LAPACK-Style Algorithms and Software for Computing the Generalized Singular Value Decompositions: CSD, QSVD, and PSVD

by

JENNY JIAN WANG
B. S. (University of California, Davis) 2003

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved by:

Zhaojun Bai (Chair Advisor)

Premkumar T. Devanbu

Vladimir Filkov

Committee in Charge

2005

LAPACK-Style Algorithms and Software for Computing the Generalized Singular Value Decompositions: CSD, QSVD, and PSVD

by

Jenny Jian Wang

Contents

List of Tables

List of Figures

Acknowledgments

First, I would like to express my greatest gratitude to my chair advisor, Professor Zhaojun Bai, for his excellent guidance, invaluable advice, and countless amount of effort on evaluating the early drafts of my work. Meanwhile, I appreciate the cooperation of my committee members, Professor Premkumar Devanbu and Professor Vladimir Filkov. Thanks also to the theory laboratory for the well-accommodated resources and a comfortable place to work in all seasons.

Next I would especially thank Mike Zhang for his understanding and caring. I am grateful to his thoughtful and genuine advice that encourages me to do my best and help retain my sanity during difficulty times. I am also indebted to all my friends in Davis. I never forget the time when we laughed, triumphed, stressed, and struggled together. It is them who filled this small and peaceful town with plenty of exciting and unforgettable memories.

Last but not the least, I deeply appreciate the unconditional love of my parents and grandparents and the sacrifice they made to their accustomed lifestyle and secured future for brightening up mine. I'd always be thankful to them for the support throughout all these years and the way they shape who I am today.

Abstract

LAPACK has become a popular mathematical library for solving most of the common numerical linear algebra problems on modern computers. This thesis intends to extend the functionality of LAPACK by adding new subroutines for computing the cosine-sine decomposition (CSD), the quotient singular value decomposition (QSVD), and the product singular value decomposition (PSVD).

The CSD is defined for a partitioned matrix with orthonormal columns. This decomposition measures the separation between two subspaces and can be viewed as a special case of the QSVD. In fact, the algorithm for computing the QSVD that we present in this thesis is based on the CSD. While an implementation of the QSVD is available in current LAPACK, we demonstrate that with the trade off in memory requirement, not only is our algorithm stable, but it also achieves significant speed-up. The PSVD is defined for any two matrices with multiplicative compatibility. Among several ideas that are proposed to tackle the problem recently, we select to extend the study on a QR-like method. This method first implicitly bidiagonalizes the product of the input matrices; it then computes the SVD of the bidiagonal matrix.

The thesis researches and generalizes the stable algorithms for computing each of these decompositions; LAPACK-style software from those algorithms are developed. Calling sequences of subroutines are listed toward the end of this paper.

Chapter 1

Introduction

As interesting problems spawned from various fields and intriguing mathematical ideas emerge, it becomes crucial to tackle those problems by exploring new innovation using computing resources. Linear Algebra Package, *LAPACK*, is an open source library that contains subroutines for solving the most commonly occurring problems in numerical linear algebra, including linear equations, linear least square problems, eigenvalue problems and singular value problems. LAPACK also handles related computations such as matrix decompositions, estimation of condition numbers, etc. It supersedes LINPACK and EISPACK by carefully restructuring the software to adapt well to high-performance machines with deep memory hierarchies, constantly adding new functionality, and seeking improved algorithms.

The backbone of typical LAPACK subroutines is supported by multiple levels of the Basic Linear Algebra Subprograms, BLAS, which implements the basic linear algebra operations such as matrix multiplication, rank-k matrix updates, and solving triangular systems [6]. Those subprograms are optimized for various types of machines in order to enable LAPACK to achieve high efficiency and portability.

This thesis consists of three major topics, which each deals with a matrix decomposition that is intended to add to the LAPACK family.

The first topic is computing the cosine-sine decomposition (CSD) of a partitioned matrix with orthonormal columns. This decomposition is motivated by the questions on how to simplify the representation of an orthogonal projector and to measure the

separation between subspaces [3]. An explicit CSD form was first provided by Stewart [27]. The CSD is used to analyze problems related to the Riccati equation, subset selection algorithm and signal processing [33, 34]. Recently, it is applied as a key technique in compilers for quantum computers [31]. The results of this decomposition also shed light on the computation of the quotient singular value decomposition.

The second topic is concerned with the quotient singular value decomposition (QSVD). This decomposition was originally introduced by Von Loan [36]. It is later improved by Paige and Saunders who suggested a formulation which is more amenable for numerical computation [21]. The QSVD is one of the essential matrix decompositions; it rises in many applications such as signal processing, constrained least square, discriminant analysis, and DNA microarrays analysis [1, 3, 22, 23, 25, 37]. Currently, LAPACK provides a solution to this decomposition using the Jacobi-like iterative approach [20, 4]. It requires only a linear work space but may suffer from a slow rate of convergence. We study extensively an alternative method that is based on the computation of the CSD. This method ought to require a quadratic work space, but with the accommodation of extra memory, we are able to show that, not only is this alternative method stable, but it also achieves significant speed-up depending on the size of the input matrices.

The third topic deals with computing the product singular value decomposition (PSVD). Although both QSVD and PSVD are extended from the singular value decomposition for a single matrix, their origins, applications, and numerical computations are very different from each other. The notion of PSVD was first proposed by Fernando and Hammarling [11]; more extensive study on this decomposition is pursued by De Moor [8]. Some of the applications related to the PSVD include solving the orthogonal Procrustes problem, balancing of state space models, and computing the Kalman decomposition [9]. Many researchers proposed ways of approaching the PSVD problem recently [13, 16, 24, 26]; from them, we select to focus the study on a QR-like method. This method first implicitly bidiagonalizes the product of a pair of input matrices; it then computes the SVD of the bidiagonal matrix. Numerical

experiments demonstrate that this method gives stable results and reasonable speed.

This thesis has two main objectives: (1) to research and generalize the existing algorithms for these three types of decompositions, and (2) to develop a set of LAPACK-style software from those algorithms using real data in single precision.

Throughout this thesis we shall adopt the notational convention used in common linear algebra literature [14, 29]. Specifically, matrices are denoted by upper case italic letters, vectors by lower case italic letters, and scalars by lower case Greek letters or lower case italic whenever there is no confusion. The symbol $\mathbb{R}$, $\mathbb{R}^n$ and $\mathbb{R}^{m\times n}$ denote the set of real numbers in dimension 1, n, and $m \times n$ respectively. The matrix A^T is the transpose of A. $\|A\|_p$ denotes the p–norm of the matrix A. The rank of A is denoted by $\text{rank}(A)$. The condition number of A is denoted by $\kappa(A)$. Furthermore, we reserve the letter I and O for the identity and zero matrices, respectively. Also, we use MATLAB notations whenever necessary. For example, $A(i{:}j,p{:}q)$ denotes the submatrix of A taken from its rows i to j and columns p to q.

The rest of this thesis is organized as follows. Chapter 2 provides the background for understanding problems related to the three types of generalized singular value decompositions and specifies our computing environment. Chapters 3, 4, and 5 are devoted to describing the details of the algorithms and software for computing the CSD, QSVD, and PSVD, respectively. The layouts of these chapters are rather similar. Each of these chapters is started out by establishing the formal definition and basic properties of the decomposition followed by presenting the numerical algorithms that relate to its computation. Next, some software implementation issues are addressed. Toward the end of each chapter, the stability and speed of the software at focus are assessed by a wide range of numerical experiments. Finally, Chapter 6 wraps up the thesis with a conclusion.

Chapter 2

Background

We shall review some of the most fundamental matrix decompositions that are relevant to this thesis. These are the QR, QL, RQ and the singular value decompositions.

Definition 2.0.1. Let $A \in \mathbb{R}^{m\times n}$ be a general matrix. A *QR decomposition* of A is the factorization

$$A = Q \begin{bmatrix} R \\ O \end{bmatrix} \tag{2.1}$$

such that Q is $m \times m$ and orthogonal and the zero matrix has size $(m-r)\times n$ where $r = \min(m, n)$. If $m < n$, R is $m \times n$ and upper trapezoidal. If $m \geq n$, R is $n \times n$ and upper triangular; in this case, (2.1) is sometimes known as the *full* QR decomposition of A while its *compact* form is defined by

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

where $\hat{Q}$ is the first n columns of Q in (2.1).

There are many sophisticated algorithms that are built using the QR decomposition, such as solving the least square problem and computing the SVD. LAPACK provides a subroutine, named SGEQRF, for computing the QR decomposition via Householder transformation. This subroutine returns a sequence of elementary reflectors that represents Q. If the explicit forming of Q is desired, one may call the

subroutine SORGQR. Pre- or post-multiplying Q with a given matrix can be done by calling SORMQR.

Definition 2.0.2. Let $A \in \mathbb{R}^{m\times n}$ be a general matrix. A *QL decomposition* of A is the factorization

$$A = Q \begin{bmatrix} O \\ L \end{bmatrix} \tag{2.2}$$

such that Q is $m \times m$ and orthogonal and the zero matrix has size $(m-r)\times n$ where $r = \min(m, n)$. If $m < n$, L is $m \times n$ and lower trapezoidal. If $m \geq n$, L is $n \times n$ and lower triangular; in this case, (2.2) is sometimes known as the *full* QL decomposition of A and its *compact* form is defined by

$$A = \hat{Q}\ L$$

where $\hat{Q}$ is the $(m-n+1)$-th to m-th columns of Q in (2.2).

The QL decomposition is not as popular as the QR decomposition, but it is vital in the development of the CSD algorithm as we will show in Chapter 3. Both the implementation of QR and QL decomposition are very similar except that the former annihilates the columns of A from left to right whereas the latter performs annihilation from right to left. In LAPACK, the subroutine named SGEQLF is responsible for performing the QL decomposition and it returns a sequence of elementary reflectors representing Q, which may be formed explicitly by calling SORGLQ. Pre- or post-multiplying Q with a given matrix can be done by calling SQRMQL.

Definition 2.0.3. Let $A \in \mathbb{R}^{m\times n}$ be a general matrix. A *RQ decomposition* of A is the factorization

$$A = RQ$$

such that R is $m \times n$ and upper triangular and Q is $n \times n$ and orthogonal.

Although not as common as the QR decomposition, the RQ decomposition rises in applications such as computing the generalized QR decomposition. The LAPACK

subroutine named SGERQF is used to compute this decomposition using Householder transformation. The subroutine SORGRQ forms the Q explicitly from a sequence of elementary reflectors. The subroutine SORMRQ is used to pre- or post-multiply Q with another matrix.

Definition 2.0.4. Let $A \in \mathbb{R}^{m\times n}$ be a general matrix. A *singular value decomposition* (SVD) of A is the factorization

$$A = U\Sigma V^T$$

such that U is $m \times m$ and orthogonal, V is $n \times n$ and orthogonal, and Σ is $m \times n$ and diagonal. The diagonal entries, σ_i, in Σ are the *singular values* of A; they are real, non-negative, and conventionally sorted non-increasingly. The column vectors in U and V are the *left* and *right singular vectors* of A respectively.

The SVD plays a crucial role in linear algebra. It provides fundamental information about a matrix. Some examples include $\sigma_{\max}(A)$ gives the 2-norm of A and the number of non-zero singular values of A is the rank of A. The SVD is also a basis for many modern scientific computing algorithms, such as solving the linear least square problems. In bioinformatics, SVD is a valuable tool to preform the gene expression analysis, similar to the principle component analysis [32].

Currently, LAPACK provides two subroutines for computing the SVD. One uses QR iterations and is named SGESVD. The other is based on the Cuppen's divide-and-conquer algorithm and is named SGESDD. The latter is faster than the former but requires more work space.

The analysis of matrix algorithms usually requires the use of matrix norms. Throughout this thesis, we use the 2–norm unless explicitly specified otherwise. All the stability criteria are measured in 1–norm because it is much easier to compute numerically and all the norms on $\mathbb{R}^{m\times n}$ are equivalent within constants. Specifically, the relation between $\|A\|_1$ and $\|A\|_2$ for $A \in \mathbb{R}^{m\times n}$ is:

$$\frac{1}{\sqrt{m}}\|A\|_1 \leq \|A\|_2 \leq \sqrt{n}\|A\|_1 \ .$$

Tests reported in this thesis are carried out on a 1.0GHz Intel Itanium2 processor with 2GB of RAM. The relative machine precision, denoted by ϵ, is around 10^{-7}. Unless otherwise stated, tests are compiled using Intel's Math Kernel Library (IMKL) 7.2.1 [17]. This library includes most of the common BLAS and LAPACK subroutines that are highly optimized for Intel processors [1].

[1] Source codes from NetLib is used for some auxiliary subroutines that are not included in the library.

Chapter 3

Cosine-Sine Decomposition

This chapter focuses on developing a stable and efficient software for computing the cosine-sine decomposition (CSD) and carrying its implementation in LAPACK style. Section 3.1 introduces the definition and some basic properties of the CSD. Our algorithm for its computation is described in Section 3.2. Section 3.3 explains the software implementation of the CSD in terms of existing LAPACK subroutines and various levels of BLAS. Section 3.4 presents the numerical experiments on stability testing and timing of the software.

3.1 Definition and Basic Properties

Definition 3.1.1. Let $Q \in \mathbb{R}^{(m+p)\times\ell}$ be a matrix with orthonormal columns and partitioned into Q_1 and Q_2 such that

$$Q = \begin{array}{c} \ell \\ \left[\begin{array}{c} Q_1 \\ Q_2 \end{array}\right] \begin{array}{l} m \\ p \end{array} \end{array},$$

where $m + p \geq \ell$. A *cosine-sine decomposition (CSD)* of Q_1 and Q_2 is the joint factorization

$$Q_1 = UCZ^T \quad \text{and} \quad Q_2 = VSZ^T\,, \tag{3.1}$$

where $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{p\times p}$ and $Z \in \mathbb{R}^{\ell\times\ell}$ are orthogonal matrices; $C \in \mathbb{R}^{m\times\ell}$ and $S \in \mathbb{R}^{p\times\ell}$ are diagonal matrices satisfying

$$C^TC + S^TS = I. \tag{3.2}$$

If we write $C^TC = \text{diag}\ (\alpha_1^2, \alpha_2^2, \cdots, \alpha_\ell^2)$ and $S^TS = \text{diag}\ (\beta_1^2, \beta_2^2, \cdots, \beta_\ell^2)$, then it immediately follows from (3.2) that

$$\alpha_i^2 + \beta_i^2 = 1 \ \text{ for } \ i = 1, 2, \ldots, \ell. \tag{3.3}$$

Clearly, (3.3) resembles the cosine and sine relation, hence it accounts for the name of the decomposition. Thus α_i and β_i are called the *cosine* and *sine values* of Q respectively. Those values are real and no greater than one. By convention, the cosine values are sorted non-increasingly whereas the sine values are sorted non-decreasingly.

Furthermore, C and S belong to one of the following structures:

1. if $m \geq \ell$ and $p \geq \ell$,

$$C = \begin{bmatrix} \Sigma_1 \\ O \end{bmatrix} \begin{matrix} \ell \\ m-\ell \end{matrix} \quad \text{and} \quad S = \begin{bmatrix} \Sigma_2 \\ O \end{bmatrix} \begin{matrix} \ell \\ p-\ell \end{matrix}; \tag{3.4}$$

2. if $m \geq \ell$ and $p < \ell$,

$$C = \begin{bmatrix} I & O \\ O & \Sigma_1 \\ O & O \end{bmatrix} \begin{matrix} \ell-p \\ p \\ m-\ell \end{matrix} \quad \text{and} \quad S = \begin{bmatrix} O & \Sigma_2 \end{bmatrix} \begin{matrix} p \end{matrix}; \tag{3.5}$$

$$\begin{matrix} \ell-p & p \end{matrix} \qquad\qquad \begin{matrix} \ell-p & p \end{matrix}$$

3. if $m \leq \ell$ and $p \geq \ell$,

$$C = \begin{array}{c} \begin{matrix} m & \ell-m \end{matrix} \\ \left[\begin{array}{cc} \Sigma_1 & O \end{array}\right] \end{array} \begin{matrix} \\ m \end{matrix} \quad \text{and} \quad S = \begin{array}{c} \begin{matrix} m & \ell-m \end{matrix} \\ \left[\begin{array}{cc} \Sigma_2 & O \\ O & I \\ O & O \end{array}\right] \end{array} \begin{matrix} \\ m \\ \ell-m \\ p-\ell \end{matrix} ; \tag{3.6}$$

4. if $m \leq \ell$ and $p < \ell$,

$$C = \begin{array}{c} \begin{matrix} \ell-p & t & \ell-m \end{matrix} \\ \left[\begin{array}{ccc} I & O & O \\ O & \Sigma_1 & O \end{array}\right] \end{array} \begin{matrix} \ell-p \\ t \end{matrix} \quad \text{and} \quad S = \begin{array}{c} \begin{matrix} \ell-p & t & \ell-m \end{matrix} \\ \left[\begin{array}{ccc} O & \Sigma_2 & O \\ O & O & I \end{array}\right] \end{array} \begin{matrix} t \\ \ell-m \end{matrix} , \tag{3.7}$$

where $t = m + p - \ell$.

There are some arbitrariness in the formation of C and S. Take the structure in (3.5) for example. Exchanging the first two rows of C, then switching the two columns of both C and S yields the forms described in [3]. Specifically, if we let

$$\Pi_1 = \begin{array}{c} \begin{matrix} \ell-p & p & m-p \end{matrix} \\ \left[\begin{array}{ccc} O & I & O \\ I & O & O \\ O & O & I \end{array}\right] \end{array} \begin{matrix} p \\ \ell-p \\ m-\ell \end{matrix} \quad \text{and} \quad \Pi_2 = \begin{array}{c} \begin{matrix} \ell-p & p \end{matrix} \\ \left[\begin{array}{cc} O & I \\ I & O \end{array}\right] \end{array} \begin{matrix} p \\ \ell-p \end{matrix} ,$$

then the CSD of Q_1 and Q_2 can be rewritten as

$$Q_1 = (U\Pi_1^{-1})(\Pi_1 C \Pi_2)(\Pi_2^{-1} Z^T) \quad \text{and} \quad Q_2 = V(S\Pi_2)\Pi_2^{-1} Z^T.$$

The structures of C and S then become:

$$C = \begin{array}{c} \begin{matrix} p & \ell-p \end{matrix} \\ \begin{bmatrix} \Sigma_1 & O \\ O & I \\ O & O \end{bmatrix} \end{array} \begin{array}{l} p \\ \ell-p \\ m-\ell \end{array} \quad \text{and} \quad S = \begin{array}{c} \begin{matrix} p & \ell-p \end{matrix} \\ \begin{bmatrix} \Sigma_2 & O \end{bmatrix} \end{array} \begin{array}{l} p \end{array} .$$

By the similar argument, one can also swap the ordering of the cosine and sine values by pre- and post- multiplying some permutation matrices. Therefore, the orthogonal components of the CSD, the structures of C and S, and the ordering of the diagonal entries in Σ_1 and Σ_2 are not uniquely determined. Meanwhile, the CSD in (3.1) also amounts to a pair of decoupled SVD's for Q_1 and Q_2, with singular values in C and S, respectively. Since the singular values of any matrix are uniquely determined [30], the cosine and sine values are therefore uniquely determined as well.

Example 1: Consider a 7–by–4 matrix orthogonal matrix that is partitioned into a 5–by–4 matrix Q_1 and a 2–by–4 matrix Q_2 as follows:

$$Q = \left[\begin{array}{c} Q_1 \\ \hline Q_2 \end{array}\right] = \left[\begin{array}{cccc} \frac{1}{\sqrt{7}} & 0 & 0 & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{7}} & \frac{-2}{\sqrt{10}} & \frac{-1}{2} & \frac{-1}{2\sqrt{3}} \\ \frac{1}{\sqrt{7}} & \frac{-1}{\sqrt{10}} & \frac{1}{4} & \frac{3}{4\sqrt{3}} \\ \frac{1}{\sqrt{7}} & 0 & \frac{3}{4} & \frac{-3}{4\sqrt{3}} \\ \frac{1}{\sqrt{7}} & 0 & 0 & 0 \\ \hline \frac{1}{\sqrt{7}} & \frac{1}{\sqrt{10}} & \frac{-1}{4} & \frac{-3}{4\sqrt{3}} \\ \frac{1}{\sqrt{7}} & \frac{2}{\sqrt{10}} & \frac{-1}{4} & \frac{1}{4\sqrt{3}} \end{array}\right].$$

The dimensions of Q_1 and Q_2 imply that the C and S in their CSD follow the structure in (3.5). With the computed cosine and sine values, they are:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.8886814 & 0 \\ 0 & 0 & 0 & 0.3019895 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$S = \begin{bmatrix} 0 & 0 & 0.4585252 & 0 \\ 0 & 0 & 0 & 0.9533111 \end{bmatrix}.$$

The orthogonal matrices, U, V, and Z are:

$$U = \begin{bmatrix} 0.1270003 & -0.4511065 & -0.5559637 & -0.3695505 & -0.5785418 \\ 0.6350006 & -0.3909590 & 0.6365165 & 0.0397774 & -0.1928473 \\ 0.0000001 & -0.6215246 & -0.3030671 & 0.6108169 & 0.3856946 \\ -0.7620008 & -0.4009837 & 0.4377696 & -0.0284440 & -0.2571298 \\ 0.0000000 & -0.3107624 & 0.0475357 & -0.6985299 & 0.6428244 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.7716442 & -0.6360544 \\ -0.6360544 & -0.7716442 \end{bmatrix},$$

$$Z = \begin{bmatrix} 0.0000001 & -0.8221998 & 0.1117674 & -0.5581179 \\ -0.4016097 & 0.4438074 & -0.3451517 & -0.7229210 \\ -0.8890008 & -0.2606394 & -0.0739271 & 0.3691602 \\ 0.2199707 & -0.2430833 & -0.9289311 & 0.1720764 \end{bmatrix}.$$

Example 2: Now consider a different partition on the same 7–by–4 orthogonal matrix Q so that Q_1 has size 3–by–4 and Q_2 has size 4–by–4. That is:

$$Q = \left[\begin{array}{c} Q_1 \\ \hline Q_2 \end{array}\right] = \left[\begin{array}{cccc} \frac{1}{\sqrt{7}} & 0 & 0 & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{7}} & \frac{-2}{\sqrt{10}} & \frac{-1}{2} & \frac{-1}{2\sqrt{3}} \\ \frac{1}{\sqrt{7}} & \frac{-1}{\sqrt{10}} & \frac{1}{4} & \frac{3}{4\sqrt{3}} \\ \hline \frac{1}{\sqrt{7}} & 0 & \frac{3}{4} & \frac{-3}{4\sqrt{3}} \\ \frac{1}{\sqrt{7}} & 0 & 0 & 0 \\ \frac{1}{\sqrt{7}} & \frac{1}{\sqrt{10}} & \frac{-1}{4} & \frac{-3}{4\sqrt{3}} \\ \frac{1}{\sqrt{7}} & \frac{2}{\sqrt{10}} & \frac{-1}{4} & \frac{1}{4\sqrt{3}} \end{array}\right].$$

The dimensions of Q_1 and Q_2 imply that the C and S in their CSD should have the structure in (3.6). With the computed cosine and sine values, they are:

$$C = \begin{bmatrix} 0.9646989 & 0 & 0 & 0 \\ 0 & 0.9118780 & 0 & 0 \\ 0 & 0 & 0.2882230 & 0 \end{bmatrix},$$

$$S = \begin{bmatrix} 0.2633552 & 0 & 0 & 0 \\ 0 & 0.4104612 & 0 & 0 \\ 0 & 0 & 0.9575632 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The orthogonal matrices U, V, and Z are:

$$U = \begin{bmatrix} -0.4255832 & -0.5658049 & 0.7062178 \\ -0.7287463 & 0.6769618 & 0.1032063 \\ -0.5364771 & -0.4707308 & -0.7004319 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.1851486 & -0.2114852 & -0.6767798 & -0.6804138 \\ -0.9507410 & -0.1372403 & 0.0564159 & -0.2721656 \\ 0.0636928 & 0.7712921 & 0.3236593 & -0.5443310 \\ 0.2403227 & -0.5844206 & 0.6588101 & -0.4082482 \end{bmatrix},$$

$$Z = \begin{bmatrix} -0.6624501 & -0.1490399 & 0.1429285 & -0.7200823 \\ 0.6536220 & -0.3062800 & 0.5420197 & -0.4303314 \\ 0.2386796 & -0.5002463 & -0.7865822 & -0.2721655 \\ -0.2774348 & -0.7960736 & 0.2589873 & 0.4714045 \end{bmatrix}.$$

In both examples above, one may easily verify that they satisfy (3.1), (3.2), and all requirements in the CSD definition with seven decimal digits, which is the number of digits displayed.

3.2 Numerical Algorithms

The CSD always exists and various constructive proofs of existence can be found in [3, 28, 14], which all shed lights on its numerical computation. Essentially, the proofs suggest that the CSD can be computed through the following three steps:

1. perform SVD on Q_2: $Q_2 = VSZ^T$
2. set $T = Q_1 Z$
3. diagonalize T.

The last step above, out of all three, is most tricky. Van Loan discovered that using the conventional methods, such as column normalization or even the QR decomposition failed to produce a numerical stable result due to the fact that a nearly orthogonal matrix with columns norms significantly smaller than one is ill-conditioned. The problem can be revealed in the following theorem:

Theorem 3.2.1. *Suppose X is a $m \times \ell$ matrix with rank(X) = min(m, ℓ), and*

$$X^T X = D^T D + E,$$

where $\|E\| = \mathcal{O}(\epsilon)$ *and* $D = diag\ (\ \|x_1\|, \|x_2\|, \ldots, \|x_\ell\|)$ *with* x_i *being the* i*-th column of* X*. Let*

$$X = QR \tag{3.8}$$

be the full QR *factorization of* X*, where* Q $(m \times m)$ *is orthogonal and* R $(m \times \ell)$ *is upper triangular when* $m \geq \ell$ *(otherwise upper trapezoidal). Let* X_i *be the first* i *columns of* X*. Then for all* i *and* j *where* $i < j$*, we have*

$$|\ R_{(i,j)}\ | \leq\ \ min\ \{\|x_j\|, \frac{\|E\|}{\sigma_{min}(X_i)}\}.$$

Proof. Let $G = X^T X$ and $R_{ii} = R_{(1:i,1:i)}$. Since $X = QR$, $G = X^T X = (QR)^T(QR) = R^T R$. It follows that

$$G_{(1:i,j)} = R_{ii}^T \begin{bmatrix} R_{(1,j)} \\ \vdots \\ R_{(i,j)} \end{bmatrix} = X_i^T x_j \quad \text{for } j = i+1, \cdots, \ell.$$

Thus,

$$|R_{(i,j)}| \leq \|R_{(1:i,j)}\| = \| \begin{bmatrix} R_{(1,j)} \\ \vdots \\ R_{(i,j)} \end{bmatrix} \| = \|R_{ii}^{-T} X_i^T x_j\| \leq \|R_{ii}^{-T}\| \|X_i^T x_j\|.$$

Furthermore, note that if we let Q_i be the first i columns of Q, then (3.8) can be re-written as

$$X = [\ X_i\ |\ \cdots\ ] = [\ Q_i\ |\ \cdots\ ] \begin{bmatrix} R_{ii} & * \\ O & * \end{bmatrix} = [\ Q_i R_{ii}\ |\ \cdots\ ]\ .$$

Hence $\|X_i\| = \|R_{ii}\|$. Then $\|R_{ii}^{-T}\| = \frac{1}{\sigma_{min}(R_{ii})} = \frac{1}{\sigma_{min}(X_i)}$. The theorem follows since $\|X_i^T x_j\| \leq \|E\|$ and $\|R_{(1:i,j)}\| = \|x_j\|$. □

Theorem 3.2.1 is similar to the one proven by Von Loan in [35] and it suggests that a well-conditioned matrix having nearly orthogonal columns can be safely diagonalize

by a QR decomposition. Thereafter, Van Loan proposed an algorithm for computing the CSD that exploits this property. The description of Von Loan's algorithm constrained the size of Q_1 such that $m \geq \ell$, and hence his theorem may neglect the case where $m < \ell$. We will generalize his idea to all possible combinations of m, p, and ℓ; that is, both Q_1 and Q_2 may be "tall", "wide", or "square". Therefore we establish Theorem 3.2.1 regardless the relation between m and ℓ.

A complete description of our CSD algorithm is branched into two cases: (1) $m \leq p$ and (2) $m > p$. The computed C and S output from the algorithm conform to the structures in (3.4)-(3.7). Both algorithms consist of eight major steps.

For the rest of the explanation, we let $q_1 = \min(m, \ell)$ and $q_2 = \min(p, \ell)$.

We now present the case where $m \leq p$.

