounds predicted using $\hat{\mu}$ are lower than the proven bounds. In some cases they do not actually bound the error in the computed bases and singular values.

An experiment was run using the 1-D flow field dataset described in Chapter 3, with $k = 5$ and $l = 1$. The resulting errors in the singular values and computed subspaces are shown in Tables 5.1 and 5.2. In this test, $\mu = 0.0236$, while the approximation is $\hat{\mu} = 0.0070$ (both relative to the norm of A). This caused the approximate bounds on the error in the

computed left subspace and the singular values to be too low, and they do not, in fact, bound the error in the computed singular values and subspaces.

Table 5.1: Predicted and computed angles (in degrees) between computed subspace and exact singular subspace, for the left (θ) and right (ϕ) computed singular spaces, for the 1-D flow field dataset, with $k = 5$ and $l = 1$.

	θ_k	$\hat{\theta}_k$	ϕ_k	$\hat{\phi}_k$
$k = 5, l = 1$	29.1222	5.2962	38.4531	87.8249

Table 5.2: Singular value error and approximate bound for the 1-D flow field data set, with $k = 5$, $l = 1$, $\sigma_6 = 1.3688e + 03$.

i	σ_i	$\hat{\sigma}_i$	$\lvert\sigma_i - \hat{\sigma}_i\rvert / \lVert A\rVert_2$	$\hat{\mu}^2/(2\hat{\sigma}_i)$
1	6.4771e+04	6.4771e+04	1.2501e-07	2.4766e-05
2	3.5980e+03	3.5471e+03	7.8558e-04	4.5224e-04
3	2.6262e+03	2.5696e+03	8.7311e-04	6.2427e-04
4	2.0342e+03	1.9387e+03	1.4746e-03	8.2744e-04
5	1.6909e+03	1.5655e+03	1.9362e-03	1.0247e-03

Note that, while the fifth canonical angle (θ_5) between the exact and computed subspace is not very good, the fifth computed singular value ($\hat{\sigma}_5$) matches well with the fifth singular value (σ_5) of A. This is because the gap between σ_5 and σ_6 is relatively small, meaning that any linear combination of the fifth and sixth singular vectors can capture nearly as much energy as the fifth singular vector.

While the performance of the incremental algorithm for this matrix is very good (in term of the approximation of the singular values), a poor estimation of μ by $\hat{\mu}$ yields poor approximations to the bounds, causing them to be overly optimistic about the error in the computed subspaces and singular values. The next section proposes measures to provide better estimates of the bounds derived by Chahlaoui et al. and demonstrates the effectiveness of these measures.

5.3 Improving Bound Approximations

This section outlines two methods for improving the approximation of μ, with the goal of producing better approximations to the bounds derived by Chahlaoui et al. The first method seeks to improve the approximation $\hat{\mu}$ by increasing the block size of the algorithm. The second method seeks to approximate μ directly, by estimating the norm of the matrix $\|A_2\|_2$ (the discarded data).

Consider again the term μ, defined $\mu = \|A_2\|_2 = \left\| \left[\tilde{Q}_1 \tilde{R}_1 \quad \ldots \tilde{Q}_f \tilde{R}_f \right] \right\|_2$, and $\hat{\mu} = \max_{i=1}^{f} \hat{\sigma}_{k_i+1}^{(i)} = \max_{i=1}^{f} \|\tilde{R}_i\|_2$. It is easily shown that the following relationship exists between μ and $\hat{\mu}$:

$$\hat{\mu} \leq \mu \leq \sqrt{f}\hat{\mu} \approx \sqrt{\frac{n-k_0}{l}}\hat{\mu}.$$

As the block size becomes larger, the right hand term approaches $\hat{\mu}$, and in the limit, $\hat{\mu}$ and μ become equal. Figure 5.2 shows the effect of a larger block size on $\hat{\mu}$ and μ.

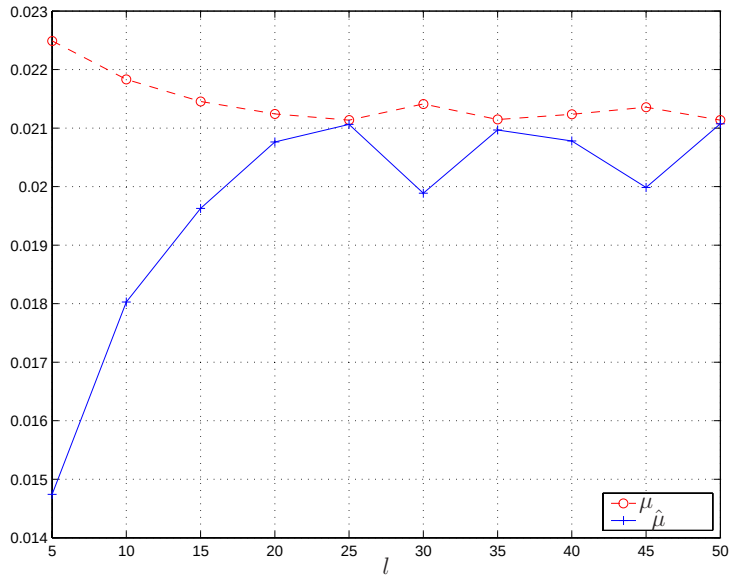

Figure 5.2: The effect of a larger block on μ and $\hat{\mu}$ for the 1-D flow field data set, with $k = 5$ (numbers relative to $\|A\|_2$).

As described before, the reason that $\hat{\mu}$ for $l = 1$ is a poor estimator is because of correlation between the discarded vectors at each time step, invalidating the assumption

in Geman's theorem. By increasing the block size from $l = 1$, more vectors are discarded at each time step as well. Recall the application of Geman's theorem to estimate the norm of the matrix A_2. This analysis occurred for a block size of $l = 1$. For values of $l > 1$, there are more columns discarded at each step. Because the data discarded at each step has a known norm, the matrix A_2 can be approximated (for the purpose of finding its norm) by one of rank $\frac{n-k_0}{l}$, instead of $n - k_0$. If Geman's theorem is applied to this lower rank matrix, with a norm denoted by γ, then the result is the following:

$$\hat{\mu} \leq \gamma \leq c\hat{\mu}, \quad c \approx 1 + \sqrt{\frac{n-k}{nl}} \approx 1 + \sqrt{\frac{1}{l}}.$$

It is proposed then that the applicability, as well as the quality, of the approximation of μ by $\hat{\mu}$ should improve as the block size of the algorithm increases. More work is needed to put this on rigorous analytical ground.

Furthermore, it is intuitive that increasing the block size should improve this estimate. As the block size is increased to the limit, the algorithm is run with a single incremental step, which uses the entire remaining part of A to update the initial singular value decomposition. Then the singular values and vectors computed are in fact the exact (up to finite precision) values and vectors and $\hat{\mu} = \mu$.

Table 5.3: Predicted and computed angles (in degrees) between computed subspace and exact singular subspace, for the left (θ) and right (ϕ) computed singular spaces, for the 1-D flow field dataset, with $k = 5$ and $l = \{1, 5\}$.

	μ	$\hat{\mu}$	θ_k	$\hat{\theta}_k$	ϕ_k	$\hat{\phi}_k$
$k = 5, l = 1$	0.0236	0.0070	29.1222	5.2962	38.4531	87.8249
$k = 5, l = 5$	0.0225	0.0147	21.6580	28.1646	29.4404	89.2113

The resulting improvement in $\hat{\mu}$ yields bounds that are closer to those predicted by [12]. Consider the experiment from Section 5.2: the incremental algorithm applied to the 1-D flow field dataset with $k = 5$, but with $l = 5$. Table 5.3 shows the resulting approximations for the error in the subspaces, compared against those produced with $l = 1$. Note that the $\hat{\mu}$ is a much better approximation for μ. Furthermore, the approximate bounds on the subspace errors (Table 5.3) and the singular values errors (Table 5.4) now hold.

Table 5.4: Singular value error and approximate bound for the 1-D flow field data set, with $k = 5$, $l = 5$.

i	$\lvert\sigma_i - \hat{\sigma}_i\rvert / \lVert A \rVert_2$	$\hat{\mu}^2/(2\hat{\sigma}_i)$
1	5.6597e-08	1.0866e-04
2	3.9020e-04	1.9699e-03
3	4.7044e-04	2.7114e-03
4	8.3129e-04	3.5539e-03
5	1.1414e-03	4.3526e-03

Another method for improving the approximate bounds is to directly estimate the value μ based on the truncated data. Recall the definition of μ:

$$\mu \doteq \|A_2\|_2 = \left\| \begin{bmatrix} \tilde{Q}_1\tilde{R}_1 & \cdots & \tilde{Q}_f\tilde{R}_f \end{bmatrix} \right\|_2 .$$

Because the discarded data is large ($m \times (n-k)$), its norm cannot be computed exactly. In fact, it infeasible even to store this matrix. But, as the individual matrices $\tilde{Q}_i$ and $\tilde{R}_i$ may be produced at each time step, a low-rank incremental algorithm may be used to approximate the dominant SVD of A_2, thereby providing an estimated norm of this matrix. This requires forming the products $\tilde{Q}_i\tilde{R}_i$ and conducting an incremental algorithm in addition to the main one. This is expensive and complicated.

An alternative is to perform the incremental algorithm on A as usual, but not keeping the k_i directions dictated by the threshold parameter. Instead, keep the factorization of rank $k_i + 1$, except on the very last step, where this extra tracked direction is dropped, its current norm being a provably better approximation to μ than the $\hat{\mu}$ proposed by Chahlaoui et al.

This can be shown as follows. Because $\hat{\mu} = \max_i \mu_i$, there is some step i where $\hat{\mu}_{old} = \mu_i = \hat{\sigma}^{(i)}_{k+1}$. When running with rank $k+1$, this extra singular value and its directional information are kept. It was shown earlier that the tracked singular values increase monotonically, so that when processing the last block of columns of A, the rank is selected to be k, discarding the extra singular triplet that was being tracked, and obtaining $\hat{\mu} \doteq \hat{\sigma}^{(f)}_{k+1} \geq \hat{\sigma}^{(i)}_{k+1} = \hat{\mu}_{old}$.

The benefit of this method is that only one incremental algorithm must be run. By keeping more than just one extra column, increased performance is expected. To illustrate this, the previous experiments are run again, with $k = 5(+1)$, for $l = 1$ and $l = 5$. For the

$l = 1$ case, $\mu = 0.0220$ and $\hat{\mu} = 0.0192$ (relative to the norm of A). Note from Tables 5.5 and 5.6 that the errors in the subspaces and singular values fall within the bounds predicted. As with the increased block size example from before, there is an increase in accuracy due to the larger tracking rank being able to preserve more information from step to step. However, there is a much greater increase in the bounds, because of the better approximation of μ by $\hat{\mu}$. For comparison with Tables 5.2 and 5.4, the $l = 5$ test is run again, here with the extra dimension of tracking. Here $\mu = 0.0218$ is much better approximated by $\hat{\mu} = 0.0195$, and the errors in the subspaces and singular values again fall within the approximate bounds.

Table 5.5: Predicted and computed angles (in degrees) between computed subspace and exact singular subspace, for the left (θ) and right (ϕ) computed singular spaces, for the 1-D flow field dataset, with $k = 5(+1)$ and $l = \{1, 5\}$.

	μ	$\hat{\mu}$	θ_k	$\hat{\theta}_k$	ϕ_k	$\hat{\phi}_k$
$k = 5(+1), l = 1$	0.0220	0.0192	17.9886	55.4335	27.1941	89.6211
$k = 5(+1), l = 5$	0.0218	0.0195	16.3042	56.2570	25.0185	89.6277

Table 5.6: Singular value error and approximate bound for the 1-D flow field data set, with $k = 5(+1)$, $l = \{1, 5\}$.

	$l = 1$		$l = 5$	
i	$\lvert\sigma_i - \hat{\sigma}_i\rvert / \lVert A\rVert_2$	$\hat{\mu}^2/(2\hat{\sigma}_i)$	$\lvert\sigma_i - \hat{\sigma}_i\rvert / \lVert A\rVert_2$	$\hat{\mu}^2/(2\hat{\sigma}_i)$
1	8.8422e-08	1.8429e-04	7.1704e-08	1.8922e-04
2	5.7350e-04	3.3522e-03	4.7251e-04	3.4355e-03
3	5.7055e-04	4.6102e-03	4.7948e-04	4.7226e-03
4	9.3447e-04	6.0480e-03	7.8978e-04	6.1803e-03
5	1.1545e-03	7.3862e-03	9.8091e-04	7.5313e-03

Finally, note that the quality of the estimate $\hat{\mu}$ is independent of the quality of the computed factorization. Consider the matrix A:

$$A = \begin{bmatrix} Q_0 R_0 & Q_1^{(1)} R_1^{(1)} & \dots & Q_1^{(n)} R_1^{(n)} \end{bmatrix},$$

with Q_0, Q_1 having orthonormal columns, $Q_0 \perp Q_1$ and $\sup(R_0) \geq \left\| \begin{bmatrix} R_1^{(1)} & \dots & R_1^{(n)} \end{bmatrix} \right\|_2 = \sqrt{n}\|R_1\|_2$. (Superscripts are simply to illustrate repetition and make explicit the count.)

The rank of A is clearly just the rank of Q_0 plus that of Q_1, and Q_0 is easily shown to be a basis for the dominant left singular subspace. If the incremental algorithm is initialized with Q_0, then none of the update steps allow a change in the current subspace, and the algorithm trivially produces correct bases for the left and right singular subspaces, along with the singular values of A. Note, however, that when updating each step with $Q_1 R_1$, the discarded data has norm $\|R_1\|_2$, so that $\hat{\mu} = \|R_1\|_2$. However, it is clear that $\mu = \sqrt{n}\|R_1\|_2 = \sqrt{n}\hat{\mu}$. The term μ can be made arbitrarily larger than $\hat{\mu}$ by increasing the number n, so long as the singular values of R_0 are increased to fulfill the hypothetical assumptions.

5.4 Summary

This chapter illustrates some of the errors inherent in the incremental algorithms, due to the truncation of data at each time step. The attempts by previous authors to a priori bound these errors are reviewed and updated for the block version of the algorithm. In addition, these a priori bounds are tightened. The algorithm block size is shown to increase the quality of the factorization (as it approximates A). However, contrary to arguments made by Levy et al., this improvement is shown to have its source in an increasing factorization rank.

This chapter also reviews the a posteriori bounds on the error in the computed subspaces and singular values, derived by Chahlaoui et al. [12]. Their recommended approximations to these bounds are discussed, and these approximations are shown to sometimes perform poorly. New techniques are proposed to increase the quality of the approximation of the μ term, so that the approximations to the error bounds perform more reliably. The first technique works simply by increasing the block size of the algorithm, showing again the importance of this parameter with respect to the performance of the algorithm. The second technique suggests increasing the rank of the factorization until all of the columns have been processed.

In spite of the improvement in the bound approximations, the a posteriori bounds can fail, without any indication. This can occur even when the algorithm itself performs very well. This suggests that, in addition to the approximations for the bounds proposed by Chahlaoui et al., there is a need for some other mechanism for testing the quality of the results and signaling when the subspaces might be inaccurate. The next chapter proposes a set of methods that reveal when improvement in the computation are possible, by exploiting multiple passes through the matrix A.

CHAPTER 6

IMPROVING COMPUTED BASES

One of the motivating factors for the class of incremental algorithms described in this thesis is that the matrix A is too large to store in local memory. The result is that the matrix must be stored at a distance (e.g., physical vs. virtual memory, local vs. remote storage, storage vs. destruction). In such a case, it is not plausible to perform operations on the matrix (or the current decomposition) that make random accesses reading and writing across all columns of the matrix, as one would do in a standard block algorithm on a hierarchical memory. The incremental methods reduce the cost here by processing the matrix in pieces, taking a group of columns of A, using them to update a low-rank decomposition, and then discarding this group of columns to make room for the next. Each group of columns of the matrix is fetched from remote storage to local storage only once, i.e. the matrix is read-only and any writing is to a significantly smaller amount of storage containing the updated bases and work space.

As described in Chapter 5, part of the inaccuracy in the method comes from truncating data at each step and not having access to it again. This gives the method a local view of A at each step; the only information available consists of the current group of columns and an approximation to the dominant space of the preceding columns. The result, as shown before, is an approximation to the dominant SVD, whose performance varies based on the gaps between the singular values in A and the sizes of the blocks used to update the decomposition at each step.