Step 1. Compute the SVD of Q_2 and conventionally, the singular values are returned in non-increasing order. Namely, $Q_2 = VSZ^T =$

$$\underbrace{\left[\begin{array}{c|c|c|c|c|c} v_1 & \dots & v_{q_2} & v_{q_2+1} & \dots & v_p \end{array}\right]}_{V} \underbrace{\left[\begin{array}{ccc|c} \beta_\ell & & & \\ & \ddots & & O \\ & & \beta_{\ell-q_2+1} & \\ \hline & O & & \end{array}\right]}_{S} \begin{array}{c} \\ p-q_2 \end{array} \underbrace{\left[\begin{array}{c} z_1^T \\ \hline z_2^T \\ \hline \vdots \\ \hline z_\ell^T \end{array}\right]}_{Z^T},$$

(column block label above S: $\ell-q_2$)

where $V \in \mathbb{R}^{p \times p}, Z \in \mathbb{R}^{\ell \times \ell}$ and $1 \geq \beta_\ell \geq \cdots \geq \beta_{\ell-q_2+1} \geq 0$.

We re-order the singular values of Q_2. To preserve equivalence, we must also swap the columns of V and rows of Z such that $Q_2 =$

$$\underbrace{\left[\begin{array}{c|c|c|c|c|c} & & & & & \\ & & & & & \\ v_{q_2} & \cdots & v_1 & v_{q_2+1} & \cdots & v_p \end{array}\right]}_{V} \underbrace{\left[\begin{array}{c|c} & \beta_{\ell-q_2+1} \\ O & \ddots \\ & \beta_\ell \\ \hline & O \end{array}\right]}_{S} \begin{matrix} \ell-q_2 \\ \\ \\ \\ p-q_2 \end{matrix} \underbrace{\left[\begin{array}{c} z_\ell^T \\ \hline z_{\ell-1}^T \\ \hline \vdots \\ \hline z_1^T \end{array}\right]}_{Z^T}.$$

Since Q_2 has $\ell - q_2$ zero singular values, we can assert that $\beta_1 = \cdots = \beta_{\ell-q_2} = 0$ and correspondingly, $\alpha_1 = \cdots = \alpha_{\ell-q_2} = 1$.

Step 2. Determine the index r such that $0 \leq \beta_{\ell-q_2+1} \leq \cdots \leq \beta_r \leq 1/\sqrt{2} < \beta_{r+1} \leq \cdots \leq \beta_\ell \leq 1$. The decision for choosing $1/\sqrt{2}$ as the threshold value is justified in Section 3.4.

Step 3. Perform the matrix-matrix multiplication: $T = Q_1 Z$. Since we started out with $Q_1^T Q_1 + Q_2^T Q_2 = I$, it follows that $T^T T = I - S^T S = \text{diag}(1 - \beta_1^2, \cdots, 1 - \beta_2^2, 1 - \beta_\ell^2)$. Thus the norm of each columns of T are be sorted non-increasingly, opposite with the order of S.

Step 4. Compute the QR decomposition of T: $T = UR$, where

$$R = \begin{array}{c} \begin{matrix} \ell-q_2 & r-\ell+q_2 & q_1-r & \ell-q_1 \end{matrix} \\ \left[\begin{array}{ccc|c} R_{11} & R_{12} & R_{13} & R_{14} \\ O & R_{22} & R_{23} & R_{24} \\ O & O & R_{33} & R_{34} \\ \hline O & O & O & O \end{array}\right] \end{array} \begin{matrix} \ell-q_2 \\ r-\ell+q_2 \\ q_1-r \\ m-q_1 \end{matrix}.$$

The non-increasing ordering of the norm of columns in T implies that $\sigma_{min}(T_1) \geq \sigma_{min}(T_2) \geq \cdots \geq \sigma_{min}(T_r)$, where T_i denotes the first i columns of T. Using the $1/\sqrt{2}$ threshold with Theorem 3.2.1, we derive that

$$| R(i,j) | \leq \frac{\epsilon}{\sigma_{min}(T_i)} \leq \frac{\epsilon}{\sqrt{1-\beta_r^2}} < \sqrt{2}\epsilon \,, \qquad \text{for } i = 1, 2, \cdots, r.$$

This means that R_{12}, R_{13}, R_{14}, R_{23}, and R_{24} are effectively the zero matrices, while R_{11} and R_{22} are diagonal and the diagonal entries are the first r sine values of Q_2. In particular, $\text{diag}([\ R_{11}\ \ R_{22}\ ]) = \text{diag}(\sqrt{1-\beta_1^2}, \cdots, \sqrt{1-\beta_r^2}) = \text{diag}(\alpha_1, \cdots, \alpha_r)$. Since the first $(\ell - q_2)$ diagonal entries in S are zeros, those corresponding entries in R must be ones in order to accommodate (3.3). Therefore $R_{11} = \text{diag}(\ \alpha_1, \cdots, \alpha_{\ell-q_2})$ $= I$.

In exact arithmetics, $T^TT = R^TR$ is diagonal and R is upper triangular; this implies that R must be diagonal. Then $U^TQ_1Z = R$ can be regarded as the SVD of Q_1. Hence Q_1 has $(\ell - q_1)$ zero singular values. Namely, $\alpha_i = 0$ and correspondingly $\beta_i = 1$ for $i = q_1, \ldots, \ell - 1, \ell$.

To summarize, we can rewrite R as

$$R = \begin{array}{c} \begin{array}{cccc} \scriptstyle \ell-q_2 & \scriptstyle r-\ell+q_2 & \scriptstyle q_1-r & \scriptstyle \ell-q_1 \end{array} \\ \left[\begin{array}{cccc} I & \epsilon & \epsilon & \epsilon \\ O & R_{22} & \epsilon & \epsilon \\ O & O & R_{33} & R_{34} \\ \hline O & O & O & O \end{array}\right] \end{array} \begin{array}{l} \scriptstyle \ell-q_2 \\ \scriptstyle r-\ell+q_2 \\ \scriptstyle q_1-r \\ \scriptstyle m-q_1 \end{array}$$

where $R_{22} = \text{diag}\ (\alpha_{\ell-q_2+1}, \cdots, \alpha_r)$.

Meanwhile, note that $\| [\ R_{33}\ R_{34}\ ] \| \simeq \sqrt{1-\beta_{r+1}^2} < 1/\sqrt{2}$. This submatrix is not well-conditioned; hence we cannot neglect its upper-diagonal entries.

Step 5. Compute the SVD of $[\ R_{33}\ R_{34}\ ]$: Let $[\ R_{33}\ R_{34}\ ] = U_rC_rZ_r^T$, with non-zero singular values $\alpha_{r+1}, \cdots, \alpha_{q_1}$ laying on the main diagonal of C_r.

Let $\Sigma_1 = \mathrm{diag}(\alpha_{\ell-q_2+1}, \cdots, \alpha_{q_1})$ and $\Sigma_2 = \mathrm{diag}(\beta_{\ell-q_2+1}, \cdots, \beta_{q_1})$, then C and S resulted from the CSD of Q_1 and Q_2 computed so far can be written as

$$C = \begin{array}{c} \begin{matrix} \ell-q_2 & t & \ell-q_1 \end{matrix} \\ \left[\begin{array}{ccc} I & O & O \\ O & \Sigma_1 & O \\ \hline O & O & O \end{array}\right] \end{array} \begin{array}{l} \ell-q_2 \\ t \\ m-q_1 \end{array} \quad \text{and} \quad S = \begin{array}{c} \begin{matrix} \ell-q_2 & t & \ell-q_1 \end{matrix} \\ \left[\begin{array}{ccc} O & \Sigma_2 & O \\ O & O & I \\ \hline O & O & O \end{array}\right] \end{array} \begin{array}{l} t \\ \ell-q_1 \\ p-q_2 \end{array} \tag{3.9}$$

where $t = q_1 + q_2 - \ell$.

The dimensions of the matrices above may look cumbersome at first because it is generalized to allow any arbitrary sizes of Q_1 and Q_2. However, if one substitutes for the four cases: (1) $q_1 = q_2 = \ell$, (2) $q_1 = \ell$, $q_2 = p$, (3) $q_1 = m$, $q_2 = \ell$, and (4) $q_1 = m$, $q_2 = p$, it is relatively easy to verify (neglect the submatrices whose dimension are small than one) that those the structures in (3.9) boil down to the four cases in (3.4)-(3.7), respectively.

The remaining part of the algorithm is to form the orthogonal matrices U, V, and Z. Essentially, this is done by a few matrix multiplications plus one more QR decomposition.

Step 6. The final matrix U is formed by post-multiplying U_r to it. Note that this affects only the $(r+1)$ to q_1 columns in U:

$$U := U \begin{array}{c} \begin{matrix} r & q_1-r & m-q_1 \end{matrix} \\ \left[\begin{array}{ccc} I & O & O \\ O & U_r & O \\ O & O & I \end{array}\right] \end{array} \begin{array}{l} r \\ q_1-r \\ m-q_1 \end{array} .$$

Step 7. The final matrix Z is formed by post-multiplying Z_r to it. This only affects the last $\ell - r$ columns of Z:

$$Z := Z \begin{array}{c} \begin{matrix} r & \ell-r \end{matrix} \\ \begin{bmatrix} I & O \\ O & Z_r \end{bmatrix} \begin{matrix} r \\ \ell-r \end{matrix} \end{array} .$$

Step 8. The product $V^T Q_2 Z$ would no longer be diagonal; to restore the diagonality, we first set W as follows:

$$W = \underbrace{\begin{bmatrix} \beta_{r+1} & & \\ & \ddots & \\ & & \beta_{q_2} \end{bmatrix}}_{\tilde{S}} \underbrace{Z_r(1{:}q_2-r, 1{:}q_2-r)}_{\tilde{Z}_r} = \widetilde{S}\ \widetilde{Z}_r\ .$$

Fortunately, the smallest singular value of $\tilde{S}$ is at least $1/\sqrt{2}$, so W can be safely diagonalized by QR decomposition. Let the QR decomposition of W be

$$W = Q_w R_w.$$

Let $\tilde{Z}_r := Z_r(1{:}q_2-r, 1{:}q_2-r)$. Since $W = \tilde{S}\tilde{Z}_r = Q_w R_w$, and R_w is diagonal, so $Q_w R_w \tilde{Z}_r{}^T$ is the SVD of $\tilde{S}$. Therefore $R_w = \tilde{S}$. Lastly, the final U can be obtained by applying Q_w to its right. Let $n = \min(\mathrm{r},\ \ell - q_2)$. This transformation affects the middle $(q_2 - r)$ columns of V:

$$V := V \begin{array}{c} \begin{matrix} r-n & q_2-r & p-q_2+n \end{matrix} \\ \begin{bmatrix} I & O & O \\ O & Q_w & O \\ O & O & I \end{bmatrix} \begin{matrix} r-n \\ q_2-r \\ p-q_2+n \end{matrix} \end{array} .$$

The eight steps above constitute an algorithm for computing the CSD when $m \leq p$. Putting it together, this algorithm can be summarized in the following.

Algorithm 3.2.1. Computing CSD of a partitioned orthonormal matrix when $m \leq p$

1: Set $q_1 = \min(m, \ell)$, $q_2 = \min(p, \ell)$.

2: Compute the SVD of Q_2: $Q_2 = VSZ^T$,

denote the singular values of Q_2 as $\beta(\ell) \geq \cdots \geq \beta(\ell-q_2+1)$.

3: Arrange the values of S in non-decreasing order.

4: Swap V: $V(:,1{:}q_2) := V(:,q_2{:}-1{:}1)$.

5: Swap Z: $Z(:,1{:}\ell) := Z(:,\ell{:}-1{:}1)$.

6: Set $\alpha(1{:}\ell-q_2) = 1$, $\beta(1{:}\ell-q_2) = 0$.

7: Find r such that $\beta(1) \leq \cdots \leq \beta(r) \leq 1/\sqrt{2} < \beta(r+1) \leq \cdots \leq \beta(\ell)$.

8: Compute $T = Q_1 Z$.

9: Compute the QR decomposition of T: $T = UR$.

label $R_2 = R(\ell-q_2+1{:}r, \ell-q_2+1{:}r)$ and $R_3 = R(r+1{:}q_1, r+1{:}\ell)$.

10: Compute the SVD of R_3: $R_3 = U_r C_r Z_r^T$ where

C_r is diagonal containing entries $\alpha(r+1) \geq \alpha(r+2) \geq \cdots \geq \alpha(q_1)$.

11: Set $\alpha(q_1+1{:}\ell) = 0$, $\beta(q_1+1{:}\ell) = 1$.

12: Set $\alpha(\ell-q_2+1{:}r) = \mathrm{diag}(R_2)$.

13: Update U: $U(:,r+1{:}q_1) := U(:,r+1{:}q_1)U_r$.

14: Update Z: $Z(:,r+1{:}\ell) := Z(:,r+1{:}\ell)Z_r$.

15: Set $W = \tilde{S} Z_r(1{:}q_2-r, 1{:}q_2-r)$ where $\tilde{S} = \mathrm{diag}(\beta_{(r+1)}, \beta_{(r+2)}, ..., \beta_{(q_2)})$.

16: Compute the QR of W: $W = Q_w R_w$.

17: Update V: $V(:,1+r-n{:}q_2-n) := V(:,1+r-n{:}q_2-n)Q_w$ where $n = \min(r,\ \ell - q_2)$.

The algorithm when $m > p$ is rather symmetric, yet different, than Van Loan's description in [35] mainly due to the performing of an QL instead of QR factorization as a mean for safe diagonalization. The idea is illuminated by the following theorem:

Theorem 3.2.2. *Suppose X is a $m \times n$ matrix with rank(X) $= n$ and*

$$X^T X = D^T D + E,$$

where $\|E\| = \mathcal{O}(\epsilon)$ and $D = diag\ (\ \|x_1\|, \|x_2\|, \ldots, \|x_n\|)$ with x_i being the i-th column of X. Let

$$X = QL \tag{3.10}$$

be the compact QL decomposition of X, where Q $(m \times n)$ is orthogonal and L $(n \times n)$ is lower triangular. Let $X_i = [\ x_i \mid x_{i+1} \mid \ \ldots \ \mid x_n\ ]$. Then for all i and j where $i > j$, we have

$$\mid L_{(i,j)} \mid \leq \ min\ \{\|x_j\|, \frac{\|E\|}{\sigma_{min}(X_i)}\}.$$

Proof. Let $G = X^T X$ and $L_{ii} = L_{(i:n,i:n)}$. Since $X = QL$, $G = X^T X = (QL)^T(QL) = L^T L$. It follows that

$$G_{(i:n,j)} = L_{ii}^T \begin{bmatrix} L_{(i,j)} \\ \vdots \\ L_{(n,j)} \end{bmatrix} = X_i^T x_j \quad \text{for } j = 1, 2, \cdots, i-1.$$

Thus,

$$|L_{(i,j)}| \leq \|L_{(i:n,j)}\| = \| \begin{bmatrix} L_{(i,j)} \\ \vdots \\ L_{(n,j)} \end{bmatrix} \| = \|L_{ii}^{-T} X_i^T x_j\| \leq \|L_{ii}^{-T}\| \|X_i^T x_j\|.$$

Now if we let Q_i be $Q_{(1:m,i:n)}$, then (3.10) can be re-written as

$$X = [\ \cdots \mid X_i\ ] = [\ \cdots \mid Q_i\ ] \begin{bmatrix} * & O \\ * & L_{ii} \end{bmatrix} = [\ \cdots \mid Q_i L_{ii}\ ].$$

Hence $\|X_i\| = \|L_{ii}\|$. Then $\|L_{ii}^{-T}\| = \frac{1}{\sigma_{min}(L_{ii})} = \frac{1}{\sigma_{min}(X_i)}$. The theorem follows since $\|X_i^T x_j\| \leq \|E\|$ and $\|L_{(i:n,j)}\| = \|x_j\|$. □

We now set to specify the algorithm where $m > p$.

Step 1. Compute the full SVD of Q_1: $Q_1 = UCZ^T$, with singular values $1 \geq \alpha_1 \geq \cdots \geq \alpha_{q_1} \geq 0$. Since Q_1 has $\ell - q_1$ zero singular values, $\alpha_i = 0$ and $\beta_i = 1$ for $i = q_1 + 1, \ldots, \ell - 1, \ell$.

Step 2. Determine the index r such that $1 \geq \alpha_1 \geq \cdots \geq \alpha_r \geq 1/\sqrt{2} > \alpha_{r+1} \geq \cdots \geq \alpha_\ell \geq 0$.

Step 3. Perform the matrix-matrix multiplication: $T = Q_2 Z$. Since $T^T = I - C^T C$ $= \text{diag}(\, 1-\alpha_1^2, \ldots, 1-\alpha_\ell^2\,)$, the norm of columns of T is placed in non-decreasing order.

Step 4. Since the norm of the columns of T is sorted from small to large, $\sigma_{min}(t_1) = \sigma_{min}([\, t_1 \mid t_2\,]) = \cdots = \sigma_{min}([\, t_1 \mid \cdots \mid t_\ell\,])$ where t_i denotes the i-th column of T. If $T = QR$ were the QR decomposition of T, then Theorem 3.2.1 would lead to

$$\begin{aligned} \mid R_{(i,j)} \mid &\leq min\{\|t_j\|, \frac{\|\epsilon\|}{\sigma_{min}(t_1)}\} \\ &= min\{\|t_j\|, \frac{\|\epsilon\|}{\sigma_{min}(Q_2)}\}\,, \qquad \text{for } i < j\,. \end{aligned}$$

Hence that QR decomposition would fail to diagonalize any rows of T as long as Q_2 has a tiny singular value. This problem can be rectified by preforming the QL decomposition instead.

Let the QL decomposition of T be $T = VL$. Applying the $1/\sqrt{2}$ bound with Theorem 3.2.2 leads to

$$\sigma_{min}([\, t_1 \mid t_2 \mid \cdots \mid t_\ell\,]) \leq \sigma_{min}([\, t_2 \mid \cdots \mid t_\ell\,]) \leq \cdots \leq \sigma_{min}([\, t_\ell\,]),$$

and

$$\mid L_{(i,j)} \mid \leq \frac{\epsilon}{\sigma_{min}([\, t_i \mid t_{i+1} \mid \ldots \mid t_\ell\,])} \leq \frac{\epsilon}{\sqrt{1-\alpha_r^2}} < \sqrt{2}\epsilon, \quad \text{for } i = p - q_2 + r + 1, \ldots, p.$$

Therefore, we have L in the form:

$$L = \left[\begin{array}{c|ccc} O & O & O & O \\ \hline L_{11} & L_{12} & O & O \\ \epsilon & \epsilon & L_{23} & O \\ \epsilon & \epsilon & \epsilon & I \end{array}\right] \begin{array}{l} p-q_2 \\ r \\ t \\ \ell-q_1 \end{array}$$

$$\begin{array}{cccc} \ell-q_2 & r & t & \ell-q_1 \end{array}$$

where $L_{23} = \mathrm{diag}(\beta_{r+1+\ell-q_2}, \cdots, \beta_{q_1})$ and $t = q_1 + q_2 - \ell - r$.

In order to conform with the structures of S in (3.4) and (3.6), we pre-multiply L with the following permutation matrix so that the first $p - q_2$ rows of zero in L are brought to the bottom of the matrix.

$$\Pi = \begin{array}{c} \begin{array}{ccc} p-q_2 & r & q_2-r \end{array} \\ \left[\begin{array}{ccc} O & I & O \\ O & O & I \\ I & O & O \end{array}\right] \begin{array}{l} r \\ q_2-r \\ p-q_2 \end{array} \end{array} ,$$

Step 5. Compute the SVD of $[\ L_{11}\ L_{12}\ ]$: Let $[\ L_{11}\ L_{12}\ ] = V_\ell S_\ell Z_\ell^T$ where S_ℓ contains the singular values $\beta_{\ell-q_2+1} \geq \cdots \geq \beta_{\ell-q_2+r}$. Swap the order of the singular values. Accordingly, swap the left and right singular vectors.

Since Q_2 has $\ell - q_2$ zero singular values, $\beta_i = 0$ and $\alpha_i = 1$, for $i = 1, 2, \ldots, \ell - q_2$.

Step 6. The final matrix V is formed by combining it with $V_\ell \Pi^{-1}$:

$$\begin{aligned} V &:= V \left[\begin{array}{ccc} I & O & O \\ O & V_\ell & O \\ O & O & I \end{array}\right] \underbrace{\left[\begin{array}{ccc} O & O & I \\ I & O & O \\ O & I & O \end{array}\right]}_{\Pi^{-1}} \\ &= V \begin{array}{c} \left[\begin{array}{ccc} O & O & I \\ V_\ell & O & O \\ O & I & O \end{array}\right] \begin{array}{l} p-q_2 \\ r \\ q_2-r \end{array} \end{array} . \end{aligned}$$

We multiply Π^{-1} from the right because Π was pre-multiply to L in *Step 4*.

Step 7. Formulate the final matrix Z:

$$Z := Z \begin{bmatrix} Z_\ell & O \\ O & I \end{bmatrix} \begin{matrix} r+\ell-q_2 \\ q_2-r \end{matrix} .$$

Step 8. Set $W = \mathrm{diag}(\alpha_1, \ldots, \alpha_{r+\ell-q_2}) Z_\ell$. Let the QR decomposition of W be $W = Q_w R_w$. Then the final U is formed by setting

$$U := U \begin{bmatrix} Q_w & O \\ I & O \end{bmatrix} \begin{matrix} r+\ell-q_2 \\ m-r-\ell+q_2 \end{matrix} .$$

Putting those eight steps together, algorithm for computing the CSD when $m > p$ can be summarized in the following.

Algorithm 3.2.2. Computing CSD of a partitioned orthonormal matrix when $m > p$

1: Set $q_1 = \min(m, \ell)$, $q_2 = \min(p, \ell)$.

2: Compute the SVD of Q_1: $Q_1 = UCZ^T$,

let the singular values of Q_1 be α_i such that $\alpha(1) \geq \cdots \geq \alpha(q_1)$.

3: Set $\alpha_{(q_1+1:\ell)} = 0$, $\beta_{(q_1+1:\ell)} = 1$.

4: Find r such that $1 \geq \alpha_{(1)} \geq \cdots \geq \alpha_{(r)} \geq 1/\sqrt{2} > \alpha_{(r+1)} \geq \cdots \geq \alpha_{(\ell)} \geq 0$.

5: Compute $T = Q_2 Z$.

6: Compute the QL decomposition of T: $T = VL$.

label $L_1 = L_{(p-q_2+1:p-q_2+r,1:\ell-q_2+r)}$ and $L_2 = L_{(p-q_2+r+1:q_1-\ell+p,r+1+\ell-q_2:q_1)}$.

7: Compute the SVD of L_1: $V_\ell^T L_1 Z_\ell = S_\ell$,

8: Arrange the singular values so that

S_ℓ is diagonal and containing entries $\beta_{(\ell-q_2+1)} \leq \cdots \leq \beta_{(r+\ell-q_2)}$.

9: Swap V_ℓ: $V_{\ell(:,1:r)} := V_{\ell(:,r:-1:1)}$.

10: Swap Z_ℓ: $Z_\ell(:,1{:}r+\ell-q_2) := Z_\ell(:,r+\ell-q_2{:}-1{:}1)$.

11: Set $\alpha(1{:}\ell-q_2) = 1$, $\beta(1{:}\ell-q_2) = 0$. $\beta(r+\ell-q_2+1{:}q_1) = \mathrm{diag}(L_2)$.

12: Set $\widetilde{V}(1{:}p\,,\,1{:}p) = 0$ except

$\widetilde{V}(1{:}p-q_2\,,\,\ell+1{:}p) = I$, $\widetilde{V}(p-q_2+1{:}p-q_2+r\,,\,1{:}r) = V_\ell$, and $\widetilde{V}(p-q_2+r+1{:}p\,,\,r+1{:}q_2) = I$.

13: Update V: $V := V\widetilde{V}$.

14: Update Z: $Z(1{:}\ell,1{:}r+\ell-q_2) := Z(1{:}\ell,1{:}r+\ell-q_2)Z_\ell$.

15: Set $W = \tilde{S}Z_\ell$ where $\tilde{S} = \mathrm{diag}(\alpha_{(1)}, \alpha_{(2)}, \cdots, \alpha_{(r+\ell-q_2)})$.

16: Compute the QR of W: $W = Q_w R_w$.

17: Update U: $U(1{:}m,1{:}r+\ell-q_2) := U(1{:}m,1{:}r+\ell-q_2)Q_w$.

3.3 LAPACK-Style Software

The CSD algorithm described in the previous section has been implemented in FORTRAN 77 using real data, supported by subroutines from LAPACK and BLAS. Algorithms 3.2.1 and 3.2.2 are put to subroutines named SORCS2 and SORCS1, respectively. Above these two, there is a subroutine named SORCSD which determines which algorithm should be used by checking the size of the input matrix. This section explains each step of the implementation of SORCS2 and herein points out the efficient memory management that optimizes the storage requirement. Similar storage scheme applies to SORCS1 and it is hence omitted.

3.3.1 Implementation Details

We shall use calligraphic fonts to denote the memory spaces. Suppose that caller has allocated the following memory spaces: $\mathcal{U} \in \mathbb{R}^{m\times m}$, $\mathcal{V} \in \mathbb{R}^{p\times p}$, and $\mathcal{Z} \in \mathbb{R}^{\ell\times\ell}$, and two length-$\ell$ vectors $\mathcal{C}$ and $\mathcal{S}$. The spaces $\mathcal{A} \in \mathbb{R}^{m\times\ell}$ and $\mathcal{B} \in \mathbb{R}^{m\times\ell}$ store the input matrices Q_1 and Q_2 at the time SORCS2 is called, but their contents are destroyed upon exit. In addition, $\mathcal{W}$ is a vector of work space. We determine its size toward the end of the section. The steps and notations of the computed values in the fol-

lowing presentation adhere to the description of the algorithm in the previous section.

Step 1. There are two LAPACK subroutines for computing the SVD, SGESVD and SGESDD. This algorithm chooses to use the former mainly for space saving concern and flexibility in placing the resulted matrices. Upon computing the SVD of Q_2, the results V and Z^T are stored in $\mathcal{V}$ and $\mathcal{Z}$, respectively. The first $\ell - q_2$ elements in $\mathcal{S}$ is zero. The singular value of Q_2 are stored in $\mathcal{S}(\ell-q_2+1:\ell)$. Note that SGESVD returns Z^T rather than Z so that we need to swap the rows in $\mathcal{Z}$ and the columns in $\mathcal{V}$. This is done by successively calling the level-2 BLAS subroutine, SSWAP, to swap two rows or columns at a time. Reversing the elements in $\mathcal{S}$ is trivial.

Step 2. This step does not involve any BLAS or LAPACK subroutine calls. We simply loop through each elements in $\mathcal{S}$ and find the gap where the two singular values are separated by $1\sqrt{2}$ and save this index, r.

Step 3. The multiplication of $Q_1 Z$ is done by invoking the level-3 BLAS subroutine, SGEMM. Since Z^T is stored in $\mathcal{Z}$, we actually need to do $Q_1(Z^T)^T$. The result of the multiplication can overwrite $\mathcal{B}$ since Q_2 is no longer needed and its size is at least as large as Q_1 in $\mathcal{A}$. This highlights another advantage in computing the initial SVD on a larger matrix because otherwise we may need an extra $m \times \ell$ matrix to store this temporary result.

Step 4. As we use the LAPACK subroutine, SGEQRF, to compute the QR decomposition of T (stored in $\mathcal{B}$), the upper triangular part of $\mathcal{B}$ is overwritten by R while its lower diagonal is overwritten by a sequence of Householder reflectors that implicitly represent U. In addition, we store the scalar factors of the elementary reflector, τ, in the first q_1 elements of $\mathcal{W}$.

Since the first r diagonal entries of R are actually the cosine values, they should be non-negative, but QR decomposition via Householder transformation may or may not

yield negative diagonal entries in R since reflector with either signs may be chosen due to stability issue. Mathematically, we may always find a diagonal matrix Π (contains only 1 or -1) such that ΠR is non-negative on the diagonal, thus the decomposition becomes $U\Pi^{-1}\Pi R = U\Pi\,\Pi R$. Practically, ΠR is translated into changing the sign of the rows of R whose diagonal entries are negative. However, we cannot do $U\Pi^{-1}$ since U is stored implicitly in reflector form. Hence we need a vector of length q_1 to save the diagonal elements in Π. This vector is stored in $\mathcal{W}(q_1+1:q_1+q_1)$. Meanwhile, we can assign $\mathcal{C}(1:r)$ to be the diagonal entries of R stored in $\mathcal{B}$, and $\mathcal{C}(q_1+1:\ell) = 0$.

Step 5. The SVD of [R_{33} R_{34}] (stored in part of $\mathcal{B}$) is computed via a call to SGESVD. However, the SVD subroutine does not assume any special structure of its input matrix and would mess up the lower triangular contents in $\mathcal{B}$. To circumvent this, we copy the contents of [R_{33} R_{34}] to the upper triangular part in $\mathcal{A}(1:q_1-r,1:\ell-r)$ and zero out the lower triangular part of $\mathcal{A}(1:q_1-r,1:\ell-r)$. Meanwhile, $\mathcal{U}$ is initialized to the identity matrix because part of it will be used to store the result of the SVD.

Upon computing the SVD of [R_{33} R_{34}], the result U_r and Z_r^T are stored in $\mathcal{U}(r+1:q_1,r+1:q_1)$ and $\mathcal{W}$, respectively. The singular values are stored in $\mathcal{C}(r+1:q_1)$.

One may stop the computation here if U, V, and Z are not desired. The cosine and sine values are computed and have been placed in $\mathcal{C}$ and $\mathcal{S}$, respectively.