In some applications, such as computational fluid dynamics or active target recognition, the columns of A are produced incrementally and size constraints may prevent the storage of the full matrix in main memory, i.e. fast reads/writes, but may allow the storage on a high-density remote medium that is essentially read-only. In this case, the matrix A is

available for further read-only incremental passes, in order to improve the quality of the subspaces produced by the initial incremental routine.

This chapter describes three methods for making a second pass (or more, if possible) through the columns of A, with the intent of improving the bases produced by the incremental methods described in the previous chapters. As in the single-pass algorithm, the accesses to the matrix A are restricted to read-only. Section 6.1 describes a pivoting method, wherein the columns of A are processed in a modified order, determined at run-time. Section 6.2 describes Echoing, which works by applying an incremental method to the matrix $B = \begin{bmatrix} A & \dots & A \end{bmatrix}$. Section 6.3 discusses Partial Correction, which considers the projection of A onto a low-rank space and uses the SVD of the projected matrix to correct the computed singular vectors.

6.1 Pivoting

It has been repeatedly stated that part of the error in the low-rank incremental algorithms is due to the truncation of information at step i that, while not important to the SVD of the system local to time i, would have contributed to the (global) SVD of A. It is natural to consider whether some permutation of the columns of A could have produced better results. Building on the explanation of error stated above, a simple heuristic follows: at time step i, if an incoming vector has a strong component in the direction of the currently tracked subspace, then process the column. Otherwise, defer its inclusion to a later time.

This permutation of columns can occur at different levels of granularity. One extreme is to allow a permutation only inside of the Gram-Schmidt update step. This allows the usage of a pivoted QR algorithm without comprising the triangular structure, which is necessary in some variants of the incremental algorithm. Another extreme is to allow a column to be delayed indefinitely and multiple times. A more reasonable method in practice would be to store delayed columns in a look-aside buffer, and process them at such point in time as they become relevant (according to some replacement policy) or the buffer becomes full.

The technique described below is part of a family of methods for determining a column-permutation of A, each characterized by making the decision to accept or to defer a column after some processing at step i. It is attractive because it performs this step as early as possible, so that as little computation as possible is performed on a column of A that may not be accepted into the decomposition at the current time step. Recall the process for updating the decomposition at step i with l new columns of A (operation count in square

brackets):

- compute coefficients $C = Q_{i-1}^T A_+$ $[2mkl]$
- compute QR of residual $Q_\perp R_\perp = A_+ - Q_{i-1}C$ $[2mkl + 4ml^2]$
- compute SVD of $\hat{R}$ $[O(k^3 + l^3)]$

By using a decision criteria based on C, the columns for update are selected early in the process, before the expensive QR decomposition. After selecting the columns for inclusion, the appropriate columns of C and A_+ must be retained to reflect this selection, and the update proceeds as usual from there.

The result of this method is that the entire matrix A is processed in a different column-order than initially specified. That is, a factorization is produced of the following form:

$$\begin{aligned} APV_f &= \begin{bmatrix} Q_f R_f & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix} \\ &\Rightarrow \\ A &= \begin{bmatrix} Q_f R_f & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix} \begin{bmatrix} W_f^T P^T \\ W_{\bar{f}}^{\perp T} P^T \end{bmatrix} \\ &\approx Q_f R_f (W_f^T P^T). \end{aligned}$$

The method is very heuristic and is outlined here as an example of a class of methods which process the columns of A in a different order. This idea was tested on a random matrix A of size $m \times n$, with $m = 10000$ and $n = 200$. At each step, the total energy of each incoming vector in the currently tracked subspace was used to decide if the vector would be used to update the current decomposition or if it would be pushed to the back of AP. Acceptance occurred if the vector contained at least 50% as much energy as the most energetic vector of the current vectors. The rank of the tracked algorithm was $k = 4$.

Table 6.1 shows the results from this experiment. For the pivoting code, the error in the computed subspaces is much less than that of the non-pivoting code. The number of column delays was 134, corresponding to an increase in work of approximately $m\frac{n}{2}k$. No column was rejected more than 4 times.

More important than these results, this method illustrates how pivoting methods may be implemented to increase the performance of incremental algorithms. The term performance is left unspecified, as a practitioner may choose heuristics which favor the preservation of orthogonality or some other measure, over that of accuracy of subspaces or representation.

Table 6.1: Performance (with respect to subspace error in degrees) for pivoting and non-pivoting incremental algorithms for random A, $K = 4$.

	Columns Processed	θ_k	ϕ_k
Non-pivot	200	89.44	89.46
Pivot	334	81.76	82.01

6.2 Echoing

In constructing a technique to take advantage of multiple passes through the matrix A, the first method to come to mind is to start processing the first columns of A again after processing the final columns of A. This naive method applies GenInc to the matrix $B = \left[\begin{array}{cc} A & A \end{array} \right]$. It is shown below that the SVD of this matrix is intimately related to the SVD of A and that applying GenInc to B provides a better approximation to the dominant SVD of A than does applying GenInc to A.

Take a matrix $A \in \mathbb{R}^{m \times n}$, with $m \gg n$ and $rank(A) = r \leq n$, and consider its SVD,

$$\begin{aligned} A &= U \left[\begin{array}{c} \Sigma \\ 0 \end{array} \right] V^T \\ &= \left[\begin{array}{cc} U_1 & U_2 \end{array} \right] \left[\begin{array}{cc} \Sigma_1 & 0 \\ 0 & 0 \end{array} \right] \left[\begin{array}{cc} V_1 & V_2 \end{array} \right]^T \\ &= U_1 \Sigma_1 V_1^T, \end{aligned}$$

where $U_1 \Sigma_1 V_1^T$ is the thin SVD of A, containing only the left and right singular vectors of A associated with the r non-zero singular values of A. Consider now the matrix $B = \left[\begin{array}{cc} A & A \end{array} \right]$. Because the range of B is clearly equal to the range of A, the rank of B is equal to the rank of A. Then the SVD of B can also be expressed in a rank-r thin form,

$$\begin{aligned} B &= \bar{U} \bar{\Sigma} \bar{V}^T \\ &= \bar{U} \bar{\Sigma} \left[\begin{array}{cc} \bar{V}_1^T & \bar{V}_2^T \end{array} \right] \\ &= \left[\begin{array}{cc} A & A \end{array} \right], \end{aligned}$$

where $\bar{U} \in \mathbb{R}^{m \times r}$, $\bar{\Sigma} \in \mathbb{R}^{r \times r}$, and $\bar{V}_1, \bar{V}_2 \in \mathbb{R}^{n \times r}$. Then it is known from this and the SVD of

A given above that

$$\begin{aligned} A &= U_1 \Sigma_1 V_1^T && (6.1) \\ &= \bar{U} \bar{\Sigma} \bar{V}_1^T && (6.2) \\ &= \bar{U} \bar{\Sigma} \bar{V}_2^T. && (6.3) \end{aligned}$$

Note that the matrices $\bar{V}_1$ and $\bar{V}_2$ are sub-matrices of the orthogonal matrix $\bar{V}$, and are not orthogonal. Therefore, the decompositions of A given by Equation (6.2) and Equation (6.3), though they do diagonalize A, are not singular value decompositions. However, there exists a relationship between these factorizations and the SVD of A in Equation (6.1).

Consider the matrix BB^T:

$$\begin{aligned} BB^T &= \begin{bmatrix} A & A \end{bmatrix} \begin{bmatrix} A^T \\ A^T \end{bmatrix} \\ &= AA^T + AA^T \\ &= U_1 \Sigma_1^2 U_1^T + U_1 \Sigma_1^2 U_1^T \\ &= U_1 (2\Sigma_1^2) U_1^T. \end{aligned}$$

Also recall that B has SVD $B = \bar{U} \bar{\Sigma} \bar{V}^T$, so that

$$\begin{aligned} BB^T &= (\bar{U} \bar{\Sigma} \bar{V}^T)(\bar{U} \bar{\Sigma} \bar{V}^T)^T \\ &= \bar{U} \bar{\Sigma}^2 \bar{U}^T. \end{aligned}$$

Both of these are eigen-decompositions for the matrix BB^T, so that the left singular vectors of B must be $\bar{U} = U_1$, and the singular values of B must be $\bar{\Sigma} = \sqrt{2\Sigma_1^2} = \sqrt{2}\Sigma_1$.