Step 6. Mathematically, this step computes $(U\Pi)U_r$. But since U and Π are not stored explicitly and matrix multiplication is associative, we compute ΠU_r first by swapping the sign of the rows in U_r for which the corresponding elements in Π (stored in $\mathcal{W}$) is negative.

This step is completed by multiplying the Householder reflectors stored in $\mathcal{B}$ and τ with contents in $\mathcal{U}$. This is done by calling the LAPACK subroutine, SORMQR. After this step, we will no longer need the Householder reflectors and the scalar factor τ, thus freeing up all spaces in $\mathcal{B}$ and $\mathcal{W}$.

Step 7. This is the second matrix-matrix multiplication done by calling SGEMM in level-3 BLAS. Space in $\mathcal{B}$ is used to store the result. Since both the transposes of Z and Z_r are returned in part of $\mathcal{W}$ and $\mathcal{Z}$ respectively, the operation actually computes $(Z_r)^T Z(_{1:\ell-r,:})^T$. To complete the formation of Z, we must move the result of the operation in $\mathcal{B}$ to $\mathcal{Z}$. Note that $\mathcal{Z}$ stores Z^T, not Z.

Step 8. Recall that to formulate the final V, we first need to compute $W = \tilde{S}\tilde{Z}_r$ with $\tilde{S}$ stored in $\mathcal{S}$ and $\tilde{Z}_r^T$ stored in $\mathcal{W}$. This can be accommodated by the equivalence that $\tilde{S}\tilde{Z}_r = (\tilde{Z}_r^T \tilde{S}^T)^T = (\tilde{Z}_r^T \tilde{S})^T$. Moreover, multiplying a diagonal matrix to the right of a dense matrix is equivalent to scaling the columns of the dense matrix. Thus the formation of W is done by successively calling the level-2 BLAS subroutine, SSCAL, to scale each column of Z_r^T by the corresponding element in $\mathcal{S}$, followed by placing the transposed of the post-scaled matrix in leading part of $\mathcal{B}$.

Computing the QR of W can be done by calling the LAPACK subroutine, SGEQRF. Here the $q_2 - r$ scalar factors are stored in $\mathcal{W}$.

Theoretically, the diagonal elements in the resulted upper triangular matrix should be $\text{diag}(s_{r+1}, ..., s_{q_2})$, so their diagonal entries are non-negative. Again, QR decomposition via Householder transformation does not guarantee this, so we seek the matrix Π_2 that ensure the diagonal of $\Pi_2 R$ be positive. The operation of $\Pi_2 R$ is implemented by altering the sign of the rows in R that contain negative diagonal elements. The sign representation of Π_2 is stored in $\mathcal{W}$ since we cannot yet apply the equivalent negation in Q_w.

To multiply the selected columns of V with the implicit Q_w, SORMQR is invoked once more. Note that only $V(_{:,\ 1+r-n:q_1-n})$ where n=min$(r, \ell - q_2)$ is affected by the Householder transformation. Result overwrites the same columns in $\mathcal{V}$.

Lastly, we still need to alter the sign of the columns in $\mathcal{V}$ according to Π_2 that is stored in $\mathcal{W}$.

As the program exits, the orthogonal matrices U, V, Z^T and the cosine and sine values are stored in $\mathcal{U}$, $\mathcal{V}$, $\mathcal{Z}$, $\mathcal{C}$, and $\mathcal{S}$, respectively.

3.3.2 Work Space Requirement

Among all the subroutines invoked in computing the CSD, the most expensive use of work space amounts to SGESVD. For a general input matrix of size $\hat{m}$-by-$\hat{n}$, SGESVD requires the minimum size of work space, $\mathcal{L}$, to be

$$\mathcal{L} = \max\{3 \times \min(\hat{m}, \hat{n}) + \max(\hat{m}, \hat{n}), 5 \times \min(\hat{m}, \hat{n})\}.$$

In SORCS2, the input matrices are $Q_1 \in \mathbb{R}^{m \times \ell}$ and $Q_2 \in \mathbb{R}^{p \times \ell}$. Let $q_1 = \min(m, \ell)$ and $q_2 = \min(p, \ell)$. In the case where $p \geq m$, the first SVD is computed on Q_2 and the minimum amount of space required is $\mathcal{L}_1$ where

$$\begin{aligned} \mathcal{L}_1 &= \max\{3 \times min(p, \ell) + \max(p, \ell), 5 \times \min(p, \ell)\} \\ &= \max\{3 \times q_2 + \max(p, \ell), 5 \times q_2\}. \end{aligned}$$

The second SVD is computed on a matrix of size $(q_1 - r)$-by-$(\ell - r)$ and the size of work space required is $\mathcal{L}_2$ where

$$\begin{aligned} \mathcal{L}_2 &= \max\{3 \times \min(q_1 - r, \ell - r) + \max(q_1 - r, \ell - r), 5 \times \min(q_1 - r, \ell - r)\} \\ &\leq \max(3 \times q_1 + \ell, 5 \times q_1). \end{aligned}$$

Since SORCS2 has $p \geq m$, $q_2 \geq q_1$. Hence re-using the work space for the first SVD is sufficient to compute the second SVD. However, the QR decomposition in *Step 4*, which takes place between the two SVD calls, took away the $q_1 + q_1$ in the work space vector $\mathcal{W}$ to store τ and Π. This requires an extra space of $q_1 + q_1$. Furthermore, the result of the right singular vectors in the second SVD are stored in $\mathcal{W}$, so that it requires extra ℓ-by-ℓ space. The rest of the subroutine may re-use the work space to complete the computation. Hence the size of work space required by SORCS2 is at least $\mathcal{L}_3$ such that

$$\mathcal{L}_3 = \max\{3 \times q_2 + \max(p, \ell), 5 \times q_2\} + q_1 + q_1 + \ell^2.$$

To generalize for both cases where $m \leq p$ and $m > p$, we let $\gamma = \min\{\max(m,p),\ell\}$, $\zeta = \min(m,p,\ell)$, and $\varphi = \max(m,p,\ell)$, then the size of work space $\mathcal{W}$ for SORCSD is thus at least $\mathcal{L}_4$ where

$$\mathcal{L}_4 \quad = \quad \max(3\gamma+\varphi, 5\gamma) + 2\zeta + \ell^2.$$

Simply put it, SORCSD requires work space no more than $7 \times \max(m,p,\ell) + \ell^2$.

It should be emphasized that the work space requirement for our SORCSD is dominated by the amount used to perform the first and second SVD, in addition to the ℓ^2 space used to store the right singular vectors. The SVD algorithm we choose, SGESVD, essentially uses QR iterations to compute the singular vectors. If we picked the alternative method, SGESDD, which based on the divide-and-conquer algorithm, at least one quadratic term must be added to the size of total work space.

3.4 Testing and Timing

As described in previous sections, our CSD method works on all combinations of input size m, p, and ℓ as long as $m+p \geq \ell$. This section, we design several tests that evaluate the stability and timing performance on both SORCS1 and SORCS2. We now present the details of those tests.

3.4.1 Notion of Stability

Before we proceed, it is constructive to first define the notion of stability for a CSD algorithm. Suppose $\widetilde{U}$, $\widetilde{V}$,$\widetilde{Z}$, $\widetilde{C}$, and $\widetilde{S}$ are the computed values of U, V, Z, C, and S respectively, as they are in (3.1). Assume that the input Q_1 and Q_2 form a matrix with orthonormal columns within ϵ, that is:

$$\|Q_1^T Q_1 + Q_2^T Q_2 - I_\ell\| \simeq \epsilon.$$

Then the following expressions measure the stability of the algorithm for computing the CSD :

$$res_{Q_1} = \frac{\|\widetilde{U}\widetilde{C}\widetilde{Z}^T - Q_1\|_1}{\max(m,\ell)\|Q_1\|_1\,\epsilon}\,, \qquad res_{Q_2} = \frac{\|\widetilde{V}\widetilde{S}\widetilde{Z}^T - Q_2\|_1}{\max(p,\ell)\|Q_2\|_1\,\epsilon}\,, \tag{3.11}$$

$$orthogU = \frac{\|\widetilde{U}^T\widetilde{U} - I\|_1}{m\,\epsilon}\,, \qquad orthogV = \frac{\|\widetilde{V}^T\widetilde{V} - I\|_1}{p\,\epsilon}\,, \tag{3.12}$$

$$orthogZ = \frac{\|\widetilde{Z}^T\widetilde{Z} - I\|_1}{\ell\,\epsilon}\,. \tag{3.13}$$

First two expressions in (3.11) measure the residual errors of the CSD, while (3.12) and (3.13) measure the orthogonality of the computed orthogonal matrices. The algorithm for computing the CSD is *stable* if all of those conditions are bounded by $\mathcal{O}(1)$.

Recall that *Step 4* of our CSD algorithm chooses the ad-hoc value $1/\sqrt{2}$ as the threshold value. We now justify this choice. Since we have $Q_1^TQ_1 + Q_2^TQ = I$, $\|Q_1^TQ_1 + Q_2^TQ_2\| = 1$. Hence we know that the singular values of Q_1 and Q_2 are between 1 and 0; $1/\sqrt{2}$ is the middle point since $(1/\sqrt{2})^2 + (1/\sqrt{2})^2 = 1$. Because the size of this threshold affects the measure of backward error bounds in (3.11)-(3.13) and the amount of subsequent work, Von Loan in [35] suggested choosing $1/\sqrt{2}$ so that it balances between the backward error bound and speed performance. Any further theoretical or empirical outcomes on the variation of this threshold value has not yet been discovered.

3.4.2 Testing Routines

The driver that tests all the real LAPACK subroutines for the matrix eigenvalue-type problem is SCHKEE. To test SORCSD, we have modified SCHKEE into SCHKEE2 by adding a call the main CSD testing subroutine, SCKCSD. This subroutine consists of a single loop over the triples (m, p, ℓ). On each iteration of the loop, a random orthogonal $(m+p)$-by-ℓ matrix, Q, is generated. It follows that the first m rows of Q is copied to Q_1 and the last p rows of Q is copied to Q_2. SCKCSD then calls SCSDTS

to compute the quantities of the stability criteria. The sequence of the subroutines in the testing procedure is summarized below.

3.4.3 Input File Format

An annotated example of an input file for testing of the CSD subroutine is shown below:

```
CSD: Data file for testing CSD subroutine
12                                      Number of values of M, P, N
20 41 36  50 67 34  28 32 41  17 28 37 Values of M (row dimension)
20 23 47  30 46 31  39 50 63  17 42 31 Values of P (row dimension)
20 16 22  40 67 32  39 47 52  34 47 52 Values of N (column dimension)
1.5                                     Threshold value of test ratio
T                                       Put T to test if error exits
2                                       Code to interpret the seed
3 27 61 59
CSD 8
```

The first line of the input files must contain the character CSD in columns 1-3.

Lines 2-9 are read using list-directed input and specify the following values:

line 2: The number of values m, p, and ℓ

line 3: Values of m (row dimension)

line 4: Values of p (row dimension)

line 5: Values of ℓ (column dimension)

line 6: The threshold value for the test ratios.

We may set this one to 0 in order to display all the test ratios.

line 7: Flag to test if error exists

line 8: An integer code to interpret the random number seed.

= 0: Set the seed to a default value before each run.

= 1: Initialize the seed to a default value only before the first run.

= 2: Like 1, but use the seed values on the next line.

line 9: If line 8 was 2, four integer values for the random seed.

Otherwise, the path CSD followed by the number of matrix types.

Note: CSD only deals with orthogonal matrices, so this parameter is irrelevant. An integer is placed here to maintain consistency with format of all other testing files.

3.4.4 Stability Testing

Each of the CSD illustrated in Example 1 and 2 toward the end of Section 3.1 is actually computed by calling SORCSD. The orthogonal matrix U, V, Z^T, and the cosine and sine values are returned. In Example 1, plugging the computed results into the stability expressions in (3.11)-(3.13) yields:

$$\begin{aligned} res_{Q_1} &= 0.5273\,, \qquad res_{Q_2} = 1.0760\,, \\ orthog_U &= 1.3944\,, \quad orthog_V = 0.1442\,, \\ orthog_Z &= 0.9635\,. \end{aligned}$$

Likewise, calculating the stability of the computation in Example 2 yields:

$$\begin{aligned} res_{Q_1} &= 0.7175\,, \qquad res_{Q_2} = 0.6682\,, \\ orthog_U &= 0.7340\,, \quad orthog_V = 0.9675\,, \\ orthog_Z &= 1.1539\,. \end{aligned}$$

Now we follow the standard procedure as explained in Section 3.4.2 and use the input file described in Section 3.4.3 to perform a sequence of tests on samples with larger dimensions in all possible combinations. Table 3.1 displays the result of the tests. All of these quantities are around or less than one, indicating that the subroutine, SORCSD, results in exact decomposition of a pair of slightly perturbed input matrices.

m	p	ℓ	res_{Q1}	res_{Q2}	$orth_U$	$orth_V$	$orth_Z$
20	20	20	0.24414	0.35714	0.71097	1.08700	1.17090
41	23	16	0.13169	0.26449	0.63198	0.72559	1.30420
36	47	22	0.20006	0.15254	0.83840	0.67989	1.81214
50	30	40	0.17695	0.17865	0.81327	0.64602	0.93008
67	46	67	0.16504	0.13370	0.84895	0.60308	1.04746
34	31	32	0.22381	0.19818	1.16388	0.73813	1.09946
28	39	39	0.19092	0.20197	1.07646	0.96881	0.93671
32	50	47	0.16655	0.16541	0.69578	0.81962	0.91608
41	63	52	0.16590	0.14002	0.72467	0.70486	1.08833
17	17	34	0.12383	0.12788	0.53498	0.60022	0.57055
28	42	47	0.12584	0.22129	0.87345	0.81936	0.76248
37	31	52	0.15072	0.12972	0.84068	0.52432	0.68538

Table 3.1: Stability Tests for SORCSD

3.4.5 Timing

The program SORCS2 primarily uses the LAPACK subroutines, SGESVD and SGEQRF to compute the SVD and QR decomposition respectively. SORCS1 also uses those subroutines plus a call to SGEQLF in order to compute the QL decomposition. In this section, we measure the time spent on each of these calls. Let t_{s1} and t_{s2} be the time spent in computing the first and second SVD, and t_{q2} be the time takes in computing the QR decomposition toward the end of the CSD decomposition in both SORCS1 and SORCS2. Also, let t_{q1} be the time spent in preforming the QL decomposition in SORCS1 or the first QR decomposition in SORCS2. Let t_{all} be the total time spent on SORCSD. Furthermore, we calculate the percentage of time of the two SVDs and the two QR decompositions (or a QR and a QL decompositions) relative to entire SORCSD as follows:

$$p_s = (\frac{t_{s1} + t_{s2}}{t_{all}})\% \ , \qquad p_q = (\frac{t_{q1} + t_{q2}}{t_{all}})\% \ .$$

Table 3.2 below reports the timing results (in seconds) on samples of all possible combinations of input dimensions. As expected, the time on computing the two SVDs accounts for the majority portion of the overall work.

m	p	ℓ	t_{s1}	t_{s2}	p_s	t_{q1}	t_{q2}	p_q	t_{all}
200	200	300	0.10254	0.00879	75.50%	0.01270	0.00000	8.61%	0.14746
410	230	160	0.13086	0.03027	85.94%	0.00586	0.00195	4.17%	0.18750
360	470	220	0.26465	0.04492	81.07%	0.01660	0.00293	5.12%	0.38184
500	300	400	0.70703	0.07422	85.47%	0.03223	0.01172	4.81%	0.91406
670	460	670	2.50586	0.14551	84.42%	0.11816	0.04004	5.04%	3.14062
340	310	320	0.43262	0.07227	85.17%	0.02539	0.00391	4.94%	0.59277
280	390	390	0.60840	0.04785	85.28%	0.03027	0.00879	5.08%	0.76953
320	500	470	0.98633	0.07129	84.87%	0.04492	0.01660	4.94%	1.24609
410	630	520	1.45215	0.16406	84.61%	0.07227	0.02246	4.96%	1.91016
170	170	340	0.06152	0.00000	67.02%	0.01172	0.00000	12.77%	0.09180
280	420	470	0.67480	0.03418	83.64%	0.03809	0.00879	5.53%	0.84766
370	310	520	0.55664	0.02051	76.16%	0.04688	0.01562	8.25%	0.75781

Table 3.2: Timing Profile for SORCSD

For the sake of comparing the performance of SORCSD due to the employment between the optimized subroutines from the Intel's math library (IMKL) and those un-optimized LAPACK and BLAS codes obtained from NetLib [19], we have designed a series of test matrices with incremental sizes. In this experiment, we let $m = p = \ell$ and test SORCSD at $m = 50, 100, 150, \ldots, 1000$. The overall time spent on using the optimized (denote by $t^o(m)$) versus un-optimized (denoted by $t^u(m)$) subroutines are collected in Table 3.3 and plotted on Figure 3.1 using logarithmic scales on both the X- and Y-axes.

$m = p = \ell$	$t^o(m)$	$t^u(m)$	speed-up
50	0.0117	0.0068	0.5830
100	0.0469	0.0439	0.9370
150	0.1201	0.1347	1.1213
200	0.2334	0.2879	1.2336
250	0.4600	0.5475	1.1904
300	0.7627	0.9389	1.2310
350	1.0059	1.4513	1.4429
400	1.5293	2.2995	1.5036
450	1.9502	3.4736	1.7811
500	2.5918	4.9620	1.9145
550	3.6338	7.3346	2.0185
600	4.2646	9.3189	2.1851
650	5.3926	12.1912	2.2607
700	7.6641	16.0991	2.1006
750	8.1230	19.8421	2.4427
800	9.5693	24.3044	2.5398
850	13.3486	30.5352	2.2875
900	13.5732	35.3634	2.6054
950	16.1904	42.6639	2.6351
1000	18.2598	49.7224	2.7231

Table 3.3: Overall Timing for SORCSD, using IMKL vs NetLib

For $m < 150$, the comparison tests show that using the NetLib codes requires less time than relying on the IMKL. This suggests that for small matrices, the overhead of calling the library functions outweighs its optimization provided, which is insignificant for small problems. However, as the size of matrices grows beyond 150, performance based on the IMKL picks up quickly. The fact that both sets of data form nearly a straight line in the log-log graph suggests that their rate of growth is polynomial. The trend of the data collected by using the IMKL and the NetLib code can be best approximated by the function $f^{(o)}(m) = 10^{-7}m^{2.75}$ and $f^{(u)}(m) = 10^{-7}m^{2.9}$, respectively, where m is the size of the test matrix. Computing $\frac{f^{(u)}(m)}{f^{(o)}(m)}$ leads to the conclusion that using optimized codes from the library gains $m^{0.15}$ improvement in the long run with respect to using the un-optimized alternatives.

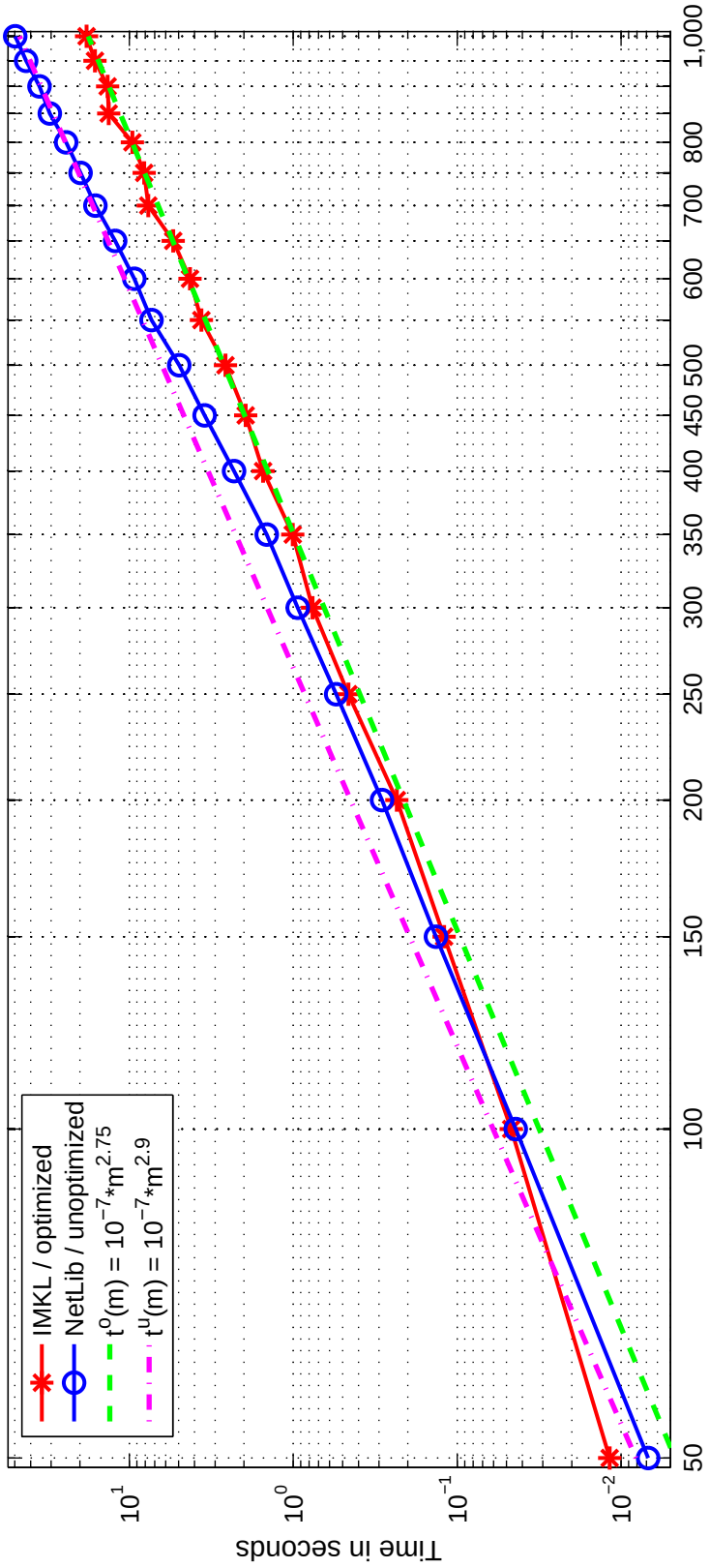


Figure 3.1: Plot of Timing Results for SORCSD, using IMKL vs NetLib

Chapter 4

Quotient Singular Value Decomposition

The notion of a quotient singular value decomposition (QSVD) was first introduced by Van Loan [36]. This chapter begins by establishing the definition and some basic properties of the QSVD in Section 4.1. Section 4.2 describes the numerical algorithms for computing the QSVD and some essential auxiliary decompositions that used as building blocks. We then discuss some implementation details in Section 4.3. Section 4.4 presents the result of a series of tests for verifying the stability and evaluating the performance of the software.

4.1 Definition and Basic Properties

Definition 4.1.1. Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times n}$ be two general matrices. A *quotient singular value decomposition (QSVD)* of A and B is the joint factorization

$$A = UC\,[\,O\;\;R\,]\,Z^T \quad \text{and} \quad B = VS\,[\,O\;\;R\,]\,Z^T,$$

where $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{p\times p}$, and $Z \in \mathbb{R}^{n\times n}$ are orthogonal matrices; $R \in \mathbb{R}^{r\times r}$ is non-singular and upper triangular where $r = \text{rank}(\,[\,A^T\;\;B^T\,]^T)$. The zero block is

of size r-by-$(n-r)$. $C \in \mathbb{R}^{m\times r}$ and $S \in \mathbb{R}^{p\times r}$ are diagonal, satisfying

$$C^TC + S^TS = I.$$

We write $C^TC = \mathrm{diag}(\alpha_1^2, \cdots, \alpha_r^2)$ and $S^TS = \mathrm{diag}(\beta_1^2, \cdots, \beta_r^2)$. The values of α_i and β_i are between 0 and 1 and the tuple (α_i, β_i) is called the *quotient singular value pairs* of A and B. It is sometimes preferable to define the *quotient singular values*, σ_i, as follows:

$$\sigma_i = \begin{cases} \alpha_i/\beta_i & \text{if } \beta_i \neq 0 \\ \infty & \text{otherwise .} \end{cases}$$

Furthermore, C and S belong to one of the following structures:

1. if $m \geq r$, then

$$C = \begin{array}{c} \begin{matrix} k & \ell \end{matrix} \\ \begin{bmatrix} I & O \\ O & \Sigma_1 \\ O & O \end{bmatrix} \end{array} \begin{array}{l} k \\ \ell \\ m-r \end{array} \quad \text{and} \quad S = \begin{array}{c} \begin{matrix} k & \ell \end{matrix} \\ \begin{bmatrix} O & \Sigma_2 \\ O & O \end{bmatrix} \end{array} \begin{array}{l} \ell \\ p-\ell \end{array} ; \tag{4.1}$$

2. if $m < r$, then

$$C = \begin{array}{c} \begin{matrix} k & m-k & r-m \end{matrix} \\ \begin{bmatrix} I & O & O \\ O & \Sigma_1 & O \end{bmatrix} \end{array} \begin{array}{l} k \\ m-k \end{array} \quad \text{and} \quad S = \begin{array}{c} \begin{matrix} k & m-k & r-m \end{matrix} \\ \begin{bmatrix} O & \Sigma_2 & O \\ O & O & I \\ O & O & O \end{bmatrix} \end{array} \begin{array}{l} m-k \\ r-m \\ p-\ell \end{array} . \tag{4.2}$$

The integer ℓ in (4.1) and (4.2) is the rank of B, and $k = r - \ell$.

The QSVD is also known as the GSVD, *generalized singular value decomposition*. Similarly, the quotient singular values of A and B is another name for the *generalized singular values* of the same pair of matrix. The structures of C and S in (4.1) and (4.2) are equivalent to those Σ_1 and Σ_2 of the GSVD defined in LAPACK Users' Guide [2].

Interestingly, the structures of C and S of the QSVD of A and B are very similar with the ones described for CSD in Chapter 3. In fact, if $[\ A^T\ \ B^T\ ]^T$ has orthonormal columns, then its QSVD and CSD are identical (i.e. $r = n$ and $R = I$). Thereby the CSD can be viewed as a special case of the QSVD.

If B is square and non-singular (i.e. $p = n = \ell$) , then the QSVD of A and B gives the SVD of AB^{-1}: $U^T(AB^{-1})V = CS^{-1}$, and the quotient singular values of A and B are equal to the singular values of AB^{-1}. Thus if $B = I$, then QSVD of A and B is just the SVD of A.

The connection between the QSVD of $[\ A^T\ \ B^T\ ]^T$ and the *generalized eigenvalue decomposition* (GEVD) of $A^TA - \lambda B^TB$ is analogous to that of the SVD of A and *eigenvalue decomposition* (EVD) of A^TA: The SVD of A provides a solution to the EVD of A^TA while the QSVD of $[\ A^T\ \ B^T\ ]^T$ can solve the GEVD problem of $A^TA - \lambda B^TB$.