Substituting this back into Equations 6.2 and (6.3) yields

$$\begin{aligned} A &= U_1 \Sigma_1 V_1^T \\ &= U_1 (\sqrt{2}\Sigma_1) \bar{V}_1^T \\ &= U_1 (\sqrt{2}\Sigma_1) \bar{V}_2^T. \end{aligned}$$

so that

$$V_1 = \sqrt{2}\bar{V}_1 = \sqrt{2}\bar{V}_2$$

are all orthogonal matrices. It follows that

$$A = U_1 \Sigma_1 V_1^T = \bar{U} (\frac{1}{\sqrt{2}} \bar{\Sigma}_1)(\sqrt{2} \bar{V}_2^T)$$

relates the SVD of B to that of A.

The proof above, for simplicity, derived the SVD of the matrix A from that of the matrix $B = \begin{bmatrix} A & A \end{bmatrix}$. A proof for $B = \begin{bmatrix} A & \ldots & A \end{bmatrix}$ is *mutatis mutandis* the same.

Applying GenInc to $B = \begin{bmatrix} A & \ldots & A \end{bmatrix}$ produces an approximation to the left singular vectors. However, the computed singular values and right singular vectors are those of the matrix B. While a full SVD of B will produce the singular values and right singular vectors of A, the approximate dominant SVD produced by the incremental algorithm does not. The desired information can be recovered.

Consider the decomposition produced by applying GenInc to $B = \begin{bmatrix} A & A \end{bmatrix}$. Only two repetitions of A are considered here, for simplicity. The algorithm computes a matrix W_f such that

$$\begin{aligned} B &= \begin{bmatrix} A & A \end{bmatrix} \\ &\approx U\Sigma V^T \\ &= \begin{bmatrix} U\Sigma V_1^T & U\Sigma V_2^T \end{bmatrix}. \end{aligned}$$

Unlike in the proof above, $U\Sigma V^T$ is not the SVD of B, so that neither $U\Sigma V_1^T$ nor $U\Sigma V_2^T$ is the SVD of A. However, an approximate singular value decomposition can be extracted from these matrices. Taking the QR decomposition of the matrix $V_2 = Q_v R_v$ yields

$$\begin{aligned} U\Sigma V_2^T &= U(\Sigma R_v^T)Q_v^T \\ &= (U\hat{U})\hat{\Sigma}(Q_v\hat{V})^T. \end{aligned}$$

This describes an approximate SVD for A. Note that while $\hat{U}$ rotates U to singular vectors for A, the computed U already approximates the dominant left singular subspace. A similar statement is true for the dominant right singular subspace, with this subspace being approximated by any orthonormal basis for the range of V_2. Finally, the matrix R_v^T has the effect of scaling the computed approximations of the singular values of B to values more suitable for approximating A.

The question remains as to whether the bases computed via echoing are a better approximation? Intuition suggests that on a second pass through A, the incremental algorithm will capture at least as much energy as it did the first time around. That is to say, the basis produced for the dominant left singular subspace is expected to be a better

approximation to the actual left singular subspace of A. While there is not yet a proof for this assertion, it is supported by empirical evidence.

One explanation is that on later passes through A, more energy has been accumulated in the middle matrix R_i, so that local directions are less able to move the left singular subspace. Because, by definition of the dominant singular subspace of A, those directions should have more energy when used to update A, and are better able to steer the current basis towards the true subspace.

Figure 6.1 shows the error in the computed subspaces as a function of the number of passes through the matrix. This error is measured as θ_k and ϕ_k, the largest canonical angles (in degrees) between the left and right singular subspaces computed using the incremental algorithm and the R-SVD. First note that the computed subspaces do indeed converge to the true subspaces of A. Note also that the improvement from one pass to the next decreases. This is because the decomposition develops an "inertia", such that even the columns of A in the direction of the dominant subspace are decreasingly able to steer the current decomposition. Most of the improvement from echoing comes in the first few passes.

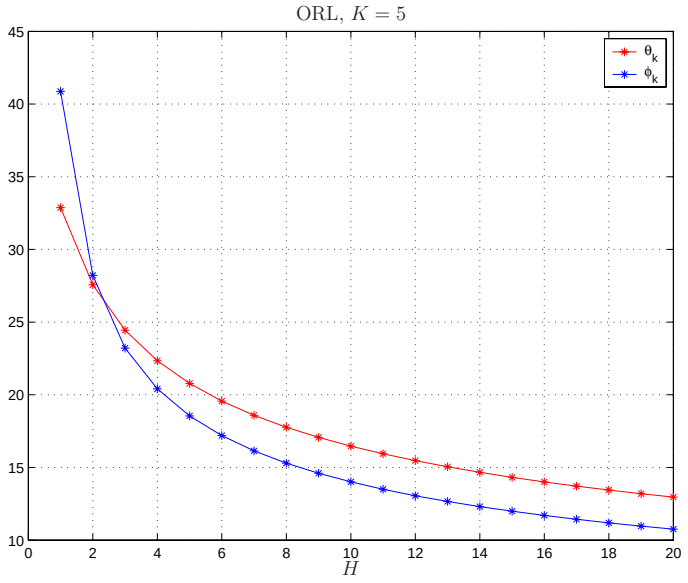

Figure 6.1: Angles between true and computed subspaces in degrees illustrates improvement from echoing, for the full ORL dataset with $K = 5$. Red is θ_k and blue is ϕ_k.

Applying GenInc to $B = \begin{bmatrix} A & A \end{bmatrix}$ requires approximately twice as much work as running one on A. The amount of post-processing necessary depends on the desired result:

- No post-processing is required to compute a basis for the dominant left singular subspace;
- Producing an orthonormal basis for the dominant right singular subspaces requires a QR factorization of $V_2^T = Q_v R_v$ (cost: $O(nk^2)$);
- Producing dominant singular values of B requires computing the SVD of ΣR_v^T (cost: $O(k^3)$), in addition to the steps above;
- Producing the dominant left and right singular vectors of B requires matrix multiplications (cost: $2mk^2$ and $2nk^2$, respectively), in addition to the steps above.

Echoing has been shown to produce a worthwhile increase in the result of the incremental algorithm. Unlike the pivoting algorithm, echoing is easy to implement, requiring few changes to the code. The only requirement is that the columns of A are available for processing multiple times in a read-only fashion. Furthermore, this method requires no additional work space, as long as the incremental algorithm does not start producing the right singular basis until the appropriate step.

6.3 Partial Correction

The incremental algorithms described in this thesis partially produce a decomposition for the matrix A of the form

$$A = UMV^T = \begin{bmatrix} \hat{U} & \hat{U}_\perp \end{bmatrix} \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix} \begin{bmatrix} \hat{V} & \hat{V}_\perp \end{bmatrix}^T. \tag{6.4}$$

The modifier "partial" is added because the incremental algorithms only produce the matrices $\hat{U}$, $\hat{\Sigma}$, and $\hat{V}$, and not their complements.

This decomposition is to be compared against the true singular value decomposition of A, partitioned conformally,

$$A = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1 & V_2 \end{bmatrix}^T. \tag{6.5}$$

The goal of the incremental algorithm is to compute approximate bases for the dominant subspaces U_1 and V_1. These approximations are the $\hat{U}$ and $\hat{V}$ shown in Equation (6.4).

The matrix $M = \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix}$ contains all of the information necessary to correct our approximations, provided the SVD of M and the complements $\hat{U}_\perp$ and $\hat{V}_\perp$ were available.

Assuming that $\hat{U}_\perp$ and $\hat{V}_\perp$ were available, the update would follow trivially. First, compute the matrix $M = U^T A V$ and the SVD $M = U_M \Sigma_M V_M^T$. Then use this SVD to update the computed matrices as follows,

$$\begin{aligned} U_{new} &= U U_M \\ \Sigma_{new} &= \Sigma_M \\ V_{new} &= V V_M \end{aligned}$$

Note that this method computes exactly the singular value decomposition of A, completely correcting the bases $\hat{U}$ and $\hat{V}$ and the singular values $\hat{\Sigma}$. This approach is very similar to the R-SVD method. This similarity includes the high cost, unfortunately. Without even discussing how $\hat{U}_\perp$ and $\hat{V}_\perp$ were obtained, this correction requires $2mn^2 + O(n^3)$ for the production of M, another $O(n^3)$ for the SVD of M, and a final $2mnk + 2n^2k$ for the production of the first k columns of U_1 and V_1. A user willing and able to invest the resources (time and storage) required for this operation could obtain the dominant SVD of A from the R-SVD.