Example 1. Consider a 5–by–4 general matrix A and a 3–by–4 general matrix B such that:

$$A = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 2 & 3 & 1 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 5 & 1 & 3 \\ 5 & 6 & 1 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 6 & 7 & 1 & 5 \\ 7 & 1 & -6 & 13 \\ -4 & 8 & 9 & -2 \end{bmatrix}.$$

Upon computed the QSVD of the pair $(A,\ B)$ with single precision, we obtain $k = 0$ and $\ell = 3$. Thus rank($[\ A^T\ \ B^T\ ]^T$) $= 3$. Since we have $m \geq k + \ell$ in this case, the structures of C and S should follow the ones in (4.1). Specifically, they are:

$$C = \begin{bmatrix} 0.8094507 & 0 & 0 \\ 0 & 0.1184501 & 0 \\ 0 & 0 & 0.0000001 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } S = \begin{bmatrix} 0.5871880 & 0 & 0 \\ 0 & 0.9929600 & 0 \\ 0 & 0 & 1.0000001 \end{bmatrix}.$$

The computed upper triangular matrix R and the orthogonal matrices U, V, Z are:

$$R = \begin{bmatrix} -7.6504660 & -13.4985952 & -5.2955680 \\ 0 & -14.8204641 & 7.5874281 \\ 0 & 0 & 12.8452349 \end{bmatrix},$$

$$U = \begin{bmatrix} -0.2260582 & 0.7408761 & 0.1622926 & -0.4151672 & -0.4486618 \\ -0.3253317 & 0.4406352 & -0.2408625 & 0.0034384 & 0.8012324 \\ -0.4246051 & 0.1403936 & 0.4138230 & 0.7824448 & -0.1285717 \\ -0.5238786 & -0.1598469 & -0.7542266 & 0.0854641 & -0.3519066 \\ -0.6231520 & -0.4600888 & 0.4189740 & -0.4561801 & 0.1279077 \end{bmatrix},$$

$$V = \begin{bmatrix} -0.9958782 & -0.0906995 & 0.0000001 \\ 0.0906997 & -0.9958783 & -0.0000001 \\ 0.0000002 & -0.0000001 & 1.0000000 \end{bmatrix},$$

$$Q = \begin{bmatrix} 0.4050956 & 0.7845642 & 0.3512648 & -0.3113995 \\ -0.5570067 & 0.3424263 & 0.4296614 & 0.6227992 \\ 0.7089177 & -0.0695111 & -0.0411706 & 0.7006491 \\ 0.1519110 & -0.5122221 & 0.8308485 & -0.1556998 \end{bmatrix}.$$

Example 2. Consider a 3–by–4 general matrix A and a 4–by–4 general matrix B such that:

$$A = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 2 & 3 & 1 & 1 \\ 3 & 4 & 1 & 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 4 & 5 & 1 & 3 \\ 5 & 6 & 1 & 4 \\ 6 & 7 & 1 & 5 \\ 7 & 1 & -6 & 13 \end{bmatrix}.$$

Upon computed the QSVD of the pair (A, B), we obtain $k = 2$ and $\ell = 2$. Hence rank($[\ A^T \quad B^T\ ]^T$) $= 4$. Since we now have $m < k + \ell$, the C and S follows the structure in (4.2). Specifically, they are:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.9013880 & 0 \end{bmatrix} \quad \text{and} \quad S = \begin{bmatrix} 0 & 0 & 0.4330123 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

The computed upper triangular matrix R and the orthogonal matrices U, V, Z are:

$$R = \begin{bmatrix} -2.2800045 & 1.9215150 & -1.8185604 & 1.7103494 \\ 0 & 0.4024834 & 1.9864100 & 1.6640470 \\ 0 & 0 & -3.6234777 & -0.6394508 \\ 0 & 0 & 0 & -8.3343592 \end{bmatrix},$$

$$U = \begin{bmatrix} -0.1230329 & 0.9924024 & -0.0000017 \\ -0.9924024 & -0.1230332 & 0.0000001 \\ 0.0000001 & -0.0000017 & -1.0000000 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.1601281 & -0.8006408 & 0.5766981 & -0.0274353 \\ -0.8006406 & -0.1601281 & -0.0274352 & -0.5766980 \\ 0.4803843 & -0.3202564 & -0.6041333 & -0.5492628 \\ -0.3202564 & -0.4803845 & -0.5492628 & 0.6041333 \end{bmatrix},$$

$$Z = \begin{bmatrix} -0.0654902 & -0.8558932 & 0.2247960 & 0.4611124 \\ 0.5610318 & 0.3260307 & -0.1322331 & 0.7493075 \\ -0.7760576 & 0.3668473 & 0.2247960 & 0.4611124 \\ 0.2805158 & 0.1630152 & 0.9388546 & -0.1152781 \end{bmatrix}.$$

In both examples, it is easy to verify that the computed matrices satisfy the requirements in the QSVD definition with single precision accuracy.

4.2 Numerical Algorithms

The GSVD subroutine in current LAPACK is computed based on a Jacobi-type iterative approach [20, 4], which may suffer from a slow rate of convergence. The algorithm that we are about to describe is based on the computation of the CSD and was proposed by Van Loan [36]. This section first explains two decompositions that contribute to the computation of the QSVD followed by presenting the overall algorithm for the QSVD computation.

4.2.1 Pre-processing Step

Like Jacobi-type method for solving the GSVD in current LAPACK, our method also requires a pre-processing step. This step reduces the input matrices A and B to two upper triangular matrices while revealing their rank information. Specifically, given input A and B, we compute U, V, and Q such that

$$U^T A Q = \begin{array}{c} \begin{matrix} n-k-\ell & k & \ell \end{matrix} \\ \begin{bmatrix} O & R_{12}^{(a)} & R_{13}^{(a)} \\ O & O & R_{23}^{(a)} \\ O & O & O \end{bmatrix} \end{array} \begin{matrix} k \\ \ell \\ m-k-\ell \end{matrix}, \quad V^T B Q = \begin{array}{c} \begin{matrix} n-k-\ell & k & \ell \end{matrix} \\ \begin{bmatrix} O & O & R_{13}^{(b)} \\ O & O & O \end{bmatrix} \end{array} \begin{matrix} \ell \\ p-\ell \end{matrix}. \tag{4.3}$$

where $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{p\times p}$, and $Q \in \mathbb{R}^{n\times n}$ are orthogonal matrices; $R_{12}^{(a)}$ and $R_{13}^{(b)}$ are upper triangular; ℓ is the rank of B, $k+\ell$ is the rank of $[\ A^T \ \ B^T\ ]^T$. If $m \geq k+\ell$, then $R_{23}^{(a)}$ is $\ell \times \ell$ and upper triangular. Otherwise, $R_{23}^{(a)}$ is $(m-k)\times \ell$ and upper trapezoidal and the bottom zero block of UAQ^T does not appear.

An algorithm for this decomposition has been developed and implemented in LAPACK [5]. The core of the algorithm is using a QR decomposition with pivoting as the rank revealing mechanism followed by a RQ decomposition that reduces the left most columns to zero.

4.2.2 Split QR

The other main module for computing the QSVD is to reduce two upper triangular matrices to one upper triangular matrix via QR decomposition. Specifically, let $R_1 \in \mathbb{R}^{m\times n}$ and $R_2 \in \mathbb{R}^{n\times n}$ both be upper triangular and $m \leq n$. We wish to find an orthogonal matrix $Q \in \mathbb{R}^{(m+n)\times(m+n)}$ or its first n columns, $Q_\perp$, and an upper triangular matrix $R \in \mathbb{R}^{n\times n}$ such that

$$\begin{bmatrix} R_1 \\ R_2 \end{bmatrix} = Q \begin{bmatrix} R \\ O \end{bmatrix}$$

or

$$\begin{bmatrix} R_1 \\ R_2 \end{bmatrix} = Q_\perp R.$$

Mathematically, this is a simple matrix operation. In practice, however, note that explicitly forming the matrix $[\ R_1^T \ \ R_2^T\ ]^T$ requires $(m+n)$-by-n memory space. We now describe a more flexible and memory-saving approach.

The facts that R_1 and R_2 may reside on non-consecutive memory location and they are upper triangular suggest that Givens rotation is a more attractive technique than Householder transformation. While there are many freedom in the ordering of rotation, applying Givens rotations by rows would lead to optimal saving in work space requirement.

In the following figures associated with both Example 1 and 2, each integer subscript of $\times$ denotes the sequence in which the element is annihilated via Givens rotation.

$$R_1 = \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}$$

$$R_2 = \begin{bmatrix} \times_1 & \times_2 & \times_3 \\ 0 & \times_4 & \times_5 \\ 0 & 0 & \times_6 \end{bmatrix}$$

Figure 4.1: Illustration of the Order of Annihilation (Example 1)

Example 1: As illustrated in Figure 4.1, we start by working on the first row of R_2. The elements $\times_1$, $\times_2$ and $\times_3$ are eliminated by rotating with the first, second and third rows of R_1, respectively. Since rotating each of those entries in R_2 modifies $Q_{(1:(m+n),4)}$, this column must be saved in the next two rotations in order to correctly update $Q_{(1:(m+n),2)}$, $Q_{(1:(m+n),3)}$. After the third rotation between $\times_3$ and $R_{1(3,3)}$, $Q_{(1:(m+n),4)}$ is fixed and can be discarded if not desired since it will no longer affect any other columns in Q. Next $\times_4$ and $\times_5$ are eliminated by rotating with the second and third rows of R_1, respectively. This time, $Q_{(1:(m+n),5)}$ is modified in the successive rotation on the same row of R_2 and it affects the update in $Q_{(1:(m+n),3)}$. We may re-used the $m+n$ vector to store $Q_{(1:(m+n),5)}$ until it is finalized. Therefore, this particular rotation ordering scheme requires total of $(m+n)$ work space.

$$R_1 = \left[\begin{array}{cccc} \times & \times & \times & \times \\ 0 & \times & \times & \times \end{array}\right]$$

$$R_2 = \left[\begin{array}{cccc} \times_1 & \times_2 & \times & \times \\ 0 & \times_3 & \times_4 & \times \\ 0 & 0 & \times_5 & \times_6 \\ 0 & 0 & 0 & \times_7 \end{array}\right]$$

Figure 4.2: Illustration of the Order of Annihilation (Example 2)

Example 2: The second ordering of annihilation, as demonstrated in Figure 4.2, is rather similar with the first case, except that some elements in R_2 now cannot be annihilated although their values may be modified during the overall rotation process. After all the elements with subscripts are eliminated, the remained non-zero elements (those $\times$'s without subscripts in the figure) form a 4–by–4 upper-triangular matrix.

Since this idea entails on computing the QR decomposition with input that is split in two non-consecutive memory locations, we refer to it as a QR decomposition implemented in split fashion or a *Split QR*.

As an interesting side note, this idea of the Split QR can be easily modified to compute the last m columns of Q so that it provides an economic way to obtain the null space of $[\ R_1^T \ \ R_2^T\ ]^T$ provided that it has full rank.

To describe the complete algorithm for the Split QR, we first lay out the two functions, `genGiv` and `appGiv`, for generating and applying a 2-by-2 Givens rotation,

respectively. The pseudo-code presented here is one of the simple versions while other robust algorithms are discussed in various linear algebra literature such as [14, 29].

Functions to generate and apply a 2-by-2 Givens rotation matrix.

---

function $[c, s, r] = \texttt{genGiv}(a, b)$

Purpose: Compute a Givens rotation to annihilate b using a. That is, compute c, s, and r such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}.$$

if $a = 0$ **then**
  $c = 0$
  $s = 1$
  $r = b$
else
  if $|a| > |b|$ **then**
    $t = b/a$
    $u = \sqrt{1 + t^2}$
    $c = 1/u$
    $s = c * t$
    $r = a * u$
  else
    $t = a/b$
    $u = \sqrt{1 + t^2}$
    $s = 1/u$
    $c = s * t$
    $r = b * u$
  end if
end if

function $[u_1, u_2] = \texttt{appGiv}(v_1, v_2, c, s)$

Purpose: Apply a Givens rotation, defined by the parameters c and s, from the left to the row vectors v_1 and v_2 such that

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$u_1 = c * v_1 + s * v_2$
$u_2 = c * v_2 - s * v_1$

Using `genGiv` and `appGiv`, we are now ready to describe the complete algorithm for the Split QR in the following.

Algorithm 4.2.1. Computing the QR decomposition in a split fashion

---

Input: $R_1 \in \mathbb{R}^{m\times n}, R_2 \in \mathbb{R}^{n\times n}$ where $m \leq n$ and both R_1 and R_2 are upper triangular.

Output: Q_1, Q_2, and R such that $[\ R_1^T\ \ R_2^T\ ]^T = [\ Q_1^T\ \ Q_2^T\ ]^T R$, where $R \in \mathbb{R}^{n\times n}$ is upper triangular; Q_1 and Q_2 have the same sizes as R_1 and R_2, respectively and $[\ Q_1^T\ \ Q_2^T\ ]^T$ is orthogonal.

Initialization: fill up Q_1 and Q_2 with 1 or 0 so that $[\ Q_1^T\ \ Q_2^T\ ]^T = I$
let R be the first n rows of $[\ R_1^T\ \ R_2^T\ ]^T$.
for $i = 1$ to n **do**
  $T = [\ 0\ ...\ 0\ 1\ 0\ ...\ 0\ ]$ where $T \in \mathbb{R}^{m+n}$ and $T_{(m+i)} = 1$
  for $j = 1$ to $\min(n, m-1+i)$ **do**
    if $j \leq m$ **then**
      % Use elements in R_1 to eliminate $R_2(j,j)$
      $[c, s, r] =$ `genGiv`$(R_{1(j,j)}, R_{2(i,j)})$
      Set $R_{1(j,j)} = r$ and $R_{2(i,j)} = 0$
      if $j+1 \leq n$ **then**
        $[R_{1(j,j+1:n)}, R_{2(i,j+1:n)}] =$ appGiv$(\ R_{1(j,j+1:n)}, R_{2(i,j+1:n)}, c, s\ )$
      end if
    else
      % Use elements in R_2 to eliminate $R_2(j,j)$
      $[c, s, r] =$ `genGiv`$(\ R_{2(j-m,j)}, R_{2(i,j)}\ )$
      Set $R_{1(j-m,j)} = r$ and $R_{2(i,j)} = 0$
      if $j+1 \leq n$ **then**
        $[R_{2(j-m,j+1:n)}, R_{2(i,j+1:n)}] =$ appGiv$(\ R_{2(j-m,j+1:n)}, R_{2(i,j+1:n)}, c, s\ )$
      end if

19: **end if**

20: *% Update a column in Q_1 and Q_2*

21: $[Q_1(1{:}m,j), T(1{:}m)] = \texttt{appGiv}(\ Q_1(1{:}m,j), T(1{:}m), c, s\ )$

22: $[Q_2(1{:}n,j), T(m{+}1{:}m{+}n)] = \texttt{appGiv}(\ Q_2(1{:}n,j), T(m{+}1{:}m{+}n), c, s\ )$

23: **end for**

24: **end for**

4.2.3 Computing the QSVD

Armed by the decompositions described so far, we are now ready to present an algorithm for computing the QSVD. The algorithm consists of six major steps.

Step 1. The computation is started out by preprocessing the input matrices A and B. The result is the same as (4.3), namely,

$$U^T A Q = R^{(a)} \quad \text{and} \quad V^T B Q = R^{(b)} \tag{4.4}$$

where

$$R^{(a)} = \begin{array}{c} \begin{matrix} n-k-\ell & k & \ell \end{matrix} \\ \begin{bmatrix} O & R_{12}^{(a)} & R_{13}^{(a)} \\ O & O & R_{23}^{(a)} \\ O & O & O \end{bmatrix} \begin{matrix} k \\ \ell \\ m-k-\ell \end{matrix} \end{array} \quad \text{and} \quad R^{(b)} = \begin{array}{c} \begin{matrix} n-k-\ell & k & \ell \end{matrix} \\ \begin{bmatrix} O & O & R_{13}^{(b)} \\ O & O & O \end{bmatrix} \begin{matrix} \ell \\ p-\ell \end{matrix} \end{array}.$$

As explained in Section 4.2.1, U, V and Q are orthogonal, $R_{12}^{(a)}$ and $R_{13}^{(b)}$ are nonsingular and upper triangular, and $\text{rank}(B) = \ell$, $\text{rank}(\ [\ A^T\ \ B^T\ ]^T) = k + \ell = r$. If $m \geq r$, $R_{23}^{(a)}$ is upper triangular. Otherwise, $R_{23}^{(a)}$ is upper trapezoidal and the bottom zero block of $R^{(a)}$ would not exist.

Step 2. Next, we compute the QR decomposition on $[\ R_{23}^{(a)T}\ \ R_{13}^{(b)T}\ ]^T$ using Algo-

rithm 4.2.1. Let the result of the decomposition be Q_1, Q_2 and $\tilde{R}_{23}$. Note that Q_1 and Q_2 are not stored consecutively. We may zoom out the decomposition and verify that

$$\begin{array}{c} \\ k \\ \ell \\ m-k-\ell \end{array} \begin{array}{c} \begin{array}{cc} k & \ell \end{array} \\ \begin{bmatrix} I & O \\ O & Q_1 \\ O & O \end{bmatrix} \end{array} \hat{R} = R^{(a)} \quad \text{and} \quad \begin{array}{c} \\ \ell \\ p-\ell \end{array} \begin{array}{c} \begin{array}{cc} k & \ell \end{array} \\ \begin{bmatrix} O & Q_2 \\ O & O \end{bmatrix} \end{array} \hat{R} = R^{(b)} ,$$

where

$$\hat{R} = \begin{array}{c} \begin{array}{ccc} n-k-\ell & k & \ell \end{array} \\ \begin{bmatrix} O & R_{12}^{(a)} & R_{13}^{(a)} \\ O & O & \widetilde{R}_{23} \end{bmatrix} \end{array} \begin{array}{c} k \\ \ell \end{array} .$$

Step 3. Compute the CSD on Q_1 and Q_2 using Algorithm 3.2.1. Let the result of the computation be U_1, V_1, Z_1, C_1, and S_1 so that

$$Q_1 = U_1 C_1 Z_1^T \quad \text{and} \quad Q_2 = V_1 S_1 Z_1^T . \tag{4.5}$$

Since Q_1 has size $\ell \times \ell$ if $m \geq r$ and $(m-k) \times \ell$ otherwise and Q_2 is always square, the structure of the cosine and sine values of Q_1 and Q_2 would fall into case 3 in (3.6). One can derive from the result of QR decomposition in *Step 2* and CSD in this step that

$$\begin{bmatrix} I & O \\ O & U_1 C_1 Z_1^T \\ O & O \end{bmatrix} \hat{R} = R^{(a)} \quad \text{and} \quad \begin{bmatrix} O & V_1 S_1 Z_1^T \\ O & O \end{bmatrix} \hat{R} = R^{(b)} .$$

It follows that the components of CSD can be split out so the two equations above are equivalent to

$$(\hat{U}C\hat{Z}^T)\hat{R} = R^{(a)} \quad \text{and} \quad (\hat{V}S\hat{Z}^T)\hat{R} = R^{(b)}$$

where

$$\hat{U} = \begin{array}{c} \begin{matrix} k & \ell & m-k-\ell \end{matrix} \\ \begin{bmatrix} I & O & O \\ O & U_1 & 0 \\ O & O & I \end{bmatrix} \end{array} \begin{array}{l} k \\ \ell \\ m-k-\ell \end{array}, \quad \hat{V} = \begin{array}{c} \begin{matrix} \ell & p-\ell \end{matrix} \\ \begin{bmatrix} V_1 & O \\ O & I \end{bmatrix} \end{array} \begin{array}{l} \ell \\ p-\ell \end{array}, \quad \hat{Z}^T = \begin{array}{c} \begin{matrix} k & \ell \end{matrix} \\ \begin{bmatrix} I & O \\ O & Z_1^T \end{bmatrix} \end{array} \begin{array}{l} k \\ \ell \end{array},$$

and

$$C = \begin{array}{c} \begin{matrix} k & \ell \end{matrix} \\ \begin{bmatrix} I & O \\ O & C_1 \\ O & O \end{bmatrix} \end{array} \begin{array}{l} k \\ \ell \\ m-k-\ell \end{array}, \quad S = \begin{array}{c} \begin{matrix} k & \ell \end{matrix} \\ \begin{bmatrix} O & S_1 \\ O & O \end{bmatrix} \end{array} \begin{array}{l} \ell \\ p-\ell \end{array}. \tag{4.6}$$

Note that in case of $m < k+\ell$, both U_1 and C_1 have only $m-k$ rows and the bottom $m-k-\ell$ rows of $\hat{U}$ and C do not appear.

Step 4. Clearly, the final U is formed by combining its $k+1$ to t columns with U_1 where $t = \min(m, l+k)$: $U_{(1:m,k+1:t)} := U_{(1:m,k+1:t)}U_1$.

Step 5. The final V is formed by combining its leading ℓ columns with V_1. That is: $V_{(1:p,1:\ell)} := V_{(1:p,1:\ell)}V_1$.

Step 6. The last major step forms the final right most orthogonal matrix Q. Unfortunately, with $\hat{R}$ locating between Q and $\hat{Z}$, the two orthogonal matrices cannot be

combined right away. First we multiply $\hat{Z}$ with $\hat{R}$:

$$\hat{Z^T}\hat{R} = \begin{bmatrix} I & O \\ O & Z_1^T \end{bmatrix} \begin{bmatrix} O & R_{12}^{(a)} & R_{13}^{(a)} \\ O & O & \tilde{R}_{23} \end{bmatrix} = \begin{bmatrix} O & R_{12}^{(a)} & R_{13}^{(a)} \\ O & O & Z_1^T\tilde{R}_{23} \end{bmatrix}.$$

This suggests that multiplying the large matrices $\hat{Z}$ with $\hat{R}$ only affects the (2,3)-block of $\hat{R}$. Secondly, RQ decomposition is performed on the product of $Z_1^T\tilde{R}_{23}$: Let the result be R_{23} and Q_3, that is $(Z_1^T\tilde{R}_{23}) = R_{23}Q_3$. The RQ decomposition of the $\hat{Z}^T\hat{R}$ is then completed by setting $R_{13} = R_{13}^{(a)}Q_3^T$. To verify that this indeed provides the RQ decomposition of the product $\hat{Z}^T\hat{R}$, one may check that

$$\hat{Z}^T\hat{R} = \begin{bmatrix} I & O \\ O & Z_1^T \end{bmatrix} \begin{bmatrix} O & R_{12}^{(a)} & R_{13}^{(a)} \\ O & O & \tilde{R}_{23} \end{bmatrix} = \underbrace{\left[\begin{array}{cc|c} O & R_{12}^{(a)} & R_{13} \\ O & O & R_{23} \end{array}\right]}_{[\,O\;R\,]} \left[\begin{array}{cc} I & O \\ \hline O & Q_3 \end{array}\right] \begin{matrix} n-\ell \\ \ell \end{matrix}.$$

Lastly, the final Q matrix is formed by multiplying its proper columns with Q_3^T:

$$\underbrace{\begin{matrix} n-\ell & \ell \\ \left[\hat{Q}_1 \;\middle|\; \hat{Q}_2\right] \end{matrix}}_{Q} \begin{bmatrix} I & O \\ O & Q_3^T \end{bmatrix} \begin{matrix} n-l \\ \ell \end{matrix} = \left[\hat{Q}_1 \;\middle|\; \hat{Q}_2Q_3^T\right].$$

Putting it together, our QSVD algorithm can be summarized in the following. Besides the preprocessing step, this algorithm employs Algorithm 3.2.1 (CSD) and 4.2.1 (Split QR) plus the QR and RQ decompositions.

Algorithm 4.2.2. Computing the QSVD based on the CSD

1: Preprocess A and B: obtaining U, V, Q , $R^{(a)}$, $R^{(b)}$, k and ℓ as in (4.4).

2: Compute the orthonormal basis for $[\; R_{23}^{(a)T} \;\; R_{13}^{(b)T} \;]^T$ using Algorithm 4.2.1, obtaining $[\; Q_1^T \;\; Q_2^T \;]^T$ (orthogonal), and $\tilde{R}_{23}$ (upper triangular).

3: Compute the CSD on Q_1 and Q_2 using Algorithm 3.2.1, obtaining U_1, V_1, and Z_1, C_1 and S_1 as in (4.5).

4: Set C and S using C_1 and S_1 as they are in (4.6).

5: Update U: $U_{(1:m,k+1:t)} := U_{(1:m,k+1:t)}U_1$ where $t = \min(m, \ell + k)$.

6: Update V: $V_{(1:p,1:\ell)} := V_{(1:p,1:\ell)}V_1$.

7: Set $T = Z_1^T \ \tilde{R}_{23}$.

8: Compute the RQ decomposition of T: $T = R_{23}Q_3$.

9: Update $R_{13}^{(a)}$: $R_{13}^{(a)} := R_{13}^{(a)}Q_3^T$.

10: Update Q: $Q_{(1:n,n-l+1:n)} := Q_{(1:n,n-l+1:n)}Q_3^T$.

4.3 LAPACK-Style Software

The QSVD algorithm described in the previous section is implemented in FORTRAN77 in LAPACK style using real data; it is named SGGQSV. In this section, we discuss the details of the implementation of SGGQSV as well as the analyze its work space requirement.

4.3.1 Implementation Details

We assume that the caller has allocated the following spaces for SGGQSV at the time it is called: $\mathcal{U} \in \mathbb{R}^{m\times m}$, $\mathcal{V} \in \mathbb{R}^{m\times m}$, $\mathcal{Q} \in \mathbb{R}^{m\times m}$, and works paces $\mathcal{W} \in \mathbb{R}^{L_w}$ and $\mathcal{W}_i \in \mathbb{N}^{L_i}$. The sizes of L_w and L_i are determined toward the end of this section. Furthermore, $\mathcal{A} \in \mathbb{R}^{m\times n}$ and $\mathcal{B} \in \mathbb{R}^{p\times n}$ store the input matrices A and B respectively upon entrance and are overwritten at exit.

Step 1. The first step of the implementation is to reduce both of the input matrices into two upper triangular forms. This is done via a call to SGGSVP. The key feature of this subroutine is to reveal the numerical rank of $[\ A^T \quad B^T\ ]^T$ without forming the matrix explicitly. SGGSVP uses SGEQPF, which is QR factorization with column pivoting, as the rank-revealing mechanism. In SGGSVP, besides the work space of size $\max(3 * n, m, p)$, it needs an integer array of dimension n to save

the pivoting information, and it needs a real array of the same dimension to save the n scalar factors of the elementary reflectors. Upon return, two of the upper triangular matrices overwrites the spaces in $\mathcal{A}$ and $\mathcal{B}$ and the orthogonal matrices are placed in $\mathcal{U}$, $\mathcal{V}$, and $\mathcal{Q}$.

Step 2. The algorithm that computes the QR decomposition in a split fashion is coded in FORTRAN77 named SGGQRJ. This subroutine mainly uses a level-1 BLAS subroutine, SROT, to generate the Givens rotation and a LAPACK's subroutine, SLARTG, to apply the rotation to two input vectors. Let $t = \min(m-k, \ell)$. This subroutine takes the appropriate t-by$-\ell$ and ℓ-by-ℓ portion of the two upper-triangular matrices in $\mathcal{A}$ and $\mathcal{B}$ as input. Upon exit, the first t rows of the resulted upper triangular matrix is overwritten to the same t-by-ℓ portion in $\mathcal{A}$, and the rest of the $\ell - t$ rows overwrite on the leading $\ell - t$ input portion of $\mathcal{B}$. Furthermore, we need spaces $\mathcal{Q}_1 \in \mathbb{R}^{t\times\ell}$, $\mathcal{Q}_2 \in \mathbb{R}^{\ell\times\ell}$ to hold the two resulted orthogonal matrices. As aforementioned, the work space for SGGQRJ is $m + n$.

Step 3. This step calls the subroutine SORCSD, which was developed in the previous chapter. It should be emphasized that this subroutine uses the areas for the input matrices as work space, yet none of the resulted components are overwritten to those areas. This means that it is necessary to allocate spaces $\mathcal{U}_1 \in \mathbb{R}^{t\times t}$, $\mathcal{V}_1 \in \mathbb{R}^{\ell\times\ell}$, and $\mathcal{Z}_1 \in \mathbb{R}^{\ell\times\ell}$ to store the returned matrices. In additional, SORCSD requires a work space of size $7\,\ell + \ell^2$.

Step 4-5. The major tasks in those two steps involve in matrix-matrix multiply, which is done efficient by SGEMM subroutine in level-3 BLAS.

Step 6. To form the final Q and R, we first multiply Z_1^T from SORCSD with $\tilde{R}_{23}$ from SGGQRJ. The result of this multiplication can be placed in $\mathcal{Q}_2$. It follows that the RQ decomposition of this product is computed via a call to SGERQF. Since

the orthogonal matrix resulted from SGERQF is stored as a sequence of elementary reflectors, the update of R and Q can be computed by calling SORMR2.

At exit, $\mathcal{A}$ and $\mathcal{B}$ are overwritten by partial of the resulted R matrix as follows:

- if $m \geq k + \ell$,

$$[\, O \; R \,] = \begin{array}{c} \begin{matrix} n-k-\ell & k & \ell \end{matrix} \\ \begin{bmatrix} O & R_{11} & R_{12} \\ O & O & R_{22} \end{bmatrix} \end{array} \begin{matrix} k \\ \ell \end{matrix}$$

  R is stored in $\mathcal{A}_{(1:k+\ell,\, n-k-\ell+1:n)}$ on exit.

- otherwise ($m < k + \ell$),

$$[\, O \; R \,] = \begin{array}{c} \begin{matrix} n-k-\ell & k & m-k & k+\ell-m \end{matrix} \\ \begin{bmatrix} O & R_{11} & R_{12} & R_{13} \\ O & O & R_{22} & R_{23} \\ O & O & O & R_{33} \end{bmatrix} \end{array} \begin{matrix} k \\ m-k \\ k+\ell-m \end{matrix}$$

  The first m rows of R are stored in $\mathcal{A}_{(1:m,\, n-k-\ell+1:n)}$, and R_{33} is stored in $\mathcal{B}_{(1:k+l-m,\, n+m-k-\ell+1:n)}$. Note that the storage location of R_{33} is the only difference in output format between SGGQSV and SGGSVD (in LAPACK).

Overall, the sequence of major subroutine calls in SGGQSV is summarized below.

Step	Major Task	subroutine name
1.	pro-processing	SGGSVP (LAPACK)
2.	compute Split QR	SGGQRJ
3.	compute CSD	SORCSD

4.3.2 Work Space Requirement

The size of the overall work space of the overall SGGQSV subroutine can be determined by examining the first three steps since rest of the steps can re-use those that are allocated earlier.

The size of $\mathcal{W}_i$ is the size needed to complete SGGSVP since this is the only place that calls for the integer work space. Hence $L_i = n$.

SGGSVP in *Step 1* requires work space of $n + \max(3n, m, p)$. After this, $\mathcal{U}$ and $\mathcal{V}$ are fixed. While SGGQRJ does not require more work space, it needs new places to store the Q_1 and Q_2 upon return. As for SORCSD, new spaces are needed to hold the computed components. Hence the work space required to complete *Step 3* would be:

$$\begin{aligned} &= \underbrace{(t \times \ell)}_{\mathcal{Q}_1} + \underbrace{(\ell \times \ell)}_{\mathcal{Q}_2} + \underbrace{(t \times t)}_{\mathcal{U}_1} + \underbrace{(\ell \times \ell)}_{\mathcal{V}_1} + \underbrace{(\ell \times \ell)}_{\mathcal{Z}_1} + 7 \times \ell + \ell \times \ell \\ &\simeq 6\ell^2 + 7\ell. \end{aligned}$$

At the time SGGQSV is called, the numerical rank is unknown, but we know that $\ell \leq \min(\ p, n\ )$ since ℓ is the rank of B. Subsequence computations can re-use this space, thus the size of work space require to complete the first three steps is sufficient to determine the size of the work space in the overall SGGQSV subroutine. That is:

$$\mathcal{L}_w \ \leq \ 6\min(p, n)^2 + 6n + \max(3n, m, p).$$

4.4 Testing and Timing

This section presents the results of the numerical experiments on testing SGGQSV. We have developed a rich set of testing cases which include various types of matrices in all possible combinations of input sizes.

4.4.1 Notion of Stability

Suppose A, B are two input matrices and the computed matrices returned upon calling SGGSVD are $\widetilde{U}, \widetilde{V}, \widetilde{Q}, \widetilde{C}, \widetilde{S}$ and $\widetilde{R}$ such that $\widetilde{U}^T A \widetilde{Q} = \widetilde{C}[\ O\ \widetilde{R}\ ]$ and $\widetilde{V}^T B \widetilde{Q} =$

$\widetilde{S}[\, O\ \widetilde{R}\,]$, then the following ratios are recorded in each of the stability tests:

$$res_A = \frac{\|\widetilde{U}^T A\widetilde{Q} - \widetilde{C}[\, O\ \widetilde{R}\,]\|_1}{max(m,n)\|A\|_1\,\epsilon}\ , \quad res_B = \frac{\|\widetilde{V}^T B\widetilde{Q} - \widetilde{S}[\, O\ \widetilde{R}\,]\|_1}{max(p,n)\|B\|_1\,\epsilon}\ , \tag{4.7}$$

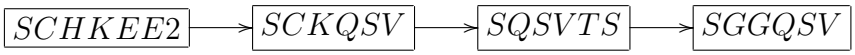
$$orthog_U = \frac{\|I - \widetilde{U}^T\widetilde{U}\|_1}{m\,\epsilon}\ , \qquad orthog_V = \frac{\|I - \widetilde{V}^T\widetilde{V}\|_1}{p\,\epsilon}\ , \tag{4.8}$$

$$orthog_Q = \frac{\|I - \widetilde{Q}^T\widetilde{Q}\|_1}{n\,\epsilon}\ . \tag{4.9}$$

We consider the method for computing the QSVD is *stable* if all of those test results are bounded by $\mathcal{O}(1)$.

4.4.2 Testing Routines

Recall from the previous chapter, SCHKEE2 was a modified version of SCHKEE that allows to test SORCSD. To enable testing of SGGQSV, we must add a section in SCHKEE2 so it calls the QSVD testing driver, SCKQSV. This subroutine consists of a doubly-nested loop: the outer one iterates over the ordered triples (m, p, n) and the inner one iterates over various of matrix types. On each iteration of the innermost loop, matrices A and B are generated and are feed into SQSVTS to compute the quantities of the testing criteria. The testing procedure is summarized below.



4.4.3 Input File Format

An annotated example of an input file for testing of the QSVD subroutine is shown below:

```
QSV:  Data file for testing Quotient SVD subroutine
12                                      Number of values of M, P, N
50 65 43  72 44 37  25 36 13  26 37 12 Values of M (row dimension)
50 31 61  22 18 29  30 66 52  60 25 12 Values of P (row dimension)
50 23 21  54 44 35  30 60 48  77 80 60 Values of N (column dimension)
2                       Threshold value of test ratio
T                       Put T to test if error exits
1                       Code to interpret the seed
QSV 8                   List types on next line if 0 < NTYPES < 8
```

The first line of the input files must contain the character QSV in columns 1-3.

Lines 2-9 are read using list-directed input and specify the following values:

line 2: The number of values m, p, and n

line 3: Values of m (row dimension)

line 4: Values of p (row dimension)

line 5: Values of n (column dimension)

line 6: The threshold value for the test ratios.

We may set this one to 0 in order to display all the test ratios.

line 7: Flag to test if error exists

line 8: An integer code to interpret the random number seed.

= 0: Set the seed to a default value before each run.

= 1: Initialize the seed to a default value only before the first run.

= 2: Like 1, but use the seed values on the next line.

line 9: If line 8 was 2, four integer values for the random seed.

Otherwise, the path QSV followed by the number of matrix types.

4.4.4 Types of Test Matrices

There are eight types of matrices used in the tests. Each type is generated with a pre-determined condition number. Table 4.1 below summarizes the properties of each of those types.

Matrix Types	Matrix A	Matrix B	$\|A\|$	$\|B\|$	$\kappa(A)$	$\kappa(B)$
1	Diagonal	Upper triangular	10	1000	100	10
2	Upper triangular	Upper triangular	10	1000	100	10
3	Lower triangular	Upper triangular	10	1000	100	10
4	Random Dense	Random Dense	10	1000	100	10
5	Random Dense	Random Dense	10	1000	$\sqrt{0.1/\epsilon}$	$\sqrt{0.1/\epsilon}$
6	Random Dense	Random Dense	10	1000	$0.1/\epsilon$	$0.1/\epsilon$
7	Random Dense	Random Dense	10	1000	$\sqrt{0.1/\epsilon}$	$0.1/\epsilon$
8	Random Dense	Random Dense	10	1000	$0.1/\epsilon$	$\sqrt{0.1/\epsilon}$

Table 4.1: Types of Matrices Used to Test SGGQSV

4.4.5 Stability Testing

Recall toward the end of Section 4.1, we illustrated two numerical examples for the QSVD. The resulted matrices are computed by calling SGGQSV. In Example 1, plugging the matrices into the stability expressions in (4.7)-(4.9) yields:

$$\begin{aligned} res_A &= 0.31395\ , \qquad res_B = 0.46324\ , \\ orthog_U &= 0.66301\ , \quad orthog_V = 1.05980\ , \\ orthog_Q &= 0.86827\ . \end{aligned}$$

Calculating the stability ratios using the computed matrices in Example 2 yields:

$$\begin{aligned} res_A &= 0.14286\ , \qquad res_B = 0.48853\ , \\ orthog_U &= 1.70833\ , \quad orthog_V = 0.85818\ , \\ orthog_Q &= 1.02750\ . \end{aligned}$$

Next, we follow the procedure as explained in Section 4.4.2 and use the input file described in Section 4.4.3 to perform a sequence of tests on matrices with larger sizes. Since there are twelve different combinations of dimension, eight different types of sample matrices are generated in each set of dimensions, and five stability ratio are calculated in each case, hence a total of 480 tests were run and they all yield ratios no greater than two.

4.4.6 Timing

As explained in prior sections, the QSVD algorithm can be mainly separated into four parts: pre-processing, Split QR, CSD, and post-processing. Here we first record the time spent in each of these parts and the total time spent in SGGQSV. Let t_{pre}, t_{qr}, t_{csd}, t_{post} and p_{all} denote those time measured in seconds. and p_{pre}, p_{qr}, p_{csd}, p_{post} denote percentages that each part spent relative to the whole subroutine in SGGQSV. Table 4.2 presents the results of the timing tests on samples of all possible combinations of input dimension using matrix type four. In SGGSVD, there are three major parts: pre-processing, Jacobi-rotation and post-processing. we record the time taken in each of these parts on the same set of input matrices and this result is displayed in Figure 4.3.

Comparing results in the two tables, the speed-up of SGGQSV over SGGSVD varies between 1 and 5. In the case where $[\ A^T\ \ B^T\ ]^T$ has full rank, the costs of computing the CSD and Jacobi-rotation dominate the overall computation in SGGQSV and SGGSVD, respectively. Since the Jacobi-rotation converges slowly, the overall computation SGGSVD takes much longer than SGGQSV. When $[\ A^T\ \ B^T\ ]^T$ has low rank, the Jacobi-rotation converges faster and it takes about the same amount of time as the pre-processing step. In SGGQSV, smaller part of the matrix is passed to SORCSD when $[\ A^T\ \ B^T\ ]^T$ has low-rank, hence it occupies significantly less proportion in the overall computation of SGGQSV, leaving the pre-processing step to be the major portion of the computational effort. Since the amount of improvement in SGGQSV over SGGSVD relies mostly on using CSD rather than Jacobi-rotation, the speed-up of SGGQSV in the low-rank is not is remarkable as in the full-rank case.

Figure 4.3 depicts the proportion of time spending in each of the parts in SGGQSV with respect to the increase of input dimensions on a logarithmic (base 10) scale. Both the plots on this figure and the percentage data in Table 4.2 indicate that the computation of CSD dominates the overall effort in the computation of a QSVD.

m	p	n	k	ℓ	t_{pre}	p_{pro}	t_{qr}	p_{qr}	t_{csd}	p_{csd}	t_{post}	p_{post}	t_{all}
500	500	500	0	500	0.55078	8.59%	1.61621	25.21%	3.61133	56.33%	0.63281	9.87%	6.41113
650	310	230	0	230	0.35156	32.73%	0.16309	15.18%	0.45703	42.55%	0.10254	9.55%	1.07422
430	610	210	0	210	0.32031	36.40%	0.12207	13.87%	0.35449	40.29%	0.08301	9.43%	0.87988
720	220	540	320	220	1.09668	63.09%	0.13867	7.98%	0.36719	21.12%	0.13574	7.81%	1.73828
440	180	440	260	180	0.36914	49.67%	0.07617	10.25%	0.22656	30.49%	0.07129	9.59%	0.74316
370	290	350	60	290	0.25879	16.81%	0.31543	20.49%	0.80762	52.47%	0.15723	10.22%	1.53906
250	300	300	0	300	0.10449	8.08%	0.33008	25.53%	0.72266	55.89%	0.13574	10.50%	1.29297
360	660	600	0	600	0.74609	10.03%	2.15820	29.02%	3.60547	48.48%	0.92773	12.47%	7.43750
130	520	480	0	480	0.28125	11.21%	0.52930	21.10%	1.24707	49.71%	0.45117	17.98%	2.50879
260	600	770	170	600	1.34863	27.03%	0.59082	11.84%	1.95410	39.17%	1.09570	21.96%	4.98926
370	250	700	370	250	0.53711	82.71%	0.00000	0.00%	0.02539	3.91%	0.08691	13.38%	0.64941
120	120	400	120	120	0.08496	84.47%	0.00000	0.00%	0.00391	3.88%	0.01172	11.65%	0.10059

Table 4.2: Timing Profile for SGGQSV

m	p	n	k	ℓ	t_{pre}	p_{pro}	t_{jr}	p_{jr}	t_{post}	p_{post}	t_{all}
500	500	500	0	500	0.35645	1.37%	25.62207	98.63%	0.00000	0.00%	25.97852
650	310	230	0	230	0.20898	7.02%	2.76855	92.98%	0.00000	0.00%	2.97754
430	610	210	0	210	0.19629	8.28%	2.17383	91.72%	0.00000	0.00%	2.37012
720	220	540	320	220	0.64941	19.38%	2.70215	80.62%	0.00000	0.00%	3.35156
440	180	440	260	180	0.24121	13.95%	1.48828	86.05%	0.00000	0.00%	1.72949
370	290	350	60	290	0.16992	3.16%	5.21191	96.84%	0.00000	0.00%	5.38184
250	300	300	0	300	0.06641	1.36%	4.81934	98.64%	0.00000	0.00%	4.88574
360	660	600	0	600	0.43262	1.40%	30.54395	98.60%	0.00000	0.00%	30.97656
130	520	480	0	480	0.17676	1.46%	11.94629	98.54%	0.00000	0.00%	12.12305
260	600	770	170	600	0.81543	4.47%	17.44141	95.53%	0.00000	0.00%	18.25684
370	250	700	370	250	0.53906	53.64%	0.46582	46.36%	0.00000	0.00%	1.00488
120	120	400	120	120	0.08301	56.67%	0.06348	43.43%	0.00000	0.00%	0.14648

Table 4.3: Timing Profile for SGGSVD

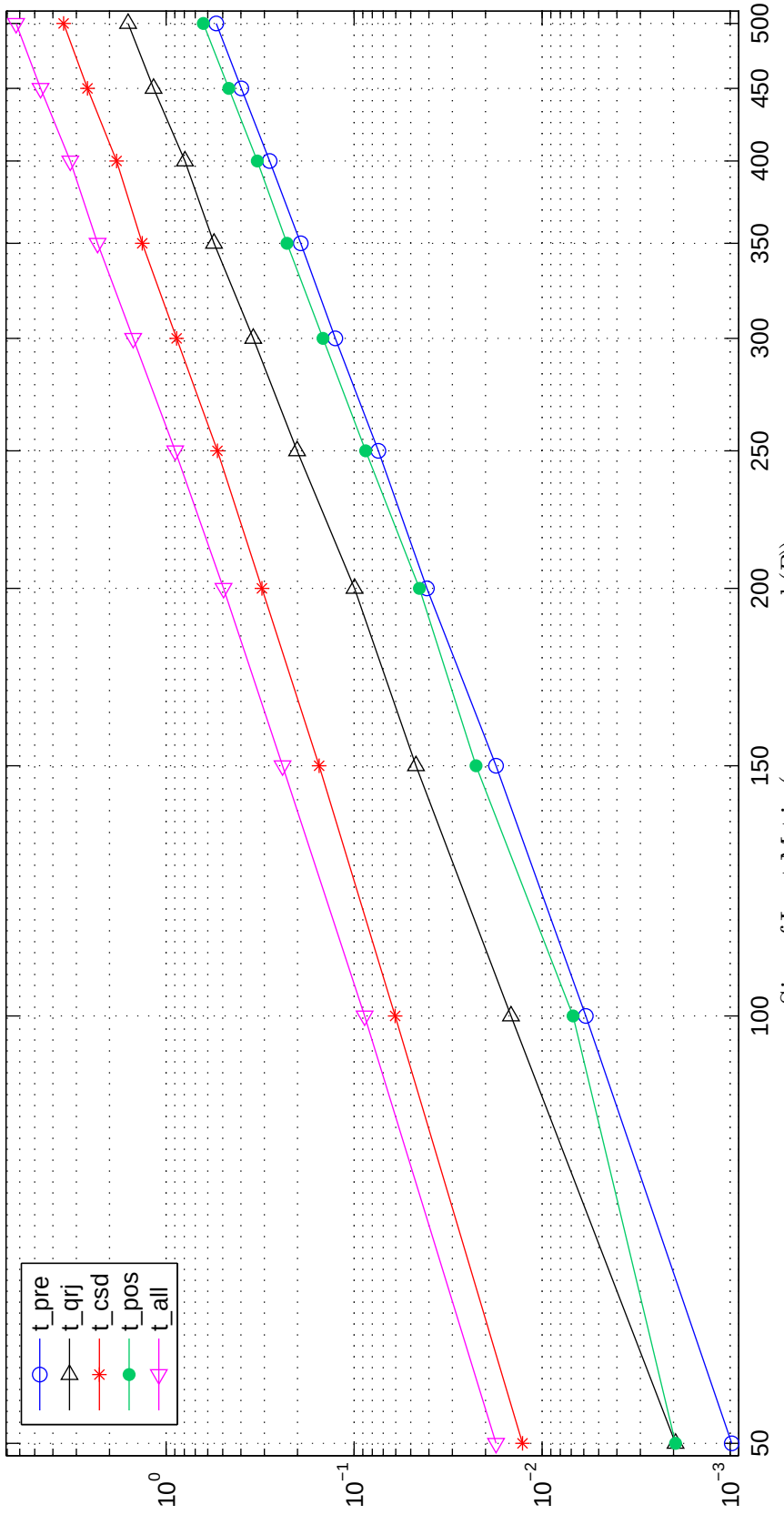


Figure 4.3: Timing Profile for SGGQSV

To analyze further, we compiled the current LAPACK subroutine, SGGSVD, and compare performance of SGGSVD with SGGQSV on square matrices. Figure 4.4 shows the tests with half rank matrices, (i.e. $\text{rank}(B) = 1/2\ m = 1/2\ n$). The first graph on this figure is the timing contest between SGGSVD and SGGQSV with square matrices. Since the data line up almost linearly on the log-log scale, they can be best approximated by the function of the form

$$log\ t = c\ log\ n + b \quad \Rightarrow \quad t = 10^b \times n^c$$

for some constant b and c where

$$c = \frac{log\ t_{i+1} - log\ t_i}{log\ n_{i+1} - log\ n_i} \text{ for some data points } (t_i, n_i) \text{ and } (t_{i+1}, n_{i+1}).$$

Following this calculation, the two plots can be best fitted by the functions $f_{gsv} \simeq 6.8466^{-8} n^{2.96}$ and $f_{qsv} \simeq 6.8466^{-8} n^{2.71}$ respectively. Thus the speed-up of SGGQSV over SGGSVD is about $n^{0.25}$, for n being the size of the input matrix. This says that the speed-up would be from $50^{0.25} \simeq 2.65$ to $500^{0.25} \simeq 4.73$ for n between 50 and 500.

The second graph on this figure plots the actual speed-up by calculating

$$\frac{\text{time in SGGSVD}}{\text{time in SGGQSV}}.$$

As the matrix grows from 50 to 500, the speed-up increases from 2.1 to 4.7. This is consistent with the prediction made by the polynomial approximations.

Next we repeat the experiments on full rank matrices (i.e. $\text{rank}(B) = m = n$). The timing comparison and the speed-up graphs are depicted in Figure 4.5. The polynomial functions that approximate the two lines are $f_{gsv} \simeq 4^{-7} n^{2.98}$ and $f_{qsv} \simeq 4^{-7} n^{2.66}$ respectively. Thus the speed-up is estimated to be $n^{0.32}$. Hence we should expect greater speed-up in SGGQSV relative to SGGSVD this time. This is confirmed by the bar graph on the bottom of the figure. With the same size of tested matrices, the speed-up now increases from 2.2 to 6.7. These results reveal that while the new

QSVD subroutine apparently outperforms the GSVD subroutine in current LAPACK, the degree of improvement varies depending not only on the size of input matrices A and B, but also the rank of B. This is because the dominant work of SGGQSV involves the call to the CSD subroutine on linearly independent columns of B, whose number is the rank of B.

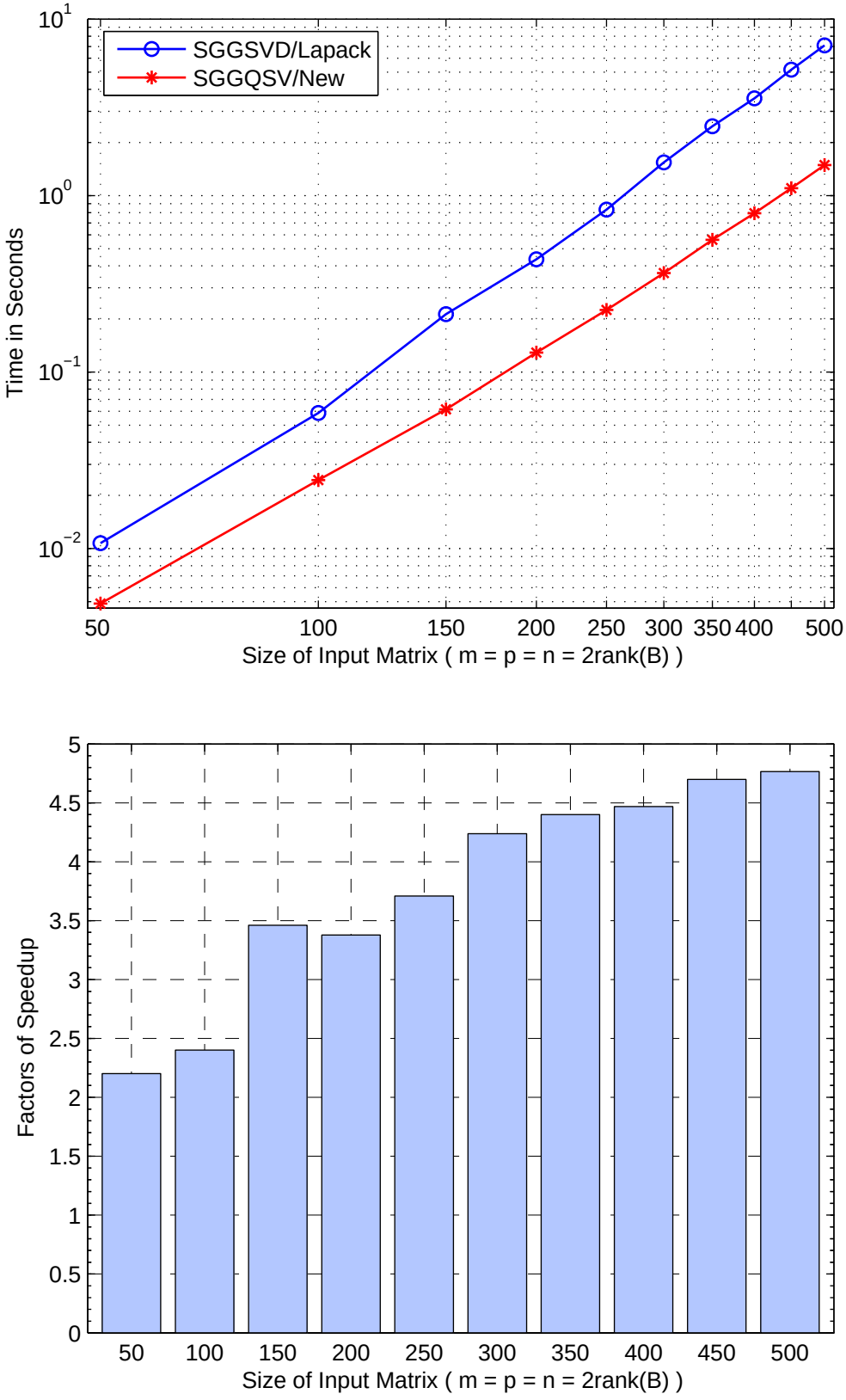


Figure 4.4: Timing Tests on SGGQSV and SGGSVD (in the half rank case)

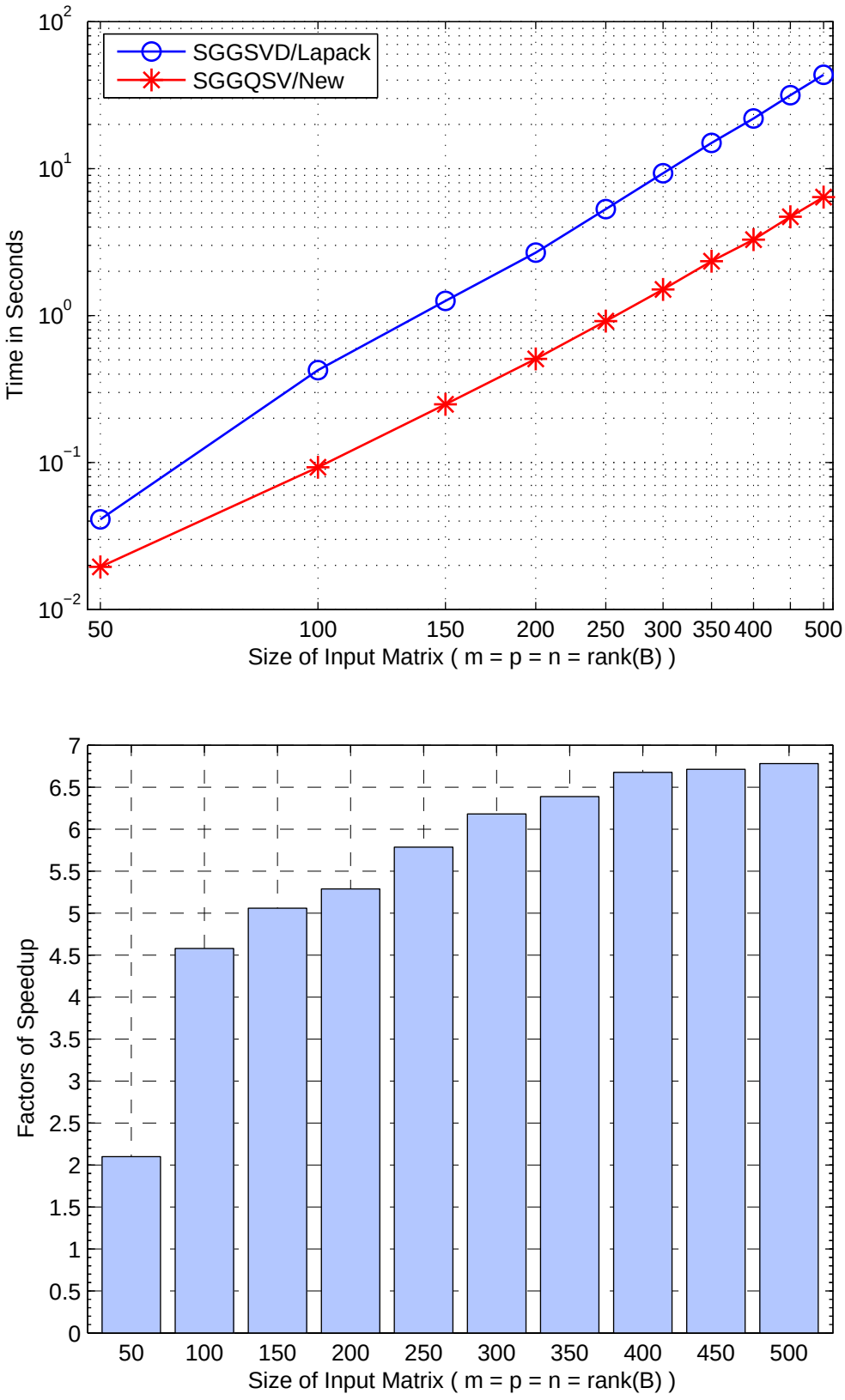


Figure 4.5: Timing Tests on SGGQSV and SGGSVD (in the full rank case)

Chapter 5

Product Singular Value Decomposition

This chapter is headed toward the product singular value decomposition (PSVD). The algorithm we develop for computing the PSVD extends the QR-like method proposed by Golub *et. al* [13]. In Section 5.1, we establish the definition and give a motivation on exploiting the QR-like method. In Section 5.2, we describe the numerical algorithms that are built up to compute the PSVD. Topics related to LAPACK-Style implementation details are discussed in Section 5.3. In Section 5.4, we present the results of the stability and timing experiments that conform the effectiveness of our software.

5.1 Definition and Basic Properties

Definition 5.1.1. Let $A \in \mathbb{R}^{m\times p}$ and $B \in \mathbb{R}^{p\times n}$ be two general matrices. A *product singular value decomposition (PSVD)* of A and B is the factorization

$$AB = U\Sigma V^T$$

where

$$\Sigma = \begin{array}{c} \overset{n}{} \\ \left[\begin{array}{ccccc} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & \sigma_{r+1} & \\ & & & & \ddots \\ & & & & & \sigma_n \\ \hline & & & O & \end{array}\right] \end{array} \begin{array}{l} n \\ \\ \\ \\ \\ \\ \\ m-n \end{array} \qquad \text{if } m \geq n, \tag{5.1}$$

or

$$\Sigma = \begin{array}{c} \overset{m}{} \quad \overset{n-m}{} \\ \left[\begin{array}{ccccc|c} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & \sigma_{r+1} & & O \\ & & & & \ddots & \\ & & & & & \sigma_m \end{array}\right] \end{array} \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ m \end{array} \qquad \text{if } m < n. \tag{5.2}$$

In those two cases above, r is the numerical rank of the product AB. The diagonal entries, σ_i, are the *singular values* of the product AB, conventionally sorted in non-increasing order. Moreover, $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices. The column vectors in U are the *left singular vectors* of the product AB, and the row vectors in V^T are the *right singular vectors* of the product AB.