Other methods exist for finding the dominant singular subspaces of M. In [16][17], Stewart and Mathias propose an iterative technique for driving a block triangular matrix to block diagonal. Applying this method to M would iteratively improve the computed subspaces. However, this method still requires that the matrix M is available, in addition to the basis complements $\hat{U}^\perp$ and $\hat{V}^\perp$.

These techniques can be approximated, requiring much less work and space. Assuming the availability of some matrix $\hat{U}$ which approximates the dominant left singular subspace of A. Given some basis $\hat{U}_p$, of rank p and orthogonal to $\hat{U}$, compute the matrix $\hat{M}$,

$$\begin{aligned} B &\doteq \begin{bmatrix} \hat{U} & \hat{U}_p \end{bmatrix} \\ \hat{M} &= B^T A. \end{aligned}$$

Compute the SVD of $\hat{M} = U_{\hat{M}} \Sigma_{\hat{M}} V_{\hat{M}}^T$ and use these matrices to update the decomposition:

$$\begin{aligned} U_{new} &= B U_{\hat{M}} \\ \Sigma_{new} &= \Sigma_{\hat{M}} \\ V_{new} &= V_{\hat{M}}. \end{aligned}$$

Because $U_{\hat{M}} \Sigma_{\hat{M}} V_{\hat{M}}^T = B^T A$, then $B U_{\hat{M}} \Sigma_{\hat{M}} V_{\hat{M}}^T = B B^T A$. Then the SVD computed by this partial correction step is the exact SVD of the projection of A onto the space $\mathcal{R}(B)$. This result has multiple consequences.

The first consequence is that, given B whose range contains the true dominant subspace of A, this correction step will produce the true dominant singular subspace and singular values of A. This is easily proven and results from the fact that the dominant SVD of A is invariant under projection down to the dominant singular subspace of A.

The second consequence is that only one correction step is worthwhile using a specific basis B. This is because the result of a correction is the dominant SVD of A projected to $\mathcal{R}(B)$; if B is kept the same, then the result will not change.

An implementation for this correction requires a basis $\hat{U}$ for which improvement is desired, along with some basis $\hat{U}_p$ to expand the space. It has already been stated that if $\mathcal{R}(U_1) \subseteq \mathcal{R}(\hat{U}) \cup \mathcal{R}(\hat{U}_p)$, then the correction will yield the exact dominant SVD of A. Having computed $\hat{U}$ by applying GenInc to A, the motivating desire is that it approximates the dominant SVD of A fairly well. However, without some other knowledge about the dominant left singular subspace of A, there is no way to select $\hat{U}_p$ to complete the space $\mathcal{R}(U_1)$.

The issue remains on how to select $\hat{U}_p$. A random basis, chosen orthogonal to $\hat{U}$, will not help to complete $\mathcal{R}(\hat{U})$ (with respect to $\mathcal{R}(U_1)$). The reason is that, because $m \gg n$, a randomly chosen basis will likely not contain any information in $\mathcal{R}(A)$, least of all in $\mathcal{R}(U_1) \subset \mathcal{R}(A)$.

One proposal for $\hat{U}_p$ is to use the discarded vectors $\tilde{Q}_f$ from the last step of the incremental algorithm. Another possibility is to take the first p columns of A and use them to construct $\hat{U}_p$, by orthogonalizing them against $\hat{U}$. This first proposal is attractive because it requires only an extra matrix multiply to produce on the last incremental step (cost: $2m(k+p)p$), while the second proposal requires a Gram-Schmidt orthogonalization of $A_{(1:p)}$ (cost: $4m(k+p)p$). However, for the first method, the value of p is determined by

the number of vectors discarded on the last step, while p is determined by the user for the second method. The partial correction method described here works increasingly well for larger values of p, so that one may not wish to be limited in the choice of p.

Once $\hat{U}_p$ is determined, the partial correction step proceeds as described above. The cost is outlined here:

- $2mn(k+p)$ for the production of $\hat{M} = B^T A$, requiring $(k+p)n$ memory to store the result,
- $4n(k+p)^2 + 2nk(k+p) + O(k^3+p^3)$ for the dominant SVD of $\hat{M}$, via the R-SVD, and
- $2mk(k+p)$ to produce the first k columns of U_{new}, requiring mk memory for work space.

Note that all of these terms are linear in both m and n, the dimensions of A. The only usage of A is in the multiplication $B^T A$, which can be easily streamed. If $p = n$, the total cost is $4mnk + 4mk^2 + 20nk^2 + O(k^3)$. Considering $m \gg n$ and $n \gg k$, this term can be simplified to $4mnk$.

Table 6.2 shows the improvement using the Partial Correction method, compared against Echoing. GenInc was applied to the COIL20-720 dataset, with $K = 5$. Running the data again with twice Echoing, the errors in the subspaces are reduced, at a cost of double that for the GenInc. However, for a partial correction with $p = k$ on the basis produced by the basic GenInc, the quality of the bases exceeds that of those produced using Echoing, at a much lower cost. Furthermore, when p is allowed to increase so that the cost of the Partial Correction approaches that of the Echoing, the quality of the produced bases increases even further. The partial complement $\hat{U}_p$ was computed in these experiments using two steps of classical Gram-Schmidt reorthogonalization on the first p columns of A.

The process described above computes the exact SVD of the projection of A onto the range of B. However, if the expense of the SVD of $\hat{M}$ in this technique is still too high, there are approximation techniques to reduce the computational and memory requirements. One possibility is to approximate the SVD of $\hat{M}$ of via a low-rank incremental, such as the GenInc. Another possibility is to use the method of Stewart and Mathias [16][17] to improve the estimated subspaces. Both of these techniques will approximately compute the SVD of the projection of A onto the range of B.

Table 6.2: Partial Completion compared against a one-pass incremental method, with and without Echoing, for the C20-720 data with $K = 5$.

	Cost	$\tan(\theta_k)$	$\tan(\phi_k)$
GenInc	$10mnk$	3.234199	4.242013
Echoing ($h = 2$)	$20mnk$	2.594525	2.740707
Partial Correction ($p = k$)	$14mnk$	2.285459	1.812507
Partial Correction ($p = 2k$)	$16mnk$	2.018349	1.589710
Partial Correction ($p = 4k$)	$20mnk$	1.851909	1.448108

6.4 Summary

This chapter proposes three methods for improving the quality of the bases computed by allowing extra read-only incremental passes through an incremental algorithm. The first method, Pivoting, outlines a general strategy for improving the quality of the bases by processing the columns of A in a different order. A simple heuristic for determining this order is proposed and improvements are demonstrated for a random matrix. A second method is described, Echoing, that processes the matrix A multiple times. This simple idea is demonstrated to yield improvements in both the computed dominant singular vectors and values, most notably in the right singular subspace. Finally, a third method is proposed, Partial Correction, which uses an approximation of a method known to compute the exact SVD. This technique is shown to be capable of better results than Echoing, with a significantly lower cost. Further approximations to Partial Correction are suggested to reduce the cost of the algorithm.

CHAPTER 7

CONCLUSION

Incremental algorithms for tracking the dominant singular subspaces of a matrix have been studied numerous times before. Many authors have described the same heuristic for tracking these subspaces, but they have each produced different methods with differing results and requirements. This thesis presents a generic algorithm for incrementally computing the dominant singular subspaces of a large matrix. This framework unifies the works of the previous authors, clarifying the differences between the previous methods in a way that illustrated the varying results. This common framework also allowed a transference of analyses to be made, from method to method. Furthermore, this framework demonstrates exactly what is necessary to compute the subspaces, and by doing so, is able to produce a more efficient method than those previously described. This efficiency is shown to be present across a greater range of operating parameters, and is illustrated empirically with application data. Recommendations are made that suggest which method should be used under which operating conditions, whether they are parametric (such as block size or data dimension) or architectural (such as primitive efficiency).

This work illustrates the benefits of a block algorithm over a scalar one. The benefits are computational, in terms of decreased runtime. Block algorithms, despite having a larger operation count, outperform scalar algorithms when using level-3 primitives which exploit a modern memory hierarchy. The benefits are numerical, with a larger block size yielding potential improvement in the computed results. The effect of increasing block size on existing error analyses is also discussed, with numerous corrections and additions made to the work of the previous authors. In particular, the approximate bounds of Chahlaoui et al. are shown to be more reliable as the block size is increased.

Finally, novel methods are proposed to improve the quality of the computed factorization

and subspaces, when allowed a second pass through the matrix. A general "pivoting" framework is described, and a heuristic is proposed to implement it. This simple method is empirically illustrated to give the intended improvements in the computed results. Echoing, a straightforward method for passing multiple times through the data matrix, is heuristically motivated, and empirically demonstrated to be effective in producing a better result. Lastly, Partial Correction describes a method for correcting the dominant subspaces, using a low-rank approximation of a method known to produce the exact singular value decomposition.

7.1 Future Work

The generic incremental algorithm proposed by this thesis is still a very heuristic one, and it is difficult to say anything in a rigorous fashion without making some assumptions about the nature of the matrix A and its singular value decomposition. Many areas of future work exist in this thread. As the matrices G_u and G_v describe whether the current space is evolving during each update, these matrices may be analyzed for clues as to the convergence of the tracked subspaces. Also, the second-pass echoing method described in Chapter 6 should be studied in more detail. Possibilities for improvement include a method for selective echoing of input data, based perhaps on some characteristic of the previously associated update step; or an analysis of convergence of this method.

APPENDIX A

Proof of the Complete Decomposition

It is stated in [11] that, at each step i of the incremental algorithm, there exists an orthogonal matrix V_i such that

$$A_{(1:i)}V_i = A_{(1:i)}\begin{bmatrix} W_i & W_i^{\perp} \end{bmatrix} = \begin{bmatrix} Q_iR_i & \tilde{q}_1\rho_1 & \dots & \tilde{q}_i\rho_i \end{bmatrix}.$$

and such that V_i embeds W_i as shown above.

This result is used to derive error bounds on the factorization produced by the incremental algorithm. However, this result is never proven in the discussion of the scalar algorithm. This appendix shows that the above holds not only for the $l_i = 1$ scalar case, for the block case in general.

First, note that the first step of the algorithm updates the factorization $A_{(1:s_0)} = Q_0R_0$ to

$$\begin{aligned} A_{(1:s_1)} &= \bar{Q}\bar{R}G_v^T \\ &= \begin{bmatrix} Q_1 & \tilde{Q}_1 \end{bmatrix} \begin{bmatrix} R_1 & 0 \\ 0 & \tilde{R}_1 \end{bmatrix} \begin{bmatrix} W_1 & \tilde{W}_1 \end{bmatrix}^T. \end{aligned}$$

Then choosing $V_1 = G_v = \begin{bmatrix} W_1 & \tilde{W}_1 \end{bmatrix}$, it follows that

$$A_{(1:s_1)}V_1 = \begin{bmatrix} Q_1 & \tilde{Q}_1 \end{bmatrix} \begin{bmatrix} R_1 & 0 \\ 0 & \tilde{R}_1 \end{bmatrix} = \begin{bmatrix} Q_1R_1 & \tilde{Q}_1\tilde{R}_1 \end{bmatrix},$$

showing the desired result for our base case of $i = 1$.

Now, assume for the purpose of induction that this result holds after step i. Then there exists a matrix V_i such that

$$A_{(1:s_i)}V_i = A_{(1:s_i)}\begin{bmatrix} W_i & W_i^{\perp} \end{bmatrix} = \begin{bmatrix} Q_iR_i & \tilde{Q}_1\tilde{R}_1 & \dots & \tilde{Q}_i\tilde{R}_i \end{bmatrix}.$$

The $i+1^{st}$ step of the algorithm updates the current factorization, $Q_i R_i W_i^T$ with the columns $A_+ = A_{(s_i+1:s_{i+1})}$, like so:

$$\begin{aligned}\begin{bmatrix} Q_i R_i W_i^T & A_+ \end{bmatrix} &= \begin{bmatrix} Q_i & Q_\perp \end{bmatrix} G_u G_u^T \begin{bmatrix} R_i & C \\ 0 & R_\perp \end{bmatrix} G_v G_v^T \begin{bmatrix} W_i & 0 \\ 0 & I \end{bmatrix}^T \\ &= \begin{bmatrix} Q_{i+1} & \tilde{Q}_{i+1} \end{bmatrix} \begin{bmatrix} R_{i+1} & 0 \\ 0 & \tilde{R}_{i+1} \end{bmatrix} \begin{bmatrix} W_{i+1} & \tilde{W}_{i+1} \end{bmatrix}^T.\end{aligned}$$

Consider the matrix

$$V_{i+1} = \begin{bmatrix} V_i & 0 \\ 0 & I_l \end{bmatrix} P \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T,$$

where

$$P = \begin{bmatrix} I_k & 0 & 0 \\ 0 & 0 & I_{s_i} \\ 0 & I_l & 0 \end{bmatrix}.$$

Then

$$\begin{aligned} A_{(1:s_{i+1})} V_{i+1} &= \begin{bmatrix} A_{(1:s_i)} & A_+ \end{bmatrix} \begin{bmatrix} V_i & 0 \\ 0 & I_l \end{bmatrix} P \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T \\ &= \begin{bmatrix} Q_i R_i & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_i \tilde{R}_i & A_+ \end{bmatrix} P \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T \\ &= \begin{bmatrix} Q_i R_i & A_+ & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_i \tilde{R}_i \end{bmatrix} \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T \\ &= \begin{bmatrix} Q_{i+1} R_{i+1} & \tilde{Q}_{i+1} \tilde{R}_{i+1} & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_i \tilde{R}_i \end{bmatrix} P^T \\ &= \begin{bmatrix} Q_{i+1} R_{i+1} & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_{i+1} \tilde{R}_{i+1} \end{bmatrix}, \end{aligned}$$

as required.

V_{i+1} is orthogonal, obviously satisfying $V_{i+1}^T V_{i+1} = I_k$. Furthermore, it is easily shown that V_{i+1} embeds W_{i+1} as desired:

$$\begin{aligned} V_{i+1} &= \begin{bmatrix} V_i & 0 \\ 0 & I_l \end{bmatrix} P \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T \\ &= \begin{bmatrix} W_i & W_i^\perp & 0 \\ 0 & 0 & I \end{bmatrix} P \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T \\ &= \begin{bmatrix} W_i & 0 & W_i^\perp \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} G_v & 0 \\ 0 & I_{s_i-k} \end{bmatrix} P^T \\ &= \begin{bmatrix} W_{i+1} & \tilde{W}_{i+1} & W_i^\perp \\ & & 0 \end{bmatrix} P^T \\ &= \begin{bmatrix} W_{i+1} & W_i^\perp \\ & 0 \end{bmatrix} \tilde{W}_{i+1} = \begin{bmatrix} W_{i+1} & W_{i+1}^\perp \end{bmatrix}. \end{aligned}$$

Each V_i then has the correct form and affect on $A_{(1:s_i)}$, and consists of the transformation G_v from each step of the algorithm along with specific permutations, as characterized in [11]. Note also that each $W_i^{\perp}$ has a block upper trapezoidal structure:

$$W_i^{\perp} = \begin{bmatrix} W_{i-1}^{\perp} & \\ 0 & \tilde{W}_i \end{bmatrix},$$

with diagonal blocks of order d_i. This result facilitates the error analyses in this thesis.

It should also be noted that this proof made no assumptions about the structure of R_i at each step, i.e. it may be diagonal (as in the SKL), triangular (as in the Recursive SVD), or unstructured (as in the generic incremental method presented in this paper.)

Then after f steps of the incremental algorithm, where all columns of A have been processed, there exists by the result above the decomposition

$$AV_f = A \begin{bmatrix} W_f & W_f^{\perp} \end{bmatrix} = \begin{bmatrix} Q_f R_f & \tilde{Q}_1 \tilde{R}_1 & \dots & \tilde{Q}_f \tilde{R}_f \end{bmatrix}.$$

The SVD of $R_f = \hat{U}\hat{\Sigma}\hat{V}^T$ yields the factorization

$$\begin{aligned} AV &= A \begin{bmatrix} W_f \hat{V} & W_f^{\perp} \end{bmatrix} \\ &= \begin{bmatrix} Q_f \hat{U} & Q_f^{\perp} \end{bmatrix} \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix} \\ &= U \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix}, \end{aligned}$$

where $Q_f^{\perp}$ is orthogonal to Q_f. Define A_2 as follows:

$$A_2 \doteq \begin{bmatrix} A_{1,2} \\ A_{2,2} \end{bmatrix}$$

with $\mu \doteq \|A_2\|_2$. Note that this gives an additive decomposition for A:

$$\begin{aligned} A &= U \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix} V^T \\ &= \begin{bmatrix} Q_f \hat{U} & Q_f^{\perp} \end{bmatrix} \begin{bmatrix} \hat{\Sigma} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix} \begin{bmatrix} W_f \hat{V} & W_f^{\perp} \end{bmatrix}^T \\ &= Q_f R_f W_f^T + U A_2 {W_f^{\perp}}^T. \end{aligned}$$

Then the norm of the residual error in the approximation $A \approx Q_f R_f W_f^T$ is

$$\begin{aligned} \|A - Q_f R_f W_f^T\|_2 &= \|Q_f R_f W_f^T + U A_2 {W_f^{\perp}}^T - Q_f R_f W_f^T\|_2 \\ &= \|U A_2 {W_f^{\perp}}^T\|_2 = \mu. \end{aligned}$$

APPENDIX B

On WY-Representations of Structured Householder Factorizations

Bischof and Van Loan describe in [13] how the product of d Householder reflectors can be represented in a block form, $H_1 H_2 \cdots H_d = I + WY^T$. When computing the QR factorization of a structured matrix, the reflectors may in turn have some structure, and this can show up in W and Y.