Example 1. Consider a 4–by–3 matrix A and a 3–by–5 matrix B such that

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 2 \\ 3 & 2 & 1 \\ 4 & 3 & 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -1 & 0 & 1 & 2 & 3 \\ -2 & -1 & 0 & 1 & 2 \\ -3 & -2 & -1 & 0 & 1 \end{bmatrix}.$$

Since $m < n$, Σ follows the structure in (5.2):

$$\Sigma = \begin{bmatrix} 42.3009148 & 0 & 0 & 0 & 0 \\ 0 & 7.1156459 & 0 & 0 & 0 \\ 0 & 0 & 0.0000001 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The computed orthogonal matrices, U and V are:

$$U = \begin{bmatrix} 0.4395704 & 0.8225173 & -0.3262416 & -0.1543034 \\ 0.3727753 & 0.1631000 & 0.9134750 & 0.0000000 \\ 0.4550903 & -0.4310786 & -0.1087469 & -0.7715167 \\ 0.6787555 & -0.3332188 & -0.2174941 & 0.6172134 \end{bmatrix},$$

$$V = \begin{bmatrix} -0.5979241 & -0.4924296 & 0.3389115 & 0.4467183 & 0.2925436 \\ -0.2825492 & -0.4692185 & -0.0952533 & -0.3210523 & -0.7667153 \\ 0.0328255 & -0.4460072 & -0.2013745 & -0.6643083 & 0.5640416 \\ 0.3482003 & -0.4227961 & -0.6671374 & 0.5049002 & 0.0018881 \\ 0.6635751 & -0.3995850 & 0.6248535 & 0.0337421 & -0.0917580 \end{bmatrix}.$$

Example 2. Consider a 5–by–4 matrix A and a 4–by–3 matrix B such that

$$A = \begin{bmatrix} 1 & -2 & 3 & -4 \\ -2 & -1 & 2 & -3 \\ 3 & 2 & 1 & -2 \\ 4 & -3 & -2 & -1 \\ 1 & -4 & 3 & 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 4 & 6 \\ 4 & 2 & 5 \\ 6 & 5 & 3 \\ 1 & 7 & 6 \end{bmatrix}.$$

Since $m \geq n$, Σ follows the structure in (5.1):

$$\Sigma = \begin{bmatrix} 52.0377426 & 0 & 0 \\ 0 & 26.8254585 & 0 \\ 0 & 0 & 16.5972614 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The computed orthogonal matrices, U and V are:

$$U = \begin{bmatrix} 0.4162883 & -0.3860091 & 0.0953318 & 0.7145372 & -0.3975542 \\ 0.6702271 & -0.3075286 & 0.1539959 & -0.2355159 & 0.6140354 \\ -0.3980060 & -0.4171929 & -0.5415683 & 0.3556824 & 0.4977299 \\ 0.1864332 & 0.7498037 & -0.0337425 & 0.5102614 & 0.3762071 \\ -0.4293410 & -0.1420120 & 0.8202208 & 0.2170076 & 0.2750352 \end{bmatrix},$$

$$V = \begin{bmatrix} -0.1365775 & -0.9818467 & -0.1316192 \\ -0.6593506 & -0.0090608 & 0.7517810 \\ -0.7393263 & 0.1894597 & -0.6461437 \end{bmatrix},$$

In both examples, one may verify that within seven digits of accuracy, the computed results satisfy the properties as in the PSVD definition.

In general, the PSVD can be defined for the product of any number of matrices $A_1, A_2, \cdots, A_k$ (for $k \geq 2$), as long as their dimensions are compatible with matrix multiplication. We concern with the case where $k = 2$ here because the algorithm is similar when $k \geq 2$. Henceforth, we notate the two input matrices A and B. Naturally, one can compute the PSVD of A and B by first multiplying those two input matrices followed by performing the SVD on the product. However, this naive approach does not serve as a good numerical algorithm because (1) forming the product AB may introduce catastrophic round off errors, and (2) if A and B have wide range of singular values, then the small singular value of AB may not be computed accurately.

Recently, papers are written on how to compute the PSVD without explicitly forming the product of A and B. Heath *et. al.* devised an algorithm that is based on the Jacobi method [16] while Golub *et. al* proposed a QR-like method [13]. None of these idea is yet realized into LAPACK.

In this chapter, we choose to generalize the QR-like method for a pair of matrices with any compatible dimensions and implement it in LAPACK style. The reason for choosing the QR-like method is twofold: (1) it has lower complexity than the Jacobi method and (2) some of the computational subroutines required by this algorithm are available in current LAPACK.

5.2 Numerical Algorithms

The PSVD can be seen as the SVD extended to two matrices. One of the efficient methods of computing the SVD for a single matrix, A, is to bidiagonalize it followed by computing the SVD on the bidiagonal matrix reduced from A. Similarly, the QR-like method first bidiagonalizes the product AB by applying a series of Householder transformations to the left and right of A and B. Once the bidiagonalized form, G, is obtained, the SVD of G is then be computed.

Thus the primary objective here is to explain the bidiagonalization process on A and B. Golub *et al* elucidated their QR-like idea in a simple case where the given matrices are square [13]. Here we extend his idea and consider a more general case where A and B are rectangular ("tall" or "flat"). Let G denote the bidiagonalization matrix reduced from AB; it turns out that there are two cases to deal with: (1) when $m \geq n$, G needs to be upper-bidiagonal; (2) when $m < n$, G needs to be lower-bidiagonal. The necessity for the specific upper- and lower-bidiagonalized form would become evident as we describe the second major step of the PSVD algorithm.

In the bidiagonalization processes below, $\mathcal{H}_{ij}$ denotes a set of Householder transformations that annihilates the i-th row or column of the given matrix except the first j elements.

5.2.1 Upper-Bidiagonalization Process

Now let us explain the bidiagonalization process of the first case (i.e. $m \geq n$) by using an example where $m = 6$, $p = 4$, $n = 5$.

First, find a Householder transformation $\mathcal{Q}_1^{(1)} \in \mathcal{H}_{11}$ so that $\mathcal{Q}_1^{(1)T} B$ has its 2 to p elements in its first column annihilated. To maintain equivalence, we must multiply $\mathcal{Q}_1^{(1)}$ to the left of A, hence

$$(A\mathcal{Q}_1^{(1)})(\mathcal{Q}_1^{(1)T} B) = \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{A\mathcal{Q}_1^{(1)}} \underbrace{\begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}}_{\mathcal{Q}_1^{(1)T} B}. \tag{5.3}$$

Once we have updated $A := A\mathcal{Q}_1^{(1)}$, and $B := \mathcal{Q}_1^{(1)T} B$, we find a Householder transformation $\mathcal{Q}_1^{(2)} \in \mathcal{H}_{11}$ so that $\mathcal{Q}_1^{(2)T} A$ has its 2 to m elements in its first column annihilated:

$$\mathcal{Q}_1^{(2)T} AB = \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}}_{\mathcal{Q}_1^{(2)T} A} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}.$$

Again, we update $A := \mathcal{Q}_1^{(2)T} A$. Since the first column of both A and B are zeros

except their (1,1) elements, the product of AB preserves this structure:

$$AB = \begin{bmatrix} \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}. \tag{5.4}$$

At this point, we are only interested in zeroing out the 3 to n elements in the first row of the product AB. Hence we multiply the first row of A to the entire matrix B: $A(1, 1 : p)B$; this result is denoted by $\hat{\times}$ in (5.4) above. Once this vector is constructed, we can figure out a Householder transformation $\mathcal{Q}_3^{(1)} \in H_{12}$ such that $A(1, 1 : p)B\mathcal{Q}_3^{(1)} = [\ \times\ \ \times\ \ 0\ 0\ ]$. Note that this does not affect the zero elements we introduced to the first column of AB since

$$AB\mathcal{Q}_3^{(1)} = \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}.$$

Clearly, this is an iterative algorithm in which the core of each iteration is to compute and apply the Householder transformations, one for each of A, B, and AB. Thus we refer to such group of transformations as a *triad*. Updating $B := B\mathcal{Q}_3^{(1)}$ and we are done with the first triad in the bidiagonalization process.

The second triad is rather similar with the first one, except that the transformations are applied from the second, instead of the first, column of A, B and the second row of AB. In particular, we find two Householder transformations $\mathcal{Q}_1^{(2)}, \mathcal{Q}_2^{(2)} \in \mathcal{H}_{22}$ that eliminate the second column in A and B in the fashion described above. Simi-

larly, the 4-th to n elements in the second rows of the product AB is eliminated by choosing a $\mathcal{Q}_3^{(2)} \in \mathcal{H}_{23}$. After all the updating of A and B are done, the end of the second triad should yield the following product:

$$AB = \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}. \tag{5.5}$$

At this stage, we do not need to perform any transformation on the right of AB since the elements above the first upper diagonal of AB are zeroed-out by the first two triads. However, we still need to iterate two more times in order to find $\mathcal{Q}_1^{(3)}, \mathcal{Q}_1^{(3)} \in \mathcal{H}_{33}$ and $\mathcal{Q}_2^{(4)}, \mathcal{Q}_2^{(4)} \in \mathcal{H}_{44}$ so that the rest of the elements below the main diagonal of AB in (5.5) are eliminated. After the last iteration, the product of AB is completely bidiagonalized:

$$AB = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = G.$$

This algorithm is summarized below. The function `house` in the algorithm takes in a row or column vector x as input and outputs the Householder matrix Q that annihilates all but the first entries of x. The implementation of `house` can be found in standard methods for computing the QR decomposition using Householder transformation and therefore detail is omitted here.

Algorithm 5.2.1. Reducing AB into upper-bidiagonal form.

Input: $A \in \mathbb{R}^{m\times p}, B \in \mathbb{R}^{p\times n}$ where $m, n, p > 0$ and $m \geq n$.

Output: U, V and G such that $U^T ABV = G$ where G is upper-bidiagonal, U and V are orthogonal.

1: Set $j = \min(m-1, p, n)$, % number of transformations needed to eliminate entries in A.
2: Set $k = \min(m\text{–}1, p\text{–}1, n)$, % number of transformations needed to eliminate entries in B.
3: Set $\ell = \min(m, p, n-2)$, % number of transformations needed to eliminate entries in AB.
4: Set $U = I$ and $V = I$.
5: **for all** i such that $1 \leq i \leq \max(j, \ell)$ **do**
6: **if** $i \leq k$ **then**
7: $Q_1 = \texttt{house}(B_{(i:p,i)})$
8: Update B: $B := Q_1^T B$
9: Update A: $A := AQ_1$
10: **end if**
11: **if** $i \leq j$ **then**
12: $Q_2 = \texttt{house}(A_{(i:m,i)})$
13: Update A: $A := Q_2^T A$
14: Update U: $U := UQ_2$.
15: **end if**
16: **if** $i \leq \ell$ **then**
17: $t = A_{(i,:)}B$
18: $Q_3 = \texttt{house}(t_{(2:p)})$
19: Update B: $B := BQ_3$
20: Update V: $V := VQ_3$
21: **end if**
22: **end for**

We may verify the correctness of this algorithm by explicitly writing out the sequence of the Householder transformations and multiply that with the original input A and B:

$$\begin{aligned} G &= \underbrace{\mathcal{Q}_2^{(j)T} \cdots \mathcal{Q}_2^{(2)^T} \mathcal{Q}_2^{(1)^T}}_{U^T} A(\mathcal{Q}_1^{(1)} \mathcal{Q}_1^{(2)} \cdots \mathcal{Q}_1^{(k)})(\mathcal{Q}_1^{(k)T} \cdots \mathcal{Q}_1^{(2)T} \mathcal{Q}_1^{(1)T}) B \underbrace{\mathcal{Q}_3^{(1)} \mathcal{Q}_3^{(2)} \cdots \mathcal{Q}_3^{(\ell)}}_{V} \\ &= U^T ABV \end{aligned}$$

5.2.2 Lower-Bidiagonalization Process

Now we turn to the second case of the bidiagonalization process where $m < n$. Algorithm for this case is rather symmetric with the first one. Let us explain it by using an example where $m = 4$, $p = 6$, $n = 5$. Since we are interested in reducing AB into lower-bidiagonal form here, we need to perform Householder transformation on the rows of A and B, followed by reducing the columns of AB. We now explain the first triad in detail.

First, find a Householder transformation $\mathcal{Q}_1^{(1)} \in \mathcal{H}_{11}$ such that it annihilates the 2 to p entries in the first row of A. This transformation operates on the right of A. Certainly, the operation also needs to apply on the left of B to maintain equivalence:

$$(A\mathcal{Q}_1^{(1)})(\mathcal{Q}_1^{(1)T} B) = \underbrace{\begin{bmatrix} \times & 0 & 0 & 0 & 0 & 0 \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix}}_{A\mathcal{Q}_1^{(1)}} \underbrace{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}}_{\mathcal{Q}_1^{(1)T} B}.$$

Let us update A and B by setting $A := A\mathcal{Q}_1^{(1)}$ and $B := \mathcal{Q}_1^{(1)T} B$. Now find a Householder transformation $\mathcal{Q}_2^{(1)} \in \mathcal{H}_{11}$ such that it annihilates the 2 to n elements

in the first row of B:

$$AB\mathcal{Q}_2^{(1)} = \begin{bmatrix} \times & 0 & 0 & 0 & 0 & 0 \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix} \underbrace{\begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}}_{B\mathcal{Q}_2^{(1)}}.$$

It follows that B is updated into $B := B\mathcal{Q}_2^{(1)}$ and note that multiplying A and B does not destroy the zeros we introduced:

$$AB = \begin{bmatrix} \hat{\times} & 0 & 0 & 0 & 0 \\ \hat{\times} & \times & \times & \times & \times \\ \hat{\times} & \times & \times & \times & \times \\ \hat{\times} & \times & \times & \times & \times \end{bmatrix}. \tag{5.6}$$

Now we are interested in zeroing out the 3 to m elements in the first columns of AB. The elements marked by $\hat{\times}$ in (5.6) are computed by multiplying the entire A to the first column in B: $AB(1:p,\ 1)$. Then the desired entries in this column can be eliminated by applying a Householder transformation $\mathcal{Q}_3^{(1)} \in \mathcal{H}_{12}$ on the left:

$$\mathcal{Q}_3^{(1)T} AB = \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}.$$

It should now be apparent how to proceed further with this algorithm. After the

last triad, this algorithm should yield the lower bidiagonal form of AB:

$$AB = \begin{bmatrix} \times & 0 & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 & 0 \\ \times & \times & \times & \times & 0 & 0 \end{bmatrix} \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ \times & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & 0 \end{bmatrix}.$$

The algorithm is summarized below. It is similar with Algorithm 5.2.1, except there are two essential differences: (1) the order in which the entries in A and B are annihilated; (2) the number of Householder transformations that needs to be generated and operated on appropriate rows or columns of A, B, or AB.

Algorithm 5.2.2. Reducing AB into lower-bidiagonal form.

Input: $A \in \mathbb{R}^{m \times p}, B \in \mathbb{R}^{p \times n}$ where $m, n, p > 0$ and $m < n$.

Output: U, V and G such that $U^T ABV = G$ where G is lower-bidiagonal, U and V are orthogonal.

1: Set $j = \min(m, p-1)$, $k = \min(m, p)$, and $\ell = \min(m-2, p, n)$.
2: Set $U = I$ and $V = I$.
3: **for all** i such that $1 \leq i \leq \max(j, \ell)$ **do**
4: **if** $i \leq j$ **then**
5: $Q_1 = \text{house}(A_{(i,i:p)})$
6: Update A: $A := AQ_1$
7: Update B: $B := Q_1^T B$
8: **end if**
9: **if** $i \leq k$ **then**
10: $Q_2 = \text{house}(B_{(i,i:n)})$
11: Update B: $B := BQ_2$

12: Update U: $U := UQ_2$.

13: **end if**

14: **if** $i \leq \ell$ **then**

15: Set $t = AB(:,i)$

16: $Q_3 = \text{house}(t(2{:}m))$

17: Update A: $A := Q_3^T A$

18: Update V: $V := VQ_3$

19: **end if**

20: **end for**

We may verify the correctness of this algorithm by explicitly writing out the sequence of the Householder transformations and multiplying them with the original input A and B:

$$\begin{aligned} G &= \underbrace{\mathcal{Q}_2^{(\ell)^T} \cdots \mathcal{Q}_2^{(2)^T} \mathcal{Q}_2^{(1)^T}}_{U^T} A(\mathcal{Q}_1^{(1)} \mathcal{Q}_1^{(2)} \cdots \mathcal{Q}_1^{(j)})(\mathcal{Q}_1^{(j)T} \cdots \mathcal{Q}_1^{(2)T} \mathcal{Q}_1^{(1)T}) B \underbrace{\mathcal{Q}_3^{(1)} \mathcal{Q}_3^{(2)} \cdots \mathcal{Q}_3^{(k)}}_{V} \\ &= U^T ABV \end{aligned}$$

Note that both Algorithm 5.2.1 and 5.2.2 never need to form the product AB explicitly; rather, just one row of A is multiplied with B or the entire A is multiplied with one column of B. Hence the work space can be kept at linear order and the work effort for matrix-vector multiplication is quadratic as opposed to cubic in normal matrix-matrix multiplication.

5.2.3 Computing the PSVD

Now that we have the product AB implicitly diagonalized, next step is to compute the SVD of the bidiagonal matrix. This task deserves an extensive study on its own and there are at least two efficient algorithms developed currently: one uses QR iterations [7] and other uses the divide-and-conquer algorithm [15].

Once the product AB is reduced to a bidiagonal form, $AB = U_1 G V_1^T$, using the

algorithm described in previous subsections, we perform the SVD of G: $G = U_2 \Sigma V_2^T$. The PSVD of A and B is then computed by setting $U = U_1 U_2$, $V = V_1 V_2$.

We now justify the requirement for the lower-bidiagonal form in the case where $m < n$. Supposed one computed the upper-bidiagonal form, then using the example illustrated earlier where $m = 4, p = 6$ and $n = 5$, the following bidiagonalized matrix would result:

$$\tilde{G} = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

Since $\tilde{G}(4,5) \neq 0$ and the subroutine on computing the SVD of a bidiagonal matrix expects the input to be a square matrix, thus we must add an extra row of zero to $\tilde{G}$:

$$\tilde{G}' = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Besides the extra work space on constructing $\tilde{G}'$ from $\tilde{G}$, the SVD of $\tilde{G}'$ would compute $m+1$ left singular vectors, which is really unnecessary. Hence, for efficiency reason, the product of AB is decomposed into lower-bidiagonal whenever $m < n$. Lastly, using this upper- and lower-diagonal policy keeps the off-diagonal entries consistently having length of min $(m - n) - 1$.

Putting everything together, we now summarize the overall algorithm for computing PSVD of two general matrices with any compatible dimensions.

Algorithm 5.2.3. Computing PSVD of two general matrices

Input: $A \in \mathbb{R}^{m\times p}, B \in \mathbb{R}^{p\times n}$ where $m, n, p > 0$

Output: U, V and Σ such that $U^T ABV = \Sigma$ where Σ is diagonal and consists of non-negative entries, U and V are orthogonal.

1: **if** $m \geq n$ **then**
2: apply Algorithm 5.2.1 to bidiagonalize AB: $AB = U_1 G V_1^T$.
3: **else**
4: apply Algorithm 5.2.2 to bidiagonalize AB: $AB = U_1 G V_1^T$.
5: **end if**
6: apply either the QR or divide-and-conquer algorithm on computing the SVD of G: $G = U_2 \Sigma V_2^T$.
7: Set $U = U_1 U_2$ and $V = V_1 V_2$.

5.3 LAPACK-Style Software

In this section, we discuss the details of the implementation of the three algorithms described in the previous three sections. we also analyze the work space requirement for those subroutines.

5.3.1 Implementation Details

All three algorithms described in the previous section are now implemented in FORTRAN 77 using real data, conforming to LAPACK style. The bidiagonalization processes described in Algorithm 5.2.1 and 5.2.2 are combined into the subroutine named SGGBRD so callers do not need to worry the implementation difference between $m \geq n$ and $m < n$. None of the Householder matrices are explicitly generated in each triad, but the vector representations of the Householder matrix U and V are

successively stored in the triangular part of A, B and two linear vector TAUP and TAUQ. This is analogous to the storage scheme used in SGEBRD, which is the subroutine in LAPACK that bidiagonalizes a single matrix. In the first example where $m = 6, p = 4$ and $n = 5$, upon return from SGGBRD, the original contents of A and B will be destroyed and overwritten by the following:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ v_1 & 0 & 0 & 0 \\ v_1 & v_2 & 0 & 0 \\ v_1 & v_2 & v_3 & 0 \\ v_1 & v_2 & v_3 & v_4 \\ v_1 & v_2 & v_3 & v_4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & 0 & u_1 & u_1 & u_1 \\ 0 & 0 & 0 & u_2 & u_2 \\ 0 & 0 & 0 & 0 & u_3 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

A similar storage scheme is applied to the case where $m < n$. In the example where $m = 4$, $p = 6$ and $n = 5$, A and B look like the following upon exit:

$$A = \begin{bmatrix} 0 & v_1 & v_1 & v_1 & v_1 & v_1 \\ 0 & 0 & v_2 & v_2 & v_2 & v_2 \\ 0 & 0 & 0 & v_3 & v_3 & v_3 \\ 0 & 0 & 0 & 0 & v_4 & v_4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ u_1 & 0 & 0 & 0 & 0 \\ u_1 & u_2 & 0 & 0 & 0 \\ u_1 & u_2 & u_3 & 0 & 0 \\ u_1 & u_2 & u_3 & u_4 & 0 \end{bmatrix}.$$

The main diagonal entries of G are stored in a vector $\mathcal{D}$ with length $\min(m - n)$ and the upper- or lower-diagonal entries of G are stored in a vector $\mathcal{E}$ with length $\min(m - n) - 1$ upon returned. The orthogonal matrices U and V can be explicitly formed by calling the LAPACK subroutines, SORGBR, with proper input.

Algorithm 5.2.3 is written in a subroutine named SGGPSV. Assume that all the m left singular vectors and n right singular vectors are desired, this subroutine first calls SGGBRD to bidiagonalize the two input matrices. Next, there are two methods available in current LAPACK that compute the SVD of a bidiagonal matrix: SBD-

SQR [7, 10, 12] and SBDSC [15, 18]. The subroutine SBDSQR uses QR iterations when singular vectors are desired, otherwise uses the dqds algorithm. The subroutine SBDSDC uses a variation of Cuppen's divide-and-conquer algorithm to compute the singular vectors and singular values. Both subroutines are about equally accurate. Our SGGPSV chooses the former subroutine since it has much lower workspace requirement, trading off with speed. One can be easily modified SGGPSV to use the latter method for efficiency concern if necessary.

5.3.2 Work Space Requirement

The transformations in SGGBRD are done by successively calling SLARFG to generate an elementary reflector H of appropriate order followed by calling SLARF to apply H to a desired matrix from either the left or the right hand side. The subroutine SLARF requires a work space of $\min(m, n, p)$ to accomplish its task in SGGBRD. Also, we need $\min(m, n)$ of work space to save the result from a matrix-vector multiply. Thus the total work space for SGGBRD amounts to $\max(m, n, p) + \min(m, n)$.

When calling SGGBRD inside SGGPSV, it must allocate enough space to store TAUP, TAUQ, and the off-diagonal entries that are returned from SGGBRD, in addition to the size of work space needed by SGGBRD. Hence SGGPSV needs a total of $\max(m, n, p) + 4 \times \min(m, n)$ just to complete the bi-diagonalization process. After that, it is sufficient to re-use those work space to accomplish the rest of the tasks in SGGPSV. Therefore, the total work space required by SGGPSV is

$$\mathcal{L}_w = \max(m, n, p) + 4 \times \min(m, n).$$

5.4 Testing and Timing

This section devotes to the testing and timing of the PSVD subroutine. We have developed a rich set of testing cases which include various types of matrices in all possible combinations of input sizes.

5.4.1 Notion of Stability

Suppose A, B are two input matrices and the matrices computed by SGGBRD are $\widetilde{P}$, $\widetilde{Q}$, $\widetilde{D}$ such that $\widetilde{D}$ is bidiagonal and that $\widetilde{P}\ \widetilde{D}\ \widetilde{Q}^T = AB$. Subsequently, the matrices computed by SGGPSV are $\widetilde{U}, \widetilde{V}, \widetilde{\Sigma}$ such that $\widetilde{U}\widetilde{\Sigma}\widetilde{V}^T = AB$ and Σ is diagonal. The following ratios are recorded in each of the stability tests:

$$res_G \ = \ \frac{\|\widetilde{U}S\widetilde{V}^T - AB\|_1}{\|AB\|_1\ max(m,n)\ \epsilon} \ , \ orthog_U = \frac{\|I - \widetilde{U}^T\widetilde{U}\|_1}{m\ \epsilon} \ , \ orthog_V = \frac{\|I - \widetilde{V}^T\widetilde{V}\|_1}{n\ \epsilon}.$$

and

$$res_D \ = \ \frac{\|\widetilde{Q}\widetilde{D}\widetilde{P}^T - AB\|_1}{\|AB\|_1\ max(m,n)\ \epsilon} \ , \ orthog_Q = \frac{\|I - \widetilde{Q}\widetilde{Q}^T\|_1}{m\ \epsilon} \ , \ orthog_P = \frac{\|I - \widetilde{P}\widetilde{P}^T\|_1}{n\ \epsilon}.$$

Note that the quantities $resG$ and $resD$ are measured relative to $\|AB\|$ instead of $\|A\|\ \|B\|$. With the formulation of the PSVD, it is non-trivial to derive an expression in terms of the product $\|A\|\ \|B\|$. Also, since $\|AB\| \leq \|A\|\|B\|$, those residual quantities we measure provide upper bounds on the error relative to the product $\|A\|\ \|B\|$.

5.4.2 Testing Routines

We modify the main test subroutine, SCHKEE2, to enable the call to SCKPSV. Like SCKQSV, SCKPSV is constituted by a doubly-nested loop. In each iteration, the outer loop retrieves the dimension triple (m, p, k) while the inner loop generates a m-by-k matrix and a p-by-n matrix of one of the specified types followed by calling SPSVTS to compute the test ratios. The sequence of this testing procedure is drawn below.

$$\boxed{SCHKEE2} \longrightarrow \boxed{SCKPSV} \longrightarrow \boxed{SPSVTS} \longrightarrow \boxed{SGGPSV}$$

5.4.3 Input File Format

An annotated example of an input file for testing of the PSVD subroutine is shown below:

```
PSV: Data file for testing Product SVD subroutine
12                                      Number of values of M, P, N
30 15 30  15 71 57  10 44 40  13 20 38 Values of M (row dimension)
16 23 16  7  38 26  98 70 62  38 40 22 Values of P
8  7  16  9  40 57  11 57 60  77 60 47 Values of N (column dimension)
2.0                                     Threshold value of test ratio
T                                       Put T to test if error exits
1                                       Code to interpret the seed
PSV 8                   List types on next line if 0 < NTYPES < 8
```

The first line of the input files must contain the character PSV in columns 1-3.

Lines 2-9 are read using list-directed input and specify the following values:

line 2: The number of values m, p, and n

line 3: Values of m (row dimension)

line 4: Values of p (column and row dimension)

line 5: Values of n (column dimension)

line 6: The threshold value for the test ratios.

We may set this one to 0 in order to display all the test ratios.

line 7: Flag to test if error exists

line 8: An integer code to interpret the random number seed.

= 0: Set the seed to a default value before each run.

= 1: Initialize the seed to a default value only before the first run.

= 2: Like 1, but use the seed values on the next line.

line 9: If line 8 was 2, four integer values for the random seed.

Otherwise, the path PSV followed by the number of matrix types.

5.4.4 Types of Test Matrices

Recall in the previous chapter that we had eight types of matrices for testing the QSVD subroutine. Those types of matrices are used again here to test the PSVD subroutine. The properties of those types of matrices are reminded below.

Matrix Types	Matrix A	Matrix B	$\|A\|$	$\|B\|$	$\kappa(A)$	$\kappa(B)$
1	Diagonal	Upper triangular	10	1000	100	10
2	Upper triangular	Upper triangular	10	1000	100	10
3	Lower triangular	Upper triangular	10	1000	100	10
4	Random Dense	Random Dense	10	1000	100	10
5	Random Dense	Random Dense	10	1000	$\sqrt{0.1/\epsilon}$	$\sqrt{0.1/\epsilon}$
6	Random Dense	Random Dense	10	1000	$0.1/\epsilon$	$0.1/\epsilon$
7	Random Dense	Random Dense	10	1000	$\sqrt{0.1/\epsilon}$	$0.1/\epsilon$
8	Random Dense	Random Dense	10	1000	$0.1/\epsilon$	$\sqrt{0.1/\epsilon}$

5.4.5 Stability Testing

Toward the end of Section 5.1, we illustrated two numerical examples for the PSVD. The resulted matrices are computed by calling SGGPSV. In Example 1, plugging the matrices into the stability expressions in Section 5.4.1 yields:

$$res_G = 0.33069\ , \quad res_U = 0.92496\ , \quad orthog_V = 0.42500\ ,$$
$$orthog_D = 0.20668\ , \quad orthog_Q = 1.01328\ , \quad orthog_P = 0.20000\ .$$

Likewise, calculating the stability ratios using the matrices in Example 2 gives:

$$res_G = 1.07899\ , \quad res_U = 0.84377\ , \quad orthog_V = 0.46354\ ,$$
$$orthog_D = 1.08332\ , \quad orthog_Q = 0.20000\ , \quad orthog_P = 0.08333\ .$$

To test the stability of the software more seriously, we follow the procedure mentioned in Section 5.4.2 and use the input file described in 5.4.3. For each triple (m, p, n), eight types of matrices are generated. Every possible combination of the (m, p, n) triple is tested. Altogether, a total of 480 tests are run and all of them result

in residual ratios no greater than two.

5.4.6 Timing

As explained in prior sections, the PSVD algorithm mainly composed of a bidiagonalization process and a SVD computation. We now time those tasks in SGGPSV. Specifically, we measure the following quantities:

t_{brd}: time spent in SGGBRD,

t_{gen}: time spent in generating the two orthogonal matrices returned from SGGBRD,

t_{sbd}: time spent in SBDSQR.

t_{all}: time spent in the overall SGGPSV.