The method in 3.3.4.2 describes the transformation of the $k + d \times d$ matrix $U_2 S_u^T$, having the lower trapezoidal form

$$U_2 S_u^T = \begin{bmatrix} L \\ B \end{bmatrix},$$

This matrix is transformed by the matrix G_u, so that

$$G_u^T (U_2 S_u^T) = \begin{bmatrix} 0 \\ I_d \end{bmatrix}.$$

This is done by computing the QL factorization of the matrix $U_2 S_u^T$. It is also equivalent to computing the QR factorization of the matrix $\bar{U} \doteq E_1 (U_2 S_u^T) E_2$, where E_1 and E_2 are permutation matrices that reverse the order of the rows and columns of $U_2 S_u^T$, respectively. This work considers the latter case, the QR factorization of the permuted matrix, $\bar{U}$.

First, note some properties of $\bar{U}$. It is orthogonal, so the QR factorization produces a matrix which is both orthogonal and upper triangular, and therefore a diagonal matrix with unit-elements. Also note the structure of $\bar{U}$:

$$\bar{U} = \begin{bmatrix} B \\ U \end{bmatrix}.$$

Then the first column of the the matrix only has $k+1$ non-zeroes. Then the first reflector matrix, H_1, needs only be of order $k+1$ and modifies only the first $k+1$ rows of the matrix

$\bar{U}$. Furthermore, because $\bar{U}$ has orthonormal columns, then the first column is transformed to the elementary basis vector e_1 and the rest of the first row consists of zeroes. This is illustrated by the following:

$$\begin{aligned}
H_d \cdots H_2 H_1 \bar{U} &= H_d \cdots H_2 \begin{bmatrix} \tilde{H}_1 & 0 \\ 0 & I_{d-1} \end{bmatrix} \begin{bmatrix} b_0 & B_0 \\ 0 & U_0 \end{bmatrix} \\
&= H_d \cdots H_2 \begin{bmatrix} e_1 & \tilde{H}_1 B_0 \\ 0 & U_0 \end{bmatrix} \\
&= H_d \cdots H_3 \begin{bmatrix} 1 & 0 & 0 \\ 0 & \tilde{H}_2 & 0 \\ 0 & 0 & I_{d-2} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & b_1 & B_1 \\ 0 & 0 & U_1 \end{bmatrix} \\
&= H_d \cdots H_3 \begin{bmatrix} I_2 & 0 & 0 \\ 0 & b_3 & B_3 \\ 0 & 0 & U_3 \end{bmatrix} \\
&= \cdots \\
&= \begin{bmatrix} I_d \\ 0 \end{bmatrix}.
\end{aligned}$$

This process continues until $\bar{U}$ has been reduced to an upper triangular matrix, whose triangular part is the identity (up to sign). This is effected by a sequence of Householder reflectors, each of order $k+1$. Each of these reflectors has the form $H_i = I_{k+d} + \beta_i u_i u_i^T$. But the reduced order of these transformations reveals an additional structure, also shown above:

$$\begin{aligned}
H_i &= I_{k+d} + \beta_i u_i u_i^T = \begin{bmatrix} I_{i-1} & 0 & 0 \\ 0 & \tilde{H}_i & 0 \\ 0 & 0 & I_{d-i} \end{bmatrix} \\
u_i &= \begin{bmatrix} 0_{i-1} \\ 1 \\ \tilde{u}_i \\ 0_{d-i} \end{bmatrix} \\
\tilde{H}_i &= I_{k+1} + \beta_i \tilde{u}_i \tilde{u}_i^T.
\end{aligned}$$

It is assumed that the Householder vector defining each reflector has 1 as its first non-zero. This is typical (see [1], page 210), but not necessary. If this is not the case, it has only minor consequences that are discussed later.

Bischof and Van Loan describe the how W and Y may be built from β_i the individual reflectors. The resulting factorization, $I + WY^T$, has the same numerical properties as the individual Householder reflectors, but has the benefit of using level 3 primitives in

its application. Furthermore, it is later shown that when W and Y are constructed from reflectors structured like those H_i above, they have the structure:

$$W = \begin{bmatrix} B \\ U \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} L \\ B \\ U \end{bmatrix}, \tag{B.1}$$

where U represents an upper triangular matrix, B represents an unstructured matrix, and L represents a unit-diagonal, lower triangular matrix.

The structure of W and Y follows simply from their construction. Let $W_{(1)}$ and $Y_{(1)}$ be defined as follows:

$$\begin{aligned} W_{(1)} &= \beta_1 u_1 = \begin{bmatrix} \beta_1 \\ \beta_1 \tilde{u}_1 \\ 0_{d-1} \end{bmatrix} \\ Y_{(1)} &= u_1 = \begin{bmatrix} 1 \\ \tilde{u}_1 \\ 0_{d-1} \end{bmatrix}. \end{aligned}$$

Then trivially, $I + W_{(1)}Y_{(1)}^T = I + \beta u_1 u_1^T = H_1$, and W and Y have structure agreeing with that of (B.1).

Now assume for the purpose of induction that $W_{(i)}$ and $Y_{(i)}$ have the structure, dictated in (B.1):

$$W_{(i)} = \begin{bmatrix} B \\ U \\ 0_{d-i} \end{bmatrix} \quad \text{and} \quad Y_{(i)} = \begin{bmatrix} L \\ B \\ U \\ 0_{d-i} \end{bmatrix},$$

and that $I + W_{(i)}Y_{(i)}^T = H_1 \cdots H_i$, so that $W_{(i)}$ and $Y_{(i)}$ have the desired effect of the product of the first i Householder reflectors.

Bischof and Van Loan describe the update of $W_{(i)}$ and $Y_{(i)}$ as follows:

$$\begin{aligned} W_{(i+1)} &= \begin{bmatrix} W_{(i)} & (I + W_{(i)}Y_{(i)}^T)\beta_{i+1}u_{i+1} \end{bmatrix} \\ Y_{(i+1)} &= \begin{bmatrix} Y_{(i)} & u_{i+1} \end{bmatrix}. \end{aligned}$$

Then the effect of $I + W_{(i+1)}Y_{(i+1)}^T$ is trivially shown as the accumulation of the Householder reflectors up to and including H_{i+1}:

$$\begin{aligned} I + W_{(i+1)}Y_{(i+1)}^T &= I + \begin{bmatrix} W_{(i)} & (I + W_{(i)}Y_{(i)}^T)\beta_{i+1}u_{i+1} \end{bmatrix} \begin{bmatrix} Y_{(i)} & u_{i+1} \end{bmatrix}^T \\ &= I + W_{(i)}Y_{(i)}^T + (I + W_{(i)}Y_{(i)}^T)\beta_{i+1}u_{i+1}u_{i+1}^T \\ &= (I + W_{(i)}Y_{(i)}^T)(I + \beta_{i+1}u_{i+1}u_{i+1}^T) \\ &= H_1 \cdots H_{i+1}. \end{aligned}$$

As for the structure of $W_{(i+1)}$ and $Y_{(i+1)}$, this is easily demonstrated. First, consider $Y_{(i+1)}$:

$$\begin{aligned} Y_{(i+1)} &= \begin{bmatrix} Y_{(i)} & u_{i+1} \end{bmatrix} \\ &= \begin{bmatrix} L & 0_i \\ B & \\ U & 1 \\ 0 & \tilde{u}_{i+1} \\ 0_{d-i-1} & 0_{d-i-1} \end{bmatrix} \\ &= \begin{bmatrix} L' \\ B' \\ 0_{d-(i+1)} \end{bmatrix}. \end{aligned}$$

Consisting of only the u_i matrix from each step, stacked back to back, the structure of Y is a trivial result of the structure in u_i.