Also, let p_{brd}, p_{gen} and p_{sbd} be the percentage of each of these tasks spent relative to the overall time. Table 5.1 below presents the results of those timing tests using test matrices of type four with various of dimensions.

m	p	n	t_{brd}	p_{brd}	t_{gen}	p_{gen}	t_{sbd}	p_{sbd}	t_{all}
300	160	80	0.01855	33.93%	0.01855	33.93%	0.01660	30.36%	0.05469
150	230	70	0.01465	46.88%	0.00391	12.50%	0.01172	37.50%	0.03125
300	160	160	0.03027	24.41%	0.03223	25.98%	0.06055	48.82%	0.12402
150	70	90	0.00391	20.00%	0.00586	30.00%	0.00977	50.00%	0.01953
710	380	400	0.41992	30.67%	0.36133	26.39%	0.58008	42.37%	1.36914
570	260	570	0.22559	27.34%	0.32129	38.93%	0.27148	32.90%	0.82520
100	980	110	0.09180	80.34%	0.00391	3.42%	0.01758	15.38%	0.11426
440	700	570	1.15430	53.24%	0.26660	12.30%	0.74121	34.19%	2.16797
400	620	600	0.89746	50.30%	0.26465	14.83%	0.61621	34.54%	1.78418
130	380	770	0.21387	47.51%	0.17090	37.96%	0.05957	13.23%	0.45020
200	400	600	0.24902	46.79%	0.14941	28.07%	0.12988	24.40%	0.53223
380	220	470	0.11523	27.57%	0.14746	35.28%	0.15137	36.21%	0.41797

Table 5.1: Timing Profile for the SGGPSV

As suggested by the timing results, the bidiagonal process accounts between 30% and 80% of the total work in SGGPSV, depending on the size of the input matrices. On average, each of the three tasks takes about the same portion of time.

Chapter 6

Conclusion

While LAPACK continues to thrive as the leading linear algebra library, more new algorithms are added to the family and existing algorithms are constantly tuned up for modern high-performance computers. This thesis developed three main algorithms for computing the generalized singular value problem, which are implemented in LAPACK style and are nourished by the specialized subroutines in LAPACK and various levels of BLAS. We provided in-depth explanation and analysis for each of these computations whose efficiency and stability were attested.

First, we considered the cosine-sine decomposition on a partitioned matrix with orthonormal columns. The algorithm and software we developed stemmed from Von Loan's idea. This algorithm is stable because it uses the QR decomposition on a well-conditioned matrix and otherwise computes the SVD, which both are stable orthogonal transformations. The work space requirement is quadratic. Through numerical experiments on all possible combination of input sizes, we confirmed that our software computes the CSD correctly within roundoff error. Meanwhile, tests revealed that more than 50% of the overall effort of our CSD software accounts for the computation of the SVD. This suggests that obvious improvement is anticipated for the CSD subroutine when we exploit a faster version for computing the SVD.

Second, we focused on the quotient singular value decomposition for any two matrices having the same number of columns. We solved this decomposition based on the computation of the CSD. This method throws away the convergent issue

possesses in its counterpart of the Jacobi-like method. The order of work space requirement for our QSVD software is quadratic, inherited from the CSD subroutine. Although this method demands higher order of work space, we have demonstrated in a wide range of numerical experiments that it is significantly faster while the relative residuals are maintained within satisfactory roundoff level. Specifically, the speed-up of our software compared to the Jacobi-like implementation in current LAPACK grows polynomially as a function of the size of the input matrices, and the rate of the growth also varies depending on the rank of the second matrix.

The last major topic aimed on the computation of the product singular value decomposition for any two matrices with multiplicative compatibility. The recommendation of the QR-like approach among many others is mainly due to its lower complexity and availability of other supporting subroutines in LAPACK. We first implicitly reduced the product of the two matrices into upper- and lower-bidiagonalization as with the computation of the SVD for a single matrix. The PSVD is then finished up by computing the SVD of the bidiagonalized matrix. Our software which implements this method passes all the stability tests on matrices with various types and sizes. The timing test indicated that on average, the bi-diagonalization process takes about the same proportion as the subsequent computation of the SVD.

Some open questions as mentioned in earlier content include (1) the investigation of the *ad-hod* value $1/\sqrt{2}$ during the CSD computation and how it might affect the backward stability and amount of the subsequent computational effort; (2) optimizing of the work space in SORCSD which would ultimately lead to reducing the work space requirement in SGGQSV; and (3) tuning up the speed of SGGPSV and SORCSD by switching to a faster version of the SVD computation.

While further improvement is possible and always welcomed, algorithm and software developed in this thesis for computing the three types of the generalized singular value decompositions are stable and efficient. Altogether, we strongly believe that works contributed by this thesis are useful to future LAPACK releases as well as in other areas of scientific computing.

Bibliography

[1] O. Alter, P.O. Brown and D. Botstein, "Generalized singular decomposition for comparative analysis of genome-scale expression data sets of two different organisms", *Proc. Nat. Acad. Sci.* **USA 100** (2003): 3351-3356.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, Third Edition,* SIAM, Philadelphia, 1999.

[3] Z. Bai, "The CSD, GSVD, their applications and computations", *Preprint Series 958, Institute for Mathematics and Its Applications,* University of Minnesota, Minneapolis, April 1992. http://www.cs.ucdavis.edu/∼bai.

[4] Z. Bai and J. W. Demmel, "Computing the generalized singular value decomposition", *SIAM J. Sci. Comp.*, **14**, (1993): 1464-1486.

[5] Z. Bai and H. Zha, "A new preprocessing algorithm for the computation of the generalized singular value decomposition", *SIAM J. Sci. Comp.,* **14**, (1993):1007-1012.

[6] BLAS, http://www.netlib.org/blas.

[7] P. Deift, J. W. Demmel, L.-C. Li, and C. Tomei, "The bidiagonal singular values decomposition and Hamiltonian mechanics", *SIAM J. Numer. Anal.,* **28**, (1991): 1463-1516.

[8] B. De Moor, "On the structure and geometry of the product singular value decomposition", Numerical Analysis Project, Manuscript NA-89-05, Department of Computer Science, Stanford University, May 1989.

[9] B. De Moor and G. H. Golub, "The restricted singular value decomposition: properties and applications" Numerical Analysis Project, Manuscript MA-89-03, Department of Computer Science, Standford University, April, 1989.

[10] J. W. Demmel and W. Kahan, "Accurate singular values of bidiagonal matrices", *SIAM J. Sci. Stat. Comp.,* **11**, (1990):873-912.

[11] K.V. Fernando and S. J. Hammarling, "A product induced singular value decomposition for two matrices and balanced realization", *Linear Algebra in Signals, Systems, and Controls*, SIAM, Philadelphia, (1988):189-197.

[12] K.V. Fernando and B. Parlett, "Accurate singular values and differential qd algorithms", *Numerische Mathematik,* **67**, (1994): 191-229.

[13] G. H. Golub, K. Solna, and P. Van Dooren, "Computing the SVD of a gneral matrix product/quotient", *SIAM J. Mat. Anal. Appl.,* **22**(No. 1),(2000): 1-19.

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

[15] M. Gu and S. Eisenstat, "A divide-and-conquer algorithm for the bidiagonal SVD", *SIAM J. Mat. Anal. Appl.,* **16**, (1995): 79-92

[16] M. T. Heath, J.A. Laub, C.C. Paige and R.C. Ward, "Computing the singular value decomposition of a product of two matrices", *SIAM J. Sci. Stat.,* **7**(No. 4),(1986): 1147-1159.

[17] Intel's Math Kernel Library 7.2.1, http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/219823.html.

[18] E. Jessup and D. Sorensen, "A parallel algorithm for computing the singular value decomposition of a matrix", *Mathematics and Computer Science Division Report ANAL/MCS-TM-102,* Argonne National Laboratory, Argonne, IL, December 1987

[19] Netlib Repository, http://www.netlib.org.

[20] C.C. Paige, "Computing the generalized singular value decomposition", *SIAM J. Sci. Stat.*, **7**, (1986): 1126-1146.

[21] C.C. Paige and M. A. Saunders, "Towards a generalized singular value decomposition", *SIAM J. Numer. Anal.,* **18**, (1981): 398-405

[22] C. H. Park and H. Park, " Nonlinear discriminant analysis using kernel functions and the generalized singular value decomposition", *SIAM J. Mat. Ana. and Appl.*, **27**(1), (2005): 87-102.

[23] H. Park, M. Jeon, and P.J. Howland, "Dimension reduction for text data representation based on cluster structure preserving projection", Research Report (TR01-013), Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, 2001, https://wwws.cs.umn.edu/tech_reports/

[24] Sanzheng Qiao, "Approximating the PSVD and QSVD", *SVD and Signal Processing, III: Algorithms, Architectures and Applications,* (1995):149-155, http://www.cas.mcmaster.ca/∼qiao/.

[25] J. Speiser, "Linear algebra problems arising from signal processing", *Invited Presentation 3, SIAM Annual Meeting,* (1989).

[26] G. W. Stewart, "On graded QR decompositions of products of matrices", *Electronic Transactions on Numerical Analysis,* **3**,(1995): 39-49.

[27] G. W. Stewart, "On the perturbation of pseudo-inverses, projections and linear least squares problems", *SIAM Rev. 19*, (1977):634-662

[28] G. W. Stewart, "Computing the CS decomposition of a partitioned orthonormal Matrix", *Numerische Mathematik,* **40**, (1982): 297-306.

[29] G. W. Stewart, *Matrix Algorithms Volume I: Basic Decompositions* SIAM Press, Philadelphia, 1998

[30] L. N. Trefethen, D. Bau III, *Numerical Linear Algebra*, SIAM Press, Philadelphia, 1997.

[31] R.R. Tucci, "Quantum fast fourier transform viewed as a special case of recursive application of cosine-Sine decomposition", http://www.arxiv.org/abs/quant-ph/0411097.

[32] M. E. Wall, A. Rechtsteiner, L. M. Rocha. "Singular value decomposition and principal component analysis", *A Practical Approach to Microarray Data Analysis*, D.P. Berrar, W. Dubitzky, M. Granzow, eds., Kluwer: Norwell, MA (2003):91-109. LANL LA-UR-02-4001.

[33] C. F. Van Loan, "Analysis of some matrix problems using the CS decomposition", Tech. Report TR84-603, Computer Science Dept., Cornell Univ., Ithaca, NY, 1984.

[34] C. F. Van Loan and J. Speiser, "Computation of the C-S decomposition with application to signal processing", *SPIE Proceedings*, *696*, (1986).

[35] C. F. Van Loan, "Computing the CS and the generalized singular value decomposition", *Numerische Mathematik,* **46**, (1985): 479-491.

[36] C. F. Van Loan, "Generalizing the singular value decomposition", *SIAM Journal on Numerical Analysis,* **13**(No. 1), (Mar. 1976):76-83.

[37] H. Zha and Z. Zhang, "Modifying the generalized singular value decomposition with application in direction-of-arrival finding", *BIT,* **38**(1), (1998): 200-216.

Appendix A

Calling Sequences of Driver Routines

A.1 SORCSD

```
 SUBROUTINE SORCSD ( JOB, M, P, L, Q1, LDQ1, Q2,
     $                    LDQ2, ALPHA, BETA, U, LDU, V, LDV, ZT,
     $                    LDZ,  WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 0.1) --
*     Univ. of California, Davis
*     September, 2005
*
*     .. Scalar Arguments ..
      CHARACTER          JOB
      INTEGER            INFO, LDQ1, LDQ2, LDU, LDV, LDZ, M, P,
     $                   L, LWORK
*     ..
*     .. Array Arguments ..
      REAL               Q1( LDQ1, * ), ALPHA( * ), Q2( LDQ2, * ),
     $                   BETA( * ), U( LDU, * ), V( LDV, * ),
     $                   ZT( LDZ, * ), WORK( LWORK )
*     ..
*
* PURPOSE:
* ========
*
* SORCSD computes the cosine-sine decomposition (CSD) of an
* (M+P)-by-L orthogonal matrix Q = (Q1' Q2')':
*
*       U' * Q1 = D1 Z',   V' * Q2 = D2 Z'
*
*
* where U, V and Z are orthogonal matrices of size M-by-M, P-by-P,
* and L-by-L, respectively.  X' denote the transpose of X.
* ( NOTE: the actual return value is Z' instead of Z )
*
* D1 and D2 are M-by-L and P-by-L "diagonal" matrices and
* D1'D1 + D2'D2 = I.  D1 and D2 have one of the following four
* structures:
*
* (1) If M >= L and P >= L,
*
```

```
* D1 =        L
*          L ( C )
*        M-L ( 0 )
*
* D2 =        L
*          L ( S )
*        P-L ( 0 )
*
* where C = diag(ALPHA(1), ... , ALPHA(N) ),
*       S = diag(BETA(1), ..., BETA(N) ),
*       C**2 + S**2 = I.
*
* (2) If M >= L and P < L,
*
* D1 =          L-P  P
*         L-P  (  I   0 )
*           P  (  0   C )
*         M-L  (  0   0 )
*
* D2 =         L-P   P
*           P  ( 0    S )
*
* where  C = diag(ALPHA(L-P+1), ... , ALPHA(L) ),
*        S = diag(BETA(L-P+1), ..., BETA(L) ),
*        C**2 + S**2 = I.
*
* (3) If M <= L and P >= L,
*
* D1 =       M   L-M
*        M  ( C    0  )
*
* D2 =       M    L-M
*        M  ( S     0  )
*      L-M  ( 0     I  )
*      P-L  ( 0     0  )
*
* where C = diag(ALPHA(1), ... , ALPHA(M) ),
*       S = diag(BETA(1), ..., BETA(M) ),
*       C**2 + S**2 = I.
*
* (4) If M <= L and P < L,
*
* D1 =         L-P   M+P-L   L-M
*         L-P (  I      0      0  )
*       M+P-L (  0      C      0  )
*
* D1 =         L-P   M+P-L   L-M
*       M+P-L (  0      S      0  )
*         L-M (  0      0      I  )
*
* where C = diag(ALPHA(L-P+1), ... , ALPHA(M) ),
*       S = diag(BETA(L-P+1), ..., BETA(M) ),
*       C**2 + S**2 = I.
*
*
* Arguments
* =========
*
```

```
* JOB     (input) CHARACTER*1
*         = 'Y':  Orthogonal matrix U,V, and Z are computed;
*         = 'N':  Orthogonal matrices are not computed.
*
* M       (input) INTEGER
*         The number of rows of the block of Q, Q1.  M >= 0.
*
* P       (input) INTEGER
*         The number of rows of the block of Q, Q2.  P >= 0.
*
* L       (input) INTEGER
*         The number of columns of Q.
*         L >= 0, and M+P >= L.
*
* Q1      (input/output) REAL array, dimension (LDQ1,*)
*         On entry, the M-by-L matrix Q1.
*         On exit, Q1 is destroyed.
*
* LDQ1    (input) INTEGER
*         The leading dimension of the array Q1. LDQ1 >= max(1,M).
*
* Q2      (input/output) REAL array, dimension (LDQ2,*)
*         On entry, the P-by-L matrix Q2.
*         On exit, Q2 is destroyed.
*
* LDQ2    (input) INTEGER
*         The leading dimension of the array Q2. LDQ1 >= max(1,P).
*
* ALPHA   (output) REAL array, dimension (L)
* BETA    (output) REAL array, dimension (L)
*     On exit, ALPHA and BETA contain the cosine-sine values
*         of Q1 and Q2
*
*     (1) if M >= L and P >= L
*           ALPHA(1:L) = C
*           BETA (1:L) = S
*
*     (2) if M >= L and P < L
*           ALPHA(1:L-P) = 1, ALPHA(L-P+1:L) = C
*           BETA (1:L-P) = 0, BETA (L-P+1:L) = S
*
*     (3) if M <= L and P >= L
*           ALPHA(1:M) = C, ALPHA(M+1:L) = 0
*           BETA (1:M) = S, BETA (M+1:L) = 1
*
*     (4) if M <= L and P < L
*           ALPHA(1:L-P) = 1, ALPHA(L-P+1:M) = C, ALPHA(M+1, L) = 0
*           BETA (1:L-P) = 0, BETA (L-P+1:M) = S, BETA (M+1, L) = 1
*
*       ALPHA are stored in non-increasing order.
*       BETA  are stored in non-decreasing order.
*
* U       (output) REAL array, dimension (LDU,M)
*         If JOB = 'U', U contains the M-by-M orthogonal matrix U.
*         If JOB = 'N', U is not referenced.
*
* LDU     (input) INTEGER
*         The leading dimension of the array U. LDU >= max(1,M)
```

```
*         if JOB = 'Y'; LDU >= 1 otherwise.
*
* V       (output) REAL array, dimension (LDV,P)
*         If JOB = 'Y', V contains the P-by-P orthogonal matrix V.
*         If JOB = 'N', V is not referenced.
*
* LDV     (input) INTEGER
*         The leading dimension of the array V. LDV >= max(1,P) if
*         JOB = 'Y'; LDV >= 1 otherwise.
*
* ZT       (output) REAL array, dimension (LDZ,L)
*         If JOB = 'Y', Z contains the L-by-L orthogonal matrix Z'.
*         If JOB = 'N', Z is not referenced.
*
* LDZ     (input) INTEGER
*         The leading dimension of the array ZT.
*         LDZ >= max(1,L) if JOB = 'Y'; LDZ >= 1 otherwise.
*
* WORK    (workspace) REAL array, dimension (LWORK)
*
* LWORK   (input) INTEGEER
*         The dimension of the array WORK.
*         LWORK >= MAX( 3 * MIN( N ,L ) + MAX( N, L ), 5 * MIN( N,L ) )
*                 + 2 * MIN( M, P, L ) + L**2
*
*          where N = MAX( M, P )
*
* INFO    (output) INTEGER
*         = 0:  successful exit
*         < 0:  if INFO = -i, the i-th argument had an illegal value.
*
* Internal Parameters
* ===================
*
* TOL    REAL
*        Thresholds to separate the "large" and "small" singular
*        values.
*        Currently, it is set to SQT(0.5).
*        The size of TOL has the effect of minimizing the backward
*           error of the composition.
*
* ====================================================================
*
```

A.2 SGGQSV

```
 SUBROUTINE SGGQSV( JOBU, JOBV, JOBQ, M, N, P, K, L, A, LDA, B,
     $                   LDB, ALPHA, BETA, U, LDU, V, LDV, Q, LDQ,
     $                   IWORK, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 0.1) --
*     Univ. of California Davis,
*     September, 2005
*
*     .. Scalar Arguments ..
      CHARACTER          JOBQ, JOBU, JOBV
      INTEGER            M, N, P, K, L, LDA, LDB, LDU, LDV, LDQ,
```

```
     $                   LWORK, INFO
*     ..
*     .. Array Arguments ..
      INTEGER            IWORK( * )
      REAL               A( LDA, * ), ALPHA( * ), B( LDB, * ),
     $                   BETA( * ), Q( LDQ, * ), U( LDU, * ),
     $                   V( LDV, * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  SGGQSV computes the quotient ( or generalized ) singular value
*  decomposition (QSVD) of an M-by-N real matrix A and P-by-N
*  real matrix B:
*
*      U'*A*Q = D1*( 0 R ),    V'*B*Q = D2*( 0 R )
*
*  where U, V and Q are orthogonal matrices, and Z' is the transpose
*  of Z.  Let K+L = the effective numerical rank of the matrix (A',B')',
*  then R is a (K+L)-by-(K+L) nonsingular upper triangular matrix,
*  D1 and D2 are M-by-(K+L) and P-by-(K+L) "diagonal" matrices and
*  have one of the following structures:
*
*  If M >= K+L,
*
*         D1 =      K   L
*             K  (  I   0  )
*             L  (  0   C  )
*         M-K-L  (  0   0  )
*
*         D2 =      K   L
*             L  (  0   S  )
*           P-L  (  0   0  )
*
*                   N-K-L  K    L
*    ( 0 R ) = K (  0    R11  R12 )
*              L (  0     0   R22 )
*
*  where
*
*       C = diag( ALPHA( K+1 ), ... , ALPHA( K+L ) ),
*       S = diag( BETA( K+1 ), ..., BETA( K+L ) ),
*       C**2 + S**2 = I
*
*       R is stored in A( 1:K+L , N-K-L+1:N ) on exit.
*
*  (2) If M < K+L,
*
*        D1 =         K   M-K  K+L-M
*                 K ( I    0     0  )
*               M-K ( 0    C     0  )
*
*         D2 =        K  M-K   K+L-M
*               M-K ( 0   S     0  )
*             K+L-M ( 0   0     I  )
*             P-K-L ( 0   0     0  )
*
```

```
*                    N-K-L   K   M-K  K+L-M
*    ( 0 R ) =    K ( 0    R11  R12  R13  )
*               M-K ( 0     0   R22  R23  )
*             K+L-M ( 0     0    0   R33  )
*
*  where
*
*       C = diag( ALPHA(K+1), ... , ALPHA(M) ),
*    S = diag( BETA(K+1), ..., BETA(M) ),
*       C**2 + S**2 = I
*
*    (R11 R12 R13 ) is stored in A(1:M, N-K-L+1:N), and R33 is stored
*    ( 0  R22 R23 )
*    in B(1:K+L-M, N+M-K-L+1:N) on exit.
*
*  The routine computes C, S, R, and optionally the orthogonal
*  transformation matrices U, V and Q.
*
*  In particular, if B is an N-by-N nonsingular matrix, then the GSVD of
*  A and B implicitly gives the SVD of A*inv(B):
*                       A*inv(B) = U*(D1*inv(D2))*V'.
*  If ( A',B')' has orthonormal columns, then the GSVD of A and B is
*  also equal to the CS decomposition of A and B. Furthermore, the GSVD
*  can be used to derive the solution of the eigenvalue problem:
*                       A'*A x = lambda* B'*B x.
*  In some literature, the GSVD of A and B is presented in the form
*                   U'*A*X = ( 0 D1 ),   V'*B*X = ( 0 D2 )
*  where U and V are orthogonal and X is nonsingular, D1 and D2 are
*  ``diagonal''.  The former GSVD form can be converted to the latter
*  form by taking the nonsingular matrix X as
*
*                       X = Q*( I   0    )
*                             ( 0 inv(R) ).
*
*  Arguments
*  =========
*
*  JOBU    (input) CHARACTER*1
*          = 'U':  Orthogonal matrix U is computed;
*          = 'N':  U is not computed
*
*  JOBV    (input) CHARACTER*1
*          = 'V':  Orthogonal matrix V is computed;
*          = 'N':  V is not computed
*
*  JOBQ    (input) CHARACTER*1
*          = 'Q':  Orthogonal matrix Q is computed;
*          = 'N':  Q is not computed
*
*  NOTE: It is assumed that JOBU, JOBV, JOBQ are all turned on
*        or off at the same time.
*
*  M       (input) INTEGER
*          The number of rows of the matrix A.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrices A and B.  N >= 0.
*
```

```
*  P       (input) INTEGER
*          The number of rows of the matrix B.  P >= 0.
*
*  K       (output) INTEGER
*  L       (output) INTEGER
*          On exit, K and L specify the dimension of the subblocks
*          described in the Purpose section.
*          K + L = effective numerical rank of (A',B')'.
*
*  A       (input/output) REAL array, dimension (LDA,N)
*          On entry, the M-by-N matrix A.
*          On exit, A contains the triangular matrix R, or part of R.
*          See Purpose for details.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A. LDA >= max(1,M).
*
*  B       (input/output) REAL array, dimension (LDB,N)
*          On entry, the P-by-N matrix B.
*          On exit, B contains the triangular matrix R if M-K-L < 0.
*          See Purpose for details.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B. LDA >= max(1,P).
*
*  ALPHA   (output) REAL array, dimension (N)
*  BETA    (output) REAL array, dimension (N)
*          On exit, ALPHA and BETA contain the generalized singular
*          value pairs of A and B;
*            ALPHA( 1:K ) = 1,
*            BETA( 1:K )  = 0,
*          and if M-K-L >= 0,
*            ALPHA( K+1:K+L ) = C,
*            BETA( K+1:K+L )  = S,
*          or if M-K-L < 0,
*            ALPHA( K+1:M )=C, ALPHA( M+1:K+L )=0
*            BETA( K+1:M ) =S, BETA( M+1:K+L ) =1
*          and
*            ALPHA(K+L+1:N) = 0
*            BETA(K+L+1:N)  = 0
*
*  U       (output) REAL array, dimension (LDU,M)
*          If JOBU = 'U', U contains the M-by-M orthogonal matrix U.
*          If JOBU = 'N', U is not referenced.
*
*  LDU     (input) INTEGER
*          The leading dimension of the array U. LDU >= max(1,M) if
*          JOBU = 'U'; LDU >= 1 otherwise.
*
*  V       (output) REAL array, dimension (LDV,P)
*          If JOBV = 'V', V contains the P-by-P orthogonal matrix V.
*          If JOBV = 'N', V is not referenced.
*
*  LDV     (input) INTEGER
*          The leading dimension of the array V. LDV >= max(1,P) if
*          JOBV = 'V'; LDV >= 1 otherwise.
*
*  Q       (output) REAL array, dimension (LDQ,N)
```

```
*          If JOBQ = 'Q', Q contains the N-by-N orthogonal matrix Q.
*          If JOBQ = 'N', Q is not referenced.
*
*  LDQ     (input) INTEGER
*          The leading dimension of the array Q. LDQ >= max(1,N) if
*          JOBQ = 'Q'; LDQ >= 1 otherwise.
*
*  IWORK   (workspace) INTEGER array, dimension ( N )
*
*  WORK    (workspace) REAL array, dimension (LWORK)
*
*  LWORK   (input) INTEGEER
*         The dimension of the array WORK.
*         LWORK >= 6 * MIN( P, N )**2 + 6 * N + max( 3M, N, P )
*
*  INFO    (output)INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value.
*
*  Internal Parameters
*  ===================
*
*  TOLA    REAL
*  TOLB    REAL
*          TOLA and TOLB are the thresholds to determine the effective
*          rank of (A',B')'. Generally, they are set to
*                   TOLA = MAX(M,N)*norm(A)*MACHEPS,
*                   TOLB = MAX(P,N)*norm(B)*MACHEPS.
*          The size of TOLA and TOLB may affect the size of backward
*          errors of the decomposition.
*
*  =====================================================================
*
```

A.3 SGGPSV

```
 SUBROUTINE SGGPSV( JOBU, JOBVT, M, K, N, A, LDA, B, LDB,
     $                   S, U, LDU, VT, LDVT, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine ( version 0.1 ) --
*     Univ. of California Davis,
*     April, 2005
*
*     .. Scalar Arguments ..
      CHARACTER          JOBU, JOBVT
      INTEGER            M, K, N, LDA, LDB, LDU, LDVT, INFO, LWORK
*     ..
*     .. Array Arguments ..
      REAL               A( LDA, * ), B( LDB, * ),
     $                   S( * ), U( LDU, * ), VT( LDVT, * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  SGGPSV computes the produce singular value decomposition (PSVD)
*  of an M-by-N real matrix A and N-by-P real matrix B:
```

```
*
*      A * B = U * SIGMA * V'
*
*  where SIGMA is the diagonal matrix that contains the singular
*  values of the product A*B, U and V are orthogonal matrices.
*  Z' denotes the transpose of Z.
*  The first min(M, N) columns of U and V are the left and right
*  singular vectors of A*B.
*
*  Note that the routine returns V', not V.
*
*  Arguments
*  =========
*
*  JOBU    (input) CHARACTER*1
*          = 'U':  Orthogonal matrix U is computed;
*          = 'N':  U is not computed.
*
*  JOBVT    (input) CHARACTER*1
*          = 'V':  Orthogonal matrix VT is computed;
*          = 'N':  VT is not computed.
*
*  M       (input) INTEGER
*          The number of rows of the input matrix A.  M >= 0.
*
*  K       (input) INTEGER
*          The number of columns of the input matrix A.
*          The number of rows of the input matrix B.  K >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrix B. N >= 0.
*
*  A       (input/output) REAL array, dimension (LDA,K)
*          On entry, the M-by-K matrix A.
*          On exit, entries are those as returned from SGGBRD.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A. LDA >= max(1,M).
*
*  B       (input/output) REAL array, dimension (LDB,N)
*          On entry, the K-by-N matrix B.
*          On exit, entries are those as returned from SGGBRD.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B. LDB >= max(1,K).
*
*  S       (output) REAL array, dimension (min(M, N))
*          The singular values of A*B, sorted so that S(i) >= S(i+1).
*
*  U       (output) REAL array, dimension (LDU, N)
*          If JOBU = 'U', U contains the M-by-M orthogonal matrix U.
*          If JOBU = 'N', U is not referenced.
*
*  LDU     (input) INTEGER
*          The leading dimension of the array U.
*          If JOBU = 'U', LDU >= M.
*          Otherwise, LDU >= 1
*
```

```
*  VT      (output) REAL array, dimension (LDVT, N)
*          If JOBV = 'VT', VT contains the N-by-N orthogonal matrix VT.
*          If JOBV = 'N',  VT is not referenced.
*
*  LDVT    (input) INTEGER
*          The leading dimension of the array VT.
*          If JOBVT = 'V', LDVT >= N.
*          Otherwise, LDVT >= 1.
*
*  WORK    (workspace) REAL array, dimension (LWORK)
*
*  LWORK   (input) INTEGER
*          The dimension of the array WORK.  LWORK >= 1
*          LWORK >= max( M, N, K ) + 4*min( M, N )
*          Note: LWORK is somewhat conservative.
*
*  INFO    (output)INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value.
*
*  =====================================================================
*
```