Next, consider $W_{(i+1)}$. The last column of $W_{(i+1)}$ is the product $(I + W_{(i)}Y_{(i)}^T)\beta_{i+1}u_{i+1}$. Its structure is shown to be:

$$\begin{aligned} (I + W_{(i)}Y_{(i)}^T)\beta_{i+1}u_{i+1} &= \beta_i(u_{i+1} + W_{(i)}z) \\ &= \beta_i(\begin{bmatrix} 0_i \\ 1 \\ \tilde{u}_{(i+1)} \\ 0_{d-i-1} \end{bmatrix} + \begin{bmatrix} Bz \\ Uz \\ 0 \\ 0_{d-i-1} \end{bmatrix}) \\ &= \begin{bmatrix} w_1 \\ 0_{d-i-1} \end{bmatrix} + \begin{bmatrix} w_2 \\ 0_{d-i-1} \end{bmatrix} = \begin{bmatrix} w \\ 0_{d-i-1} \end{bmatrix}. \end{aligned}$$

Then $W_{(i+1)}$ has the structure

$$W_{(i+1)} = \begin{bmatrix} B & \\ U & w \\ 0 & \\ 0_{d-i-1} & 0_{d-i-1} \end{bmatrix} = \begin{bmatrix} B' \\ U' \\ 0_{d-(i+1)} \end{bmatrix}.$$

After the accumulation of d reflectors, then it follows that W and Y have the structure

$$W = \begin{bmatrix} B_w \\ U_w \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} L_y \\ B_y \\ U_y \end{bmatrix},$$

where B are unstructured matrices, U are upper triangular matrices, and L_y is a unit-diagonal, lower-triangular matrix. Note that in the case that the leading non-zero of each

Householder vector u_i is not 1, then L_y does not have unit diagonal elements. This has little bearing on the algorithm, with respect to computation or storage.

It has been shown then the form and construction method of a structured transformation that sends $\bar{U}$ to $\begin{bmatrix} I \\ 0 \end{bmatrix}$. This method can be modified to operate on the matrix $U_2 S_u^T$, yielding the same transformation. Note that

$$\begin{aligned}
\begin{bmatrix} 0_k \\ I_d \end{bmatrix} &= E_1 \begin{bmatrix} I_d \\ 0_k \end{bmatrix} E_2 \\
&= E_1 (H_d \cdots H_1) \bar{U} E_2 \\
&= E_1 (I + Y W^T) \bar{U} E_2 \\
&= E_1 (I + Y W^T) E_1 E_1 \bar{U} E_2 \\
&= (I + E_1 Y E_2 E_2 W^T E_1)(U_2 S_u^T) \\
&= G_u^T (U_2 S_u^T),
\end{aligned}$$

where $G_u = I + (E_1 W E_2)(E_1 Y E_2)^T$, and has structure

$$\begin{aligned}
G_u &= I + (E_1 W E_2)(E_1 Y E_2)^T \\
&= I + \left(E_1 \begin{bmatrix} B_w \\ U_w \end{bmatrix} E_2 \right) \left(E_1 \begin{bmatrix} L_y \\ B_y \\ U_y \end{bmatrix} E_2 \right)^T \\
&= I + \begin{bmatrix} L'_w \\ B'_w \end{bmatrix} \begin{bmatrix} U'_y & B'_y & L'_y \end{bmatrix}.
\end{aligned}$$

APPENDIX C

Testing Specifications

The timings presented in Section 4.1 were performed on a Sun Ultra-80, with 1 gigabyte of memory and dual 450MHz Ultra-II processors. The compiler used was the Fortran 95 compiler from the Sun Workshop 6, Update 2. The version of ATLAS was version 3.5.0. The version of the Netlib BLAS and LAPACK libraries was version 3.0 with the May, 2000 update.

The compiler options used to compile both the incremental libraries and the testing executable were:

```
FLAGS=-fast -ftrap=%none -fsimple=0 -fns=no
F90=f95 -dalign -w2 ${FLAGS}
```

Timing was performed in the executable, using the `gethrtime` subroutine.

REFERENCES

[1] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, third edition, 1996. 1, 4, 76

[2] G. W. Stewart. *Matrix Algorithms, Volume I: Basic Decompositions*. Society for Industrial and Applied Mathematics, Philadelphia, 1998. 1, 21

[3] L. Sirovich. Empirical eigenfunctions and low dimensional systems. In L. Sirovich, editor, *New Perspectives in Turbulence*, pages 139–163. Springer, New York, 1991. 3

[4] J. Lumley. The structure of inhomogeneous turbulent flows. In A. Yaglom and V. Tatarsky, editors, *Atmospheric Turbulence and Radio Wave Propagation*, page 166. Nauka, Moscow, 1967. 3

[5] L. Sirovich. Turbulence and the dynamics of coherent structures. Part I: Coherent structures. *Quarterly of Applied Mathematics*, 45(3):561–571, October 1987. 3

[6] M. Gu and S. C. Eisenstat. A stable and fast algorithm for updating the singular value decomposition. Technical Report YALEU/DCS/RR-966, Yale University, New Haven, CT, 1993. 7, 17

[7] S. Chandrasekaran, B. S. Manjunath, Y. F. Wang, J. Winkeler, and H. Zhang. An eigenspace update algorithm for image analysis. *Graphical models and image processing: GMIP*, 59(5):321–332, 1997. 8, 44

[8] B. S. Manjunath, S. Chandrasekaran, and Y. F. Wang. An eigenspace update algorithm for image analysis. In *IEEE Symposium on Computer Vision*, page 10B Object Recognition III, 1995. 8, 43, 48

[9] A. Levy and M. Lindenbaum. Sequential Karhunen-Loeve basis extraction and its application to images. *IEEE Transactions on image processing*, 9(8):1371–1374, August 2000. 9, 45, 46, 47

[10] M. Brand. Incremental singular value decomposition of uncertain data with missing values. In *Proceedings of the 2002 European Conference on Computer Vision*, 2002. 10

[11] Y. Chahlaoui, K. Gallivan, and P. Van Dooren. Recursive calculation of dominant singular subspaces. *SIAM J. Matrix Anal. Appl.*, 25(2):445–463, 2003. 10, 11, 72, 74

[12] Y. Chahlaoui, K. Gallivan, and P. Van Dooren. An incremental method for computing dominant singular spaces. In *Computational Information Retrieval*, pages 53–62. SIAM, 2001. 16, 19, 21, 42, 48, 49, 50, 51, 54, 57

[13] C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8:s2–s13, 1987. 25, 75

[14] G. W. Stewart and J.-G. Sun. *Matrix Perturbation Theory*. Academic Press, San Diego, 1990. 49

[15] S. Geman. A limit theorem for the norm of random matrices. *Annals of Probability*, 8(2):252–261, April 1980. 51

[16] G. W. Stewart. On an algorithm for refining a rank-revealing URV decomposition and a perturbation theorem for singular values. Technical Report CS-TR 2626, Department of Computer Science, University of Maryland, 1991. Revised version appeared as [17]. 66, 68

[17] R. Mathias and G. W. Stewart. A block QR algorithm and the singular value decomposition. *Linear Algebra and Its Applications*, 182:91–100, 1993. 66, 68, 82

BIOGRAPHICAL SKETCH

Christopher G. Baker

Christopher Grover Baker was born on September 5, 1979, in Marianna, Florida, to Frank and Lynn Baker. In the spring of 2002, he completed his Bachelors degree in Computer Science and Pure Mathematics at The Florida State University. Under the advisement of Prof. Kyle Gallivan, he obtained his Masters degree in summer of 2004, also from the Department of Computer Science at FSU. He enrolled in the doctoral program at FSU in the fall of 2004. Legend holds that he still remains there.

Chris's research interests include numerical linear algebra, computer graphics, scientific visualization, numerical optimization, and high performance computing.

Chris lives in Tallahassee, FL, with his wife Kelly Baker, a doctoral student in the Department of Religious Studies at FSU. They have two dogs and one cat.

Chris is a member of ACM, IEEE, and SIAM.