Appendix B

Calling Sequences of Computational Routines

B.1 SORCS1

```
      SUBROUTINE SORCS1 ( JOB, M, P, L, Q1, LDQ1, Q2,
     $                    LDQ2, ALPHA, BETA, U, LDU, V, LDV, ZT,
     $                    LDZ, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 0.1) --
*     Univ. of California, Davis
*     September, 2005
*
*     .. Scalar Arguments ..
      CHARACTER          JOB
      INTEGER            INFO, LDQ1, LDQ2, LDU, LDV, LDZ, M, P,
     $                   L, LWORK
*     ..
*     .. Array Arguments ..
      REAL               Q1( LDQ1, * ), ALPHA( * ), Q2( LDQ2, * ),
     $                   BETA( * ), U( LDU, * ), V( LDV, * ),
     $                   ZT( LDZ, * ), WORK( LWORK )
*     ..
*
* PURPOSE:
* ========
*
* SORCS1 computes the cosine-sine decomposition (CSD) of an
* (M+P)-by-L orthogonal matrix Q = (Q1' Q2')':
*
*       U' * Q1 = D1 Z',   V' * Q2 = D2 Z'
*
*
* where U, V and Z are orthogonal matrices of size M-by-M, P-by-P,
* and L-by-L, respectively.  X' denote the transpose of X.
* ( NOTE: the actual return value is Z' instead of Z )
*
* D1 and D2 are M-by-L and P-by-L "diagonal" matrices and
* D1'D1 + D2'D2 = I.   For details, refer to SORCSD.f.
*
* This is an internal subroutine called by SORCSD to handle the
* case where assumes M > P.  For other case, see SORCS2.f.
*
```

```
* Arguments
* =========
*
* JOB    (input) CHARACTER1
*        = 'Y':  All three orthogonal matrices, U, V, Z are computed;
*        = 'N':  None of orthogonal matrices, U, V, Z are not computed.
*
* M       (input) INTEGER
*         The number of rows of the matrix Q1.  M >= 0.
*
* P       (input) INTEGER
*         The number of rows of the matrix Q2.  P >= 0.
*
* L       (input) INTEGER
*         The number of columns of the matrices Q1 and Q2.
*         L >= 0, and M+P >= L.
*
* Q1      (input/output) REAL array, dimension (LDQ1,L)
*         On entry, the M-by-L matrix Q1.
*         On exit, Q1 is destroyed.
*
* LDQ1    (input) INTEGER
*         The leading dimension of the array Q1. LDQ1 >= max(1,M).
*
* Q2      (input/output) REAL array, dimension (LDQ2,L)
*         On entry, the P-by-L matrix Q2.
*         On exit, Q2 is destroyed.
*
* LDQ2    (input) INTEGER
*         The leading dimension of the array Q2. LDQ1 >= max(1,P).
*
* ALPHA   (output) REAL array, dimension (L)
* BETA    (output) REAL array, dimension (L)
*     On exit, ALPHA and BETA contain the cosine-sine value pairs
*         of Q1 and Q2
*
*     (1) if M >= L and P >= L
*           ALPHA(1:L) = C
*           BETA (1:L) = S
*
*     (2) if M >= L and P < L
*           ALPHA(1:L-P) = 1, ALPHA(L-P+1:L) = C
*           BETA (1:L-P) = 0, BETA (L-P+1:L) = S
*
*     (3) if M < L and P < L
*           ALPHA(1:L-P) = 1, ALPHA(L-P+1:M) = C, ALPHA(M+1, L) = 0
*           BETA (1:L-P) = 0, BETA (L-P+1:M) = S, BETA (M+1, L) = 1
*
*       ALPHA are stored in non-increasing order.
*       BETA  are stored in non-decreasing order.
*
* U       (output) REAL array, dimension (LDU,M)
*         If JOB = 'Y', U contains the M-by-M orthogonal matrix U.
*         If JOB = 'N', U is not referenced.
*
* LDU     (input) INTEGER
*         The leading dimension of the array U. LDU >= max(1,M) if
*         JOB = 'Y'; LDU >= 1 otherwise.
```

```
*
* V       (output) REAL array, dimension (LDV,P)
*         If JOB = 'Y', V contains the P-by-P orthogonal matrix V.
*         If JOB = 'N', V is not referenced.
*
* LDV     (input) INTEGER
*         The leading dimension of the array V. LDV >= max(1,P) if
*         JOB = 'Y'; LDV >= 1 otherwise.
*
* ZT       (output) REAL array, dimension (LDZ,L)
*         If JOB = 'Y', ZT contains the L-by-L orthogonal matrix Z'.
*         If JOB = 'N', ZT is not referenced.
*
* LDZ     (input) INTEGER
*         The leading dimension of the array ZT.
*         LDZ >= max(1,L) if JOB = 'Y'; LDZ >= 1 otherwise.
*
* WORK    (workspace) REAL array, dimension (LWORK)
*
* LWORK   (input) INTEGEER
*         The dimension of the array WORK.
*         LWORK >= MAX( 3 * MIN( M, L ) + MAX( M, L ), 5 * MIN( M, L ) )
*                 + 2 * MIN( P, L ) + L**2
*
* INFO    (output) INTEGER
*         = 0:  successful exit
*         < 0:  if INFO = -i, the i-th argument had an illegal value.
*
*
* Internal Parameters
* ===================
*
* TOL    REAL
*        Thresholds to separate the "large" and "small" singular
*        values.
*        Currently, it is set to SQT(0.5).
*        The size of TOL has the effect of minimizing the backward
*           error of the composition.
*
* ====================================================================
*
```

B.2 SORCS2

```
 SUBROUTINE SORCS2 ( JOB, M, P, L, Q1, LDQ1, Q2,
     $                    LDQ2, ALPHA, BETA, U, LDU, V, LDV, ZT,
     $                    LDZ, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 0.1) --
*     Univ. of California, Davis
*     September, 2005
*
*     .. Scalar Arguments ..
      CHARACTER          JOB
      INTEGER            INFO, LDQ1, LDQ2, LDU, LDV, LDZ, M, P,
     $                   L, LWORK
*     ..
```

```
*     .. Array Arguments ..
      REAL               Q1( LDQ1, * ), ALPHA( * ), Q2( LDQ2, * ),
     $                   BETA( * ), U( LDU, * ), V( LDV, * ),
     $                   ZT( LDZ, * ), WORK( LWORK )
*     ..
*
* PURPOSE:
* ========
*
* SORCS1 computes the cosine-sine decomposition (CSD) of an
* (M+P)-by-L orthogonal matrix Q = (Q1' Q2')':
*
*       U' * Q1 = D1 Z',   V' * Q2 = D2 Z'
*
*
* where U, V and Z are orthogonal matrices of size M-by-M, P-by-P,
* and L-by-L, respectively.  X' denote the transpose of X.
* ( NOTE: the actual return value is Z' instead of Z )
*
* D1 and D2 are M-by-L and P-by-L "diagonal" matrices and
* D1'D1 + D2'D2 = I.   For details, refer to SORCSD.f.
*
* This is an internal subroutine called by SORCSD to handle the
* case where assumes M <= P.  For other case, see SORCS1.f.
*
* Arguments
* =========
*
* JOB    (input) CHARACTER1
*        = 'Y':  All three orthogonal matrices, U, V, Z are computed;
*        = 'N':  None of orthogonal matrices, U, V, Z are not computed.
*
* M       (input) INTEGER
*         The number of rows of the matrix Q1.  M >= 0.
*
* P       (input) INTEGER
*         The number of rows of the matrix Q2.  P >= 0.
*
* L       (input) INTEGER
*         The number of columns of the matrices Q1 and Q2.
*         L >= 0, and M+P >= L.
*
* Q1      (input/output) REAL array, dimension (LDQ1,L)
*         On entry, the M-by-L matrix Q1.
*         On exit, Q1 is destroyed.
*
* LDQ1    (input) INTEGER
*         The leading dimension of the array Q1. LDQ1 >= max(1,M).
*
* Q2      (input/output) REAL array, dimension (LDQ2,L)
*         On entry, the P-by-L matrix Q2.
*         On exit, Q2 is destroyed.
*
* LDQ2    (input) INTEGER
*         The leading dimension of the array Q2. LDQ1 >= max(1,P).
*
* ALPHA   (output) REAL array, dimension (L)
* BETA    (output) REAL array, dimension (L)
```

```
*     On exit, ALPHA and BETA contain the cosine-sine value pairs
*         of Q1 and Q2
*
*     (1) if M >= L and P >= L
*           ALPHA(1:L) = C
*           BETA (1:L) = S
*
*     (2) if M < L and P >= L
*           ALPHA(1:M) = C, ALPHA(M+1:L) = 0
*           BETA (1:M) = S, BETA (M+1:L) = 1
*
*     (3) if M < L and P < L
*           ALPHA(1:L-P) = 1, ALPHA(L-P+1:M) = C, ALPHA(M+1, L) = 0
*           BETA (1:L-P) = 0, BETA (L-P+1:M) = S, BETA (M+1, L) = 1
*
*       ALPHA are stored in non-increasing order.
*       BETA  are stored in non-decreasing order.
*
* U       (output) REAL array, dimension (LDU,M)
*         If JOB = 'Y', U contains the M-by-M orthogonal matrix U.
*         If JOB = 'N', U is not referenced.
*
* LDU     (input) INTEGER
*         The leading dimension of the array U. LDU >= max(1,M) if
*         JOB = 'Y'; LDU >= 1 otherwise.
*
* V       (output) REAL array, dimension (LDV,P)
*         If JOB = 'Y', V contains the P-by-P orthogonal matrix V.
*         If JOB = 'N', V is not referenced.
*
* LDV     (input) INTEGER
*         The leading dimension of the array V. LDV >= max(1,P) if
*         JOB = 'Y'; LDV >= 1 otherwise.
*
* ZT       (output) REAL array, dimension (LDZ,L)
*         If JOB = 'Y', ZT contains the L-by-L orthogonal matrix Z'.
*         If JOB = 'N', ZT is not referenced.
*
* LDZ     (input) INTEGER
*         The leading dimension of the array ZT.
*         LDZ >= max(1,L) if JOB = 'Y'; LDZ >= 1 otherwise.
*
* WORK    (workspace) REAL array, dimension (LWORK)
*
* LWORK   (input) INTEGEER
*         The dimension of the array WORK.
*         LWORK >= MAX( 3 * MIN( P, L ) + MAX( P, L ), 5 * MIN( P, L ) )
*                 + 2 * MIN( M, L ) + L**2
*
* INFO    (output) INTEGER
*         = 0:  successful exit
*         < 0:  if INFO = -i, the i-th argument had an illegal value.
*
*
* Internal Parameters
* ===================
*
* TOL    REAL
```

```
*        Thresholds to separate the "large" and "small" singular
*        values.
*        Currently, it is set to SQT(0.5).
*        The size of TOL has the effect of minimizing the backward
*           error of the composition.
*
* ====================================================================
*
```

B.3 SGEQRJ

```
 SUBROUTINE SGGQRJ  ( M, N, A, LDA, B, LDB, U, LDU, V, LDV,
     $                     WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 0.1) --
*     Univ. of California, Davis
*     September, 2005
*
*     .. Scalar Arguments ..
      INTEGER            LDA, LDB, LDU, LDV, M, N, LWORK, INFO
*     ..
*     .. Array Arguments ..
      REAL               A( LDA, * ), B( LDB, * ),
     $                   U( LDU, * ), V( LDV, * ),
     $                   WORK( LWORK )
*     ..
*
*
* PURPOSE:
* ========
*
* SGGQRJ computes the compact QR factorization of a (M+N)-by-N
* matrix where A is M-by-N upper-trapezoidal and B is
* N-by-N upper-triangular:
*
*      Q R = (A' B')'.
*
*
* Arguments
* =========
*
* M       (input) INTEGER
*         The number of rows of the matrix A.  M >= 0.
*
* N       (input) INTEGER
*         The number of columns of the matrix A
*         and he number of rows and columns of the matrix B.
*         N >= 0, N >= M.
*
* A       (input/output) REAL array, dimension (LDA, N)
*         On entry, the M-by-N upper-trapezoidal matrix A.
*         On exit, the first min( M, N ) rows of the upper triangular
*         matrix R.
*
* LDA     (input) INTEGER
*         The leading dimension of the array A. LDA >= max(1, M).
*
```

```
* B       (input/output) REAL array, dimension (LDB, N)
*         On entry, the N-by-N upper triangular matrix B
*         On exit, the origional content of B is destroyed.
*         If M < N, B contains the M+1 to N rows of the upper
*         triangular matrix R.
*
* LDB     (input) INTEGER
*         The leading dimension of the array B. LDB >= max(1, N).
*
* U       (output) REAL array, dimension (LDU, N)
*         U contains the first M rows of the orthogonal matrix Q.
*
* LDU     (input) INTEGER
*         The leading dimension of the array U. LDU >= max(1, M).
*
* V       (output) REAL array, dimension (LDV, N)
*         V contains the last N rows of the orthogonal matrix Q.
*
* LDV     (input) INTEGER
*         The leading dimension of the array V. LDV >= max(1, N).
*
* WORK    (workspace) REAL array, dimension (LWORK)
*
* LWORK   (input) INTEGEER
*         The dimension of the array WORK.
*         LWORK = M+N.
*
* INFO    (output) INTEGER
*         = 0:  successful exit
*         < 0:  if INFO = -i, the i-th argument had an illegal value.
*
* =====================================================================
*
```

B.4 SGGBRD

```
 SUBROUTINE SGGBRD( M, K, N, A, LDA, B, LDB, D, E, TAUQ, TAUP,
     $                   WORK, LWORK,  INFO )
*
*  -- LAPACK driver routine ( version 0.1 ) --
*     Univ. of California Davis,
*     April, 2005
*
*     .. Scalar Arguments ..
      INTEGER            M, N, K, LDA, LDB, LWORK, INFO
*     ..
*     .. Array Arguments ..
      REAL               A( LDA, * ), B( LDB, * ),
     $                   D( * ), E( * ), TAUP( * ), TAUQ( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  SGGBRD reduces the product of two general matrices, A of size M-by-N
*  and B of size N-by-P, to upper bidiagonal form G by a sequence of
*  orthogonal transformations:
```

```
*
*              Q' * (A * B) * P = G
*
*  where Q and P are orthogonal matrices, Z' denotes the transpose of Z.
*  If m >= n, B is upper bidiagonal; if m < n, B is lower bidiagonal.
*
*
*  Arguments
*  =========
*
*  M       (input) INTEGER
*          The number of rows in the matrix A.  M >= 0.
*
*  K       (input) INTEGER
*          The number of columns in the matrix A.
*          The number of rows in the matrix B.  K >= 0.
*
*  N       (input) INTEGER
*          The number of columns in the matrix B.  N >= 0.
*
*  A       (input/output) REAL array, dimension (LDA, K)
*          On entry, the M-by-K general matrix.
*          On exit,
*          if M >= N, the elements below the diagonal,
*            with the array TAUQ, represent the orthogonal matrix Q
*            as a product of elementary reflectors.
*          if M < N, the elements below the first subdiagonal,
*            with the array TAUQ, represent the orthogonal matrix Q
*            as a product of elementary reflectors.
*          The rest of the entries in A are set to zeros.
*          See Further Details.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,M).
*
*  B       (input/output) REAL array, dimension (LDB, K)
*          On entry, the K-by-N general matrix.
*          On exit,
*          if M >= N, the elements above the first superdiagonal,
*            with the array TAUP, represent the orthogonal matrix P as
*            a product of elementary reflectors.
*          if M < N, the elements above the diagonal
*            with the array TAUP, represent the orthogonal matrix P as
*            a product of elementary reflectors.
*          The rest of the entries in B are set to zeros.
*          See Further Details.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B.  LDB >= max(1,N).
*
*  D       (output) REAL array, dimension (min(M,N))
*          The diagonal elements of the bidiagonal matrix G:
*
*  E       (output) REAL array, dimension (min(M,N) - 1)
*          if M >= N, the upper-diagonal elements of the bidiagonal
*              matrix G,
*          If M < N, the lower-diagonal elements of the bidiagonal
*              matrix G.
```

```
*
*  TAUQ    (output) REAL array dimension (min(M,N))
*          The scalar factors of the elementary reflectors which
*          represent the orthogonal matrix Q. See Further Details.
*
*  TAUP    (output) REAL array, dimension (min(M,N))
*          The scalar factors of the elementary reflectors which
*          represent the orthogonal matrix P. See Further Details.
*
*  WORK    (workspace) REAL array, dimension (LWORK)
*
*  LWORK   (input) INTEGER
*          The length of the array WORK.
*          LWORK >= max( M, N, K ) + min( M, N ).
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value.
*
*  Further Details
*  ===============
*
*  The matrices Q and P are represented as products of elementary
*  reflectors:
*
*  If M >= N,
*
*     Q = H(1) H(2) . . . H(t)  and  P = W(1) W(2) . . . W(r)
*     where t = min(m-1, k, n) and r = min(m, k, n-2).
*
*  Each H(i) and W(i) has the form:
*
*     H(i) = I - tauq * v * v'  and W(i) = I - taup * u * u'
*
*  where tauq and taup are real scalars, and v and u are real vectors;
*  v(1:i-1) = 0, v(i) = 1, and v(i+1:m) is stored on exit in A(i+1:m,i);
*  u(1:i) = 0, u(i+1) = 1, and u(i+2:n) is stored on exit in B(i,i+2:n);
*  tauq is stored in TAUQ(i) and taup in TAUP(i).
*
*  If M < N,
*     Q = H(1) H(2) . . . H(t)  and  P = G(1) G(2) . . . G(r)
*     where t = min(m, k-1) and r = min(m-2, k, n).
*
*  Each H(i) and G(i) has the form:
*
*     H(i) = I - tauq * v * v'  and G(i) = I - taup * u * u'
*
*  where tauq and taup are real scalars, and v and u are real vectors;
*  v(1:i) = 0, v(i+1) = 1, and v(i+2:m) is stored on exit in A(i+2:m,i);
*  u(1:i-1) = 0, u(i) = 1, and u(i+1:n) is stored on exit in B(i,i+1:n);
*  tauq is stored in TAUQ(i) and taup in TAUP(i).
*
*  The contents of A and B on exit are illustrated by the following
*  examples:
*
*  M = 6 K = 7 and N = 5 ( M >= N )
*   A =                         B =
*    ( -  -  -  -  -  -  - )       (  -  -  u1 u1 u1 )
```

```
*    ( v1 -  -  -  -  -  - )       (  -  -  -  u2 u2 )
*    ( v1 v2 -  -  -  -  - )       (  -  -  -  -  u3 )
*    ( v1 v2 v3 -  -  -  - )       (  -  -  -  -  -  )
*    ( v1 v2 v3 v4 -  -  - )       (  -  -  -  -  -  )
*    ( v1 v2 v3 v4 v5 -  - )       (  -  -  -  -  -  )
*                                  (  -  -  -  -  -  )
*
*  M = 5, K = 7, N = 6   ( M < N )
*   A =                          B =
*    ( -  -  -  -  -  -  - )       (  -  u1  u1  u1  u1  u1 )
*    ( -  -  -  -  -  -  - )       (  -  -   u2  u2  u2  u2 )
*    ( v1 -  -  -  -  -  - )       (  -  -   -   u3  u3  u3 )
*    ( v1 v2 -  -  -  -  - )       (  -  -   -   -   u4  u4 )
*    ( v1 v2 v3 -  -  -  - )       (  -  -   -   -   -   u5 )
*                                  (  -  -   -   -   -   -  )
*                                  (  -  -   -   -   -   -  )
*
*
*  where vi denotes an element of the vector defining H(i),
*  and ui an element of the vector defining W(i),
*  and the dash denotes an zero element.
*  The main diagonal entries of A*B are stored in D and the
*  off-diagonal entries of A*B are stored in E.
*
*  =====================================================================
*
```

Appendix C

Makefile

```
##--------------------------------------------------------
## UC Davis
##  2003 Jenny Wang
## Purpose:
##   This file generates the executable for testing
##   new LAPACK-like routines: SORCSD, SGGQSV, SGGPSV,
##   and LAPACK routine, SGGSVD, as a reference to SGGQSV.
##
## Nov. 28, 2005
##
## Usage: make {imkl/netlib} < infile
##--------------------------------------------------------

help:
    @echo
    @echo "Usage: make {imkl/netlib} < infile"
    @echo
    @echo "imkl   - use the Intel's Math Kernel Library 7.2.1"
    @echo "netlib - use the source codes obtained from Netlib"
    @echo "infile - see .in files in \"input\" folder"
    @echo

imkl:   use_imkl

netlib: use_netlib

m = lapack/modified
u = lapack/not-in-IMKL
n = new

FC = ifort -w -D_Linux
G2C = /usr/lib/gcc-lib/ia64-redhat-linux/3.2.3

# modified LAPACK routines
OBJM = $m/alahdg.o $m/slatb9.o \
       $m/sgsvts.o $m/sckgsv.o $m/sggsvd.o

# un-modified LAPACK or BLAS routines that are not in IMKL
OBJU = $u/alareq.o $u/alasum.o $u/lsamen.o $u/sbdt01.o $u/second.o \
       $u/slabad.o $u/slacpy.o $u/slagge.o $u/slagsy.o $u/slamch.o \
       $u/slaran.o $u/slarnd.o $u/slaror.o $u/slarot.o $u/slatm1.o \
       $u/slatms.o $u/sort01.o $u/ssdot.o
```

```
# new LAPACK-style routines
OBJN = $n/csd/sorcs1.o $n/csd/sorcs2.o $n/csd/sorcsd.o \
       $n/qsv/sggqrj.o $n/qsv/sggqsv.o                 \
       $n/psv/sggbrd.o $n/psv/sggpsv.o                 \
       $n/tests/scsdts.o $n/tests/sckcsd.o             \
       $n/tests/sqsvts.o $n/tests/sckqsv.o             \
       $n/tests/spsvts.o $n/tests/sckpsv.o             \
       $n/tests/schkee2.o                              \
       $n/tests/ptmatrix.o

# ---- compile, link and execute using IMKL ----
use_imkl: $(OBJM) $(OBJU) $(OBJN)
    $(FC) $(OBJM) $(OBJU) $(OBJN) -L. \
    -L$(IMKL)/lib/64 -lmkl_lapack -lmkl_ipf -lguide  -o $@.out
    ./$@.out

#-------------------------------------------------

b = blas
# dependent BLAS routines for CSD
OBJBc = $b/for-csd/blasRoutines.o    $b/for-csd/scopy.o   \
        $b/for-csd/second.o          $b/for-csd/sgemm.o   \
        $b/for-csd/snrm2.o           $b/for-csd/srot.o    \
        $b/for-csd/sscal.o           $b/for-csd/sswap.o   \
        $b/for-csd/strmm.o           $b/for-csd/xerbla.o

# dependent LAPACK routines for CSD
c = lapack/for-csd
OBJC =  $c/ql/sgeql2.o $c/ql/sgeqlf.o $c/ql/sorg2l.o      \
        $c/ql/sorgql.o $c/ql/sorm2l.o $c/ql/sormql.o      \
        $c/svd/util/ieeeck.o     $c/svd/util/ilaenv.o     \
        $c/svd/single/sbdsqr.o   $c/svd/single/sgebd2.o   \
        $c/svd/single/sgebrd.o   $c/svd/single/sgelq2.o   \
        $c/svd/single/sgelqf.o   $c/svd/single/sgeqr2.o   \
        $c/svd/single/sgeqrf.o   $c/svd/single/sgesvd.o   \
        $c/svd/single/slabrd.o   $c/svd/single/slacpy.o   \
        $c/svd/single/slamch.o   $c/svd/single/slange.o   \
        $c/svd/single/slapy2.o   $c/svd/single/slarf.o    \
        $c/svd/single/slarfb.o   $c/svd/single/slarfg.o   \
        $c/svd/single/slarft.o   $c/svd/single/slartg.o   \
        $c/svd/single/slas2.o    $c/svd/single/slascl.o   \
        $c/svd/single/slaset.o   $c/svd/single/slasq1.o   \
        $c/svd/single/slasq2.o   $c/svd/single/slasq3.o   \
        $c/svd/single/slasq4.o   $c/svd/single/slasq5.o   \
        $c/svd/single/slasq6.o   $c/svd/single/slasr.o    \
        $c/svd/single/slasrt.o   $c/svd/single/slassq.o   \
        $c/svd/single/slasv2.o   $c/svd/single/sorg2r.o   \
        $c/svd/single/sorgbr.o   $c/svd/single/sorgl2.o   \
        $c/svd/single/sorglq.o   $c/svd/single/sorgqr.o   \
        $c/svd/single/sorm2r.o   $c/svd/single/sormbr.o   \
        $c/svd/single/sorml2.o   $c/svd/single/sormlq.o   \
        $c/svd/single/sormqr.o                            \
        \
        $c/tests/sasum.o         $c/tests/sbdt01.o        \
        $c/tests/slagge.o        $c/tests/slansy.o        \
        $c/tests/slaran.o        $c/tests/slarnd.o        \
        $c/tests/slarnv.o        $c/tests/slaror.o        \
```

```
        $c/tests/slaruv.o        $c/tests/sort01.o        \
        $c/tests/ssyrk.o

# dependent LAPACK routines for GSVD
g = lapack/for-gsv
OBJG = $g/slags2.o       $g/slapll.o       $g/stgsja.o

# dependent BLAS routines for PSVD
OBJBp = $b/for-psv/sdot.o            $b/for-psv/sgemm.o   \
        $b/for-psv/snrm2.o           $b/for-psv/srot.o    \
        $b/for-psv/sscal.o           $b/for-psv/ssdot.o   \
        $b/for-psv/sswap.o           $b/for-psv/strmm.o   \
        $b/for-psv/xerbla.o

# dependent LAPACK routines for PSVD
p = lapack/for-psv
OBJP = $p/sbdsqr.o       $p/slacpy.o       $p/slamch.o   \
       $p/slapy2.o       $p/slarf.o        $p/slarfb.o   \
       $p/slarfg.o       $p/slarft.o       $p/slargv.o   \
       $p/slartg.o       $p/slas2.o        $p/slascl.o   \
       $p/slaset.o       $p/slasq1.o       $p/slasq2.o   \
       $p/slasq3.o       $p/slasq4.o       $p/slasq5.o   \
       $p/slasq6.o       $p/slasr.o        $p/slasrt.o   \
       $p/slasv2.o       $p/sorg2r.o       $p/sorgbr.o   \
       $p/sorgl2.o       $p/sorglq.o       $p/sorgqr.o   \
       \
       $p/tests/sasum.o        $p/tests/sbdt01.o         \
       $p/tests/slagge.o       $p/tests/slange.o         \
       $p/tests/slansy.o       $p/tests/slaran.o         \
       $p/tests/slarnd.o       $p/tests/slarnv.o         \
       $p/tests/slaruv.o       $p/tests/slassq.o         \
       $p/tests/sort01.o       $p/tests/ssyrk.o

# dependent BLAS routines for QSVD
OBJBq = $b/for-qsv/blasRoutines.o    $b/for-qsv/isamax.o  \
        $b/for-qsv/scopy.o           $b/for-qsv/sgemm.o   \
        $b/for-qsv/snrm2.o           $b/for-qsv/srot.o    \
        $b/for-qsv/sscal.o           $b/for-qsv/sswap.o   \
        $b/for-qsv/strmm.o           $b/for-qsv/xerbla.o

# dependent LAPACK routines for QSVD
q = lapack/for-qsv
OBJQ = $q/sgeqpf.o       $q/sgerq2.o       $q/sgerqf.o  \
       $q/sggsvp.o       $q/slapmt.o       $q/sormr2.o

# dependent BLAS routines
OBJB =  $b/blasRoutines.o $b/xerbla.o $b/sscal.o $b/srot.o  \
        $b/scopy.o        $b/sdot.o   $b/snrm2.o $b/saxpy.o \

# ---- compile, link and execute ----
use_netlib: $(OBJM) $(OBJU) $(OBJN) $(OBJB)   \
       libcsd libgsv libpsv libqsv
    $(FC) $(OBJM) $(OBJU) $(OBJN) $(OBJB) \
        -L.  -lcsd_lib -lqsv_lib  -lpsv_lib -lgsv_lib \
        -L$(G2C) -lg2c -o $@.out
    ./$@.out

libcsd: $(OBJBc) $(OBJC)
```

```
    $(FC) -c $(OBJBc) $(OBJC) -L$(G2C) -lg2c
    ar rcv libcsd_lib.a $b/for-csd/*.o $c/ql/*.o \
    $c/svd/single/*.o $c/svd/util/*.o $c/tests/*.o
    ranlib libcsd_lib.a

libgsv: $(OBJG)
    $(FC) -c $(OBJG) -L$(G2C) -lg2c
    ar rcv libgsv_lib.a $g/*.o
    ranlib libgsv_lib.a

libpsv: $(OBJBp) $(OBJP)
    $(FC) -c $(OBJBp) $(OBJBp) $(OBJP) -L$(G2C) -lg2c
    ar rcv libpsv_lib.a $b/for-psv/*.o $p/*.o $p/tests/*.o
    ranlib libpsv_lib.a

libqsv: $(OBJBq) $(OBJQ)
    $(FC) -c $(OBJBq) $(OBJQ) -L$(G2C) -lg2c
    ar rcv libqsv_lib.a $b/for-qsv/*.o $q/*.o
    ranlib libqsv_lib.a

#-----------------------------------------------

clean:
    find . -name '*.o' -exec rm -f {} \;
    rm -f *.a *.out
#-----------------------------------------------
```